***p*-Cycle Spare Capacity Allocation for Large-Scale Networks**

by

Tiange Shi

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Engineering Management

Department of Mechanical Engineering
University of Alberta

ABSTRACT

Optical transport network failures are destructive, costly and inevitable. Therefore, extensive work has been conducted on survivable network designs. Survivable network designs are primarily built around network span failures, and a common approach for improving network resilience against span failures is by adding redundancy in span capacities. In recent years, a relatively new *p*-cycle survivable network design has drawn much attention due to its ring-mesh dichotomy that allows *p*-cycles to provide ring-like restoration speed with mesh-like protection efficiency. One approach to enhancing the *p*-cycle survivable network design is by optimizing allocation of its spare capacities. The spare capacity allocation problem optimizes spare capacity designs by placing selective pre-enumerated candidate cycles onto the network to achieve 100% network survivability with minimal allocation cost. Integer linear programming (ILP), heuristics, and meta-heuristics are commonly-adopted approaches for optimizing the *p*-cycle spare capacity allocation problem. Although extensive work has been conducted on *p*-cycle survivable network designs and *p*-cycle SCA problems using either linear programming or heuristic methods, there are still great opportunities for enhancement. For example, to the best of our knowledge, there has not been a *p*-cycle spare capacity allocation method proposed and tested particularly for large-scale networks.

The work herein addresses the problem of *p*-cycle spare capacity allocation by developing and evaluating two new algorithms. The first algorithm is a novel heuristic algorithm for generating efficient candidate *p*-cycles, referred to as the disjoint-paths Dijkstra cycle development (DDCD) algorithm. The DDCD is an iterative cycle development

method that is capable of generating high-performance candidate *p*-cycles in networks of any size. The DDCD outperforms some conventional cycle enumeration methods in small and large networks and is particularly desirable for large-scale networks that are over 80 nodes. The second algorithm is a novel GA model for optimizing the *p*-cycle spare capacity allocation problem, referred to as a GA-SCA model. The GA-SCA model provides a better optimized spare capacity allocation solution than that of CIDA. The GA operators, especially the mutation operator, are specifically designed and tuned to enhance the performance of the GA-SCA model.

To the ones who dream and dare to challenge.

For my father, Xiaonan Shi, mother, Jiangling Zhao, and HH.

ACKNOWLEDGEMENT

The work herein would not have been possible without the support, continuous guidance, and infinite patience of my supervisors, Dr. John Doucette and Dr. Tetsu Nakashima, who have set examples of excellence and professionalism.

I would like to extend my gratefulness to my MSc. examining committee members, Dr. Michael Lipsett and Dr. Dan Sameoto, for attending my virtual oral defence despite this challenging time of a global pandemic and offering valuable advice.

I would especially like to thank my research group members and friends, with whom I have shared many discussions and good laughs. I would like to thank Samira Doostie and Juan Tzintzun Ramos for countless fantastic conversations, discussions and collaborations. I would like to thank Abril Alvaro Munos, Farhad Haftani, and Shirin Geranmayeh for sharing knowledge and advice and always reminding me of the power of hugs (and virtual hugs) and smiles. I would also like to thank Andres Castillo and Wenjing Wang for their kind assistance and valuable advice when I started my thesis research.

Finally, I would like to dedicate this thesis to my parents, Beth DeBlock and Joel DeBlock, for their countless love and education. Thank you, HH, for being the wildest dreamers with me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

1.1 Background

Network survivability is the capability of a network to maintain proper functioning in the event of a network failure [1]. It is a fascinating topic to consider due to our growing social dependency on telecommunication and a pivotal aspect to consider when designing *optical transport networks* due to their ultra-high capacities [1]. With the advent of 5G technologies, artificial intelligence, cloud computing, etc., modern life and business transactions depend increasingly heavily on faster and more reliable *network traffic*. In the *2016-2020 ICT Market Review and Forecast* released by the Telecommunication Industry Associated (TIA), it was reported that some rapidly growing markets would experience an increase of 13.7% compound annual growth rate from 2015 to 2020. These markets include cloud computing services, business ethernet, Internet of Things (IoT), network virtualization, and intelligent transportation services, where IoT and network virtualization will experience more than 30% compound annual growth rate [2]. This prediction indicates a significant increase in the Internet demands in recent years and dramatic internet traffic growth that will continue in the years to come. To support increasing volumes of data traffic, the *Internet Protocol* (IP) and optical transport networks have become dominant forms of transmission [3].

Because significant volumes of data is carried on network traffic, any *failures* or disruptions on the transport network may lead to tremendous financial loss and social impacts on the services and transactions that rely on these infrastructures. For example, Amazon Web Services (AWS) experienced a significant outage in their US East region that lasted over 20 hours in April 2011 [4]. Since AWS is a major cloud computing platform that provides cloud services to a number of online sites and services, this major outage caused downtimes and service interruptions to various popular online sites including Reddit, Quora, SCVNGR (now known as LevelUp), Foursquare, etc. [5]. Based on a study by Cérin *et al.* [6] from the International Working group on Cloud Computing Resiliency (IWGCR), some major cloud service providers experienced an increase on annual downtime from 2012 to 2013. For example, the downtimes at SalesForce and Microsoft Windows Azure almost tripled from 2012 to 2013

(34.36 hours and 111.5 hours in 2012 to 84.72 hours and 272.04 hours in 2013, respectively) [6]. Therefore, designing networks with faster data transmission and better survivability are highly desirable [2]-[3].

Despite the fact that optical transport network failures or disruptions are destructive and costly, studies have shown that optical network outages and cable cuts are inevitable despite a considerable amount of effort taken to physically protect the cables [3], [7]. A network failure can occur at various network components, e.g., a *node* of a network, or a *span* of a network [1]. Because nodes of a network are often considered perfect, survivable network designs are primarily built around network *span* failures [1], [5]. A common approach for improving network resilience against span failures is by adding redundancy in span capacities, such as adding *spare capacity* units on each network span. These spare capacity units do not carry any traffic and are not cross-connected until restoration of a failure is needed [8]. Examples of such survivable network designs include *1+1 automatic protection switching (APS)* [9], survivable rings [10], *span restoration* [11], *path restoration* [12], *shared backup path protection (SBPP)* [13], and *p-cycles* [14]-[16]. Except for the 1+1 APS and survivable ring designs, allocation of spare capacity is carefully optimized when designing survivable networks [17].

In recent years, the relatively new *p*-cycles survivable network design has drawn a lot of attention. The *p*-cycles concept was proposed in the late 1990s [14]-[16] and has been widely studied due to its ring-mesh dichotomy, which allows *p*-cycles to provide ring-like restoration speed with mesh-like protection efficiency. Various studies have been conducted on *p*-cycle designs and selection methods which will be elaborated further in CHAPTER 3 of this thesis. *Integer linear programming (ILP)*, *heuristics*, and *meta-heuristics* are commonly-adopted approaches for optimizing *p*-cycle survivable network design problems [3]. Integer linear programming is a type of linear programming where some decision variables are integers. It is often solved by the *branch-and-bound* method, where a branching process explores the entire solution space by creating new sub-problems with new bounds [18]. An ILP can provide an optimal solution to a problem, however, the computational runtime increases extensively as a problem gets more complicated [19]. A heuristic method is a problem-specific method that is used for all or part of a problem [20]. Compared to LP or ILP, a heuristic method generally provides sub-optimal solutions; however, it can solve a complex problem within a relatively

shorter run time. Meta-heuristics are a type of heuristic method that are not problem-specific, and that can be widely used in a wide variety of optimization problems [3]. They are high-level structures or processes for solving optimization problems and can easily be customized into case-specific problems.

1.2 Motivation & Goals

Although extensive work has been conducted on *p*-cycle survivable network designs using either linear programming or heuristic methods, there are still great opportunities for further exploration of this topic. For example, to the best of our knowledge, there has not been a design proposed in particular for large-scale networks and tested using large test networks that are over 100 nodes. Therefore, the primary focus of this work herein is to scale up and advance the *p*-cycle protection design optimization problem for large-scale networks. In this thesis, we will challenge and scale up the *p*-cycles designing problem for large networks using heuristic and meta-heuristic methods. To be specific, a novel heuristic algorithm will be developed for enumerating highly efficient *p*-cycles in large-scale networks. Also, a suitable genetic algorithm model will be proposed for optimizing *p*-cycle network protection against single-span failures.

The research goals of this thesis can be generalized as the following:

1) Develop a novel heuristic *p*-cycle enumeration algorithm with strong performance
2) Propose a genetic algorithm model for *p*-cycle spare capacity allocation, including:
    - Proposing suitable chromosome encoding and optimal GA operators
    - Developing suitable repair mechanism for disrupted chromosomes
    - Developing effective and problem-specific mutation operation

1.3 Thesis Outline

The remainder of this thesis will provide a thorough discussion of relevant background concepts regarding *p*-cycles and relevant mathematical tools, followed by experimental and

computational set-ups, and two main contributions to the *p*-cycle network protection design problem for large-scale network topologies.

CHAPTER 2 introduces fundamental concepts and terminologies in transport network, graph theory and network survivability which will be used extensively throughout this thesis. This chapter also highlights some mathematical tools and algorithms (e.g., key search algorithms, linear programming, heuristics and meta-heuristics), which will build a solid foundation for developing our heuristic algorithm and GA model in later chapters of this thesis.

In CHAPTER 3, *p*-cycles fundamental concepts and metrics, and relevant previous research and studies will be presented. Previous work on *p*-cycle pre-selection methods, *p*-cycle enumeration methods, *p*-cycle protection design and optimization methods will be discussed in detail in this chapter.

CHAPTER 4 will introduce a key meta-heuristic method called a *genetic algorithm* (GA). Fundamental GA concepts, genetic operators designs, and applications of genetic algorithm in optimizing *p*-cycle protection in network survivability will also be discussed.

CHAPTER 5 is devoted to introducing the experimental methods and computational set-ups. The calibration network topologies (also referred to as "calibration networks") and test case network topologies used in the experiments conducted for this thesis works will also be introduced in this chapter.

CHAPTER 6 will propose a new heuristic *p*-cycle design method, which is referred to as *disjoint-path Dijkstra cycle development (DDCD)*. Detailed case-specific experimental designs and set-ups will be presented. The experimental results will be compared to a benchmark heuristic method from the literature and a conventional *depth-first search (DFS)* approach.

CHAPTER 7 will present a meta-heuristic approach to solving the *p*-cycle *spare capacity allocation problem* using genetic algorithm (GA). This novel GA design developed for this thesis work is referred to as a GA-SCA model. Specific chromosome encoding and GA operator design rationales (selection, crossover, mutation) will be explained in detail. The proposed GA-SCA model will be applied to various test case network topologies that are up to 140 nodes. Comprehensive experimental results and discussions will follow.

Finally, in CHAPTER 8, this thesis will conclude with a summary discussion of the main contributions of this thesis work, as well as recommendations for future studies.

CHAPTER 2. TRANSPORT NETWORK BASICS & METHODS

2.1 Graph Theory in Transport Networks

A *graph* can be denoted as $G = (V, E)$, which has a finite set of *vertices* $V = \{v_1, v_2, v_3, \ldots\}$, and a finite set of *edges* $E = \{e_1, e_2, e_3, \ldots\}$ [21]. Each edge in E connects two vertices in V and can be referred to as $e_1 = \{v_1, v_2\}$. An edge in E (e.g., e_1) connects two adjacent vertices in V (e.g., v_1, v_2). If v_1 is considered the origin of the edge e_1 and v_2 is the destination, and swapping these designations will change properties of the edge, then e_1 is a *directed edge*. A directed edge is drawn using an arrowhead pointing its direction. A graph with at least one directed edge is referred to as a *directed graph*. An *undirected graph* is a graph with no directed edge. In the case of an optical transport network, the terms *nodes* and *spans* are used to refer to vertices and edges, respectively [3]. In the context of this thesis, all transport network models are illustrated and treated as undirected network graphs.

A graph is considered a *weighted graph* if a value w_{ei} is associated with each edge e_i, and w_{ei} is considered the *weight* of the edge. In transport networks, weights of edges may be span costs, distances, capacities, etc. [3] Where edge weights indicates span capacities, the graph is considered a *capacitated graph*.

The number of edges incident on a vertex is referred to as the *degree* of the vertex. In transport networks, degrees of vertices are referred to as *nodal degrees*. The degree of a node is determined by the number of spans traverses that node. A network's average nodal degree can be calculated by $\bar{d} = \frac{2 * |E|}{|V|}$, where $|E|$ is the total number of edges (spans) and $|V|$ is the total number of vertices (nodes) [3].

Figure 2.1. illustrates two key terms used in transport networks, a *link* and a *span*. A link is the fundamental unit of capacity where nodal switching devices (OXC, ADM) function to interconnect capacity [7]. A span refers to a physical unity of a collection of such links in parallel between a pair of adjacent nodes. A failure of a span will cause the failure of all the links associated with the span simultaneously [3].

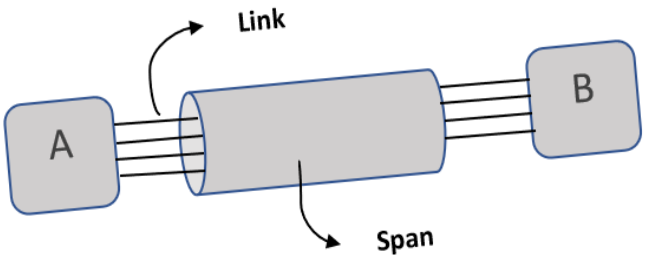


Figure 2.1. Illustration of a link and a span.

Figure 2.2 distinguishes a *route* from a *path* in the context of transport networks. A route *r*efers to a general outline of the sequence of connected spans, whereas a *path* is a particular cross-connected sequence of links that are embedded within a route. Therefore, a route can consist of multiple paths. As shown in Figure 2.2, there are two paths on the same route.

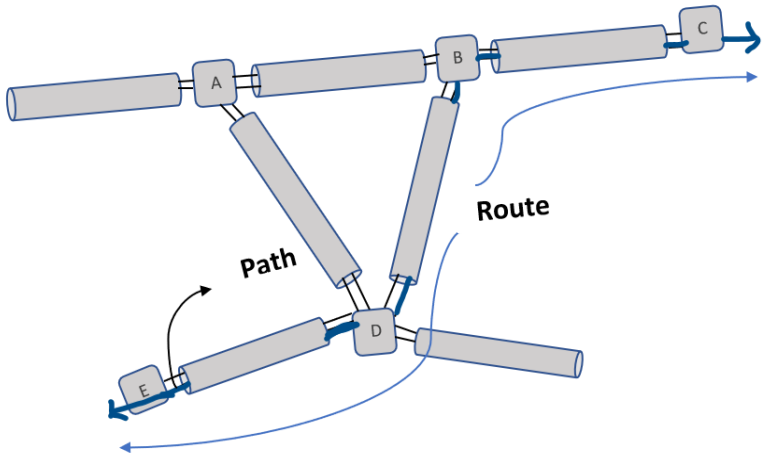


Figure 2.2. Illustration of a path and a route.

Disjoint Routes

Two routes are considered *disjointed* if they do not share any element in common. Two *span-disjoint* routes do not share any span; however, they may share common nodes other than their end nodes (as shown in Figure 2.3(a)). In the case of *node-disjoint* routes, two routes share no common nodes except the two end nodes (as shown in Figure 2.3(b)). In an undirected graph G = (V, E), *survivability* of the graph is determined by the number of node-disjoint paths (x_{ij})

between pair of nodes i, j. To be specific, transmission is ensured between i and j until x_{ij} -1 of nodes fail [21].

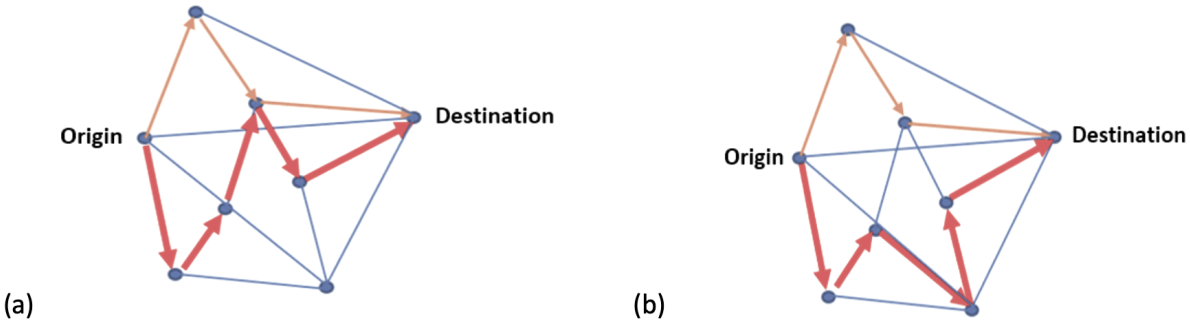


Figure 2.3 Illustrations of a span-disjoint route (a) and a node-disjoint route (b) between an origin node and destination nodes.

Two-connectivity vs. Bi-connectivity

As was explained in [3], a network graph is a valid *two-connected* graph if there are at least two span-disjoint routes between every pair of nodes (see Figure 2.4 (a)). Compared to a two-connected graph, a valid *bi-connected* graph has at least two node-disjoint routes between every node pair (see Figure 2.4 (b)). In the context of transport networks, *two-connectivity* is required for any single span failure survivability in a transport network, whereas *bi-connectivity* is required for survivability of any single-node failure.

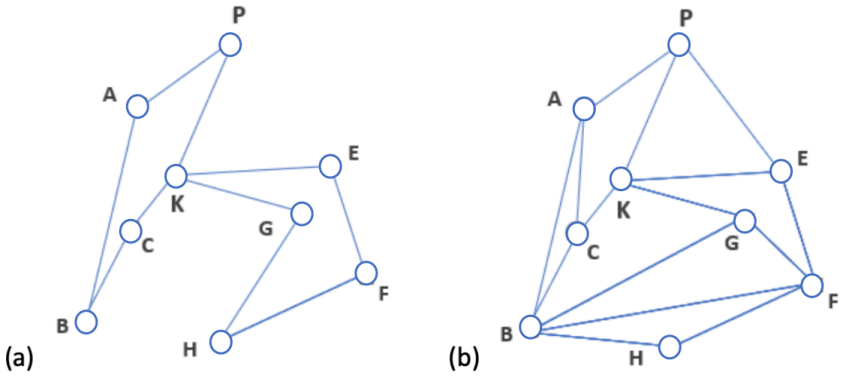


Figure 2.4 Examples of (a) a two-connected graph and (b) a bi-connected graph.

Hamiltonian Cycle vs. Eulerian Cycle

A *Hamiltonian cycle* is a cycle that traverses all the nodes in a graph where each node is passed through once and only once, whereas a *Eulerian cycle* is a cycle that visits all the spans once and only once [3] (see Figure 2.5). In a Eulerian cycle, a node may be traversed more than once. As shown in Figure 2.5, the Eulerian cycle F-G-D-B-F-E-A-B-C-D-F passes through nodes B and D twice. A Eulerian cycle protects every span of a network with an on-cycle relationship. A Hamiltonian cycle also has the potential to protect every span of a graph, however, through both on-cycle and off-cycle protection relationships [3]. This is a particularly important concept in *p*-cycle network protections. As shown in Figure 2.5, the Hamiltonian cycle F-G-D-C-B-A-E-F can also protect the spans B-F and B-D which *straddle* the cycle. CHAPTER 3 of this thesis will further discuss the concepts and mechanisms of *p*-cycles and *straddling spans*.

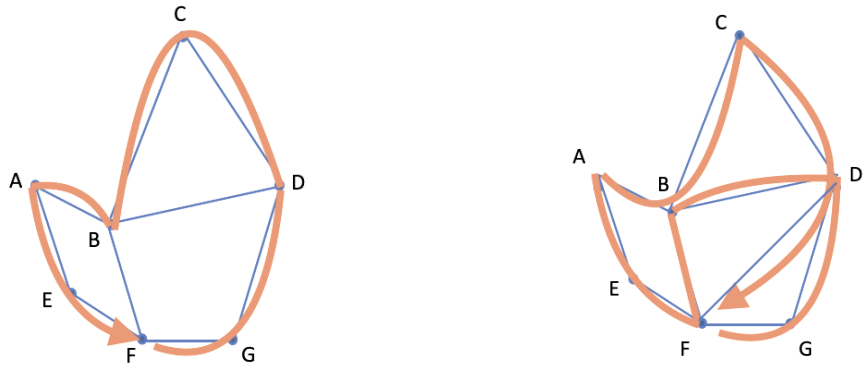


Figure 2.5 Examples of a Hamiltonian cycle F-G-D-C-B-A-E-F (left) and a Eulerian cycle F-G-D-B-F-E-A-B-C-D-F (right).

2.2 Optical Transport Network Basics

A *transport network* (also known as a *backbone network*) provides fast and efficient network data transmission for various types of data, such as voice, images, videos, etc. Sets of multi-channel point-to-point data transmissions are managed through transport networking, in order to create a virtual network domain for other services to operate on [22]. Current optical transport networks, using *dense wavelength division multiplexing (DWDM)* [22] and *optical*

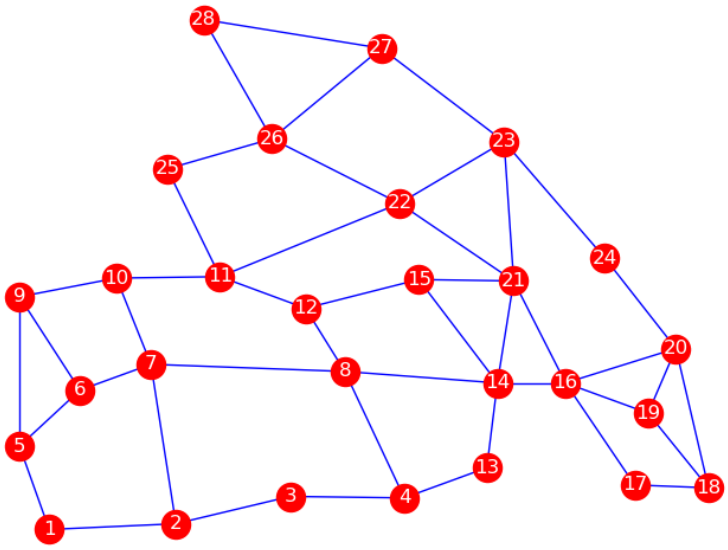


Figure 2.6 *US long haul* network with 28 nodes and 45 spans

cross-connects (OXC) [17], allow a massive amount of data to be transmitted at ultra-high speed [3]. A transport network consists of nodes and spans that connect two *neighbouring* nodes. In the case of an optical transport network, a node can be an OXC device, an a*dd/drop multiplexer* (ADM) device, or an *Internet Protocol* (IP) *router*. A span generally refers to an optical fibre cable. Figure 2.6 illustrates an example of an optical transport network topology, which is the *US long haul network* with 28 nodes and 45 spans [78]. The nodes and spans represent various cities and connecting fibres, respectively.

In transport network operations, various network components play crucial roles. Although a detailed introduction of all these key components is beyond the scope of this thesis, some key terms and definitions regarding high-level optical transport networks, which will be referenced throughout this thesis, are explained in detail as follows:

Wavelength Division Multiplexing (WDM)

Wavelength division multiplexing (WDM) is an optical transport network technology that has favourable long-haul network transport capacity efficiency. In WDM, each optical fibre is capable of carrying multiple optical carrier wavelengths [22]. Each wavelength has its distinct

payload (for instance, a formatted signal). Different payloads are appropriately spaced apart and are modulated onto each optical carrier. With growing service demands over long haul transportation, it is suggested installing WDM systems are more economical than installing and upgrading fibre systems [3].

A preferred such system is called *dense* wavelength division multiplexing (DWDM), where a larger amount (> 40) of wavelengths are carried on each fibre with tighter frequency spacing [80]. In contrast to DWDM, an earlier technology which is referred to as a *coarse wavelength division multiplexing (CWDM)* allows only two to four wavelengths with wider frequency spacing [3].

Add/Drop Multiplexer (ADM) vs. Optical Cross-Connect (OXC)

The ADM and the OXC systems are two types of nodal switching devices in optical transport networks, where fibre optic cables are connected to [17]. An ADM line-terminating device has two line-rate interfaces, *East* and *West* lines. An ADM also has local add/drop ports to allow tributary signals to enter (add) or exit (drop) the main line-signals that pass through the device from *West* and *East* lines. ADMs are generally used in survivable ring designs, such as in *bidirectional line-switched rings* (BLSR) [10] and *unidirectional path-switched rings* (UPSR) [10]. An OXC terminates and switches optical signals in a fibre optic network, and are usually used in mesh-based survivable networks.

Network Demands

Demands on a transport network refer to an aggregation of all traffic flows from an origin node to a destination node of the transport network [24]. A *demand unit* is expressed in terms of the number of transmission capacity units required by the traffic flow aggregates, such as whole *lightpaths* and *OC-48* [24]. OC-48 standards for a particular category of *Optical Carrier* (hence, *OC-n*) line with transmission rate of 2488.32 Mb/s (48 multiplies the base rate of 51.84 Mb/s), where an optical backbone network is usually managed [24]. In the context of this thesis, one unit of traffic demand takes on a whole lightpath, hence one unit of working

capacity on a span between an origin node and a destination node of a network. In addition, only integer units of optical network traffic demands will be considered in this thesis unless stated otherwise.

Working Path vs Spare Path

A *working path* is a path that carries traffic demands (or, units of *working capacity*) during normal operations. A *protection path*, however, carries spare traffic capacity (or units of spare capacity) that is available for deployment when a failure occurs [25]. In general, spare capacities may refer to protections fibres in APS or rings or spare channels designed in mesh-based restorable networks [3]. Lengths of the working and spare paths vary based on the number of spans along the path between the origin node and the destination node. A network's working capacity status and spare capacity status will be looked at while assessing a network's *capacity efficiency*, which is the reciprocal of the *network redundancy*.

Figure 2.7 illustrates a working path of D-C-B and outlines a spare path of D-G-F-B that can be used to restore the working path in case of a failure. The spans D-C and C-B carries 4 and 3 units of working capacity, respectively. The spans D-G, G-F and F-B carry 4, 4, and 5 units of spare capacity that are available for restoring a failure.

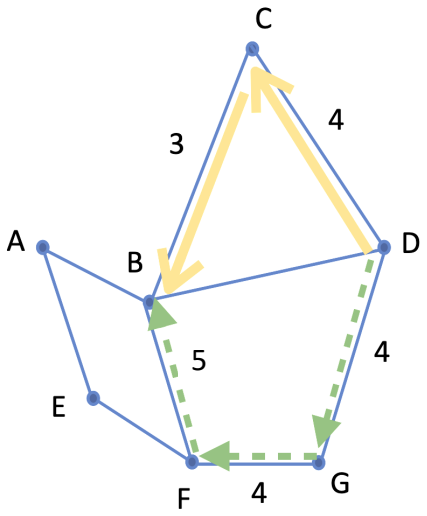


Figure 2.7 Example of a working path (and units of working capacity) and a spare path (and units of spare capacity) in a network.

Network Redundancy

In the context of this thesis, a redundancy calculation is used to measure a network's efficiency. When all span costs (c_i) are equal ($c1 = c2 = c3 = c4 = ...$) or are not available for efficiency calculation, redundancy of a transport network is calculated as a *capacity redundancy* (R_{cap}) of the network. R_{cap} is defined as the total units of all the spare capacity (S_i) allocated on all spans (S) divided by the total units of all the working capacity (W_i) across all spans of the transport network [7], as shown below in Eq. 2.1.

$$Rcap = \Sigma Si \,/\, \Sigma Wi, \quad \forall i \in S \qquad \text{(Eq. 2.1)}$$

When capacity costs are considered, and span costs are not identical, *capacity cost redundancy* (R_{cost}) is calculated for transport network design efficiency. The R_{cost} is defined as the total cost of spare capacity (summation of costs of spare capacity on each span) divided by the total cost of working capacity (summation of costs of working capacity on each span)(Eq. 2.2) [7].

$$Rcost = \Sigma(Si \times ci) \,/\, \Sigma(Wi \times ci), \quad \forall i \in S \qquad \text{(Eq. 2.2)}$$

While designing a survivable transport network in this thesis, network redundancy calculation will be taken into consideration (R_{cap} or R_{cost} depending on the design of the span costs in test networks) because the calculation indicates how efficient the design is in terms of using its spare capacity. A lower redundancy indicates a lower total spare capacity is used to protect the total working wavelengths, hence better efficiency. For the same network topology and the same set of network traffic demands, a survivable ring network design will have higher redundancy than a mesh-based design or a *p*-cycle design.

2.3 Network Survivability

As mentioned in CHAPTER 1, Network survivability is the capability of a network to maintain proper functioning in the event of a network failure [1]. A network failure event can be either a node failure or a span failure [21]. In practice, a node failure of a network can be a

failure of a data centre, a city, or a switching point. A span failure, however, is a complete cable cut between two nodes [21]. Failure of a node will cause the failure of all spans that traverse that node simultaneously. As discussed previously in this thesis, network failures are costly and inevitable; therefore, studying and advancing survivable network designs that restore or protect network failures is much needed [17]. Over the past two decades, various types of network *restoration* and *protection* mechanisms have been studies by researchers [9]-[13], [17]. In this section herein, we will introduce some common survivable network schemes.

2.3.1 Survivable Rings

Survivable rings are pre-configured closed-loop structures (ring-like) that use ADM nodal devices to switch optical signals. In a survivable ring structure, half of the transmission capacity can be used for working lightpath traffic routing [10]. In contrast, the other half is reserved as a backup channel for rerouting disrupted lightpath when a failure occurs.

There are two types of survivable rings: *unidirectional path-switched rings* (UPSR) and *bidirectional line-switched rings* (BLSR) [3]. A UPSR contains a working fibre and a protection fibre. The working fibre transmits the working signals in one direction, and the protection fibre transmits the backup signal in the opposite direction. In the event of a single span failure, two end nodes next to the failure perform a tail-end transfer and *switch* the signals to the backup channel to reroute the signal around the ring [7]. Figure 2.8 shows a UPSR during normal operation (left) and its protection switching operation after a failure occurs (right).

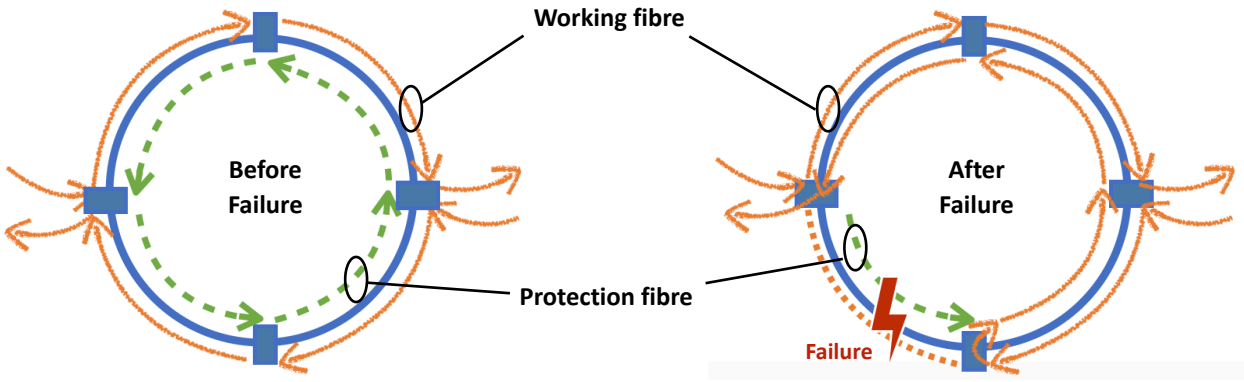


Figure 2.8 Illustration of UPSR protection mechanism after a failure occurs.

In a bidirectional line-switched ring, a failure may involve a span or several spans. When a failure occurs, working demands are protected through a *loop-back* mechanism. Figure 2.9 shows a *4-fibre* BLSR where a pair of bidirectional fibres is designated for working demands and a separate pair is for protection. When a failure occurs, the entire working demands are looped back from working fibres to the protection fibres at the nodes on both ends of the failed section [3]. Because the protection signal is share across the entire ring and the same path (unoccupied spans) can be reused for other demand units, a BLSR is more capacity-efficient than a UPSR under general demand patterns [3], [10].

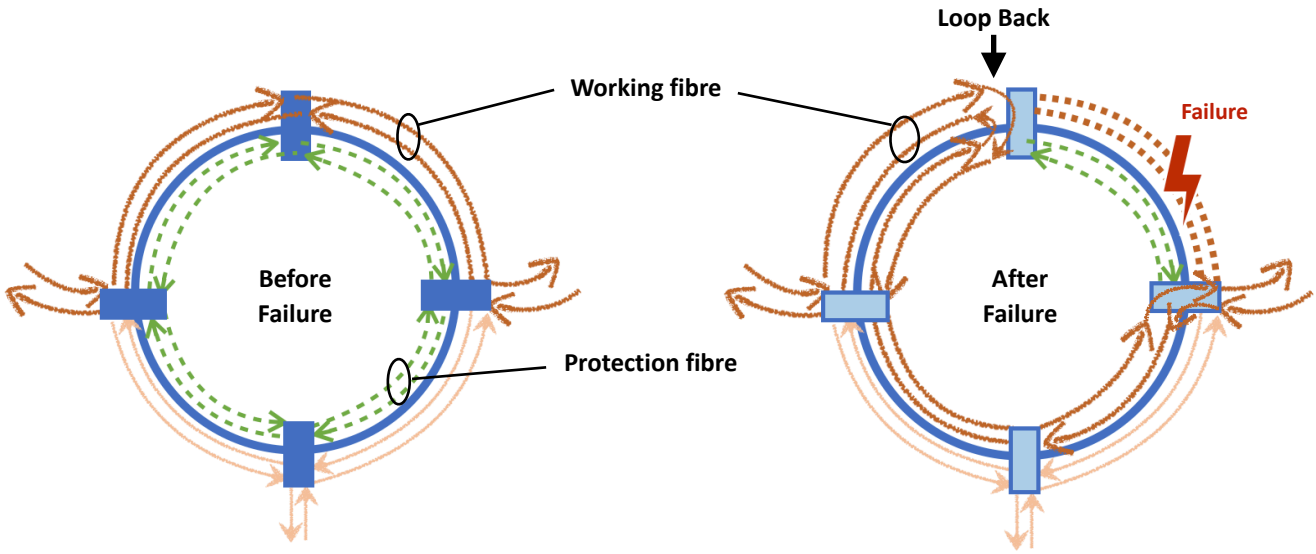


Figure 2.9 Illustration of BLSR protection mechanism after a failure occurs.

2.3.2 Mesh Network Survivability

Mesh network survivability is a type of network survivability scheme, where protection or restoration of paths utilizes spare capacities distributed across entire network instead of following pre-constructed capacity structure (as in survivable rings) [17]. Sharing of spare capacities allows survivable network designs with lower capacity redundancy [17]. The working paths routing in mesh network survivability usually follow a shortest path method [3].

Mesh network survivability mechanisms can be further categorized under *localized* and *end-to-end* restoration and protection mechanisms [25]. Localized restoration refers to the

survivability mechanism where the restoration paths are constructed between the two end nodes that are adjacent to the failure itself [17]. Span restoration is an example of a localized mesh restoration mechanism [11], where a set of local restoration paths are formed between immediate end nodes of a failed span to re-route the demands as illustrated in Figure 2.10.

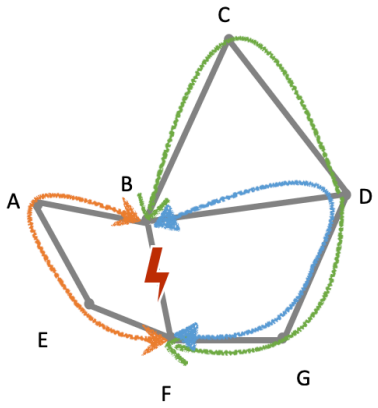


Figure 2.10 An illustration of span restoration.

End-to-end restoration or protection mechanisms establish spare routes between origin and destination nodes for any affected demand units [17]. This approach effectively deploys more network resources (e.g., exploring more disjoint routes or utilizing unoccupied existing-working routes). Therefore, an end-to-end restoration or protection scheme tends to be more capacity-efficient than a localized restoration mechanism [17], [25]. Shared backup path protection (SBPP) [13] and path restoration (PR) [12] are examples of end-to-end restoration or protection schemes. In SBPP, a set of end-to-end backup routes are formed and are fully disjoint from corresponding working routes. By doing so, capacity efficiency may be compromised. However, SBPP allows the sharing of spare capacities on the backup routes and allows activation of signal switchover by affected end nodes without prior knowledge of a failure [13]. Therefore, SBPP is considered "failure-independent" and is also referred to as *failure independent path protection* (FIPP). As shown in Figure 2.11, node pairs C-E and D-G route their demands via completely disjoint working paths (C-A-E and D-G, respectively) as drawn in solid lines. Therefore, no single-span failure will trigger both working paths (C-A-E and D-G) to fail at the same time so that their backup paths (dotted lines) will be occupied simultaneously. In this case, both backup paths can share the spare capacity on their common span B-F.

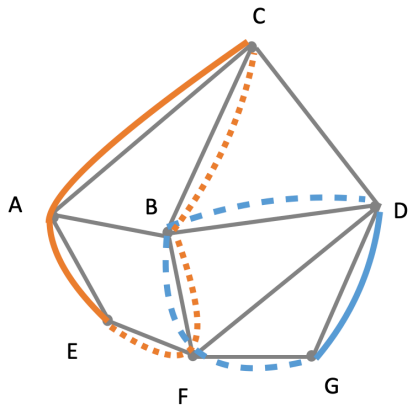


Figure 2.11 An illustration of shared backup path protection.

Path restoration (PR) is also an end-to-end restoration scheme. However, unlike in the case of an SBPP, the formation of end-to-end backup routes occurs in real-time in response to an actual failure occurs on a working path [12]. Therefore, PR is considered "failure-dependent". Because the failure is known prior to restoration, it is possible to re-use the working capacity on the surviving segments of the affected working route. This special mechanism of "releasing" the surviving "stub" portions to be re-used for restoration is referred to as "stub-release", which makes it the most capacity-efficient survivability mechanism [12]. Figure 2.12 shows a path restoration with sub-release. C-B-F and C-B-A-E are two working paths that share a common failure (a cut on span C-B). With stub-release, the restoration paths C-D-B-F and C-A-E each picks up a portion of the surviving working paths (B-F and A-E, respectively) and re-use their working capacities for restoration.

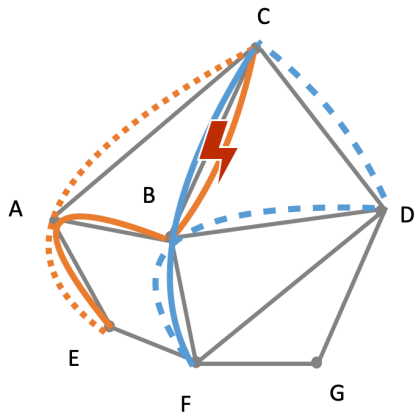


Figure 2.12 An illustration of path restoration with stub-release.

2.3.3 *p*-Cycles

As discussed in previous sections, survivable rings may be preferred due to simpler mechanisms and faster restoration processes; however, they are less capacity-efficient than mesh-restorable network designs. Mesh network survivability schemes are favoured for their enhanced capacity efficiency, despite relatively longer restoration time [17].

p-Cycle protection is based on the optimal formation of *p*-cycles in the spare capacity of a mesh-restorable network, which was proposed in [14]-[16]. The *p*-cycles are formed prior to any network failures and are formed out of the previously un-connected spare links of a mesh-restorable network [14]. Figure 2.13 (a) illustrates a *p*-cycle (bolded), where the spans traversed by the *p*-cycle are on-cycle spans. A *p*-cycle provides one unit of spare capacity to all the on-cycle spans and offers two units of spare capacity for the spans that straddle over the cycle. This type of spans, which has its two end nodes on the cycle but not the span itself, is referred to as straddling spans to the *p*-cycle. Straddling spans carry two working paths for each protection path on the *p*-cycle they straddle, but they do not retain any spare capacity themselves [3]. Figure 2.13(b)-2.13(c) indicate two scenarios where a *p*-cycle can protect a span failure. When an on-cycle span fails (as shown in Figure 3.2(b)), the surviving portion forms one restoration path between both nodes adjacent to the failure. This protection operation is similar to that of a bidirectional line-switched ring. In Figure 3.2(c), if a failure occurs not on any part of the *p*-cycle but on a span that straddles the *p*-cycle, two restoration paths will be available for restoring the failure around the failed spans.

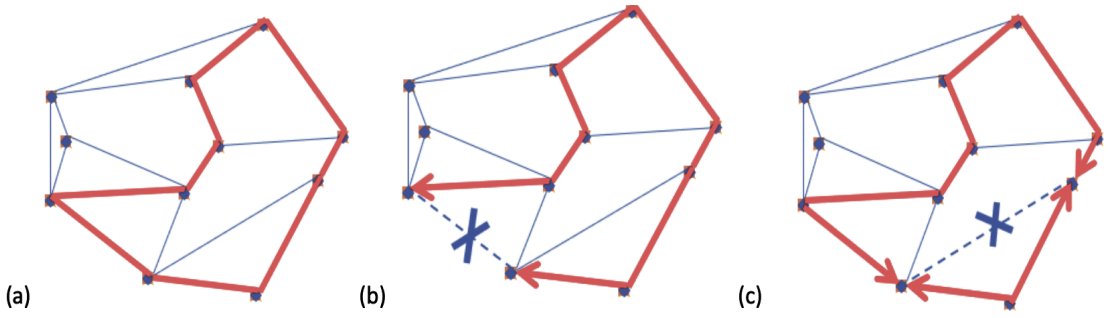


Figure 2.13 (a) a *p*-cycle. (b) a failure occurs on an on-cycle span, and (c) a failure occurs on a straddle span.

p-Cycle is a relatively new network survivability scheme that combines both *ring-like* restoration speed and *mesh-like* protection efficiency [14]-[16]. It is "ring-like" mostly because the topology of a *p*-cycle is similar to that of a ring. Additionally, the signal switching mechanism of a *p*-cycle is functionally similar to that of a BLSR. The restoration speed of a *p*-cycle depends on the time taken for two end nodes to perform signal switching from a failed working path to a pre-cross-connect protection path. This resembles the mechanism of a survivable ring where routing of the protection path is determined before a failure occurs. Despite its ring-like signal switching mechanism and restoration time, a *p*-cycle is more structurally accessible for restoration in more ways, which contributes to its "mesh-like" efficiency [14]. Details regarding *p*-cycles will be further discussed in CHAPTER 3 of this thesis.

2.3.4 *p*-Cycles vs. Survivable Rings

Even though rings and *p*-cycles share some similarities topologically and functionally, various studies have discussed the comparison between rings and *p*-cycles in optical networks [26]-[28]. Some studies have suggested more efficient spare capacity utilization in *p*-cycles protection as compared to that of ring protections [29]. In general, a *p*-cycle protection scheme is a more plausible protection mechanism in the following aspects:

Firstly, when a span on a *p*-cycle fails, two end nodes adjacent to the failure are involved in switching the traffic signals to the protection path. No real-time signalling is needed between the two end nodes for the switching (unlike in rings) [26]. Secondly, in survivable rings, the protection path routing must comply with the working routing and is structurally restricted by it. However, compared to rings, *p*-cycles are formed in the sparing capacity pool of a network, which does not collide with the routing of working paths [26]. Additionally, *p*-cycles have the flexibility to be restructured and modified, whereas rings cannot be modified once they are deployed [27]. Finally, *p*-cycles can provide network protection with redundancy less than 100%, whereas ring-based protection has a redundancy of at least 100%. With its capability to protect straddling spans, *p*-cycles can provide more efficient overall protection to the whole network [27].

2.3.5 Network Protection vs. Network Restoration

Previously, there have been two types of survivability mechanisms against network failures that we used to differentiate from one another: *network protection* and *network restoration*. "Network protection" refers to the type of network survivability mechanism where the cross-connections and allocation of spare capacities are pre-determined. The backup routes are determined prior to a failure [14]. Some survivable network schemes provide dedicated protection where each traffic demand is protected by a dedicated backup path (e.g., in 1:1 APS), whereas some provide shared protection (e.g., in SBPP). "Network restoration" does not require the provision of spare capacity in advance. In response to a network failure, network restoration schemes allow an alternative route to be configured in real-time to restore each failed connection (e.g., in span restoration and path restoration).

In recent years, "network protection" and "network restoration" are often used interchangeably. Going forward in this thesis, we will be using these two terms interchangeably.

2.4 Mathematical Tools - Selected Search Algorithms

2.4.1 Finding the Shortest Path - Dijkstra's Algorithm

Dijkstra's algorithm is a search algorithm that finds the shortest path between a pair of nodes in a weighted network with non-negative edge weights, and it can be applied in either a directed or an undirected graph [30]. The problems discussed in this thesis will simulate the real-world network cases with non-negative edge weights (e.g., *Euclidean* distances). Therefore, studies conducted in this thesis will be using Dijkstra's algorithm to find the shortest path from an origin node to a destination node [30]-[31]. Dijkstra's algorithm starts at a node of origin and explores the entire network in a process as outlined in the following steps [30]-[31]:

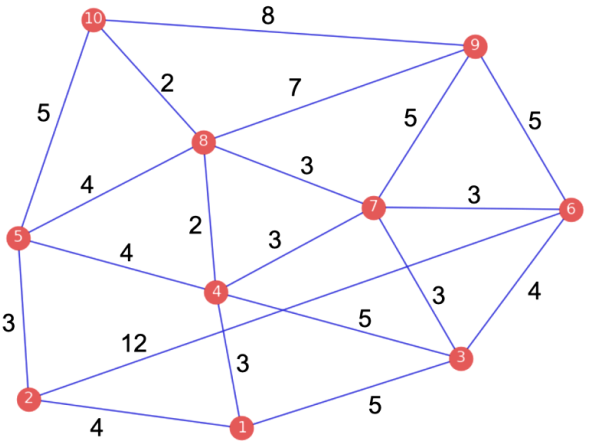
Step 1: The process starts at a node of origin. *Scanning* all the neighbouring nodes that are adjacent to the node of origin (also referred to as the *predecessor node*) and assign them *temporary labels* in the format of {T, total distance to the node of origin, predecessor node ID}, where "T" represents a temporary label.

Step 2: Choose the temporarily labelled node with the smallest total distance from the node of origin and replace the temporary label with a *permanent label* in the format of {P, total distance to the node of origin, predecessor node ID}, where "P" stands for a permanent label. If more than one temporarily labelled node has the same smallest total distance, which indicates more than one shortest path, we can randomly pick one of them. If the permanently labelled node has no neighbouring nodes, skip it and move on to the next permanently labelled node.

Step 3: Continue scanning all the adjacent nodes of the newly permanently labelled node. Assign temporary labels to the nodes that are not yet labelled and skip the ones with permanent labels. For previously labelled nodes, update their temporary labels if the new total distances to the node of origin are smaller than previously assigned values.

Step 4: Repeat Steps 2 and 3 until the node of destination is permanently labelled. Obtain the shortest path from the node of origin to the node of destination by following along all predecessor nodes in permanent labels.

Figure 2.14 illustrates an example of finding the shortest path between Node 1 to Node 10 using Dijkstra's algorithm in a network with 10 nodes and 20 spans (10n20s). Figure 2.14(a) presents the network topology where each span weight (Euclidean distance between two end nodes of a span) is indicated next to each span. Figure 2.14(b) tabulates the scanning process as described above in Steps 1-4. Therefore, the shortest path between Node 1 to Node 10 in the 10n20s network is {Node 1 → Node 4 → Node 8 → Node 10} with the shortest distance of 7.



(a)

Source Node	Neighbouring Nodes	Distance	Temporary Labels	Permanent Labels
Node 1	Node 4	3	{T, 3, Node 1}	{P, 3, Node 1}
	Node 2	4	{T, 4, Node 1}	
	Node 3	5	{T, 5, Node 1}	
Node 4	Node 5	4	{T, 7, Node 4}	
	Node 8	2	{T, 5, Node 4}	{P, 5, Node 4}
	Node 7	3	{T, 6, Node 4}	
Node 8	Node 10	2	{T, 7, Node 8}	{P, 7, Node 8}
	Node 9	7	{T, 14, Node 8}	
Node 10	Terminate process	-	-	-

(b)

Figure 2.14 Example of Dijkstra's algorithm in a network with 10 nodes and 20 spans.

A pseudo-code for Dijkstra's algorithm [30] can be generalized as follows:

```
function Dijkstra(Graph, source):
    initialize node set N
    for each vertex v in Graph:
        initialize distance[v], visited[v] as empty
        add v to N
    distance[source] ← 0
    while N is not empty:
        u ← node in N with min distance[u]
        remove u from N
        for each neighbour v of u:
```

```
                distance2 ← distance[u] + length(u, v)
                if distance2 < distance[v]:
                    distance[v] = distance2
                    visited[v] = u
    return distance[], visited[]
```

In *p*-cycle enumeration problems, Dijkstra's algorithm can be used to generate eligible *p*-cycles [32]-[34] by applying the algorithm twice between a pair of nodes. For example, Zhang and Yang [32] proposed a heuristic *p*-cycles generation algorithm where a *p*-cycles is formed by running Dijkstra's algorithm twice consecutively on a pair of origin-destination (O-D) nodes to generate two node-disjoint paths. The two node-disjoint paths are then joined to form a bi-connected cycle with one straddling link. Dijkstra's algorithm is also used for working path routing in various *p*-cycle protection optimization studies [32]-[37].

2.4.2 Finding All Distinct Routes - Depth-First Search (DFS) Algorithm

Depth-first search (DFS) algorithm is a search algorithm used for a thorough exploration of all the reachable vertices of an undirected or directed graph. The algorithm starts at an arbitrary *root node* on a graph, exploring all neighbouring nodes (either towards the left side of the graph or to the right side of the graph) and spans as "deep" as possible until all the nodes and spans on the current path are explored [38]. If there is any *unexplored* nodes remain in the graph, the DFS process will start to *backtrack* to the next reachable unexplored node. The entire search process is repeated until all the nodes in the graph are explored. During the search, nodes in a graph can be "marked" with different colours indicating their status [38]. For example, all nodes are *white* initially ("unexplored") and are marked *grey* once they are "discovered". Once exploration of a node and its adjacency list is completed, the node will be coloured *black*. The output of a DFS algorithm is a *depth-first forest* with a collection of *spanning tree* [38].

Figure 2.15 illustrates an overall search process of a DFS algorithm. In this example, the search process starts at Node A in the original graph (as shown in Figure 2.15(a)) and explores the network from the left side (Node A → Node B). Figure 2.15(b) outlines the full details of the search process. The solid directed lines marks the *forward edges* that point from a node of

origin to a *descendant* node, whereas the dotted directed lines indicates the *back edges* which point from a node to its *ancestor* node. The numbers next to the directed lines indicate its search sequence. For example, the forward edge of "Node A → Node B" is the first edge explored during the process whereas the back edge of "Node B → Node A" is explored at the 12th step.

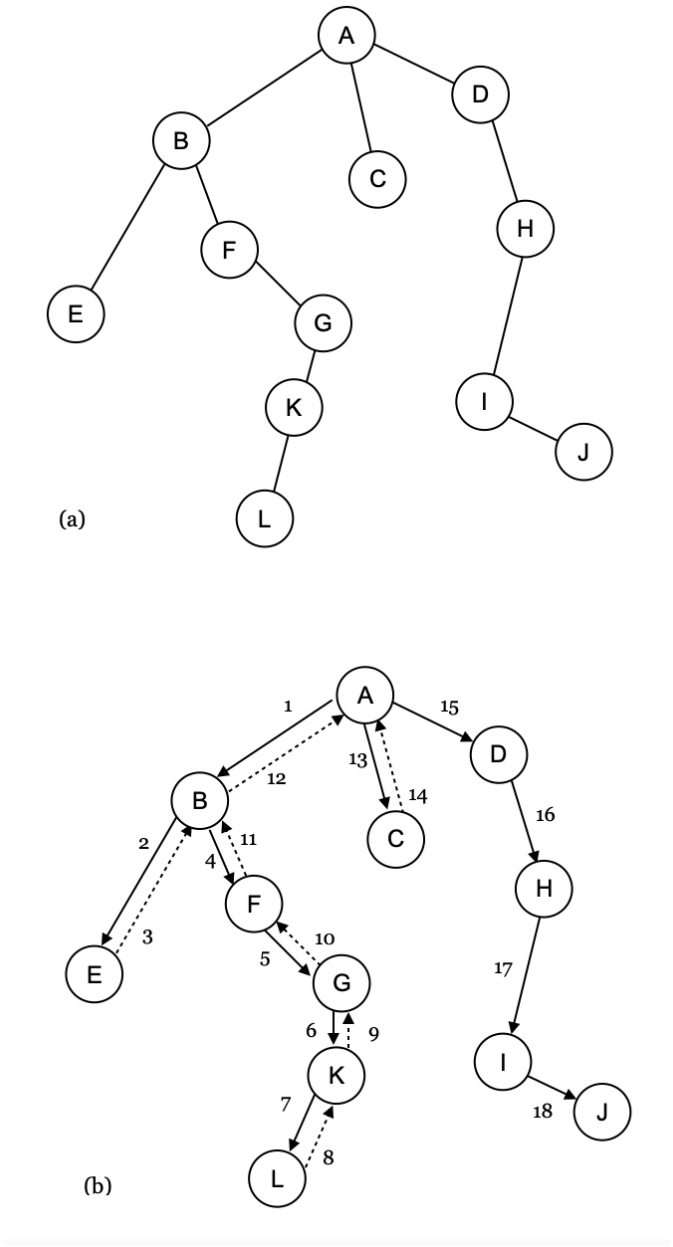


Figure 2.15 Example of the depth-first search algorithm.

Pseudo-code for a classic DFS algorithm [38] is outlined as follows, where the graph *G* can be either a directed graph or an undirected graph:

```
function DFS(G):
 for each vertex u ← V[G]:
       do colour[u] ← WHITE
           pred[u] ← null
  for each vertex u ← V[G]:
       if colour[u] == WHITE:
            do DFS-Visit(u)

function DFS-Visit(u):
   colour[u] ← GREY
   for each v adjacent to [u]:
       if colour[v] == WHITE:
            pred[v] ← u
                DFS-Visit(v)
colour[u] ← BLACK
```

In the above pseudo-code, whenever a vertex *v* is discovered as a result of scanning the adjacency list of a predecessor vertex *u* (already explored), this action is recorded as setting the predecessor field of *pred[v]* to *u* (`pred[v] ← u`). In addition, all vertices are initialized with colour WHITE and are updated as GREY once they are "discovered". Once exploration of a vertex is completed, the vertex is labelled as BLACK.

In *p*-cycle protection design problems, DFS algorithms can be used to enumerate sets of candidate cycles as a preparatory step for the subsequent cycle placement process[33]-[34], [39]-[40] or for the working path routing [39]. In addition, DFS algorithm can also be used find the *k*th shortest path by conducting a *topological sorting* because it is capable of finding all the paths in a network [3], [25]. In this thesis herein, the depth-first search algorithm will be implemented to enumerate candidate cycles in CHAPTER 6 as one of the benchmark algorithms to be compared with our novel heuristic algorithm.

2.5 Mathematical Tools - Linear Programming

2.5.1 Key Terminology

A *linear programming* (LP) method is a type of *mathematical programming* method used for planning decisions, where an *objective function* is optimized (e.g., maximized or minimized) and one or more *constraints* are satisfied [19]. A linear programming model is a constrained *optimization* model which include *decision variables*, an objective function, and constraints [19]. *Optimization* is a process of searching for the best solution(s) to a particular *operations research* problem; *operations research* refers to a scientific decision making process that aims at pursuing the best way to plan and operate a system [19], [41].

In a linear programming model:

An *objective function* is a function that represents a problem to be either maximized or minimized, and it reveals how much each decision variable contributes to the value to be optimized [19]. The contribution to the objective function from each decision variable is *proportional* to its value [3]. An objective function is usually denoted by $f(x)$.

Constraints are the limitations and conditions that must be satisfied in search of solutions [19]. Each constraint of a LP model includes one of the three relationships: "≤", "≥" or "=" [19]. "≤" means that the constraint function on the left-hand side (LHS) of the relationship is less than or equal to the right-hand side (RHS). "≥" indicates that the LHS of the constraint relationship appears to be greater than or equal to the RHS. Whereas, in a "=" relationship, the LHS and RHS are equal. For example, a constraint may look like $f_1(x) \geq a_1$ or $f_2(x) \leq b_1$.

Decision variables are values that must be determined to solve an optimization problem. They are under our control and their values have direct impacts on performance of an LP model [3]. Decision variables are usually represented by $\boldsymbol{x_i} = \{x_1, x_2, x_3, \ldots, x_n\}$ for a problem with n variables. A *continuous* decision variable refers to a decision variable that can take on either a fractional value or an integer value [19].

The objective function and all constraints in a LP model are linear, and the decision variables are continuous [3]. LP models can be solved using the *simplex method*, which finds the

optimal *feasible solution* in a cascade of pivot moves and matrix operations [19]. A feasible solution satisfies all constraints on the variable values [3]. An *optimal solution* for a LP model is the feasible solution that provides the best value to the problem's objective function [19].

A decision-making environment for LP models can be either *deterministic* or *probabilistic*. In a deterministic decision-making environment, the uncertainty about the decision outcome is so trivial that it can be ignored [19]. On the contrary, a probabilistic environment is where the uncertainty about the decision outcome is significant enough that it requires special consideration [19].

2.5.2 Integer Linear Programming

When one or more of the decision variables are required to be integers, the mathematical programming problem is referred to as an *integer linear program* (ILP) [3]. A *pure integer programming* problem is an ILP where all decision variables are integers. A *mixed integer programming* (MIP) problem, however, refers to an ILP where some decision variables are integers and some are continuous. *Integer variables* are the variables that must have integer values, whereas, *binary variables* are a type of variables whose value must be either 1 or 0.

An ILP or LP can be reduced to a standard algebraic form as shown in Figure 2.16. Unless otherwise specified, all LPs discussed in this thesis are ILPs, and they are pure ILPs in most cases.

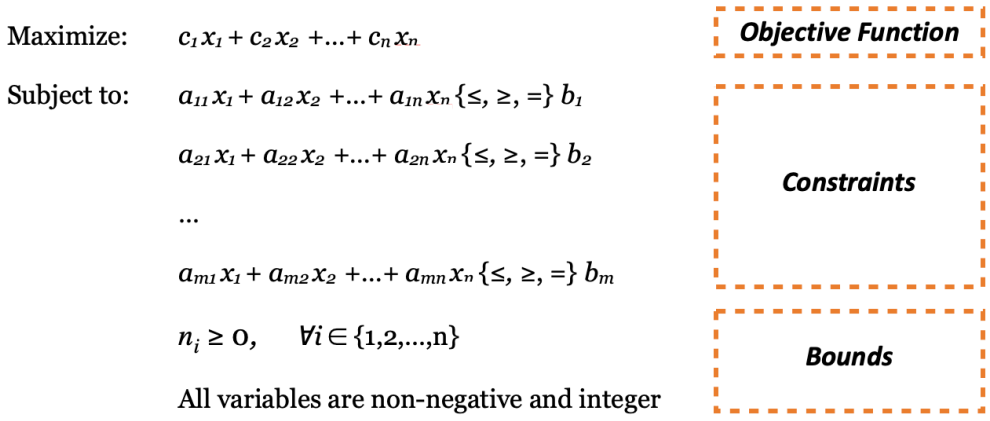


Figure 2.16 Example of a standard algebraic form for an ILP.

As mentioned briefly in the CHAPTER 1 of this thesis, an ILP model is typically solved by the branch-and-bound method. In the branch-and-bound method [18], the "branching" process creates new sub-problems with new bounds. The "bounds" are used to snip off any branch LPs that do not contain an overall optimal solution. Branching continues down on each sub-problem until all the branches are explored, and the best integer solution is found.

2.5.3 ILP Formulation in Survivable Network Design

ILPs are formulated to solve optimization problems in various survivable transport network designs (i.e., span restoration [11], path restoration [12], SBPP [13], *p*-cycles [14]). Given a network topology, a set of working traffic demands, and any constraints that must be satisfied, this type of ILP formulations finds the minimum cost associated with working and spare capacity distribution. In these formulations, the cost of placing the required capacity units in the network is minimized. Constraints that must be satisfied include: ensuring full working lightpath routing and achieving 100% restorability to single-span failures [3].

To optimize this type of network design problem, we can adopt either a single-step or two-step approach. The single-step approach to optimizing a survivable network design problem is generally referred to as a *joint capacity allocation* (JCA) problem. The JCA optimizes the working units routing in conjunction with the allocation of spare capacities, so that the total capacity cost (working capacity plus spare capacity) will be minimized [17], [26]. The *spare capacity allocation* (SCA) problem, on the other hand, is a two-step approach where the working paths are usually routed in advance via preferred routing methods, such as the shortest path method [26] or the DFS algorithm [32]-[39], [42]-[43]. Given the working capacities, the allocation of spare capacities with minimum cost will be determined to ensure 100% protection for all working paths.

To demonstrate how an ILP may be formulated to optimize a survivable network design, we will provide an example of a SCA problem in a span-restorable network. The SCA problem in a span-restorable network can be formulated using an *arc-path* approach [11], [44]. An arc-path approach assumes that all distinct eligible restoration routes are already enumerated and working demands are routed. Based on specific spare capacity values on each span , the model

optimizes assignment of restoration flows over eligible restoration routes that crosses each span in a network topology so that the total cost of spare capacity allocation is minimized. The ILP formulation of arc-path approach of SCA in span-restorable network is as shown below.

Objective function: $$\text{Minimize} \quad \sum_{\forall i \in S} c_i \bullet s_i \qquad \text{(Eq. 2.1)}$$

Subject to: $$s_j \geq \sum_{\forall p \in \text{Pi}} x_{i,j,p} \bullet f_{i,p} \quad \forall (i,j) \in S^2 \quad i \neq j \qquad \text{(Eq. 2.2)}$$

$$w_i = \sum_{\forall p \in \text{Pi}} f_{i,p} \qquad \text{(Eq. 2.3)}$$

All variables are non-negative and integer

In the model presented above, S is a set of all network spans and P_i *is a s*et of distinct eligible routes available to carry restoration flow for failure on span i ($\forall_i \in S$). i is a failed span in a network scenario, whereas j refers to a span on an eligible restoration route p. In the objective function (Eq. 2.1), c_i and s_j are the cost of each unit capacity and the number of spare capacity on span i, respectively. The objective function minimizes the cost of placing spare capacities to restore a failed span i. In the first constraint (Eq. 2.2), $x_{i,j,p}$ is a binary parameter that equals one if span j on the restoration route p traverses the failed span i, zero otherwise. $f_{i,p}$ is the number of restoration flow assigned to the eligible restoration route p for failure of span i. The s_j on the left hand side indicates the number of total spare capacities on span j. This constraint ensures that the spare capacities on span j can support all the eligible restoration routes that cross the span i. The second constraint (Eq. 2.3) ensures that the total protection provided by the restoration route p for each failed span i is enough to fully restore the working capacity on the failed span. The total number of working capacity to be protected on span i is denoted by w_i. Number of restoration flow on restoration route p for failure of span i *is denoted by* $f_{i,p}$. The arc-path ILP formulation provides a detailed specification of restoration routes and flows.

2.6 Mathematical Tools - Heuristics & Meta-Heuristics

2.6.1 Heuristics vs. Meta-Heuristics

Apart from the ILP method, which provides the optimal solution to a problem, *heuristic* methods were also adopted in some studies to optimize survivable network design problems. Heuristic methods are problem-specific, purpose-specific methods to be used for all or part of a problem [20]. Compared to LP or ILP methods, heuristic methods generally provide sub-optimal solutions; however, they can solve a complex problem within a relatively shorter run time. In reality, large-scale problems may take exceedingly long runtime for an ILP solver to reach an optimal solution, and a sub-optimal solution with significantly shorter runtime may be preferred. Depending on the details of a heuristic algorithm and the problem it is applied, the result of a heuristic algorithm may be very close to an optimal solution. However, in reality, one may never know how close it actually is to the optimum [26]. Heuristic algorithms may be proposed and implemented in various aspects of a survivable network design problem, such as cycle enumeration [32], [39], [45], candidate cycle selection and refinement [46]-[49], optimal capacity design and optimization[39], [50]-[52], wavelength assignment [53], etc.

Like a heuristic method, a *meta-heuristic* method also does not guarantee to find an optimal solution to a problem. However, unlike a heuristic algorithm, a meta-heuristic algorithm is not a problem-specific algorithm but a general problem-formulating framework that can be implemented in any optimization problem [20].

2.6.2 Meta-Heuristics Basics & Overview

A *meta-heuristic* method is a highly general framework for solving optimization problems. The prefix “meta-” is derived from a Greek word which means high-level methodologies. Therefore, a meta-heuristic approach is not specific to a single problem but can be customized and applied to many optimization problems. Typically, meta-heuristics are applied to complex, computationally-intensive problems that cannot be easily solved by other techniques like ILPs or LPs. Meta-heuristics do not guarantee optimal results; however, they can

find near-optimal results for complex optimization problems more effectively and efficiently as compared to ILP/LPs [54]. As compared to ILP/LPs, meta-heuristic methods are exploration-focused and are suitable for handling computational-intensive problems. They also have the advantage of handling non-linear constraints or objectives [3].

In general, a meta-heuristic algorithm optimizes the objective function value by randomly explores a search space to find the most appropriate solution for the decision variable. Figure 2.17 depicts the schematic of a generic meta-heuristic algorithm process, which involves the following key elements and factors [54]:

Initial State and Input Data. All meta-heuristic algorithms start from an initial state, which can be predefined, calculated or randomly generated. The input data include the input information data that is critical for solving the optimization problem and the input parameters that are required to execute the algorithm.

Objective Function. An objective function delineates the goal of an optimization problem. The objective function values are either minimized or maximized over a set of various feasible solutions to the optimization problem.

Decision Variables. Decision variables are calculated when an algorithm is executed. The values of decision variables reflect the solutions to an objective function.

Fitness Functions. A fitness function is generated to evaluate the suitability or desirability of a possible solution. It advances a search process by promoting more desirable solutions or inhibiting less-desirable solutions.

Iterations. Meta-heuristic algorithms are executed iteratively when exploring a solution in the search space. At each iteration, a new solution to the objective function is generated, which will be used as the initial value for the next iteration. The number of iterations can serve as a termination condition.

Constraints. In optimization problems, constraints determine the boundaries of the search space for feasible solutions. All constraints must be satisfied for a solution to be feasible.

Final State and Termination Criteria. Termination criteria delineate the conditions required to terminate a meta-heuristic search process. Once the termination criteria are satisfied, the search process will stop and return the best available solution, showing the final state of the problem.

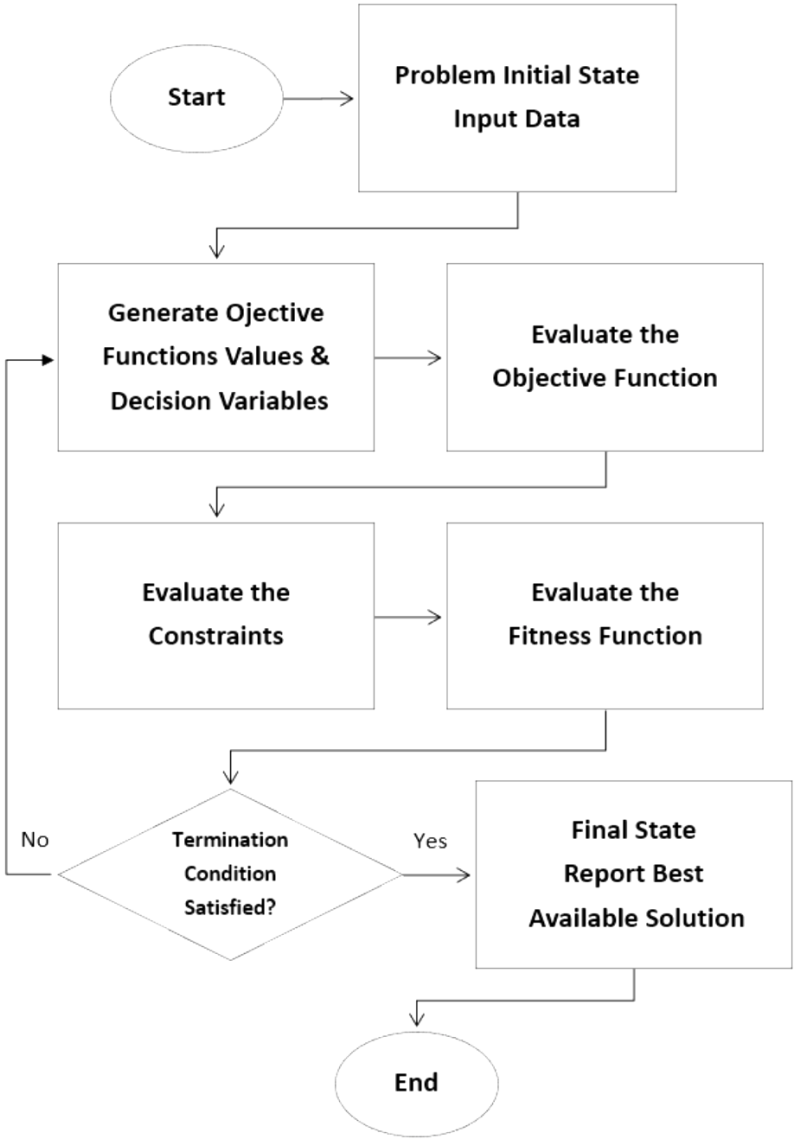


Figure 2.17 Overview of a meta-heuristic algorithmic process from start to end.

Some optimization problems are highly complex and are considered to be *NP-hard* [3]. *NP* stands for non-deterministic polynomial problems. These problems can be solved and then validated by a non-deterministic algorithm within polynomial time. *NP-hard* problems are the ones that are at least as hard as the hardest *NP* problems. In the telecommunication industry,

various optimization problems are considered to be *NP-hard*, hence cannot be solved and validated within polynomial time. For example, the network design problem, the routing problems, and the frequency assignment problems are all categorized as *NP-hard* problems. Among these problems, the network design problem is one of the most robust and crucial combinatorial optimization problems (COPs) [55]. Meta-heuristics and heuristics have been the preferred approaches for solving *NP-hard* combinatorial optimization problems [55].

2.6.3 Meta-heuristics in Survivable Network Optimization

Various meta-heuristic optimization methods have been proposed in the literature to solve network optimization problems, such as *genetic algorithms* (GAs) [56]-[57], *simulated annealing* (SA) [51], *ant colony optimization* (ACO) [58]-[59], *Tabu Search* (TS) [54], [60], etc. Genetic algorithms are developed based on the Darwinian theory on natural selection, where individuals with the most advantageous variations are selected and reproduced in subsequent generations. Simulated annealing (SA) mimics the slow metal cooling process ("annealing") in metallurgy. The algorithm randomly explores a solution at each iteration while the temperature decreases progressively. It is a stochastic algorithm for exploring the global optimal solution. Ant colony optimization (ACO) is a meta-heuristic method that imitates the social behaviour of ant colonies, where the quality of a possible solution is evaluated based on the associated amount of pheromone during the search process. Tabu Search (TS) is inspired by conventional "hill-climbing" that only accepts "uphill" moves after a local optimum is discovered. A backward step is considered "tabu" for a defined period of time.

These meta-heuristic methods can be categorized in several ways [54]-[55], which include *single-solution meta-heuristics*, *population-based meta-heuristics*, *hybrid meta-heuristics, nature-inspired vs. Non-nature-inspired*.

Single-solution meta-heuristics involve one solution in the search process. TS and SA are good examples of single-solution meta-heuristics. TS is deterministic, and SA is stochastic.

Population-based meta-heuristics are in search for a population of search points at each iteration. GA and ACO are examples of population-based meta-heuristics.

Hybrid meta-heuristics is an approach taken to combine meta-heuristics with other methods to achieve a synergistic performance. For example, a *hybrid heuristic-GA* (HGA) is proposed in [53] for survivability dynamic routed and wavelength assignment problems on IP/ WDM transport networks. At first, a heuristic algorithm was executed to generate an optimized searching space for GA. Then, a GA was applied to find the best near-optimal solution within the search space. In a survivable network design problem with relays [52], the author solved the problem in two steps using a hybrid GA-Lagrangian heuristic approach. GA was used in the first step to search for paths, and the Lagrangian heuristic was used for relay assignment in the second step.

Some meta-heuristic algorithms are designed based on behaviours of certain organisms (such as ACO) or biological theory (such as GA). As mentioned in Fernandez *et al.* [53], nature-inspired meta-heuristics are suitable for solving optimization problems regarding planning, designing and controlling. These approaches search for the global optimum by manipulating a population of possible solutions in a competitive manner using well-designed operators.

In Fernandez *et al.* [55], the authors conducted an extensive review of meta-heuristics in telecommunication applications. This paper indicated the increasing popularity of meta-heuristics in the past two decades in solving various large-scale telecommunication optimization problems, especially for addressing the population-based meta-heuristics (e.g., GA and ACO).

In our research in this thesis, we will be implementing the GA technique to optimize *p*-cycle spare capacity allocation problem. More details regarding GA and its implementation in the problem will be further elaborated in CHAPTER 4 of this thesis.

CHAPTER 3. *p*-Cycles Basics and Studies

3.1 Introduction to *p*-Cycles

Before the *p*-cycle protection mechanism was proposed, there had been two basic approaches to survivable network designs: survivable rings mechanism and mesh network survivability mechanism. As introduced in CHAPTER 2 of this thesis, the mesh-based survivability mechanism has a more favourable capacity efficiency than a ring-based protection mechanism. However, a real-time mesh-based restoration scheme requires longer restoration time than a ring-based survivability mechanism, which is undesirable. Therefore, a cycle-orientated pre-configuration of spare capacity was introduced in the mesh network restoration scheme, and a ring-mesh hybrid protection scheme was proposed to allow the faster protection speed of a ring as well as the higher protection efficiency of a mesh [14]-[16]. This novel design was named a *p*-cycle, meaning a pre-configured protection cycle.

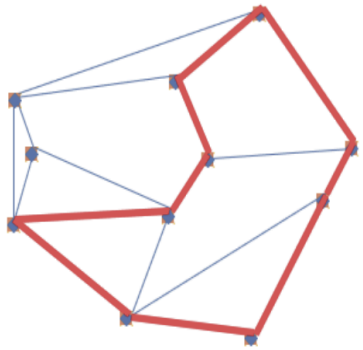


Figure 3.1 Illustration of a *p*-cycle (bolded lines).

p-Cycle protection is based on the optimal formation of *p*-cycles in the spare capacity of a mesh-restorable network and is a type of shared link protection, first proposed in late 1990s [14]-[16]. The *p*-cycles are formed prior to any network failures and are formed out of the previously un-connected spare links of a mesh-restorable network [14]. Two immediate end nodes on a failed span perform real-time optical signal switching for restoration to happen. An

exceptional feature of a *p*-cycle is that it not only provides one unit of spare capacity to all the on-cycle spans but also offers two units of spare capacity for the spans that straddle over the cycle. This type of spans, which has its two end nodes on the cycle but not the span itself, is referred to as straddling spans to the *p*-cycle. Straddling spans carry two working paths for each protection path on the *p*-cycle that they straddle, but they do not retain any spare capacity themselves [3]. Figure 3.1 shows an example of a *p*-cycle, where the spans traversed by the bolded *p*-cycle are on-cycle spans.

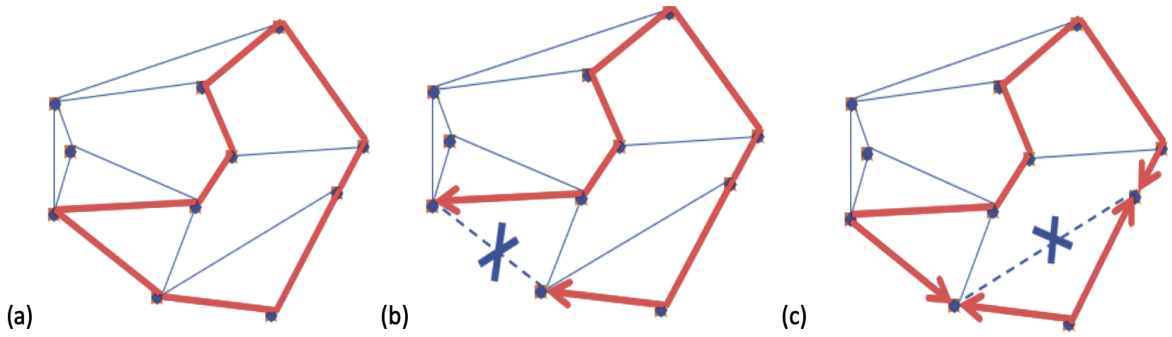


Figure 3.2 Illustration of a *p*-cycle in a network (a). (b) shows a failure on an on-cycle span, and (c) shows a failure on a straddle span.

Figure 3.2 was presented previously in section 2.3.3. The figure demonstrates examples of different failure scenarios occur on a *p*-cycle. As illustrated in Figure 3.2(b), when an on-cycle span fails, the surviving portion forms one restoration path between both nodes adjacent to the failure. This protection operation is similar to that of a bidirectional line-switched ring. In Figure 3.2(c), if a failure occurs not on any part of the *p*-cycle but on a span that straddles the *p*-cycle, two restoration paths around the failed spans are available for restoring the failure. Therefore, the *p*-cycle in Figure 3.2(a) provides one protection path for 9 of the on-cycle spans, and it provides two protection paths for each of the three straddling spans (six units of protection). Hence, the *p*-cycle provides a total of 15 units of protection from 9 units of spare capacity, a redundancy of 9/15 = 60%.

3.2 Types of *p*-Cycles

There have been various types of *p*-cycles discussed in the literature[26]-[28], [56]. They can be categorized either based on their structural relationship with the network they are in or based on specific protection functionality that they provide to the network.

p-Cycles that are named based on their structural relationship with the network are *Hamiltonian p-cycles*, *simple p-cycles*, and *non-simple p-cycles* [26]. Hamiltonian *p*-cycles are the ones that pass through all the nodes of the network once and only once (see Figure 3.3(a)). Simple *p*-cycles refer to *p*-cycles that do not pass through any nodes or spans more than once (see Figure 3.3(a)). A non-simple *p*-cycle, on the other hand, is a *p*-cycle that passes through a node or a span more than once (see Figure 3.3(b)). The type of *p*-cycles that are going to be studied in this thesis herein are the simple *p*-cycles.

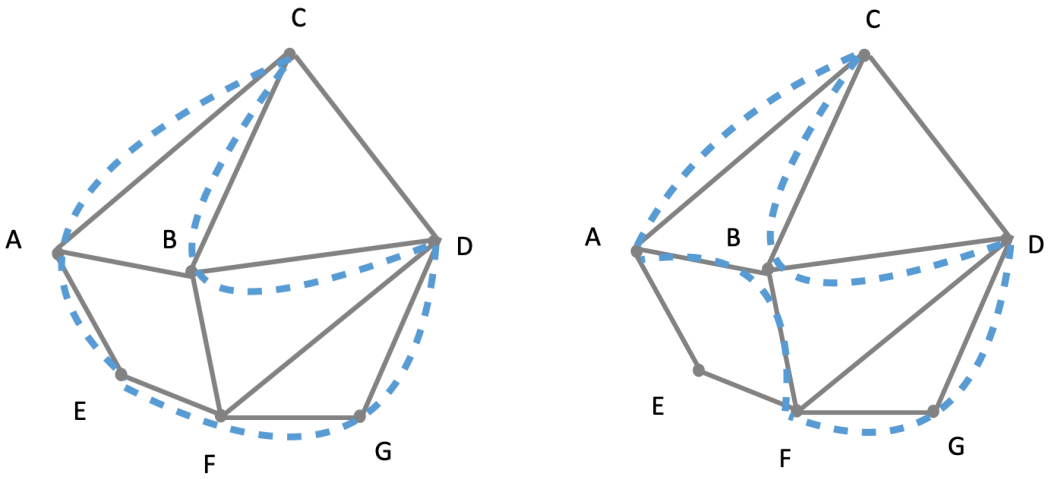


Figure 3.3 Examples of a Hamiltonian and simple *p*-cycle (left), and a non-simple *p*-cycle (right).

p-Cycles that have different protection functionalities (e.g., protecting spans, paths, nodes, or path segments against failures) are named differently. Based on their specific functionalities, these *p*-cycles are referred to in the literature as *span p-cycles*, *path protecting p-cycles*, *node encircling p-cycles*, and *flow p-cycles* [26].

A *span p-cycle* protects a particular span of a network. It refers to a *p*-cycle that allows protection to a span that is either on the cycle or straddles the cycle [26]. The *p*-cycle illustrated in Figure 3.3(a) is also a span *p*-cycle.

A *path-protecting p-cycle* protects an entire path on which its origin node and destination node incident. A *failure independent path protection* (FIPP) *p*-cycle is an example of a path-protecting *p*-cycle, which offer protection to a set of end-to-end disjoint paths whose end nodes are located on the cycle [62]. FIPPs have similar capacity efficiency than that of SBPP, and they can protect failures on both node and span[27], [62]. An important pre-requisite for FIPP *p*-cycles is that all paths to be protected must be all mutually disjoint to share spare channels, which is similar to that of SBPP [62]. In a FIPP *p*-cycle protection, a real-time cross-connection is not needed to form protection paths, and protection switching is controlled by end nodes and is entirely failure-independent [62]. As shown in Figure 3.4, the FIPP *p*-cycle in blue dotted lines protects a set of disjoint paths (green double-arrowhead lines) with their end nodes on the FIPP *p*-cycle.

Flow p-cycles are a type of path segment-protecting *p*-cycles [63]. The section of a traffic flow that lies between two intersecting nodes of the flow and the associated *p*-cycle is referred to as a *flow* or *path segment*. The *p*-cycle that protects these flows is known as a flow *p*-cycle. Flow *p*-cycles can provide both path protection (if the origin and destination nodes incident on the *p*-cycle) and node protection and are shown to be capacity efficient [27]. As demonstrated in Figure 3.4, the flow *p*-cycle (in blue dotted lines) can provide path and node protections to the flows or path segments (red double-arrowhead lines). Flow *p*-cycles and their designs for path- and node-protection are extensively studied in [63]-[66].

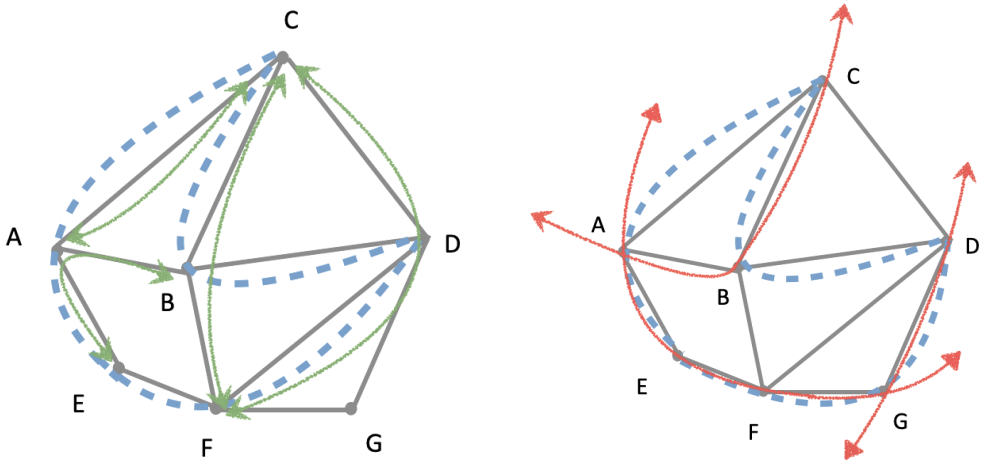


Figure 3.4 Examples of a failure-independent path protection (FIPP) *p*-cycle (left) in dotted lines, and a flow *p*-cycle (right) in dotted lines.

p-Cycles can also provide node protection by adopting the *node-encircling p-cycle* (NEPC) scheme [67]. An NEPC traverses all the neighbouring nodes of a failed node but not the failed node itself. It can protect all the traffic flows that pass through the failed node, hence protecting the failed node. The NEPC protection is illustrated in Figure 3.5; its design and mechanism are discussed extensively in [67]. As shown in Figure 3.5, the nodes that are highlighted in orange circles (Node B and Node E, respectively) are not part of the *p*-cycles, however, they are protected by the NEPC protection scheme. An NEPC can be either a simple cycle (Figure 3.5 on the left) or a non-simple cycle (Figure 3.5 on the right) [27].

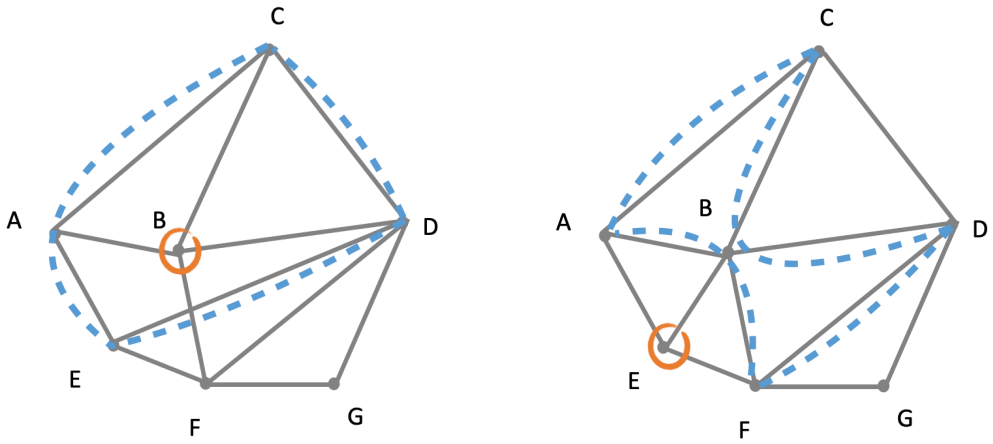


Figure 3.5 Examples of a simple node-encircling *p*-cycle (left) and a non-simple node-encircling *p*-cycle (right) in dotted lines. Circled nodes (orange) are the nodes protected by NEPC protection.

3.3 Determining *p*-Cycle Efficiency

An efficient *p*-cycle is capable of providing more units of protection per spare capacity used. Finding and selecting efficient *p*-cycles for network protection is getting more beneficial as network size increases. The most fundamental efficiency metrics which have been deployed in the literature include the *topological score* (TS), and *A Priori efficiency* score (AE). These metrics laid the foundation for the development of several other advanced efficiency metrics, such as the *actual efficiency* (*Ew*), the *demand weighted efficiency* (DWE), the *efficiency ratio* (ER), and the *efficiency of restoration* (EoR).

Topological Score (TS)

The *topological score* (TS) of a candidate *p*-cycle is a basic metric that measures the amount of protection a *p*-cycle provides to the network [46]. It is defined as $TS(j) = \sum X_{i,j}$ for all spans *i* on *p*-cycle *j*. $X_{i,j}$ takes a value of 1,2 or 0 depending upon whether the link is on the cycle, straddles the cycle, or is not a part of the cycle. However, TS only takes into account the topological structure of a cycle and does not consider the cost of using that cycle.

A Priori Efficiency (AE)

A priori efficiency (AE) score measures the potential efficiency of using a *p*-cycle [26], [46]. It calculates the total number of protections a *p*-cycle provides relative to the cost of constructing the *p*-cycle. Therefore, a *p*-cycle with a higher AE score has higher numbers of on-cycle and straddling spans per unit of the cost; hence, higher protection capacity efficiency. However, neither AE nor TS considers the actual set of demands of a network. Therefore, AE and TS both calculate only the potential efficiency of a *p*-cycle, not the weighted actual efficiency of a *p*-cycle. For a candidate *p*-cycle *j* in a *p*-cycle set, AE(*j*) is defined as in Eq. 3.1.

$$AE(j) = \frac{\sum Xi, j}{\sum \partial k, j \bullet \sum Ck}, \qquad \forall i \in S, \ \forall k \in S \qquad \text{(Eq. 3.1)}$$

In Eq. 3.1, $X_{i,j}$ is the same as in TS which equals to 1,2 or 0; $\partial_{k,j}$ equals to 1 if span *k* traverses cycle *j,* and zero otherwise. C_k refers to the cost of using span *k*. *S* is set of all spans.

Demand Weighted Efficiency (DWE)

The third metric, *demand weighted efficiency* (DWE) [26], takes into account the actual demands in a network. For a given network, the DWE assigns actual units of demands on each link and calculates the protection provided by the cycle in the presence of those demands. For example, a cycle may protect *n* straddling links in a network. However, if there is no actual

traffic flowing through those *n* links, the actual protection provided by the cycle to the network will not factor in those *n* links.

For a candidate *p*-cycle *j* in a *p*-cycle set, DWE is defined as below in Eq. 3.2.

$$DWE(j) = \frac{\sum min(Xi,j \bullet wi)}{\sum \partial k,j \bullet Ck}, \qquad \forall i \in S, \ \forall k \in S \qquad \text{(Eq. 3.2)}$$

$X_{i,j}$ is the same as in Eq. 3.1, and w_i is the working capacity of span *i*. $\partial_{k,j}$ equals to 1 if span *k* traverses cycle *j*, and zero otherwise. C_k is the cost of using span *k*. *S* is set of all spans.

Actual Efficiency (E_w) in Capacitated Iterative Design Algorithm (CIDA)

The actual efficiency metric which was proposed by Doucette *et al.* [39] considers current working capacity status (number of unprotected working capacity) on the span *i* and the protection relationship of span *i* relative to the cost of using the *p*-cycle. The E_w score provides not only an indication of a *p*-cycle's ability to protect current working capacity but also a suggestion of a *p*-cycle's actual suitability in a specific working capacity state. The *capacitated iterative design algorithm* (CIDA) calculates and ranks current working capacity state at each iteration and conducts iterative placement of *p*-cycles based on the highest-ranked actual efficiency (see details in CHAPTER 3 regarding CIDA).

The E_w of a candidate *p*-cycle *p* is defined as shown in Eq. 3.3.

$$E_w(p) = \frac{\sum Xi,j \bullet wi}{\sum \partial k,j \bullet Ck}, \qquad \forall i \in S, \ \forall k \in S \qquad \text{(Eq. 3.3)}$$

$X_{i,j}$ equals to 1,2 or 0 as in previous metrics, and w_i is the unprotected working capacity on span *i*. The binary $\partial_{k,j}$ equals to 1 if span *k* traverses cycle *j*, and zero otherwise. C_k is the cost of using span *k*. *S* is set of all spans.

Efficiency Ratio (ER)

Efficiency Ratio is an efficiency metric for a unity *p*-cycle that was proposed by Zhang *et al.* [40]. A unity *p*-cycle refers to a *p*-cycle that has one unit of capacity on each of its spans. An ER is the ratio of the number of working capacity units that are protected by a *p*-cycle to the number of spare capacity units of the *p*-cycle. ER sounds similar to the E_w score, but instead of looking at *unprotected* working capacity, ER brings *protected* working capacity into the ratio calculation. A higher ER indicates a higher spare capacity utilization efficiency of a *p*-cycle. Unlike TS and AE scores, where only topology of a *p*-cycle is considered, an ER value indicates both a *p*-cycle topology and the working capacities protected by the said *p*-cycle.

Efficiency of Restoration (EoR):

This efficiency metric was proposed by Meixner *et al.* [68] to be used in conjunction with a heuristic method for a scalable *p*-cycle selection method. The metric can be used in both ILP and Genetic Algorithm models. Calculation of EoR involves the following parameters: average *p*-cycle length (β), the standard deviation of *p*-cycle length (η), the total cost of a *p*-cycle (TC), and total protection a *p*-cycle provides (TS). A set S refers to all the spans in the network. This metric is based on the TS, but with consideration of *p*-cycle lengths.

$$EoR(S) = \frac{TS(S)}{TC(S) \bullet \beta(S) \bullet (\eta(S) + 1)} \qquad \text{(Eq. 3.4)}$$

3.4 *p*-Cycle Network Design and Optimization

3.4.1 Candidate *p*-Cycles Enumeration

The *depth-first* search (DFS)algorithm is commonly used to generate sets of candidate cycles [12]-[14]; however, the DFS algorithm is a greedy approach and is not runtime-efficient in large-scale network problems. Therefore, some studies have proposed various problem-specific heuristics to find eligible candidate *p*-cycles that are *high-merit* [32]-[33], [45]. The following algorithms are commonly used in the literature, which has also inspired the novel heuristic cycle-enumeration method that will be proposed and discussed in CHAPTER 6 of this thesis.

Straddling Link Algorithm (SLA)

In Zhang and Yang's research [32], a heuristic method called *straddling link algorithm* (SLA) was proposed to find protection cycles in a network. For each span *i* of a network, the SLA algorithm finds two other node-disjoint paths (if any) that originate from one end node (O_i) of span *i* and terminate at the other end node (D_i). An eligible *p*-cycle is formed by merging the second and third node-disjoint paths at node O_i and node D_i. The shortest path calculation of the node-disjoint paths is based on Dijkstra's algorithm. The multiple node-disjoint paths between two nodes are found by executing Dijkstra's algorithm iteratively, with the previously found paths (and all nodes along the paths, except O_i and D_i) being labelled and hidden before the next one is found.

As shown in Figure 3.6 below, two node-disjoint paths between span O_i-D_i are illustrated and highlighted. To be specific, for the span O_i-D_i (the dotted line as shown in the Figure 3.6), the SLA algorithm finds two node-disjoint paths: *A*-*B*-*C*-D_i-O_i (orange arrow) and O_i-*E*-D_i (green arrow). Merging these two paths and removing span O_i-D_i will yield a new *p*-cycle *A*-*B*-*C*-D_i-*E*-O_i. In this case, span O_i-D_i becomes a straddle link of the new *p*-cycle.

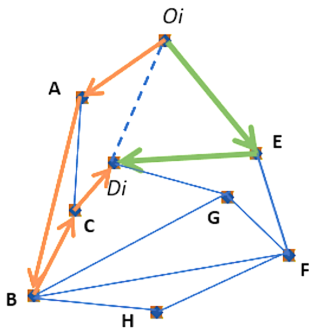


Figure 3.6 Illustration of generating a new *p*-cycle (*A-B-C-D*$_i$*-E-O*$_i$) using SLA.

Add and Join Algorithms

Based upon the SLA, two cycle-merging operations were proposed in [39] to create *p*-cycles with more straddling spans (hence, higher efficiency). The *Add* and *Join* algorithms are operations that can be applied to the primary cycles generated by SLA. In the *Add* algorithm, two primary cycles can be merged if they meet the following criteria:

1) *Span i* is a straddling span of *Cycle A,* but an on-cycle span to *Cycle B*;
2) *Span j* is a straddling span of *Cycle B,* but an on-cycle span to *Cycle A*;
3) *Cycle A* and *Cycle B* are not adjacent to one another in the network

When the *Add* algorithm is applied, *Cycle A* and *Cycle B* are merged into one. The resulting new *p*-cycle will include all the on-cycle spans of both *Cycle A* and *Cycle B* (except *Span i* and *Span j*), and *Span i* and *Span j* will be straddling spans to the new cycle. As shown in Figure 3.7, *Cycle A* is cycle A-B-C-K-P, and *Cycle B* is cycle A-C-K-P. *Span i* refers to span A-C, and *Span j* is span P-K. The resulting cycle after applying the *Add* algorithm is A-B-C-K-E-P.

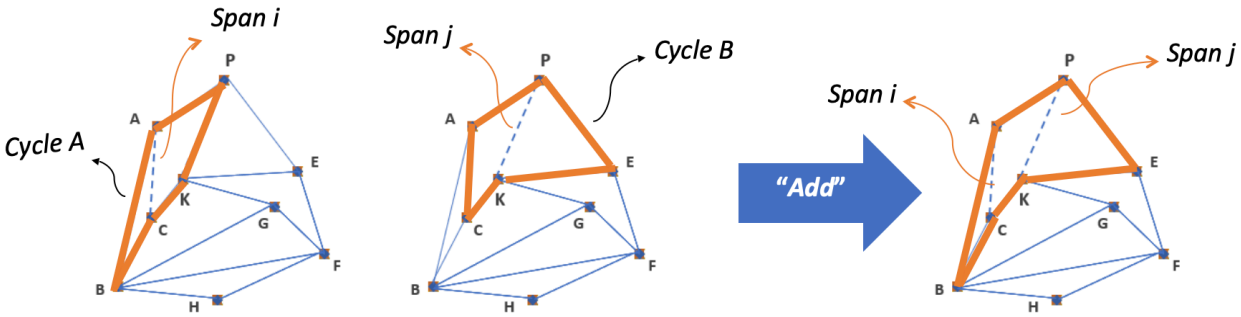


Figure 3.7 Illustration of generating a new *p*-cycle using the *Add* algorithm.

In the *Join* algorithm, two primary *p*-cycles have one and only one common span, and that span has to be an on-cycle span (not a straddling span of any of these two *p*-cycles). As shown in Figure 3.8, *Cycle A* (cycle A-B-C-K-P) and *Cycle C* (cycle P-K-G-F-E) share only one common span, the span P-K (*Span k*). The resulting cycle (cycle A-B-C-K-G-F-E-P) contains the straddling spans of both *Cycle A* and *Cycle C*, as well as their common span (*Span k*).

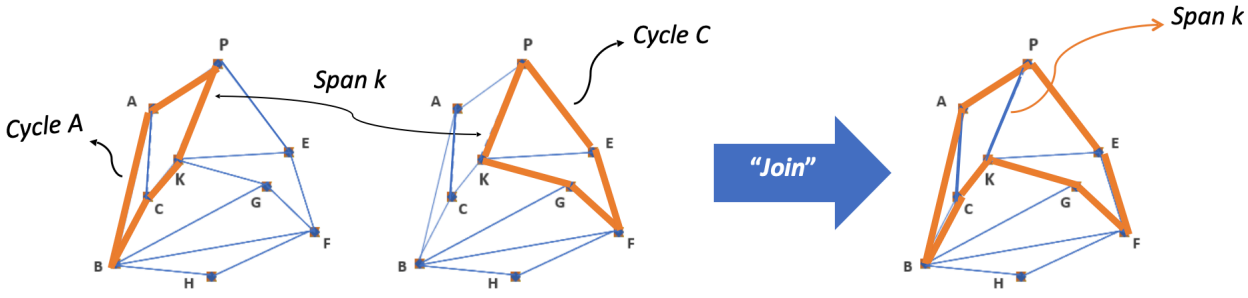


Figure 3.8 Illustration of generating a new *p*-cycle using the *Join* algorithm.

Expand and Grow Algorithms

Doucette *et al.* [39] proposed two advanced algorithms that transform primary *p*-cycles with one straddling span to larger *p*-cycles with more straddling span, hence higher efficiency. Unlike the *Add* or *Join* algorithm, either *Expand* or *Grow* algorithm takes just one primary *p*-cycle as an input cycle. In the *Expand* algorithm, an on-cycle span is *expanded* and replaced with a node-disjoint path that connects the same origin and destination nodes as the said on-cycle span. The newly formed *p*-cycle will then be added to the cycle set. The operation repeats recursively until all on-cycle spans of the original primary *p*-cycle are visited and replaced by a node-disjoint path (if existed). As shown in Figure 3.9, the *Expand* algorithm can generate multiple new *p*-cycles from the original primary *p*-cycle. In Figure 3.9 (a), span B-F (left) is replaced by path B-H-F (right), and generate a new *p*-cycle B-H-F-G-K-C. In Figure 3.9 (b), span B-C (left) is replaced by path B-A-C (right), resulting in a new *p*-cycle B-H-F-G-K-C-A. In Figure 3.9 (c), span C-K (left) is replaced by path C-A-P-K (right), which forms a new *p*-cycle B-F-G-K-P-A-C.

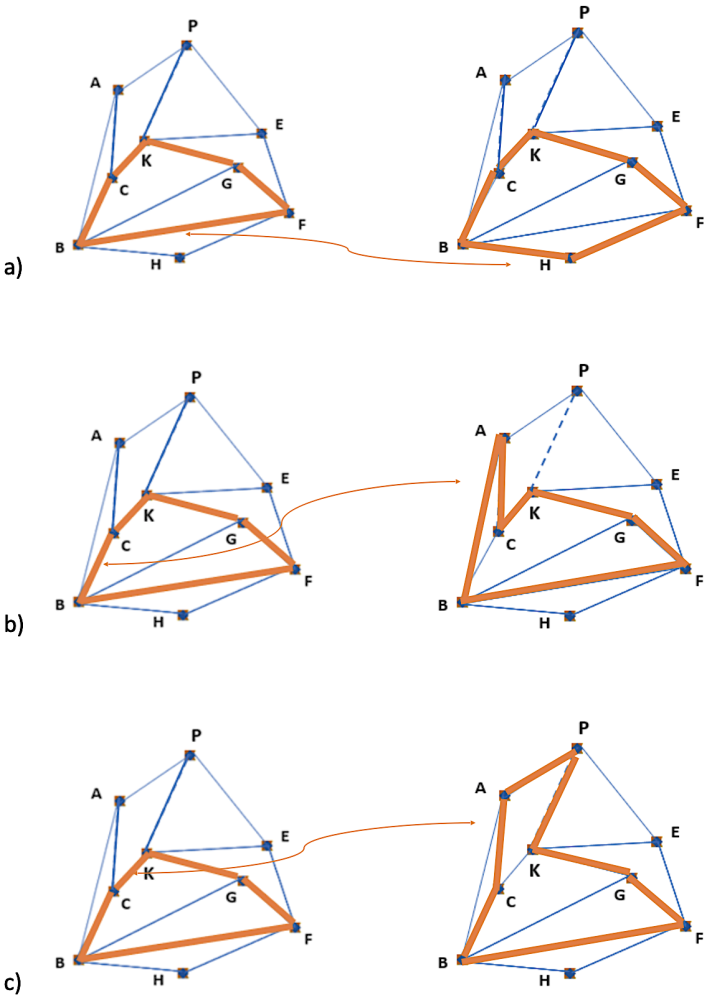


Figure 3.9 Illustration of generating new *p*-cycles using the *Expand* algorithm. The double-arrowhead lines Indicate transitions of the spans from left side of a figure to the right side of a figure.

The *Grow* algorithm is a more extensive approach that generates *p*-cycles of various sizes. The *Add* algorithm is first applied to the primary cycles to generate a set of intermediate *p*-cycles. The *Grow* operation then takes place on the new intermediate cycle set. For each cycle in the intermediate cycle set, pick one on-cycle span (*span s*) and replace it with a node-disjoint path (*path p*) that connects end nodes of the said on-cycle span. This node-disjoint path is generated using the shortest path method. If such a path exists, *span s* is removed from the cycle and *path p* is added to the cycle, resulting in a new *p*-cycle (say, *Cycle M*). The algorithm will then start on the first span of *Cycle M and* repeat the same path-finding process until no more node-disjoint paths can be found on any of the spans on any of the newly-formed cycles. During

this process, each *p*-cycle that is incrementally generated will be retained and added to the total *p*-cycle set. Therefore, the resulting total *p*-cycle set will contain an extensive profile of various small and large *p*-cycles, including primary *p*-cycles, intermediate *p*-cycles, and new *p*-cycles that are incrementally *grown* out of previous *p*-cycles. Having a wide range of *p*-cycles is very beneficial for the later stage of cycle placement.

The pseudo-code for *Grow* algorithm is as follows:

```
function Grow(OriginalCycleSetA):
    initialize SPAddCycleSetB
    AddCycleSetB = Add(OriginalCycleSetA)
    initialize NewCycleSet
    for each cycle p in SPAddCycleSetB:
        let cycle p' = cycle p
        for each span i on cycle p':
            mark all spans and nodes on cycle p'
            Dijkstra(i, unmarked spans/nodes) → r
            if returns a route r:
                add route r to cycle p'
                remove span i from cycle p'
                add cycle p' to NewCycleSet
                restart count of i to first span in p'
            unmark all spans and nodes
    add OriginalCycleSetA to NewCycleSet
    add SPAddCycleSetB to NewCycleSet
    for each span i:
        mark span i
        Dijkstra(i, unmarked spans/nodes) → r
        if returns a route r:
            let cycle x = span i and spans on route r
            add cycle x to NewCycleSet
        unmark span i
    return NewCycleSet
```

Weighted DFS-Based Cycle Search Algorithm (WDCS)

The *weighted DFS-based cycle search* (WDCS) algorithm is a DFS-inspired cycle finding algorithm that was proposed in Liu and Ruan [33], where the WDCS algorithm finds a controlled number of cycles in the graph. The algorithm explores the graph from a root node v, extending the path along the way until the root node is reached again and a new cycle is formed. While exploring, the algorithm finds a neighbouring span with the highest weight so that cycles with higher efficiency are likely to be found earlier in the process.

3.4.2 Candidate *p*-Cycles Selection

Prior studies on p-cycle designs and selection suggested two approaches to selecting candidate p-cycles, where candidate p-cycles are selected by limiting either the maximum hop-limits (or route length-limit) or the circumference-limits of all candidate p-cycles to reduce the complexity of a problem [14]-[16], [47]. When considering hop-limits as a constraint, p-cycles of any length may be used; however, the lengths of actual paths are restricted to a defined limit. In a circumference-limit approach, however, the length of a p-cycle is constrained. A comparison of hop-limit versus circumference-limit in terms of their effect on p-cycle capacity efficiency was studied by Kodian *et al.* [47]. This research indicated that a hop-limiting factor (limit of 3 or 4 hops) brings about a positive effect on providing better capacity efficiency in their test cases of a 13-node network and a 19-node network. Another approach to selecting good eligible p-cycle candidates is to exhaustively find all the p-cycles in a network, rank them based on a particular selection metric, and then select the top few candidates [46], [48]. For example, in the study by Doucette *et al.* [46], all the candidate cycles were first ranked using TS or AE scores, from which a limited number of top candidate cycles were chosen.

Both of these two approaches are greedy and runtime-consuming, as both will require exhaustively enumerating all cycles. This will be counter-efficient when it comes to studying large networks. When network size increases, finding all cycles in a network can get explicitly

calculation intensive. Some score-based approaches or heuristic approaches, however, do not require finding all the cycles and are more runtime-efficient [49], [68].

Score-based *p*-cycle selection utilizes the efficiency metrics discussed earlier in the CHAPTER 3. These metrics are used to rank *p*-cycles based on their protection efficiency. Selecting or pre-selecting a lesser amount of highly efficient *p*-cycles using an efficiency metric has shown to provide at least equally good results, and in the meantime, significantly reduce cycle enumeration time [32]-[34], [39]-[40], [68]. In addition, some heuristic *p*-cycle selection methods avoid greedy searching of cycles and suitable for solving large-scale network problems [49]. For example, Lo *et al.* designed a two-step heuristic method in selecting and grooming high-quality candidate *p*-cycles, which does not require enumeration of all cycles. However, the selection part of the heuristics can increase capacity inefficiency whereas the grooming part of this approach may prolong the program runtime [49]. To be specific, the selection approach taken in the first step can result in a considerable amount of idle capacities ("wastes") which impede the overall capacity efficiency. The second step of grooming ("refining") is to conduct a secondary search of cycles in order to replace any inefficient cycle pairs from step one. This grooming step can significantly increase program runtime, especially in large networks.

3.4.3 *p*-Cycle Protection Capacity Optimization: LP/ILP Approach

As mentioned earlier in CHAPTER 2, there are two algorithmic approaches to designing a fully restorable *p*-cycle protection: a one-step approach and a two-step approach. The one-step approach refers to the joint capacity allocation (JCA) approach, where working path routing and spare capacity placement are optimized jointly [36]. The one-step approach to solving a *p*-cycle protection optimization problem is the spare capacity allocation (SCA) approach. The SCA optimizes the allocation of spare capacities so that the total cost of capacity placement will be minimized, regardless of working path routing. Optimization of *p*-cycle protection capacity may be solved by either ILP models or case-specific heuristic algorithms.

Joint Capacity Allocation (JCA) Model

In the JCA model, the total capacity (spare capacity plus working capacity) is optimized as a whole. For each origin-destination node pair in the network, several eligible working routes are available. Working paths are selected concurrently with spare capacity allocation to minimize the cost of the operation. The ILP formulation of JCA problem, as well as the definitions of specific parameters and variables are presented as follows [12]:

S = set of network spans

C = set of eligible cycles

D = set of non-zero demands

E^t = set of eligible working routes for demand t

s_j = number of spare capacity on span j, $\forall j \in S$

w_j = number of working capacity on span j, $\forall j \in S$

c_j = cost or length of span j, $\forall j \in S$

$p_{i,j}$ = number of spare links needed on span j to form a copy of p-cycle i, $\forall i \in C, \forall j \in S$

($p_{i,j}$ =1 if cycle i traverses span j, 0 otherwise)

n_i = number of copies of p-cycle i, $\forall i \in C$

$x_{i,j}$ = number of paths that a p-cycle i provide to restore span j, $\forall i \in C, \forall j \in S$

($x_{i,j}$ = 0 if span j not on p-cycle i, $x_{i,j}$ = 1 if span j on p-cycle i, $x_{i,j}$ = 2 span j straddles i)

d^t = an integer values of demand for demand pair t

$f^{t,e}$ = integer number of demand for t^{th} demand allocated to the e^{th} eligible route, $\forall e \in E^t$

$\gamma_j^{t,e}$ = 1, if the e^{th} working route for the t^{th} demand traverses span j, 0 otherwise

Objective function: Minimize $\sum_{\forall j \in S} c_j \bullet \left(w_j + s_j \right)$ (Eq. 3.1)

Subject to: $d^t = \sum_{\forall e \in E}^{E} f^{t,e}, \ \forall t \in D$ (Eq. 3.2)

$$w_j = \sum_{\forall t \in D} \sum_{\forall e \in E} f^{t,e} \bullet \gamma_j^{t,e}, \quad \forall j \in S \qquad \text{(Eq. 3.3)}$$

$$s_j = \sum_{\forall i \in C} p_{i,j} \bullet n_i \qquad \text{(Eq. 3.4)}$$

$$w_j \leq \sum_{\forall i \in C} x_{i,j} \bullet n_i \qquad \text{(Eq. 3.5)}$$

$$n_i \geq 0, \quad \forall i \in C \qquad \text{(Eq. 3.6)}$$

All variables are non-negative and integer

The objective function of this JCA model (Eq. 3.1) minimizes the cost of both working routing and placing spare capacities to restore failed span *j*. The first constraint, as shown in the Eq. 3.2, makes sure that all the demands are routed. The second constraint (Eq. 3.3) indicates the amount of working capacity on span *j* will support routing of all demands. The constraint in Eq. 3.4 ensures that the spare capacity on span *j* can support all the *p*-cycle *i* that cross the span. Eq. 3.5 is the constraint that ensures total protection provided by the *p*-cycle *i* for each failed span *j* is sufficient to restore the working capacity on the span *j* fully. In reality, all variables must be non-negative and integer, as indicated in Eq. 3.6.

Spare Capacity Allocation (SCA) Model

In the case of a two-step *p*-cycle SCA model, working paths are routed in advance via preferred routing methods, such as the shortest path method [26] or the DFS algorithm [32], [35]-[39]. Given the working capacities, the allocation of spare capacities with minimum cost

will be determined to ensure 100% protection for all working paths. The *p*-cycle SCA ILP can be formulated by modifying the above mentioned JCA model by eliminating a few sets, parameters, variables and constraints. The resulting ILP formulation for a *p*-cycle SCA problem will be as shown below in equations 3.7 - 3.10, where the objective function (Eq. 3.7) will simply be to minimize the cost of placing spare capacities to restore failed span *j*:

Objective function: Minimize $\sum_{\forall j \in S} c_j \bullet s_j$ (Eq. 3.7)

Subject to: $s_j = \sum_{\forall i \in C} p_{i,j} \bullet n_i$ (Eq. 3.8)

$w_j \leq \sum_{\forall i \in C} x_{i,j} \bullet n_i$ (Eq. 3.9)

$n_i \geq 0, \quad \forall i \in C$ (Eq. 3.10)

All variables are non-negative and integer

Apart from the above two-step approach, an SCA problem may also be formulated without enumerating candidate cycles (step one) using ILP [69], [90]-[91]. For example, Wu *et al.* [69] proposed three ILP models to solve the *p*-cycle SCA design problem. These three ILP models were based upon recursion, flow conservation, and cycle exclusion. The study showed a linear correlation between the number of ILP variables and constraints versus network size in the flow conservation approach [69].

3.4.4 *p*-Cycle Protection Capacity Optimization: Heuristics/Meta-Heuristics

Several prior studies have proposed heuristic or meta-heuristic algorithms in solving *p*-cycle capacity optimization problems [39]-[40]. The following are some problem-specific heuristic algorithms and a meta-heuristic approach (genetic algorithm) that were proposed in the some prior studies.

Capacitated Iterative Design Algorithm (CIDA)

Capacitated Iterative Design Algorithm (CIDA) was a *p*-cycle placement operation proposed by Doucette *et al.* [39]. CIDA selects and places a set of *p*-cycles that provide full network protection with near-minimal spare capacity. The process starts by generating a set of eligible candidate cycles using a preferred cycle enumeration method, such as Add, Expand, Grow, SLA, etc. The current efficiency of candidate cycles will be calculated using capacity-weighted efficiencies, also known as the *actual efficiencies*, E_w (see details in CHAPTER 3). *p*-Cycles with the highest E_w score will be selected and placed in the network. The network traffic flows protected by this *p*-cycle is deducted from existing working capacity values. To be specific, subtracting one unit of working capacity from each on-cycle span of the *p*-cycle just placed, and two units on each straddling span (if present). Then, update the working capacity values of the current network so that they always reflect the current *actual* efficiency status at each iteration. The process is repeated recursively until all the network working capacities are protected.

The pseudo-code for CIDA is shown below:

```
function CIDA(OriginalCycleSet, i):
    initialize CycleSet, work[], and CycleUse[]
    CycleSet = Grow(OriginalCycleSet)
    while work[i] > 0 for all spans i:
        BestCycle = 0
        for each cycle p in CycleSet:
            calculate Ew(p)
            if Ew(p) > Ew(BestCycle):
         BestCycle = p
        if BestCycle not in CycleUse → add BestCycle to CycleUse
        else if BestCycle in CycleUse:
            CycleUse[BestCycle] = CycleUse[BestCycle] + 1
    for each on-cycle span i in BestCycle:
        work[i]= work[i] - 1
    for each straddling span i in BestCycle:
        work[i]= work[i] - 2
    return CycleUse
```

ER-Based Unity *p*-Cycle Design

Another example of heuristics is an *ER-Based Unity p-cycle Design* proposed in [40]. A *unity-p-cycle* refers to a unidirectional *p*-cycle with capacity equals to one unit of wavelength on each span. A *unity-p*-cycle provides one unit of protection in the opposite direction for an on-cycle span, and two units of protection for each straddling span (one unit in each direction). The *ER* refers to *efficiency ratio* (as previously described in Section 3.3), which is the ratio of the number of *protected* working units by a *p*-cycle to the number of spare capacities provided by that same *p*-cycle. This algorithm starts by enumerating all candidate cycles in the network, given network topology and traffic demand. Determine working capacity on each span of the network based on working path routing performed in advance. Then, determine the ER value of each candidate cycles. Place the cycle with the largest ER value into the network and deducting the number of working units protected by this *p*-cycle from the network. Repeat the process by selecting the second cycle with the largest ER value, and so on. If multiple cycles share the same ER value, then they can all be selected at the same time as long as they are span-disjoint to one another. The algorithm will terminate when working capacity on each span is reduced to zero, which means full protection to the network has achieved.

Genetic Algorithm

Genetic algorithms may be implemented in conjunction with heuristics in solving network survivability problems [53]. In work conducted by Pastor *et al.* [53], a hybrid heuristic-GA (HGA) approach was proposed to optimize a network routing and wavelength assignment (RWA) problem. Genetic algorithms can also be applied as a standalone method to solve various aspects of communication network survivability and optimization problems, including survivable network topology designs [70]-[72], network service reliability [73], shortest path routing in survivable networks [74], etc.

Detailed examples and discussions regarding past GA implementation in survivable network designs and *p*-cycle protection optimization will be further elaborated in CHAPTER 4 of this thesis.

CHAPTER 4. GENETIC ALGORITHMS BASICS & STUDIES

4.1 Introduction to Genetic Algorithms

Genetic algorithms (GAs) are stochastic, discrete search algorithms that are inspired by Darwin's theory of evolution. They solve problems by using an evolutionary process, where elite chromosomes are selected from a population (parents) for crossover and mutation processes to generate a new population (offspring). The more elite (better fitness) a chromosome is, the higher the probability it is to be selected for reproduction. The new population is expected to fit the objective function better than the predecessor population. This process is repeated until some predefined termination conditions are satisfied. The idea behind it is that the fittest individuals will be more likely to adapt to the evolving circumstances and have a better chance of survival. The offspring of these individuals will likely inherit these traits and evolve into even fitter descendants who are more likely to survive and reproduce.

4.1.1 GA Process Overview

Genetic algorithms start with a random population of *chromosomes*. Each chromosome is a possible solution to the optimization problem. Some of these chromosomes will be selected for reproduction based on their fitness values. Selected chromosomes will go through *crossover* and *mutation* operations to form a new population of *offspring*. The newer offspring is expected to have better performance (fitter) than the predecessor. The new generation of offspring will contain newly-produced offspring chromosomes as well as the parent chromosomes. This reproduction process will continue until the termination criteria are satisfied.

For a given optimization problem, a genetic algorithm process will first define its objective function and associated variables. Then, the genetic algorithm will commence with the following steps: generating the initial population and encoding the chromosomes, creating an evaluation mechanism (fitness function), selecting individuals for reproduction, the crossover

and mutation operations on chosen individuals, termination and decoding of the results. Figure 4.1 illustrates a flowchart of a generic GA process and its components.

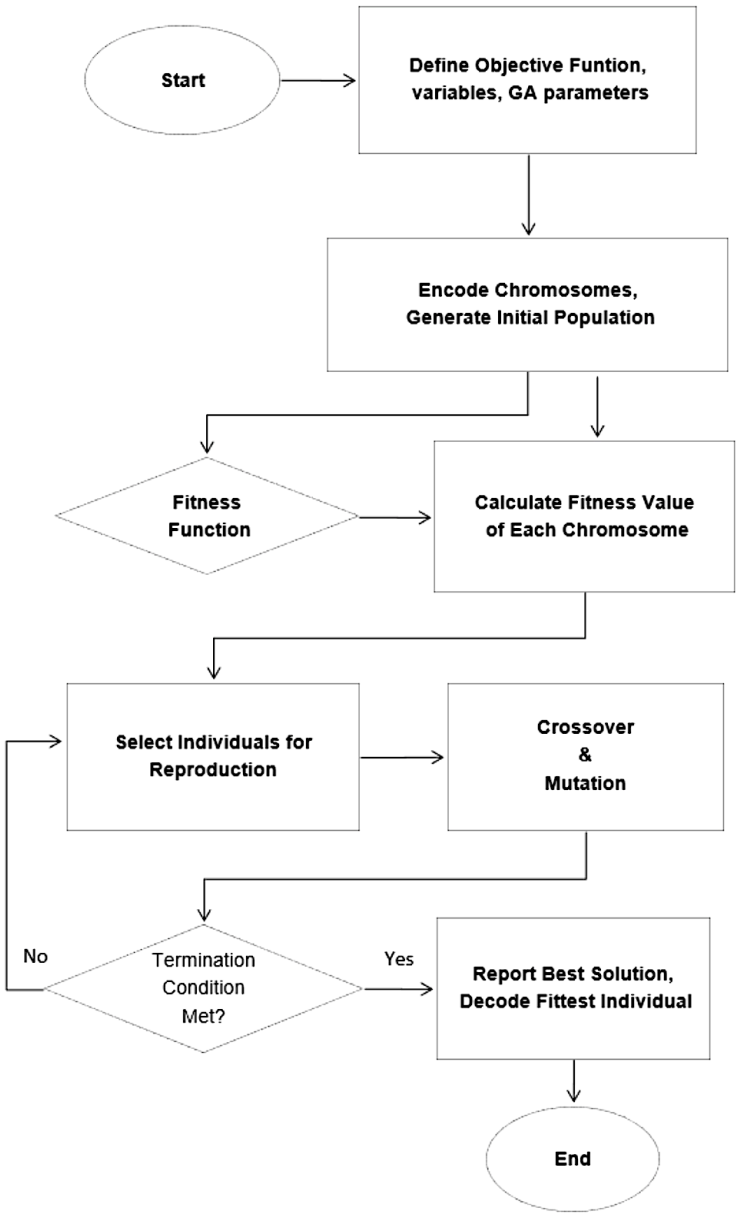


Figure 4.1 Overview of a genetic algorithmic process from start to end.

An initial population is created by generating a group of random feasible solutions (chromosomes) for the problem. Encoding a chromosome is to create a particular genetic code to represent a possible solution and to make sure it contains the required information for

solving the problem. Designing a suitable fitness function is crucial for evaluating elitism (fitness) of a solution. Based on the fitness values, two individual chromosomes will be selected for crossover using a predetermined selection method. An individual (chromosome) with a better fitness value will have a higher probability of being selected from mating, hence the survival of the fittest. For the reproduction process, a crossover operation takes place on a pair of individuals to allow the exchange of genetic information between the two chromosomes and to yield two new individuals (offspring). Besides, a mutation operator will be applied on a random chromosome and alters its genetic information at a random locus to generate a new individual. These GA operators are used to increase the diversity of a population and to prevent an algorithm from falling into local optima. Newly formed offspring from crossover and mutation and the existing parent chromosomes will create a new population, which will be used as the starting population for the next generation (iteration). This process continues iteratively until a predetermined termination condition is met, and the GA will then stop and return the best current solution. The results will be decoded to reveal the real solution to the problem.

A pseudocode for a GA is shown below, where `generate candidate individuals` is simply a space holder for any problem-specific initiation function:

```
function Genetic_Algorithm(popSize, cross_rate, mutation_rate):
    initialize spans, nodes, p-cycle sets → input_param
    generate candidate individuals → individual
    generate initial population(popSize, individuals) → init_pop
    calculate fitness values → fit_val
    rank population using fit_val → ranked_pop
    while termination condition not met:
        Initialize new set → new_pop
        Selection(ranked_pop, fit_val) → selected_parent
        Crossover(selected_parents, cross_rate) → crossover_child
        Mutation(crossover_child, mutation_rate) → mutated_child
        add crossover_child, mutated_child to new_pop
        update population, fit_val, ranked_pop for new_pop
        best_ind = ranked_pop[0]
    return best_ind
```

4.2 Key Concepts in Genetic Algorithms

4.2.1 Genomes and Chromosomes

In cell biology, genetic contents of a cell are structured into a pair of deoxyribonucleic acid (DNA) double helixes known as *chromosomes*. A complete genetic package of a cell is referred to as a *genome*. In genetic algorithms, genomes represent the search spaces [75]. The chromosomes are strings of one consistent data type (bits or real values), which represent the feasible solutions to a problem. A string chromosome can be either fixed-length or variable-length [75]. A *chromosome* is often referred to as an *individual* to a population. In this thesis, the terms *chromosome* and *individual* are used interchangeably. Two relevant concepts that are worth mentioning are *locus* and *allele*. A locus is the location of a gene in a chromosome. An allele is the value of a gene. In terms of genetic algorithms, the term *locus* means a string position which may be used to specify the location where a crossover or mutation operation takes place.

4.2.2 Schema and Schemata

According to Holland's *Schema Theorem* [75], a schema refers to a subset of a string chromosome with common features at certain positions (loci) among all chromosomes. It is composed of some common elements, and the *don't care* symbol (*) [76]. A *schema* serves as a masking template for a chromosome. It matches a specific string at every position (locus) other than the *. For example, a schema **110** would match these four strings {11101, 01101, 11100, 01100}. A *schemata* refers to sets of encoded string chromosomes that share one or more elements in common.

4.2.3 Encoding Chromosomes

Encoding a chromosome is a process of assigning an artificial data structure to each feasible solution (chromosome). Chromosome encoding allows the information of a chromosome to be processed and analyzed in a genetic algorithm. A suitable and robust chromosome design should contain essential information that is critical for resolving the objective function and evaluating fitness values. It should also be accessible to GA operators (crossover and mutation).

A common approach to encode a chromosome is a *bit string (binary) encoding*, which is relatively unsophisticated and traceable [76]-[77]. As shown in Figure 4.2, a 1 or 0 stands for two distinct features of a solution in each binary string. Another popular approach to chromosome representation is real-value chromosomes [77] (or continuous chromosomes [78]), where the bit strings are replaced with real values. Real-value encoding is a more plausible option for solving practical problems with actual parameters. For problems that involve sequencing, a *permutation encoding* may be more suitable. In a permutation encoding, each chromosome is a list of numbers that stand for their logical position in a sequence (see Figure 4.3). Permutation encoding is exceptionally suitable for optimization problems like the travelling salesman problem, where a chromosome stands for a trail of cities visited.

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
Chromosome # 1	32.0	52.4	2.5	66.1	0.3	18.8	21.9
Chromosome # 2	4	1	7	6	2	3	5
	P_1'	P_2'	P_3'	P_4'	P_5'	P_6'	P_7'

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
Chromosome # 1	0	1	0	1	1	1	0
Chromosome # 2	1	1	1	0	0	1	1
	P_1'	P_2'	P_3'	P_4'	P_5'	P_6'	P_7'

Figure 4.2 Example of two binary encoded chromosomes, where *Pi* and *Pi'* refer to the loci in Chromosome #1 and Chromosome #2, respectively.

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
Chromosome # 1	1	5	6	3	4	2	7
Chromosome # 2	4	1	7	6	2	3	5
	P_1'	P_2'	P_3'	P_4'	P_5'	P_6'	P_7'

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
Chromosome # 1	0	1	0	1	1	1	0
Chromosome # 2	1	1	1	0	0	1	1
	P_1'	P_2'	P_3'	P_4'	P_5'	P_6'	P_7'

Figure 4.3 Example of two chromosomes with permutation encoding, where each string on a chromosome refers to its position in a sequence.

4.2.4 Population and Generation

Once chromosomes are encoded, a set of individual string chromosomes (feasible solutions) will randomly form an initial population. A population can also be created by including better-fit individuals that are generated and evaluated previously. The predetermined population size will decide the number of individuals in a population. The performance of a population is improved iteratively. The current population will then serve as a starting population to reproduce a successor population in the next generation.

GA populations can be constructed in various ways. An example, as provided by Goldberg [76], is to implement a population as a series of individual string chromosomes where each chromosome contains binary strings (genotype), key variable (phenotype), and corresponding objective function value (fitness value) (see Figure 4.4).

Individual Identification	Individuals (String Chromosomes)		
	String	**Variable, *x***	**Fitness Value, *f(x)***
Ind #1	101101	3	82
Ind #2	001110	28	353
Ind #3	001010	11	198
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
Ind #*n*	111000	10	187

Figure 4.4 Example of a population of encoded chromosomes. Population size is *n*. *x* is a key variable that is plugged into the fitness function to generate corresponding fitness value *f(x)*.

Population design, population size and the number of generations play pivotal roles in enhancing the search power of a genetic algorithm and are problem-specific. If a suboptimal population size or the number of generations is used in a genetic algorithm, the algorithm may encounter insufficient exploration for a global optimum and reach a premature convergence. Alternatively, the algorithm may plateau over numbers of iterations and extend the computational runtime with no performance improvement. Typically, GA operations will significantly increase the population size, and then individuals with the lowest fitness values from both prior generation and newly generated individuals will be removed from further reproduction. By doing so, population size will remain constant in each generation.

4.2.5 Fitness Function and Objective Function

The Darwinian theory regarding the survival of the fittest can be translated into reproduction based on values of fitness, where individuals with higher values of *fitness* have higher probabilities to be selected for breeding. A problem's fitness function is designed to evaluate the level of *fitness* of each individual in a population-based on its power to provide a favourable solution. It is designed based on the problem's objective function, and the fitness value of an individual reflects its objective function value. Therefore, a fitness function links a genetic algorithm to its problem system by taking an input chromosome and generating a number that is related to its level of fitness. For example, fitness functions can be designed as measures of profits to be maximized or costs to be minimized.

Scaling of the fitness values is a commonly used practice in genetic algorithms to standardize objective function values across various problems and to maintain uniformity during the simulation process [76]. Scaling fitness values is implemented by converting a range of raw fitness values into an acceptable range that is suitable for the problem. The scaled fitness scores will assist the selection operation in picking the fittest individuals for reproduction. Without scaling of fitness values and the fitness values vary over a broad range, some outstanding super-individuals may be predominant over the selection process at the early stages of GA, which will lead to premature convergence. At later stages of GA, where the population is extensively diverse, and the fitness values vary too little, competitions among individuals will become unnoticeable. In which case, scaling up fitness values will boost rewarding the top performers [76]. There are three approaches generally adopted in literature for scaling fitness values: linear scaling, power-law scaling, and sigma truncation [76]-[77].

4.2.6 Selection and Elitism

Elitism is a strategy that allows the fittest individuals to survive and reproduce in the succeeding generations by retaining it from generation to generation. It is implemented by copying the best-performing individuals directly to a new population without any alterations.

Protecting the fittest individuals may enhance the performance of a GA rapidly for a local search; however, elitism sacrifices a GA's potential to explore a globally optimal solution [76]. Selection operation, however, chooses competent individuals from a population-based on their fitness values and places them into a new population. Selection plays a crucial role in directing a GA convergence as it determines whether an individual is to be born, and live or die [79].

A critical factor that determines the convergence rate of a GA is *selection pressure*. *Selection pressure* refers to the probability an individual with a higher fitness value is selected. To calculate the selection pressure, taking the ratio of the likelihood that the best individual is chosen to the probability that the average individual is selected. A higher selection pressure will result in a higher convergence rate, and a better individual will be more likely to be chosen. Therefore, a higher selection pressure will expedite convergence, whereas a low selection pressure may impede the process as convergence rate gets slow [80].

Various selection mechanisms have been proposed and discussed in the literature, such as fitness proportionate selection (also known as roulette wheel selection), tournament selection, ranking selection, and steady-state selection [79]. In this thesis herein (see details in CHAPTER 7), we will be implementing the two most commonly discussed selection methods: *roulette wheel selection* and *tournament selection*.

Fitness Proportionate Selection (Roulette Wheel Selection)

The *fitness proportionate selection* (also known as *roulette wheel selection*) was introduced in Holland [75] and is one of the oldest selection mechanisms. As implied by its name, the probability of an individual being selected is proportional to its standardized fitness value as compared to the sum of all fitness values. Imagine all individuals are placed on a roulette wheel. The slot sizes of the individuals are directly proportional to their fitness values, where a higher fitness value indicates a larger slot size on the wheel (as shown in Figure 4.5). The selection process starts like a marble tossed on a spinning roulette wheel. Clearly, the marble has a higher chance of stopping and pick the individual with a larger slot size on the wheel. This process will continue iteratively until a desired amount of individuals are selected.

The probability (P) for an individual i with a fitness value of f to be selected from a population (Pop) is calculated as per the following equation (Eq. 4.1):

$$P_i = \frac{f_i}{\sum f_j}, \qquad \forall j \in Pop \qquad \text{(Eq. 4.1)}$$

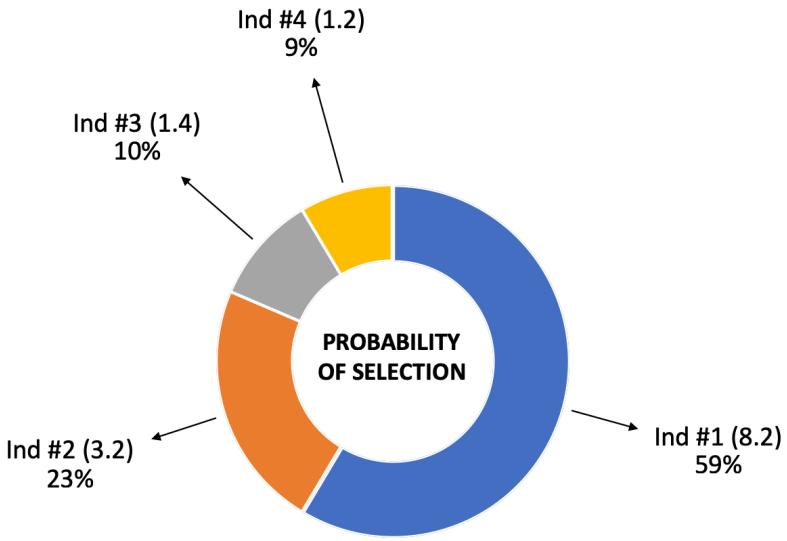


Figure 4.5 Example of a roulette wheel selection on a population with four individuals. Fitness values of these individuals are indicated in the brackets after their names.

Tournament Selection

Tournament selection starts by randomly choosing a small subset of chromosomes from the current population, where each chromosome is considered a tournament participant. The chromosome with the best fitness value in the subgroup is declared to be the winner and will be selected for reproduction. The best chromosome has a probability of p to become the winner. The second best chromosome has a probability of $p(1-p)$, while the third has a probability of $p(1-p)^2$, and so on. The size of the subset in a tournament selection will affect its selection pressure. To be specific, the more individuals in a subgroup (more competitors), the higher the selection

pressure will be [81]. Some research suggested that tournament selection is more desirable for problems with population size, where sorting gets exceedingly time-consuming [78].

Linear Ranking Selection

Linear ranking selection is the most common and straightforward type of ranking selection, where a linear equation is used as an assignment function [78]. A linear ranking selection requires sorting the population from the fittest to worst based on each individual's fitness value, extrapolating a linear function, and then assigning a number to each individual based on the linear function. The process is then followed by a proportionate selection. In a linear ranking selection, the probability of an individual being selected is directly proportional to its ranking in the sorted list of all individuals in a population [61].

Steady-State Selection

The *steady-state selection* evaluates an individual's fitness based on its linear ranking. It proceeds by choosing an individual with a higher fitness value for reproduction, and in the meantime, replacing the current worst individual with a newly generated one. Results of steady-state selection are found to follow an exponential growth, which stops when the desired population size is filled [79].

4.2.7 Crossover

The two most fundamental GA operators are *crossover* and *mutation*. *Crossover* is a GA operator that mimics the biological reproduction process, where two parent chromosomes exchange and combine their genetic contents to generate a new pair of offspring chromosomes. It is a stochastic process where the crossover point(s) is/are selected randomly.

There are three primary types of crossover: one-point crossover, two-point crossover, and uniform crossover. In a *one-point crossover*, each of the pairs of parent chromosomes

splices at the same randomly-picked crossover point along the chromosome. Then, the two parent chromosomes exchange their contents by mix-matching the portions of their genetic materials before and after the crossover point (see details in Figure 4.6).

Two-point crossover is similar to one-point crossover but with two random crossover points in each chromosome. Again, the two parent chromosomes are slitted at the same crossover points. As demonstrated in Figure 4.7, the sections between the two crossover points are swapped and combined with the remaining genetic contents in their counterpart.

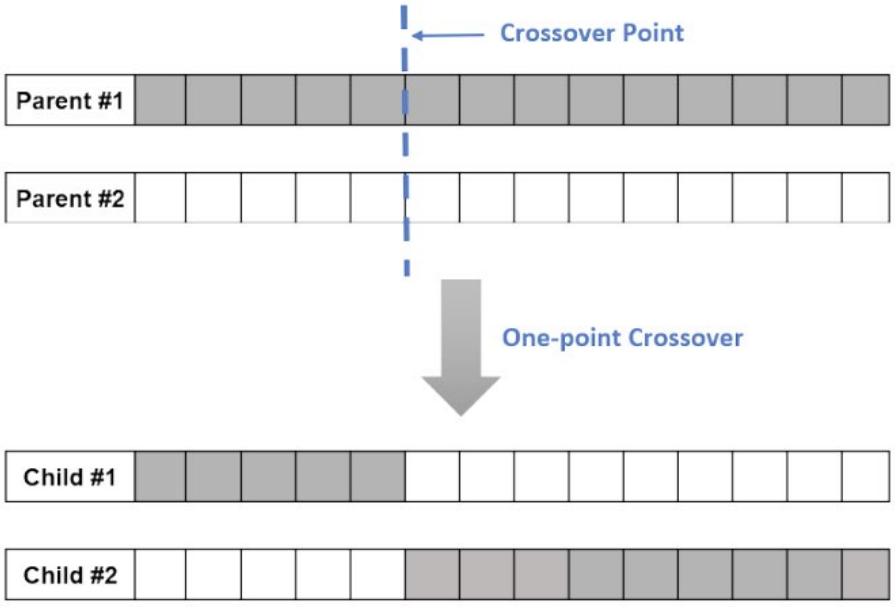


Figure 4.6 One-point crossover applied on Parent #1 and Parent #2 to generate Child #1 and Child #2. The crossover occurred at the crossover point, as indicated by the dotted line.

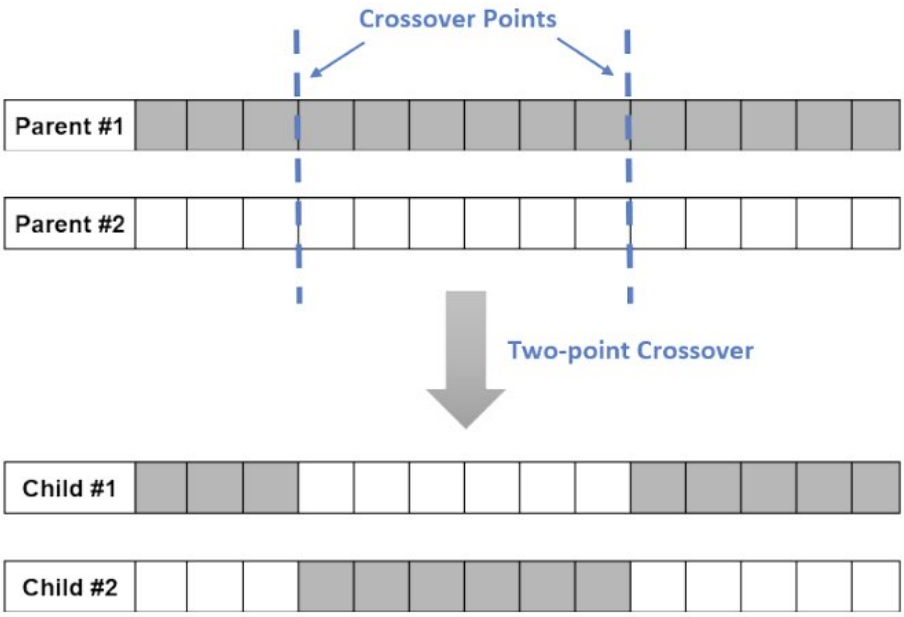


Figure 4.7 Two-point crossover applied on Parent #1 and Parent #2 to generate Child #1 and Child #2. The crossover occurred at the crossover points as indicated by the dotted lines.

Uniform crossover involves multiple crossover points in each parent chromosome. This is implemented by designing a randomly generated binary crossover mask which has the same length as the parent chromosomes. This crossover process looks at each gene on each parent chromosome individually and allocates it based on the "0" and "1" pattern along with the crossover mask. As shown in Figure 4.8, where it is a "0" on the mask, the corresponding gene in parent #1 is assigned to child #1 and parent #2 to child #2. Where it is a "1" on the mask, the corresponding gene in parent #1 will be passed on to child #2, and the gene in parent #2 goes to child #1. Therefore, the resultant pair of offspring chromosomes will contain mixed contents from both parents. The number of crossover points varies in a uniform crossover; however, the average length of a mask is found to be around half of the length of parent chromosomes [78].

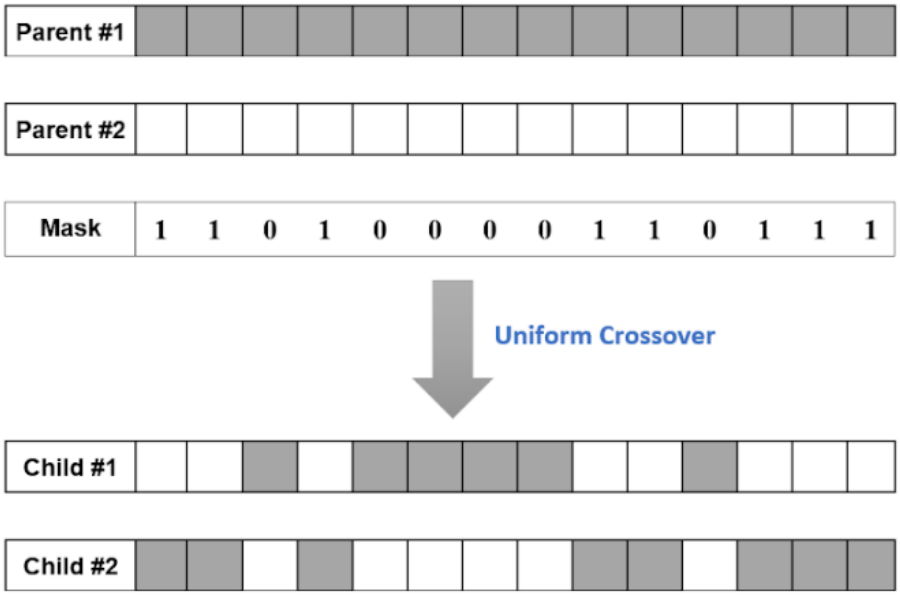


Figure 4.8 Example of uniform crossover applied on Parent #1 and Parent #2 to generate Child #1 and Child #2. The crossover mask is indicated as a binary string with the length of the parent chromosomes.

As mentioned previously, a string chromosome can be either a fixed-length string chromosome or a variable-length string chromosome. In both cases, the same types of crossover operation can be implemented (one-point, two-point or uniform); however, the crossover points vary. In the fixed-length string chromosome, the crossover points always occur at the same loci position, as shown in Figure 4.6 and 4.8. Whereas, in variable-length string chromosomes, the crossover points may shift during the reproduction process [61]. The lengths of the child chromosomes may, therefore, differ from the lengths of the parent chromosomes (as shown in Figure 4.9).

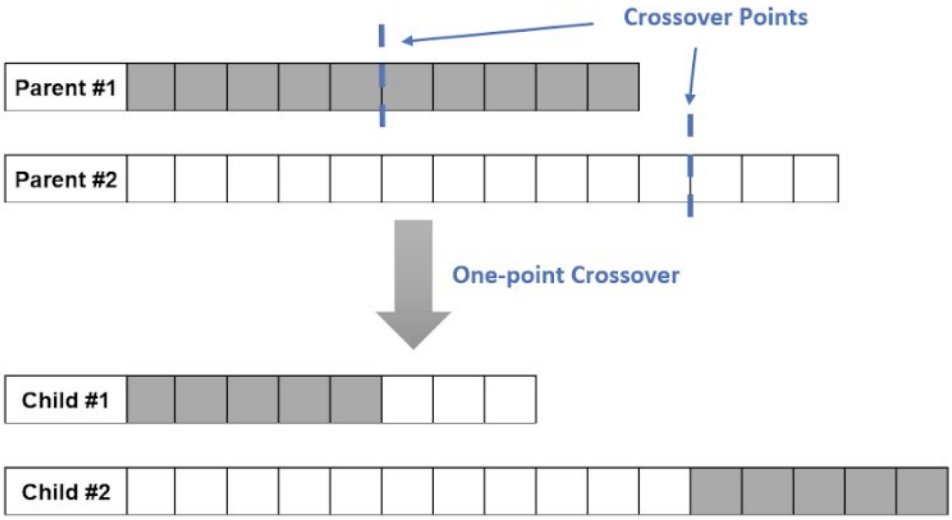


Figure 4.9 Example of one-point crossover applied on a pair of variable-length string chromosomes.

Crossover Rate

The crossover rate indicates the frequency (in percentage) a crossover operator is applied to a particular population. It determines the number of individuals from a population that is going to participate in the reproduction process. Therefore, assigning an appropriate crossover rate is critical for a valid GA. To be specific, if the crossover rate is too large, too many individuals will be disrupted by the crossover operation, and that competent individuals are less likely to be retained [78]. A GA with a small crossover rate, on the other hand, may impede the GA's search power and fail to find a local near-optimal solution.

4.2.8 Mutation

After crossover has taken place, the mutation operator is applied to each chromosome individually by altering the genes in a chromosome in a random fashion. It is executed on a population with a relatively small probability. Traditionally, a crossover has been the primary and most desirable GA operator to exploit a desirable search region by recombining highly potential chromosomes. In contrast, a mutation has been considered merely the secondary operator. However, some research has suggested that a well-designed mutation operator has the

potential to reach a different region of the search space [78]. Studies have also indicated that tailoring the population size and mutation rate has a paramount influence on enhancing GA's capability to find the local optima [78].

There are various ways to implement mutation in a GA, such as modifying values, permutation, and deletion/insertion. A more general approach to implement a mutation operator is by randomly altering genes in a chromosome. The genetic alteration may be handled as a one-point mutation or a multi-point mutation (see details in Figure 4.10). Permutation mutation is where two genes on a string chromosome are swapped (as shown in Figure 4.11)., which requires the data type of all genes to be consistent. These two approaches can be applied to the fixed-length string chromosomes [61].

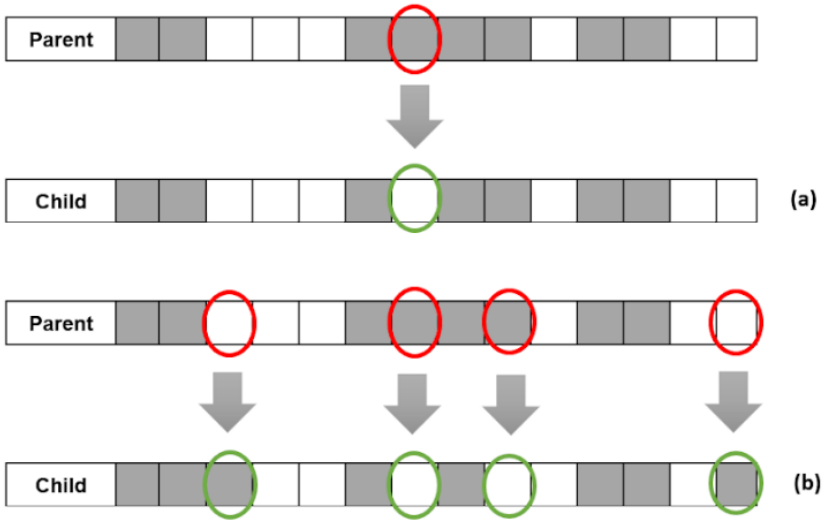


Figure 4.10 Examples of value-modifying mutation on a fixed-length string chromosome (*Parent*), where (a) shows a one-point mutation, and (b) shows a multi-point mutation.

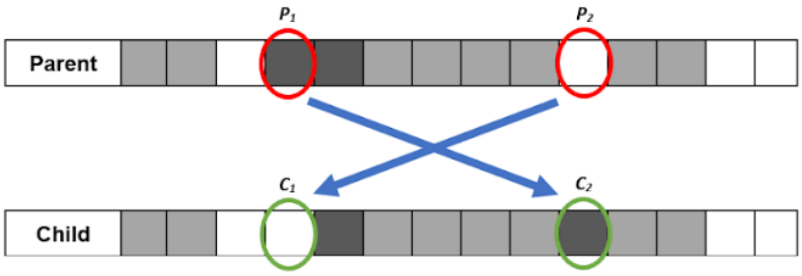


Figure 4.11 Examples of permutation mutation on a fixed-length string chromosome (*Parent*), where the gene at P_1 locus is swapped to C_2 locus in the *Child* chromosome, and the one at P_2 traded to C_1.

In the case of variable-length string chromosomes, a mutation can be implemented by either randomly inserting or deleting a few genes at random positions along a chromosome (see details in Figure 4.12).

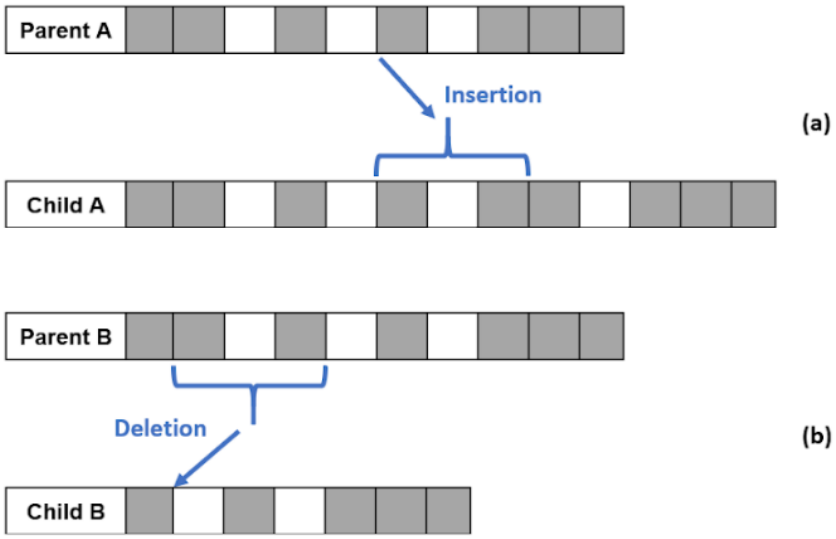


Figure 4.12 Examples of insertion (a) and deletion (b) as mutation operator on a variable-length string chromosome (*Parent A, Parent B*).

Mutation Operator vs Chromosome Encoding

Choosing a suitable type of mutation operator for a problem is closely associated with how the chromosomes are encoded. For binary encoded chromosomes, permutation can be applied to swap two different genes in a chromosome (a "1" and a "0"). The deletion/insertion of a specific gene will also work in this case. Alternatively, bit inversion is also a suitable mutation operator where a selected gene is replaced by its opposite value (a "1" substituted with a "0", or vice versa). For real-value encoded chromosomes, implementing a one-point or multi-point value modification mutation is a desirable approach. For permutation encoded chromosomes, selecting two genes and exchanging their orders in a chromosome using permutation mutation is an appropriate approach.

Mutation Rate

A mutation rate indicates the percentage of genes in a population that is going to be mutated (altered) to create new individuals. As compared to crossover rates, mutation rates are usually lower (no more than 0.1) [77]. If a high mutation rate is applied to a GA, then the GA search will ultimately be converted into a random search.

4.2.9 Other GA Operations

Various studies in the literature have discussed other advanced genetic algorithm operations reported and studied. For example, Deshpande and Kelkar [57] studied advanced genetic algorithm operator *Dominance & Diploidy* in binary string chromosomes, where new binary strings are structured regarding expressions of dominant (apparent) or recessive (hideous) alleles. *Reordering* operation is another example of advanced GA operations, which involves the movement of gene positions in a chromosome [57], [77]. It is beneficial in situations where the order of genes in a chromosome is crucial and that changing order of genes will alter phenotypic expressions. *Inversion* is a type of reordering operation where two randomly selected loci in a chromosome are reversed [77].

Detailed mechanisms of Dominance & Diploidy, reordering or inversion will not be in the scope of this research since none of these operations applies to the research problem we are going to study. For example, changing the ordering of genes in the chromosomes in our design will not affect altering the decoded string structure.

4.2.10 Termination Criteria

The GA process is repeated until predetermined termination criteria are satisfied. There are various ways to define termination criteria: a predefined fitness value [77], a total number of evolutionary iterations, a predetermined level of variations between different generations, a specific total computational runtime [54]. When the termination criteria are reached, the GA will exit the current program loop and return the best current solution.

4.3 Genetic Algorithms in Network Survivability

As mentioned previously in this chapter, genetic algorithms are capable of delivering results for NP-hard problems like network optimization problems. In fact, genetic algorithms have been applied to various aspects of communication networks survivability and optimization. Examples of typical applications include survivable network topology designs, network service reliability, shortest path routing in survivable networks, and network routing and wavelength assignment (RWA) problems.

Telecommunication network topology optimization has been studied in various research using genetic algorithms. An early study by Dengiz *et al.* [72] set the stage for optimizing communication network topology designs using genetic algorithms. Their research with 70 randomly generated test networks has proved substantial flexibility and scalability of a genetic algorithm approach in handling *NP*-hard problems. In this work, Dengiz *et al.* [72] presented a GA model to minimize the cost of a network placement. The study used variable-length string encoding of the chromosomes, uniform crossover and an effective mutation design with local search operators. For repairing disrupted chromosomes, this GA model used a repair mechanism that checks and fixes any non-two-connectivity in a candidate network.

In recent years, Morais *et al.* [70]-[71] developed a genetic algorithm for optimizing the topological designs of survivable optical transport networks with minimum capital expenditure (CAPEX). In this work, the chromosomes were encoded as binary strings with information regarding an adjacency matrix. The binary adjacency matrix carries information on node-span

correlations in a particular network. The GA model was tested with various combinations of initialization methods and GA operators. To be specific, the authors tested two initial population generators (random topology generator vs. a realistic optical network topology generator), two selection methods (roulette wheel selection vs. tournament selection), two crossover operators (one-point crossover vs. uniform crossover), and two population sizes (100 vs. 500). Simulation tests were conducted on nine test networks ranging from 9 nodes to 17 nodes.

Some studies use genetic algorithms to optimize optical network reliability for real-world communication networks, where maintaining reliable network connections and survivability against failures are critical to customer satisfaction. This type of problem focuses on connecting all customers (nodes) to infrastructure connecting points with the lowest cost while ensuring network survivability against single-span failures. To tackle this problem, Binh and Duong [73] proposed a genetic algorithm, also referred to as GA-EDP, for optimizing survivable network design problems (SNDP) to minimize network cost. The GA model was implemented with three different case-specific initialization methods, two customized crossover methods. Mutation operation for GA-EDP was conducted by adding an extraneous node and its associated spans to a selected chromosome. The proposed GA-EDP algorithm was proven to obtain promising results on both real-world and random examples.

Genetic algorithms were also used to solve the shortest path routing in network survivability. Ahn and Ramakrishna [74] developed a GA model for solving the shortest path problem in wireless networks with dynamic changes in a network topology. This study applied a GA model to the classic shortest path routing problem. It used variable-length string chromosomes to encode the problem. This design also included a simple repair mechanism to fix infeasible chromosomes after the crossover and mutation. The repair mechanism will find and remove the lethal genes (a loop) in a disrupted chromosome. This study also developed a scalable population-sizing algorithm that can be used to customize a suitable population size for a GA model, which will enhance the solution with desirable quality.

In terms of solving the routing and wavelength assignment (RWA) problems, Kavian *et al.* [82] presented a genetic algorithm for optimizing survivable optical networks that can survive single link failures. The objective function of this problem was to minimize wavelength utilization by the working and spare lightpaths while servicing the demands. The GA encoded

the chromosomes as variable-length binary strings. A single-point crossover and a binary mutation operator were used as GA operators. The GA terminated when the process reached a predetermined number of generations. This proposed approach was implemented on both dedicated path protection and shared path protection on a Pan European network with 18 nodes and 35 spans.

4.3.1 Genetic Algorithms in *p*-Cycle Protection

A recent study by Guo *et al.* [83] introduced an improved genetic algorithm using a genetic *p*-cycle combination protection strategy (GPCPS) to produce a set of *p*-cycle combinations that can fully protect the entire network topology. This result is used in secondary algorithms to optimize network spectrum allocation. The GPCPS utilizes the improved GA to optimize the arrangement of primary candidate *p*-cycle so that an effective *p*-cycle protection combination can be achieved, which fully protects the whole optical network topology with a minimum cost. The algorithm takes in network topology, a request (demand), and the number of requests to be processed, and it returns an optimized working path, a protection path and an assigned spectrum index for the particular request. The process started by generating all the primary cycles in a network topology using DFS. Then, cycles under a predetermined hop limit were selected. Individual chromosomes were encoded using a variable-length decimal encoding method where each gene is denoted with a serial number that reflects the identification of a *p*-cycle. In terms of GA operators, this study adopted the roulette-wheel selection, two-point crossover, and an adaptive mutation function. The process terminates when the maximum allowed number of generations is reached or when the objective function value remains unchanged for the maximum allowable generations. The results are verified for whether full protection has reached. For any unprotected spans, an additional minimum *p*-cycle will be formed to protect those spans, and the newly formed minimum *p*-cycle will be added to the total *p*-cycle combination. The simulation results indicated that this proposed method was effective. This study did not look at optimizing the cost-efficiency in placing *p*-cycle. To be specific, the proposed improved GA used a fitness function that calculates the total number of unit protection that a *p*-cycle can offer. This fitness function does not reflect the cost-efficiency of

protection (total protection versus the cost of cycle placement), nor does it consider how real-world demands may affect *p*-cycle protection effectiveness.

Genetic algorithms have also been effectively implemented in optimizing *p*-cycle selection for maximizing *p*-cycle protection efficiency [68], [84]. For example, Colmán *et al.* [84] proposed a GA model that takes a set of previously generated candidate *p*-cycle and calculates an optimized set of *p*-cycle with maximum protection efficiency. The chromosomes in this problem had dynamic sizes, which enveloped multiple information (a *p*-cycle ID and a value indicating the number of cycles). All required information regarding features of all candidate *p*-cycles is tabulated into a Protection Table, which can be used to decode chromosomes. This work implements a binary tournament selection and a mutation operator that involves replacing individuals. At each generation, the GA model retained the top performer by directly passing the best individual onto the next generation with no genetic alterations. Simulation tests were conducted with eleven test networks, ranging from 11 nodes to 66 nodes. This work did not consider demands as a relevant factor that may influence *p*-cycle selections.

Meixner *et al.* [68] adopted the GA model developed in [84] and incorporated a novel *p*-cycle efficiency metric as a fitness function. The novel *p*-cycle efficiency metric proposed in this study was referred to as *Efficiency of Restoration* (EoR). The metric assesses a *p*-cycle's efficiency based on its total protection, total hop count, average *p*-cycle length and fairness of cycles (standard deviation of cycle length). This study also proposed a new heuristic algorithm called *Efficient Restoration Algorithm* (ERA). GA was implemented as a benchmark, and the results were compared to the results of the ERA.

Genetic algorithms were also adopted to solve the *shared risk link group* problems in *p*-cycle network protection. Shared risk link groups (SRLG) refer to sets of optical links on a cycle that are grouped together and, therefore, will fail simultaneously in the event of a node or duct failure [58]. In the research conducted by Paez *et al.* [58], a GA design was proposed to find the best set of *p*-cycles that are capable of enhancing the survivability of network traffic flows against SRLG. The proposed algorithm was also capable of restoring network failure faster and more efficiently.

This thesis herein will be devoted to developing a novel heuristic algorithm and a GA model for solving the *p*-cycle spare capacity allocation problem with given demands. The details regarding these two novel design will unfold in CHAPTER 6 and 7 of this thesis. For the heuristic algorithm, a scalable cost-efficient and computational memory-efficient model is proposed that targets solving the spare capacity allocation problem in large-scale networks. A selection of network topologies with up to 140 nodes will be used in testing this model, which is avant-garde. For the GA approach, a problem-specific chromosome encoding method is proposed that adequately reflects all genetic information. Two efficient and objective-driven repair mechanisms are developed for this problem. Also, an additional contribution of this work includes an extensive study on novel mutation operator design and refining for this problem. To the best of our knowledge, there has not been a study conducted on designing and testing a suitable GA model for minimizing the cost of *p*-cycle spare capacity allocation with given demands, nor has there been a similar problem-specific repair or mutation method proposed.

CHAPTER 5. EXPERIMENTAL SET-UP & BENCHMARKING

So far in this thesis, key concepts regarding survivable networks and *p*-cycle protection in survivable network design have been introduced. Fundamental concepts and background information regarding heuristics, meta-heuristics and genetic algorithms, as well as their applications in network survivability and *p*-cycle protection, have also been reviewed and discussed. To the best of our knowledge, there has not been a heuristic method proposed in particular for large-scale networks and tested using multiple large test networks that are over 100 nodes. Besides,, there has not been a GA model explicitly proposed for optimizing *p*-cycle spare capacity allocation problem. In this thesis, we would like to challenge and scale up the *p*-cycle survivability problem for large networks using heuristic and meta-heuristic methods. A novel heuristic algorithm will be developed for enumerating highly efficient *p*-cycle in large-scale networks. In addition, a scalable GA model will be proposed for optimizing the *p*-cycle spare capacity allocation problem. This chapter is devoted to providing details regarding the experimental and computational set-up and test network topologies.

5.1 Experimental Network Models

For the WDM optical mesh networks discussed in this thesis, a network topology can be denoted as *G(N, E)* where *N* stands for the set of nodes and *E* is the set of edges (spans). $|N|$ and $|E|$ represent the number of nodes and the number of edges, respectively. All the network graphs used in this research are weighted and undirected, and their topologies are known in advance. Each span's weight is uniformly one for all the calibration networks (the *USA* network and the *France* network) to allow comparison between the experimental results from this study with those in Doucette *et al.* [39]. However, to simulate real-world problems, the weight of each span in all other test case networks (10n20s to 140s220s) will be adopting the Euclidean distance between two end nodes of the said span.

5.1.1 Calibration Networks

Since we will conduct comparison studies on our novel heuristic algorithm versus the *Grow* algorithm, we will be using the *USA* network [78] and the *France* network[4], [79] used in Doucette *et al.*[39] as our calibration networks in CHAPTER 6. The *USA* network and the *France* network have average nodal degrees of 3.2 and 3.3, respectively. As mentioned in CHAPTER 2 of this thesis, a network's average nodal degree is calculated as per the formulae: $d = 2 * |E||V|$. The *USA* network will again be used as a calibration network for designing and calibrating the GA model in CHAPTER 7.

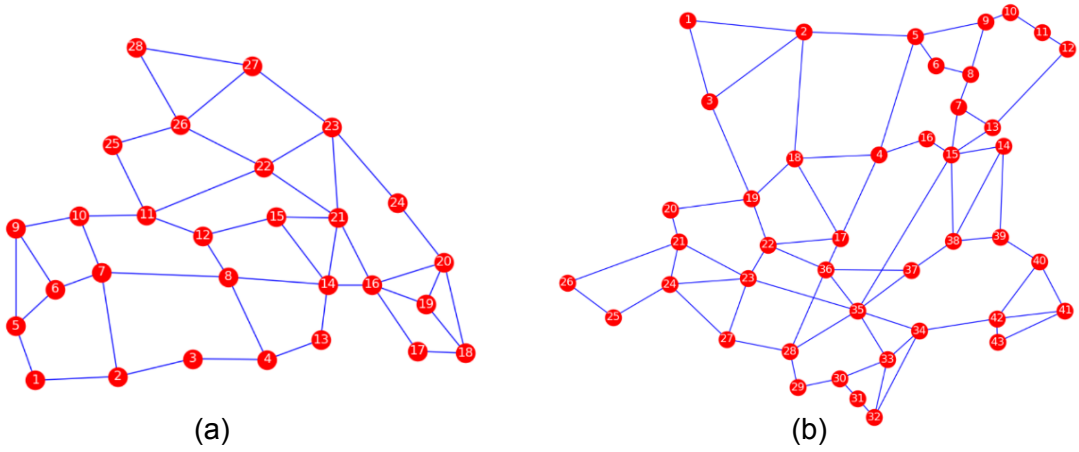


Figure 5.1 (a) *USA long-haul* network [78]; (b) *France* network [4], [79].

The two calibration network topologies are illustrated in Figure 5.1, which are generated using the *networkx* package in *Python* version 3.7.6. The nodes and spans features of these two topologies are listed in Table 5.1.

Table 5.1 Features of Calibration networks: *USA* and *France* networks

Networks	Number of Nodes	Number of Spans	Avg. Nodal Degrees
USA	28	45	3.2
France	43	71	3.3

5.1.2 Test Case Networks

A total of 14 test case network topologies will be used in the experiments conducted in this thesis, which were selected from [25]. Each of the test case networks was modified from a master network (same nodes but difference spans, average nodal degree equals to 4) by removing random span(s). By doing so, the average nodal degrees of these test cases decreased from 4 to 3 progressively. All the test case networks in this study are bi-connected and have average nodal degrees between 3 and 4. Table 5.2 presents nodes and spans features for all 14 test case networks. Network notations follow the pattern of *X-n-Y-s*, where *X* stands for the number of nodes, and *Y* is the number of spans. For example, *10n20s* refers to the test network topology with 10 nodes and 20 spans. The weight of each span of all test case networks (10n20s to 140s220s) is determined using the Euclidean distance between two end nodes of the said span. The network topology graphs for all 14 test case networks will be provided in the Appendix I, which are generated using the *networkx* package in *Python 3.7.6* [86].

Table 5.2 Features of all 14 test networks. Span costs are equivalent to Euclidean distances.

Network	Number of Nodes	Number of Spans	Avg. Nodal Degree
10n20s	10	20	4
20n34s	20	34	3.4
30n45s	30	45	3
40n60s	40	60	3
50n80s	50	80	3.2
60n96s	60	96	3.2
70n105s	70	105	3
80n128s	80	128	3.2
90n135s	90	135	3
100n150s	100	150	3
110n165s	110	165	3
120n180s	120	180	3
130n195s	130	195	3
140n210s	140	210	3

5.2 Demand Models and Working Routing

Demands of a transport network refer to an aggregation of all traffic flows from an origin node to a destination node of the transport network [3]. Working capacity on every span can be determined based on traffic demands. A working path carries traffic demands during regular operation. In this thesis, one unit of traffic demand takes on one unit of working capacity on each span between an origin node and a destination node of a network. Also, only integer units of traffic demands will be considered in this thesis unless stated otherwise. Usually, network traffic demands of optical WDM networks can be forecasted or determined beforehand [50], and the information can then be used for working capacity routing.

For the algorithms in this thesis, working capacity routing was conducted beforehand using the shortest path method in all test networks, i.e., probably solved in an SCA fashion. Therefore, the working capacity on every span of each test network is considered an input parameter for the algorithms.

5.3 Computational Set-Up

All the heuristic algorithms (e.g., DDCD and CIDA) and genetic algorithm models generated in this thesis work are programmed using *Python 3.7.6* [86]. All the *Python* coding is conducted, compiled and debugged using the *Visual Studio Code* [87]. All ILP models in this thesis herein are compiled using a third party software called AMPL [88] and are solved using IBM ILOG CPLEX Interactive Optimizer 12.6.1.0 [89]. AMPL is an LP modelling language used to describe mathematical programming formulations in general algebraic forms. AMPL reads an LP model and case-specific data and generates a readable standard file format imported to an LP solver (i.e., CPLEX) to find the optimal solution of the LP [17]. All experiments are programs run on a server with 12-core ACPI multiprocessor X64-based PC with Intel Xeon® CPU E5-2430 running at 2.2 GHz with 96 GB RAM. All ILP results include a default mipgap of 0.0001 (unless specified otherwise), which indicates that the results are ensured to be within 0.01% of optimal.

5.4 Experiment Benchmarking

Studies in this thesis will be using the experimental results presented in [39] as benchmarks, where Doucette *et al.* compared performances of the CIDA-*Grow* and pure ILP models on both the *USA* and *France* networks. The experimental results from [39] are presented in the Table 5.3 below. In this table, "# of *p*-cycles" indicates the total number of candidate *p*-cycles generated using either the *Grow* algorithm or the ILP model. The "Work" and "Spare" stand for the total working capacity and total spare capacity of a network design, respectively. "Redundancy" is the percentage of spare capacity over the working capacity. The lower the redundancy, the better the efficiency a design resembles. "% Difference" indicates how much (in %) the CIDA-*Grow* result deviates from the pure ILP approach (the true optimal or near-optimal results). "RT" stands for the total runtime of a model, which includes the time taken to generate eligible cycles plus the time taken to solve the ILP model using CPLEX 7.5 [39]. In the *USA* network, a "Pure ILP" approach is where all the eligible cycles from the network topology are enumerated and used as possible candidate cycles when optimizing the capacity cost, therefore, the resulting "Redundancy" value is the true optimal result. In the *France* network, Doucette *et al.* generated the shortest 15,000 eligible *p*-cycles out of over 500,000 possible cycles using DFS, which is used as a near-optimal solution for this problem.

Table 5.3 Test results as reported in Doucette *et al.* [39] for the *USA* and *France* networks

	USA Network			**France Network**		
	Grow + CIDA	**Grow + ILP**	**Pure ILP**	**Grow + CIDA**	**Grow + ILP**	**Pure ILP**
# of p-Cycles	839	839	7321	2407	2407	15000*
Work	1273	1273	1273	4043	4043	4043
Spare	1212	1164	1064	3890	3692	3675
Redundancy	95.2%	91.4%	83.6%	96.2%	91.3%	90.9%
% Difference	13.9%	9.4%	0.0%	5.9%	0.5%	0.0%
RT (sec)	0.32	1.70	17.29	1.93	8.25	541.08

** The shortest 15,000 eligible p-cycles out of over 500,000 possible cycles generated by DFS [39]*

CHAPTER 6. A NOVEL HEURISTIC METHOD FOR *p*-CYCLE DESIGN

6.1 Introduction

As discussed previously, the *p*-cycle is a promising network protection mechanism in WDM mesh networks due to its ring-mesh dichotomy and its capability to protect off-cycle straddling spans. In a non-joint approach to design a fully restorable *p*-cycle network protection in WDM mesh networks, the first step is to generate a set of eligible and efficient candidate *p*-cycles that can provide full protection across the entire weighted network. The second step is to optimize the allocation of selected p-cycle spare capacities to achieve 100% network survival with the minimum spare capacity cost. Since *p*-cycles are constructed in the spare capacity of a network, solving a *p*-cycle spare capacity allocation problem does not interfere with working capacity routing.

As stated previously, heuristic methods refrain from searching cycles exhaustively and are suitable for solving large-scale network problems. Prior studies on heuristics in *p*-cycle designs, as discussed in CHAPTER 3 of this thesis, have shown promising results in various network topologies that have less than 50 nodes. Among these studies, the *Grow* algorithm for cycle enumeration and the CIDA for cycle placement by Doucette *et al.* [39] were the most frequently referenced benchmark methods. The experiments conducted in this chapter of the thesis will use the *Grow* algorithm as the benchmark to the proposed novel heuristic method. The CIDA algorithm will be implemented as one of the spare capacity allocation methods in this research. The *Grow* algorithm starts by enumerating some small primary *p*-Cycles using the *straddling link algorithm* (SLA) proposed in [32]. By doing so, it generates a wide range of *p*-cycles in various sizes (small *p*-cycles, intermediate *p*-cycles, and very large *p*-cycles) based on the primary cycles. CIDA selects *p*-cycles with the highest current actual efficiency (E_w) scores and places them in the network. Once a cycle is placed onto the network, working capacity values are updated by subtracting one unit of working capacity from each on-cycle span of the selected *p*-cycle, and two units on each straddling span (if present). Working capacity values are updated at each iteration. This process is repeated iteratively until all the working capacities are protected (meaning all working capacity values are reduced to zero).

A review of prior work indicates that the majority of previous studies have conducted simulation tests using networks that have less than 40 nodes (see details in Table 6.1). Among which, *COST239* with 11 nodes and 26 spans and *USA long-haul network* of 28 nodes and 45 spans are the most frequently tested network topologies. Some studies used network topologies that have more than 60 nodes [64]-[69], [90]-[93]. However, these methods expedited the eligible cycle searching process using genetic algorithms, which is an alternative approach that we will implement in the next chapter. As mentioned in [25], there is no explicit definition for a *large* network. However, since the majority of the test case networks in prior work are no larger than 40-node networks, network topologies that have 50 nodes or more with nodal degrees of three and above will be referred to as *large* networks in this study.

Table 6.1 Test networks used in selective literature which adopted heuristic or ILP approach in *p*-cycles designs and pre-selections against single network failure. * indicates studies that used genetic algorithm.

Reviewed Literature	Test Network Used
[56]	6n8s, 17-node GERMAN network
[42]	8n13s
[28]	9n16s
[14][15][29][69][91]	SmallNet network (10n22s)
[14][16][17][29][35]-[37][42] [45][46][48][50][69][91]	COST239 network (11n26s)
[47]	12n19s, 13n23s, 15n26s
[34][42][43][45][47][56][68]*	NSFNET network (14n21s)
[92]	15n27s
[14][15][35][40]	KL Network (15n28s)
[68]*	European Core Network (ECN)(18n39s)
[40][42][43][49]	EON network (19n38n)
[14][15]	20n31s, 30n59s
[60]	28-node EUROPEAN network
[32][33][36][39][43][48][49][68]*	USA long-haul network (28n45s)
[69]	30n62s
[32][39]	France network (45n71s)
[14][15]	53n79s
[68]*	China National Backbone Network (CNBN)(66n120s)

To the best of our knowledge, there has not been a study that experimented with heuristic cycle enumeration algorithms on large networks in particular. With the rapid advancement of and growing demands for network technologies, we are motivated to pursue this study to develop methods that can effectively scale up and solve large-scale network problems. In this chapter, we will introduce and test a novel heuristic method for highly efficient candidate cycle enumeration that is proved to be effective in large networks, which is referred to as the *disjoint-paths Dijkstra cycle development* (DDCD) algorithm. The DDCD approach starts by generating short primary *p*-cycles using a double shortest path approach inspired by [33]. In the next step, a heuristic algorithm is proposed to iteratively *develop* an extensive set of *p*-cycles of various sizes using the short primary p-cycles.

The remainder of this chapter is organized as follows. Introduction and design consideration of the novel *disjoint-paths Dijkstra cycle development* (DDCD) algorithm will be presented in Section 6.2. Section 6.3 includes detailed descriptions of the experimental set-ups. The experimental results and algorithm performance will be presented in Section 6.4, which includes the results of calibration networks and all test case networks with sizes ranging from 10 nodes to 140 nodes will be presented. Finally, Section 6.5 will wrap up this chapter with key findings and conclusions.

6.2 A Novel Heuristic *p*-Cycle Enumeration Algorithm

6.2.1 Design Considerations

The heuristic cycle enumeration algorithm is referred to as the *disjoint-paths Dijkstra cycle development* (DDCD). It is a recursive cycle enumeration method which is capable of "developing" efficient *p*-cycles in small and large networks (over 100 nodes). This algorithm is designed based on the following considerations:

First, the DDCD algorithm will be generating *p*-cycles of various sizes. Liu and Ruan [33] stated in their research that a good candidate cycle set should incorporate both large cycles with multiple straddling spans as well as small-sized candidate cycles. Analyzing the set of *p*-cycles generated by *Grow* from [39] also indicated that the resulting candidate *p*-cycles contained a

diverse profile of *p*-cycles: very small *p*-cycles, intermediate *p*-cycles, and very large *p*-cycles. As discussed in [21], *p*-cycle sets with higher average cycle length (meaning more larger *p*-cycles) provide better average protection efficiency. In the meantime, small *p*-cycles are essential for preventing excessive capacity redundancies. In a situation where the working capacities of most spans are protected, short *p*-cycles are needed to protect the remaining working capacities with a lower spare capacity cost [33].

Second, the DDCD algorithm avoids duplicate *p*-cycles and is more memory-efficient, especially when applied on large-scale networks. In the *Grow* algorithm, a significant amount of duplicate cycles were generated and retained during the process. For example, the *Grow* algorithm created 839 candidate cycles for the *USA* network and 2407 cycles for the *France* network as reported in [39]. However, after duplicate *p*-cycles in both cycle sets are eliminated, only 309 and 863 candidate cycles remain in each cycle set, respectively. The DDCD algorithm is designed to check and remove duplicate cycles, and avoid the accumulation of duplicate cycles during the process as well as in the final cycle set.

Finally, the amount of *p*-cycles to be enumerated is controllable in the DDCD algorithm. The DDCD algorithm finds new *p*-cycles repetitively, and the number of iterations is an input parameter for the algorithm. While some network topologies may require more cycles, some may need less to provide equally good results (if not better) as compared to the benchmark. By controlling the number of iterations, the number of output cycles will be customized as per the input test network topologies. Correlations between the input number of iterations versus network sizes will be discussed later in this chapter.

6.2.2 *Disjoint-Paths Dijkstra Cycle Development* (DDCD)

Figure 6.1 shows an overview of the mechanism involved in the DDCD algorithm mechanism that is applied on a network topology as shown in Figure 6.1 (a). In general, there are three steps involved in the DDCD algorithm:

1) Generating starting cycles based on each span *i* in a network (see Figure 6.1 (b));

2) Larger *p*-cycles by converting on-cycle spans to straddling spans (Figure 6.1 (c));

3) Post-enumeration procedure to verify straddling spans (Figure 6.1 (d)).

The first two steps will yield a set of high-merit candidate *p*-cycles with a wide range of sizes. The last step ensures computational soundness, memory efficiency, and data correctness. All the network topologies used herein are undirected and bi-connected.

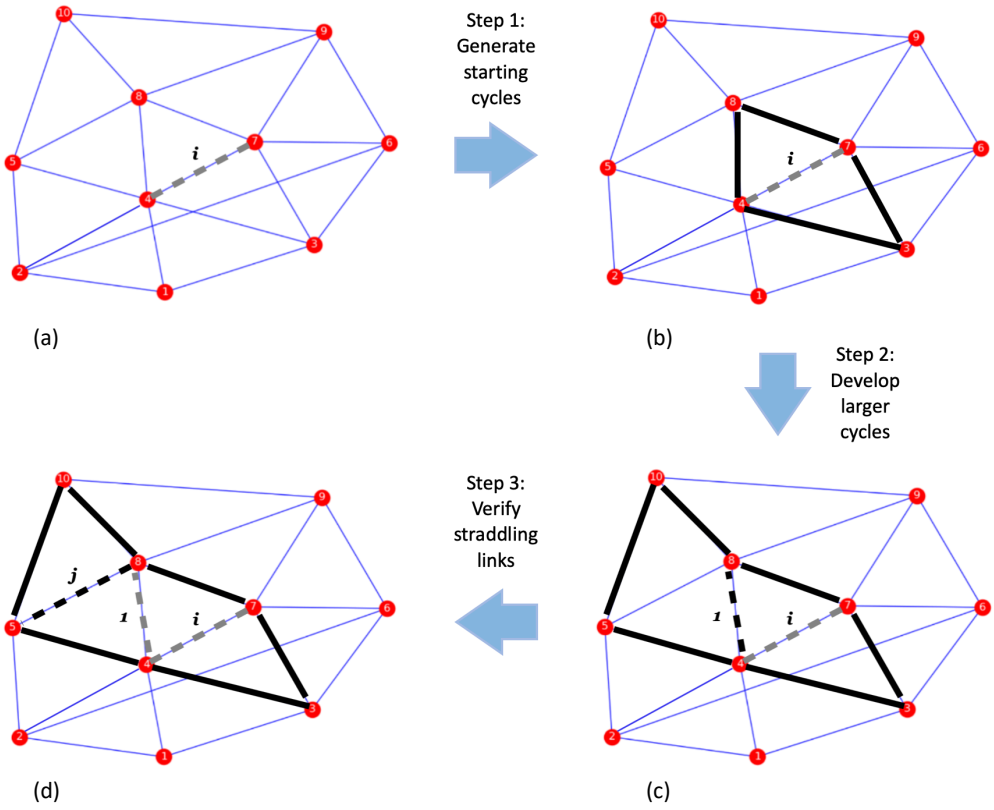


Figure 6.1 Overview of the DDCD algorithm mechanism applied on a topology shown in (a).

Step One: Generating Starting Cycles

Similar to the *Expand* and *Grow* algorithms in [39], the DDCD algorithm operates on an initial set of short *p*-cycles. These starting cycles are crucial for developing eligible and capacity-efficient *p*-cycles at later stages. Each of the short cycles generated at this step will be an input cycle for the DDCD algorithm, where further operations will be conducted to generate larger *p*-cycles with higher capacity efficiency. The starting cycles are generated using the short cycle enumeration method proposed in [33], which is referred to as a *double-shortest-path* method in our implementation. For each span *i* of a network, the double-shortest-path algorithm will find a

minimum cycle that protects *i* as an on-cycle span **and** a short *p*-cycle if such a cycle exists. Figure 6.2 demonstrates how double-shortest-path algorithm is used to generate minimum cycle and short *p*-cycle. An overview of the process can be described as follows:

1) Initialize a new cycle set where all newly enumerated short cycles will be saved:

   `New_Cycle_Set = {}`

   Note that in our *python* implementation, each item in the `New_Cycle_Set` is an eligible cycle that is distinguished by a specific cycle ID plus the cycle's span information (span ID + span relation values). The span relation values are represented as either *1* or *2*, where *1* represents an on-cycle span, and *2* represents a straddling span.
2) For each span *i* of a network *N*, locate the two end nodes (N1 and N2) of span *i*. Using Dijkstra's algorithm, find two shortest paths r_1 and r_2 between N1 and N2 that are node-disjoint from span *i*.
3) If only r_1 exists for span *i* (illustrated in Figure 6.2(a) and displayed in the pseudo-code below as `distance(`r_1`) != MAX`), then connect r_1 with *i* to form a minimum cycle *a* that includes span *i* as an on-span cycle. The algorithm will proceed no further and the cycle *a* is saved to the `New_Cycle_Set`. As shown in Figure 6.2 (a), only one shortest path (r_1) exists between two end node of span *i* (Node 1 → Node 2). Therefore, only one minimum cycle is formed by joining r_1 with span *i*.
4) If both r_1 and r_2 exist for span *i*, the algorithm will first form and keep the minimum cycle *a*. Then, a *p*-cycle *b* will be formed by joining the r_1 and r_2 at their end nodes, where span *i* is a straddling span to the cycle *b*. Save the cycle *b* to the `New_Cycle_Set`. This is shown in Figure 6.2 (b) and displayed in the pseudo-code below as `distance(`r_1`)!= MAX` and `distance(`r_2`)!= MAX`. As illustrated in Figure 6.2 (b), two node-disjoint shortest paths (r_1 and r_2) are found for span *i* (Node 4 → Node 7). In this case, a minimum cycle is formed by joining r_1 with span *i*. In addition, a short *p*-cycle is formed by joining r_1 and r_2, with span *i* straddles over this cycle.
5) This process continues until all the spans in the network *N* is explored. The process will then terminate and return the `New_Cycle_Set`.

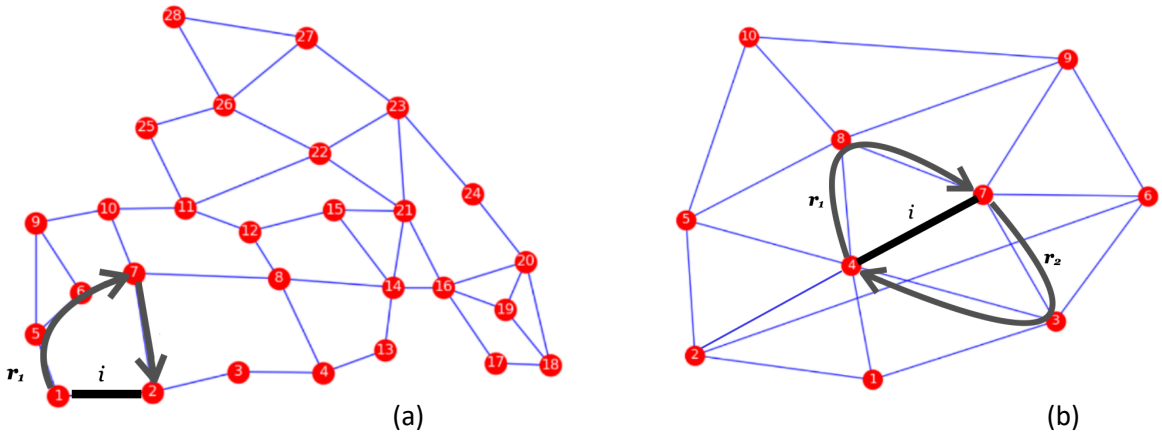


Figure 6.2 Generating starting cycles (and *p*-cycles) using double-shortest-path method.
(a) a minimum cycle is formed by joining $r_1 + i$, (b) a small *p*-cycle is formed by joining $r_1 + r_2$.

The pseudo-code for generating starting cycles and double-shortest-path are as follows:

```
# all_spans is a set of all spans in the network N;
function StartCycles(all_spans):
    initialize New_Cycle_Set
    for each span i in all_spans:
        N1 = origin_node(i)
        N2 = destination_node(i)
        double_ShortPath(Graph, N1, N2) → r1, distance(r1), r2,
                                          distance(r2)
        if distance(r1) != MAX:
            add all the spans of r1 to span i → cycle a
            save cycle a to New_Cycle_Set
        if distance(r2) != MAX:
            add all the spans of r2 to r1 → cycle b
            remove duplicate spans
            mark span i as a straddling span
            save cycle b to New_Cycle_Set
    return New_Cycle_Set
```

```
function double_ShortPath(Graph, origin, destination):
    set graph distance between origin & destination to MAX
    for each span i on Graph:
        Dijkstra(i, unmarked spans/nodes) → r1, distance(r1)
        if distance(r1) != MAX:
            mark visited nodes
            Dijkstra(i, unmarked spans/nodes) → r2, distance(r2)
        unmark all spans/nodes
    reset graph.distances()
    return r1, distance(r1), r2, distance(r2)
```

Step Two: *Disjoint-Paths Dijkstra Cycle Development* (DDCD)

The DDCD algorithm takes the starting cycles that were generated at the previous step and *develops* various larger *p*-cycles with higher capacity efficiency (more straddling spans). The DDCD algorithm adopts Dijkstra's algorithm to replace each span of a selected cycle progressively by a node-disjoint path. The DDCD algorithm starts at an arbitrary span *i* on a starting cycle *p*. The algorithm finds the shortest node-disjoint path that connects the end nodes of span *i* (if existed) and then connects this path with the remaining portion of cycle *p* to form a new *p*-cycle. Span *i* is then removed from the newly-formed cycle. Therefore, span *i* will be transformed from a on-cycle span of the old *p*-cycle to a straddling span to the newly generated *p*-cycle. Note that in the *python* code implementation for the DDCD algorithm, straddling spans are "marked" as straddling spans to be distinguished from on-cycle spans and other off-cycle spans. If such a node-disjoint path does not exist, the algorithm will move on to the next span on cycle *p* and start over. If such a path does exist, a new *p*-cycle is generated. The visited nodes along the path are then "marked" and hidden, so the same path will not be explored again. The algorithm will then find the second shortest node-disjoint path for span *i,* and the process will continue as above. If no more node-disjoint paths can be found to connect the end nodes of span *i*, the algorithm will jump to the next span on the cycle and repeat the same procedure. The algorithm will terminate when all the spans on the cycle *p* are explored.

Figure 6.3 illustrates the process of developing larger *p*-cycles using the DDCD algorithm. Figure 6.3(1) shows a starting small *p*-cycle enumerated from Step one of the DDCD algorithm. This starting *p*-cycle has four on-cycle spans, denoted as *Span 1, Span 2, Span 3,* and *Span 4*. Figure 6.3(2) illustrates the scenario when the cycle development process starts at *Span 1*. As shown in Figure 6.3(2), there exist more than one node-disjoint paths between the two end nodes of *Span 1*, including but not limited to p_1 (grey), p_2 (green), p_3 (yellow), *and* p_4 (orange). Figure 6.3(3a)-Figure 6.3(3d) presents the resulting newly formed *p*-cycles by replacing *Span 1* with the node-disjoint paths p_1 - p_4, respectively. In these newly formed *p*-cycles, *Span 1* becomes a straddling span. Therefore, Step two of the DDCD algorithm generates larger candidate *p*-cycles with at least one more straddling span (better protection efficiency). Every time when a node-disjoint path is explored, it will be "hidden" so it will not be explored again. Once a span (e.g., *Span 1*) is fully explored, the algorithm will move on to the next spans (*Span 2, Span 3,* and *Span 4*) on the topology until all spans are explored. Each new *p*-cycle formed will be saved to the `New_Cycle_Set` as a valid candidate.

Unlike the *Grow* algorithm, DDCD finds all possible node-disjoint paths for each chosen span (if existed). Therefore, more than one new *p*-cycles (if existed) are generated for each span explored on a starting cycle. As in the *Expand* and *Grow* algorithms, all the newly generated *p*-cycles are saved and added to the total *p*-cycle pool to be used in a later cycle allocation optimization process (e.g., CIDA). Therefore, the resulting total *p*-cycle pool from DDCD will incorporate a diverse set of cycles that includes minimum cycles, short *p*-cycles, medium-sized *p*-cycles and very large *p*-cycles.

The pseudo-code for developing larger *p*-cycles using DDCD is provided in the section below.

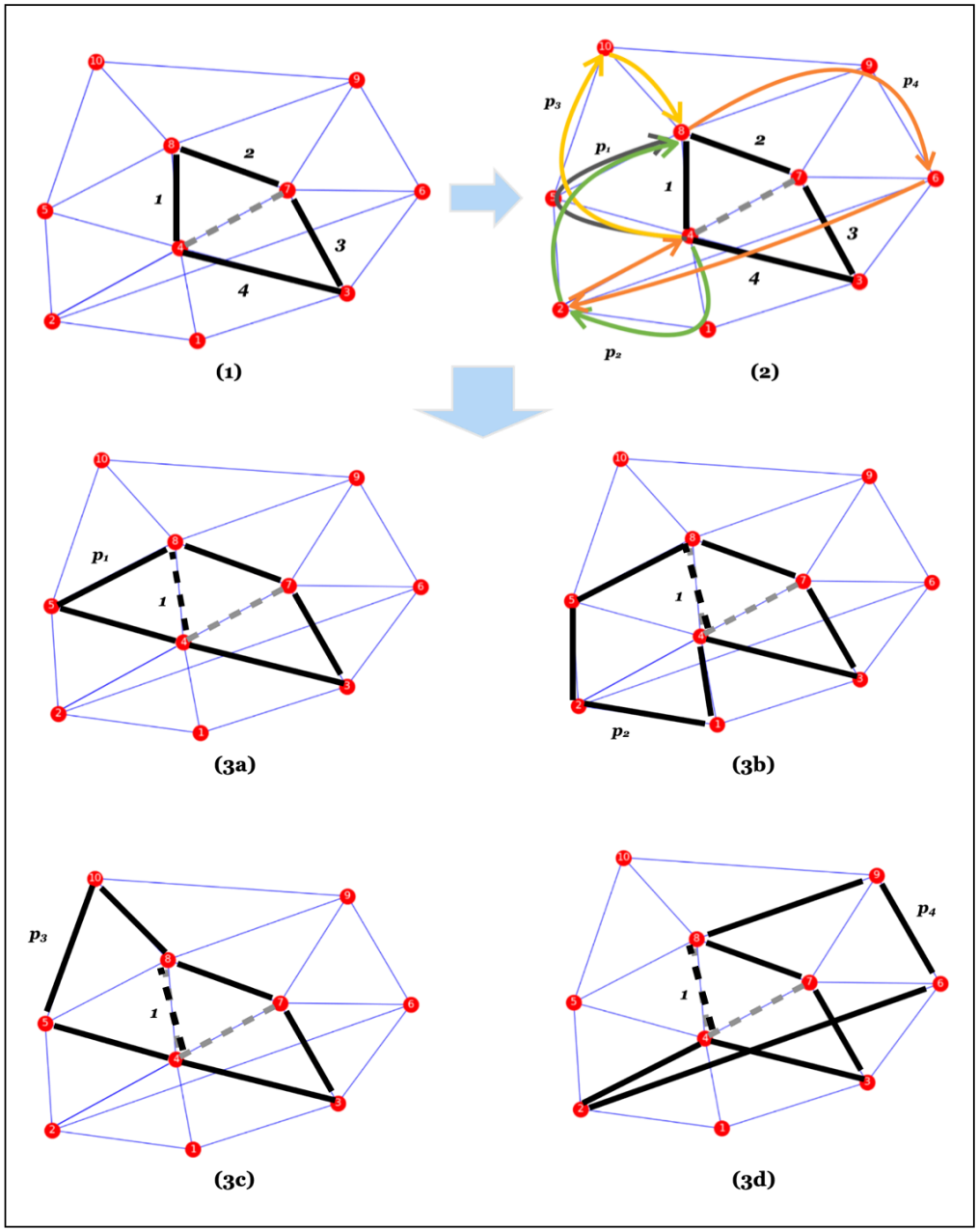


Figure 6.3 Overview of developing larger *p*-cycles from a start cycle (a) using DDCD algorithm.

```
function DDCD(input_cycle_set, Graph, new_cycle_id):
    initialize New_pCycle_Set
    for each cycle c in input_cycle_set:
        initialize marked_nodes_list
        let cycle c' = cycle c
```

```
        for on-cycle span i on cycle c':
            mark all nodes except end nodes → marked_nodes_list
            while marked_nodes_list:
                Dijkstra(i, unmarked spans/nodes) → p_i, dist(p_i)
                if path p_i exists:
                    add path p_i to cycle c' → cycle c"
                    mark span i as straddling span
                    add cycle c" to New_pCycle_Set
                    add nodes on path p_i to marked_nodes_list
                Dijkstra(i, unmarked spans/nodes) → p_i', dist(p_i')
                if Dijkstra() does not return p_i':
                   break
            unmark all nodes and spans
    return New_pCycle_Set
```

In the above pseudo-code, note that the data structure in our *python* implementation for each *p*-cycle in the `New_pCycle_Set` contains a *p*-cycle ID and its span information (span ID + span relation values). The span relation takes on a value of either *1* or *2*, where *1* represents an on-cycle span, and *2* represents a straddling span.

As stated previously, The DDCD algorithm searches cycles iteratively in a network topology, where the number of iterations (*k*) is a controllable input parameter. Note that *k* can take on any integer values from 1 to ∞. As the value shifts from 1 to ∞, the DDCD algorithm will find as many candidate cycles as possible. Once no more can be found, the DDCD algorithm will exit and return the most recent collection of *p*-Cycles. The entire DDCD process is demonstrated by the following pseudo-code (the *asterisk part will be presented later in Step Three):

```
start_cycles = startCycles(all_spans)
for k ∈ [1,∞), do:
    initialize new_cycle_id
    new_pcycles = DDCD(start_cycles, Graph, new_cycle_id)
    remove duplicate cycles* → no_dup_pcycles
    add no_dup_pcycles to start_cycles → New_pCycle_Set
    if no new cycle found → break
```

Step Three: Duplicates Removal and Validity Checks

After *p*-cycle enumeration, there are a few miscellaneous cleaning and verification steps that we take to ensure excellent memory efficiency and data correctness.

a) Remove Cycle Duplicates. At the end of each DDCD iteration, all the candidate cycles in the total cycle pool (e.g., *input_cycle_set*) are checked for possible duplicates. This duplicate removal step is implemented by first calculating all candidate cycles' costs and sorting the cycles based on their costs. Cycles with different costs are considered non-duplicates by default. Cycles with the same costs, however, are grouped together where their span information will be investigated to find out whether these cycles are identical or not. We compare the on-cycle spans of each cycle within each group with those of other cycles within the same group. Non-duplicate cycles are added to a new list (e.g., a new *p*-cycle set called *no_dup_cycle_set*), whereas the duplicates are flagged and dropped. The resulting *p*-cycle set (*no_dup_cycle_set*) will contain no duplicates. This duplicate removal step is conducted at the end of an iteration (see the *asterisk** part in previous pseudo-code for Step Two), as well as after the Step b) below. The pseudo-code for removing duplicate *p*-cycles is as shown below:

```
function Remove_Duplicates(input_cycle_set):
    initialize no_dup_cycle_set, cycle_costs_set
    for cycle c in input_cycle_set:
        calculate cycle costs → cycle_costs_set
    sorted_cycles = groupby(cycle_costs_set, cycle costs)
    for cycle c' in group i in sorted_cycles:
        if set(on-cycle span)for c' = set(on-cycle span)for c+1:
            flag duplicates
            break
        if not duplicates:
            add cycle c' to no_dup_cycle_set
        i = i+1
    return no_dup_cycle_set
```

b) Straddling Span Check. During the process of DDCD, when a new *p*-cycle is formed, only the selected span *i* was marked as a straddling span. As the size of the node-disjoint paths grows, larger cycles will be generated that may contain multiple additional bonus straddling spans that are not explicitly expressed during the process. Therefore, an additional check on all the cycles is performed to capture any missed straddling spans. If a span is found to straddle the cycle but is not previously identified, it will be "marked" as a straddling span. Since we assign values of *1* or *2* to each span on a *p*-cycle, "marking" a straddling span means to assign a value of *2* as its span relation, whereas an on-cycle span will retain a value of *1* for its span relation. This step is conducted by scanning all spans and nodes information is all candidate cycles in a *p*-cycle set (`input_cycle_set`).

For example, say a `cycle p` from `input_cycle_set` is structured as follows:

`{cycle p: {span 1: 1, span 2: 1, span 3: 2, …, span i: 1}, …}`

where `cycle p` is the cycle ID and `{span 1: 1, span 2: 1, span 3: 2, …, span i: 1}` is the cycle span information (*span ID : span relation*). For any two arbitrary nodes on the `cycle p`, if there exists a span *s* in the network graph *G*, which is not included in the span information. The span *s* is an additional straddling span to the `cycle p`, which should be added to the cycle span information and assigned a span relation of *2*.

As illustrated in Figure 6.4, the *p*-cycle in Figure 6.4(1) is a newly formed *p*-cycle developed by Step two of the DDCD algorithm. This step only identified the straddling *span 1* at which the new *p*-cycle was formed. We may notice that there is still a straddling span (*span j*) in this newly formed *p*-cycle that is not identified yet. *Span j* is identified during this verification step and is "marked" in the cycle's span information.

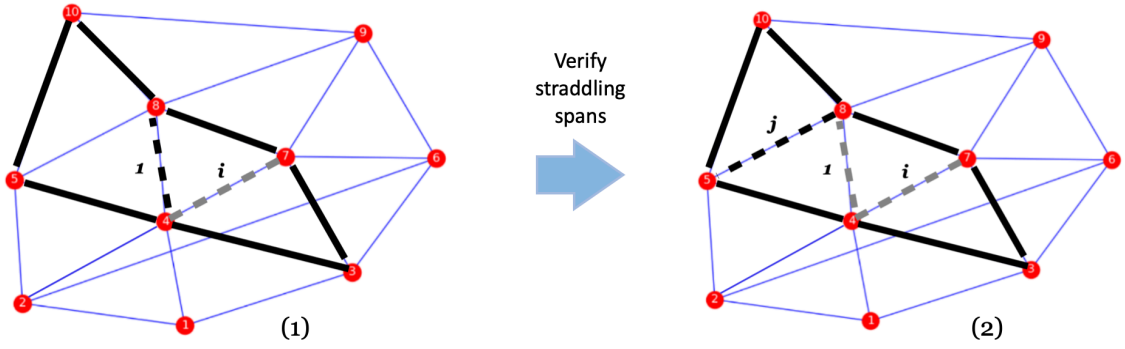


Figure 6.4 Updating span information for a newly formed *p*-cycle (1) at the DDCD verification step.

The pseudo-code for this step is shown as follows:

```
function Straddle_Spans(input_cycle_set, Graph):
    for (cycle p, cycle_p_span_info) in input_cycle_set:
        initialize cycle_node_set, updated_pcycle_set
        save all nodes on cycle p → cycle_node_set
       for each node pair from cycle_node_set:
           if span s ∈ Graph and span s ∉ cycle_p_span_info:
               add span s to cycle_p_span_info as straddling span
               update cycle p information in updated_pcycle_set
    Return updated_pcycle_set
```

c) Save to a *.pcycle* File. Last but not least, for accessibility of these *p*-cycle data at later stages of our experiments (e.g., CIDA optimization and genetic algorithms operation), all the *p*-cycle data are saved to a *.pcycle* file, which can be imported into other algorithms.

6.3 Experimental Set-Up

As mentioned earlier, the SCA approach will be implemented to solve the *p*-cycle protection design problem. The candidate *p*-cycles are generated using the novel DDCD algorithm. In addition, the *Grow* algorithm [39] and the conventional depth-first search (DFS) algorithm are implemented as our benchmark cycle enumeration methods. After the *p*-cycles are enumerated, both a heuristic algorithm (CIDA) and an ILP model are used for spare capacity allocation optimization. These two methods are used to select a set of *p*-cycles to protect the network's working capacities entirely while minimizing the cost of spare capacity.

The DDCD algorithm is programmed using *Python 3.7.6*. The *Grow* algorithm is from [94], which was programmed previously in C++ (see pseudo-code in Section 3.4.1 of this thesis). The DFS approach is from [95] and was also programmed in C++ (see pseudo-code in Section 2.4.2 of this thesis). The CIDA is implemented based on the pseudo-code provided in [39]. It is

programmed using *Python 3.7.6* (see pseudo-code in Section 3.4.4 of this thesis). The ILP spare capacity allocation model is from [96] and was presented and discussed in Section 3.4.3. The three cycle-enumeration methods and two spare capacity allocation methods are paired up as follows: DDCD + CIDA, DDCD + ILP, *Grow* + CIDA, *Grow* + ILP, DFS + CIDA, DFS + ILP. The result of each pair will be presented and compared against one another in the experimental results section in this chapter. Where applicable, all eligible cycles in a network topology will be enumerated using DFS which will be followed by SCA ILP model to find the optimal solution.

There are various input files and parameters, algorithmic models, and output files and information that are associated with developing these experiments. Detailed lists of these items (item names and explanations) can be found in the Appendix section of this thesis.

6.3.1 Benchmarks and Test Cases

As mentioned in CHAPTER 5, the calibration network topologies for this study are the *USA long-haul* network and the *France* network from [39]. Other test network topologies vary from 10 nodes with 25 spans to 140 nodes with 210 spans. The number of working capacity units on each span is determined by the shortest-path routing of all traffic demands in [94]-[96]. Since a uniform span weight of one is used in [39], we will continue adopting the same span weight in our experiments using the *USA* and *France* network to facilitate results comparison. For any experiments conducted on other test case networks, however, we will be using the Euclidean distances between two end nodes of a span as the span weights.

In our implementation and experimentation, three consecutive sets of experiments will be conducted. First, experiments will be performed on the two calibration networks using a uniform span weight of one. To be specific, the process will start by enumerating eligible *p*-cycles using the *Grow* algorithm, DDCD algorithm, and DFS, and then follow up with spare capacity allocation using CIDA and ILP. The results will be compared against the benchmark results as reported in Doucette *et al.* [39] for the *USA* and *France* networks, which is listed in Table 5.3 in Section 5.5 of this thesis. These benchmarking tests are conducted on our server, therefore, the runtimes are updated to reflect our current computational set-ups. Once benchmarking tests are completed, the same sets of combined algorithms will be applied on

small test networks between 10 nodes and 40 nodes using Euclidean distances as the span weights. For these small test networks, the DFS algorithm is capable of finding all eligible candidate cycles in a network topology within three days of runtime, which provide the optimal solutions to these test cases. Finally, all sets of combined algorithms will be applied to larger networks with sizes of 50 nodes up to 140 nodes using Euclidean distances as the span weights. In these network topologies, ILP is unable to provide an optimal result within three days of runtime. In this study, tests that take longer than three days of runtimes to obtain results will be considered unsatisfactory and unfavourable.

6.4 Experimental Results and Discussion

6.4.1 Calibration network Models

As explained in the previous section, a set of eligible *p*-cycles will be generated using a specific cycle enumeration algorithm (*Grow,* DDCD, or DFS). Then the set of cycles will be used in either the CIDA or ILP model for spare capacity optimization. Table 6.2 presents the results of such benchmark tests on the *USA* network and the *France* network. As shown in the table, three cycle enumeration algorithms were used to generate candidate cycles: *Grow*, DFS, and DDCD. Each set of candidates are then combined with either CIDA or SCA ILP model for the spare capacity allocation. Experimental result generated from each test is a minimum total cost of capacity placement for 100% network survivability that each combined algorithm is capable of achieving. Since all span costs are assumed to be one, the working capacity costs will equal to the working capacity units, and the spare capacity costs equal to the spare capacity units. Results of the "CIDA + *Grow*" and "ILP + *Grow*" are the repeats of the tests as reported in Doucette *et al.* [39] with updated experimental runtimes that reflect our current computational set-up. DFS algorithm in our implementation finds the shortest *X* number of cycles given network topology. Therefore, our DFS cycle finding algorithm is executed with a controllable input parameter, the number of output cycles. This input parameter is set to be the same as the number of candidate cycles generated by DDCD, which will allow us to compare the robustness of the results by DDCD to those by DFS. Therefore, DFS and DDCD enumerate the same amount

of eligible cycles; however, member cycles in each candidate pool are different due to distinct cycle finding methodologies. Resulting cycle pools are combined with either CIDA ("DFS + CIDA", "DDCD + CIDA") or SCA ILP model ("DFS + ILP", "DDCD + ILP"). DFS is again used to find all eligible cycles in a network topology ("All") which is used in the SCA ILP model to obtain the optimal solution. This result is reported as "(Near) Optimal" in the last column of Table 6.2, where the result for *USA* is the true optimal solution using a complete set of 7,321 eligible cycles, and the result for *France* is a near optimal solution using the shortest 50,000 *p*-cycles out of over 500,000 total possible cycles [39].

For each column of a test method in Table 6.2, the "# of cycles" shows the total number of eligible cycles generated by a corresponding cycle enumeration algorithm. Each test result yields a minimal total cost of capacity placement. The "Normalized Cost" is obtained by dividing all the capacity costs by the minimum cost among all. "Total RT" is the summation of the runtime taken to enumerate candidate cycles plus the runtime taken to optimize the spare capacity allocation by either the CIDA or ILP model.

Table 6.2 Comparing DDCD with Grow and DFS for cycle enumeration in the USA and France networks

Network Topologies		***CIDA***			***ILP***			***(Near) Optimal***
		Grow	***DFS***	***DDCD***	***Grow***	***DFS***	***DDCD***	***All + ILP***
USA (28n45s)	***# of cycles***	839	1028*	1028	839	1028*	1028	7321
	Normalized Cost	1.139	1.117	1.110	1.094	1.048	**1.045**	**1.000**
	Total RT (s)	0.06	1.23	4.13	0.39	0.64	2.46	3.26
France (43n71s)	***# of cycles***	2407	28810*	28810	2407	28810*	28810	50000**
	Normalized Cost	1.130	1.127	1.072	1.072	1.043	**1.000**	**1.026**
	Total RT (s)	0.98	361.31	966.16	1.62	295.25	887.52	760.28

** Equal amount of p-Cycles as those of DDCD but different member cycles in each set*
*** Using DFS to generate the shortest 50,000 p-cycles instead of 15,000 as reported in [39]*

When comparing normalized costs, the lower the value indicates better capacity efficiency. The lowest capacity cost has a normalized value of 1.000. As shown in Table 6.2, for the same spare capacity allocation method (CIDA or ILP), candidate *p*-cycles generated by the DDCD algorithm provided full network protection with the lowest spare capacity cost compared

to those by *Grow* or DFS. Among all the tests by combined algorithms, "DDCD + ILP" generates the best near-optimal solutions in either test network. In the *USA* network, the "DDCD + ILP" result deviates from the true optimal solution by 4.5%. In the *France* network, the "DDCD + ILP" result is 2.6% better than the near optimal solution provided by the 50,000 shortest cycles in the last column. In our implementation, we generated the shortest 50,000 eligible *p*-cycles instead of the shortest 15,000 as reported in [39]. This is because the "DDCD-ILP" capacity was found to have outperformed that of the shortest 15,000 *p*-cycles, and the number of cycles generated by DDCD (28,810) has surpassed 15,000. When comparing the total runtimes for all the tests, DDCD creates a better near-optimal solution with an approximately four times longer runtime than those of *Grow* or DFS. However, the runtimes are comparable to the (near) optimal solution provided in the last column (e.g., 4.13s and 2.46s vs. 3.26s for *USA* network, 966.16s and 887.52s vs. 760.28s for *France* network). These results suggest promising results for application in larger networks, where ILP runtimes grow exponentially as network size grow.

6.4.2 Small Test Case Networks (10 Nodes to 40 Nodes)

We now apply the DDCD algorithm on networks with the sizes of 10 nodes to 40 nodes. Again, CIDA and ILP models are used for spare capacity allocation, and the results of DDCD are compared to those of *Grow* and DFS. In all the test case networks here, we use the actual Euclidean distances between two end nodes of spans as the cost of these spans. The optimal solution to each test case network is determined by the spare capacity allocation ILP model using the complete set of eligible cycles enumerated by DFS. Same as in the calibration network models discussed in Section 6.4.1, DFS and DDCD generates the same amount of eligible cycles; however, member cycles in each cycle pool are different due to distinct cycle finding approaches. Results of all small test case networks are presented below in Table 6.3.

Same as performed previously, the resulting candidate cycle pools are combined with either CIDA (results reported as "DFS + CIDA", "DDCD + CIDA") or SCA ILP model (results reported as "DFS + ILP", "DDCD + ILP"). DFS is again used to find complete sets of cycles in each network topology ,which is combined with the SCA ILP model to generate the optimal solution in each of the small network case. These results are reported as "Optimal" in the last

column of Table 6.3. For each column of a combined test method, the "# of cycles" shows the total number of eligible cycles generated by a corresponding cycle enumeration algorithm. The last column ("All + ILP") indicates the total numbers of all the eligible cycles in a network topology. "Normalized Costs" are calculated by dividing all the capacity costs by the minimum cost among all, which indicate quality of a result. The lower the normalized cost, the better the solution. A normalized cost is 1.000 indicates a minimum capacity cost across all data. "Total RT" is the summation of the runtime taken to enumerate candidate cycles plus the runtime taken to optimize the spare capacity allocation by either the CIDA or ILP model.

Table 6.3 Comparing DDCD with Grow and DFS for cycle enumeration in small test networks (10n20s to 40n60s)

Network Topologies		***CIDA***			***ILP***			***Optimal***
		Grow	***DFS***	***DDCD***	***Grow***	***DFS***	***DDCD***	***All + ILP***
10n20s	***# of cycles***	449	376*	376	449	376*	376	416
	Normalized Cost	1.058	1.141	1.056	1.004	1.061	**1.000**	**1.000**
	Total RT (s)	0.16	0.08	0.42	0.19	2.04	0.67	0.48
20n34s	***# of cycles***	1018	699*	699	1018	699*	699	2794
	Normalized Cost	1.079	1.062	1.037	1.025	1.035	**1.000**	**1.000**
	Total RT (s)	2.35	1.31	2.11	0.22	0.27	0.78	1.03
30n45s	***# of cycles***	723	1138*	1138	723	1138*	1138	15818
	Normalized Cost	1.131	1.242	1.093	1.099	1.175	**1.023**	**1.000**
	Total RT (s)	4.49	5.74	7.00	0.46	3.98	4.89	10.58
40n60s	***# of cycles***	1158	1692*	1692	1158	1692*	1692	234065
	Normalized Cost	1.156	1.191	1.125	1.104	1.144	**1.064**	**1.000**
	Total RT (s)	15.28	20.11	23.25	0.84	7.06	5.32	357801.11

** Equal amount of p-cycles as those of DDCD but different member cycles in each set*

Table 6.3 indicates that "DDCD + CIDA" and "DDCD + ILP" outperformed both "*Grow* + CIDA" and "*Grow* + ILP" in all four small test case networks. Especially in the cases of 10n20s and 20n34s, the "DDCD + ILP" method was able to find the optimal solution (normalized costs of 1.000) within comparable runtime as compared to the normalized costs and runtimes as reported under "Optimal". Therefore, the DDCD-CIDA and DDCD-ILP are proven to show

robust performance in optimizing the capacity allocation with 100% survivability in these test cases. The DFS, on the other hand, shows the least favourable performance in almost all test cases. Compared to the optimal solution in the last column, “DDCD + ILP” is proven to provide the best near-optimal solution. For example, “DDCD + ILP” obtained the optimal solutions in both 10n20s and 20n34s. In the 30n45s and 40n60s networks, “DDCD + ILP” provided near-optimal solutions that deviate from the optimal solutions by only 2.3% and 6.4%, respectively.

In terms of the test runtimes as shown in Table 6.3, all tests combined with CIDA show an increase in runtimes as the network size grows. Although DFS-ILP is still able to find the optimal solutions for these networks (results reported under “All + ILP” in the last column), we observe a significant increase in ILP runtime as the network size grows from 10n20n to 40n60s. In the 40n60s network, test runtime soared up to 357,801.11s, which is almost 7,000 times longer than the runtime of DDCD-ILP (5.32s). The sharp increase of runtime may be associated with the tremendous amount of candidate cycles involved (234,065 candidate cycles) in generating the optimal solution. In contrast, only 1,692 candidate cycles are involved in DDCD-ILP to provide a near-optimal solution that only deviates from the optimal solution by 6.40%. As network size grows, finding the optimal solution within a satisfactory time frame (e.g., three days in our experiments) will be more challenging. Finding the best near-optimal solution within an acceptable runtime will be the primary concern in large networks, where DDCD shows promising results. Although *Grow*-CIDA and *Grow*-ILP took shorter runtimes as compared to those of DDCD-CIDA and DDCD-ILP in most cases, their normalized costs indicate much weaker performance in all test networks.

6.4.3 Large Test Case Networks (50 Nodes to 140 Nodes)

As the results of DDCD combined with either CIDA or ILP have been proved to be promising on the calibration networks and the small test case networks, we will now be implementing these methods on large test case networks that are over 50 nodes. In these networks, DFS-ILP cannot find an optimal solution within acceptable runtime (e.g., over three days of runtime). Therefore, it is our primary interest to find the best near-optimal results for

these test case networks. All span costs are the Euclidean distances between two end nodes. Results of DDCD with CIDA and ILP are again compared to those with *Grow* and DFS.

As shown in Table 6.4, the lowest spare capacity costs are highlighted with bolded font and compared with other test results. Same as previous tests, the "# of cycles" is the total number of eligible cycles generated by a specific cycle enumeration algorithm (*Grow,* DFS, or DDCD). "Normalized Costs" are assessed by dividing all capacity costs by the minimum cost among all, which indicate the quality of a result. Therefore, the lower the normalized cost, the better the solution. As mentioned before, a normalized cost is 1.000 indicates a minimum capacity cost (hence, best result) across all data. "Total RT" includes both the runtime taken to enumerate candidate cycles plus the runtime taken to optimize the spare capacity allocation by either the CIDA or ILP model. For tests that fail to provide results within three days of runtime, their normalized costs are voided ("/"), and corresponding "Total RT" is recorded as "> 3 days". These tests are considered undesirable due to long runtimes.

As demonstrated in Table 6.4, DDCD-ILP still generates the best results with the lowest spare capacity costs among all six tests in every test case network. When looking at the sets of experiments with the same spare capacity allocation methods (using either CIDA or ILP), tests using DDCD as the cycle enumeration method outperformed *Grow* or DFS. Among these, DFS provides the weakest-performing candidate cycles in every test case network. Especially when the network size reaches 80 nodes and above, DFS failed to generate the same amount of candidate cycles as DDCD within three days of runtime in almost all cases. Therefore, DFS is the least desirable cycle enumeration method as compared to *Grow* and DDCD in large networks, whereas DDCD is the most plausible method for large-scale networks.

Table 6.4 Comparing DDCD with Grow and DFS for cycle enumeration in large test networks (50n80s to 140n210s)

Network Topologies		*CIDA*			*ILP*		
		Grow	***DFS***	***DDCD***	***Grow***	***DFS***	***DDCD***
50n80s	***# of cycles***	4104	13550*	13550	4104	13550*	13550
	Normalized Cost	1.094	1.143	1.054	1.058	1.096	**1.000**
	Total RT (s)	88.74	469.74	210.32	7.92	353.51	29.79
60n96s	***# of cycles***	6626	6708*	6708	6626	6708*	6708
	Normalized Cost	1.072	1.115	1.042	1.044	1.068	**1.000**
	Total RT (s)	225.18	7880.98	186.76	9.90	7712.29	21.15
70n105s	***# of cycles***	3974	10671*	10671	3974	10671*	10671
	Normalized Cost	1.099	1.155	1.051	1.057	1.107	**1.000**
	Total RT (s)	232.52	612.62	376.84	42.54	192.51	119.03
80n128s	***# of cycles***	11018	3629*	3629	11018	3629*	3629
	Normalized Cost	1.083	/	1.042	1.049	/	**1.000**
	Total RT (s)	1613.81	*> 3 days*	173.37	64.18	*> 3 days*	17.49
90n135s	***# of cycles***	7548	2953*	2953	7548	2953*	2953
	Normalized Cost	1.103	1.408	1.051	1.075	1.337	**1.000**
	Total RT (s)	1794.62	6444.63	161.36	50.65	6178.02	17.66
100n150s	***# of cycles***	7114	4278*	4278	7114.00	4278*	4278
	Normalized Cost	1.122	/	1.039	1.083	/	**1.000**
	Total RT (s)	2091.85	*> 3 days*	12.74	69.92	*> 3 days*	12.74
110n165s	***# of cycles***	9669	3278*	3278	9669	3278*	3278
	Normalized Cost	1.068	/	1.033	1.028	/	**1.000**
	Total RT (s)	4543.99	*> 3 days*	315.55	150.65	*> 3 days*	16.22
120n180s	***# of cycles***	8485	5353*	5353	8485	5353*	5353
	Normalized Cost	1.167	/	1.040	1.131	/	**1.000**
	Total RT (s)	6545.41	*> 3 days*	1082.25	333.34	*> 3 days*	27.42
130n195s	***# of cycles***	11531	4675*	4675	11531	4675*	4675
	Normalized Cost	1.058	/	1.034	1.037	/	**1.000**
	Total RT (s)	12473.49	*> 3 days*	1228.40	532.04	*> 3 days*	30.6
140n210s	***# of cycles***	7155	7450*	7450	7155	7450*	7450
	Normalized Cost	1.251	/	1.029	1.272	/	**1.000**
	Total RT (s)	10212.74	*> 3 days*	2085.01	804.79	*> 3 days*	47.41

** Equal amount of p-cycles as those of DDCD but different member cycles in each set*

6.4.4 Runtimes vs. Network Sizes

Let us now have a look at how the runtime changes for the DDCD tests ("DDCD + CIDA" and "DDCD + ILP") versus the *Grow* tests ("*Grow* + CIDA" and "*Grow* + ILP") as the network sizes increase from 50n80s to 140n210s. All the test results are obtained from Table 6.7 and are plotted using line charts.

Figure 6.5 illustrates the comparison between the "DDCD + CIDA" tests runtimes and the "*Grow* + CIDA" tests runtimes in test cases from 50n80s to 140n210s, as outlined in Table 6.7. It is shown in the figure that the *Grow*-CIDA runtime inclines as the network size grows; however, the DDCD-CIDA runtimes are maintained at below 2,500s for all network sizes and only start to incline from 110n165s slightly. The runtime differences are especially significant for networks larger than 80n128s, where "*Grow* + CIDA" starts to show a rapid increase in runtime and "DDCD + CIDA" remains relatively steady. Therefore, the DDCD-CIDA approach is proven to be a much more runtime-efficient heuristic approach for large networks than *Grow*-CIDA.

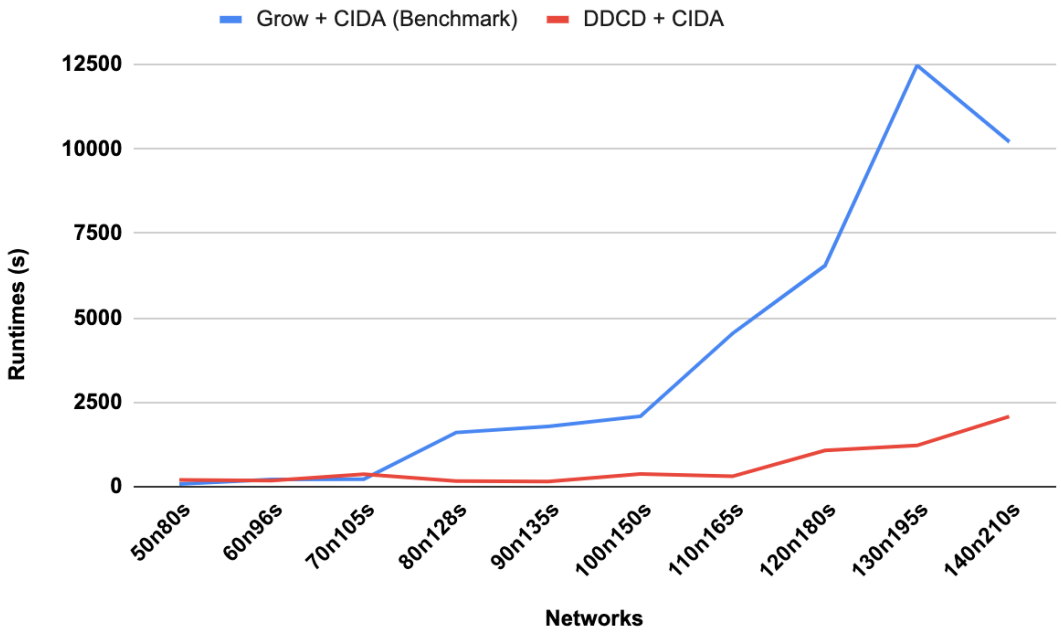


Figure 6.5 Comparing DDCD-CIDA runtimes with *Grow*-CIDA runtimes in 50n80s to 140n210s.

ILP test results are also plotted using a line chart and are presented in Figure 6.6. Figure 6.6 presents the trending of ILP test runtimes for both DDCD-ILP and *Grow*-ILP in test case networks from the 50n80s to 140n210s. As shown in the figure, all the ILP tests (maximum 804.79s) show much shorter runtimes (about 1/15) as compared to those of CIDA tests (maximum 12,473s). From 50n80s to 70n105, the DDCD-ILP test runtimes appeared less plausible than those of the *Grow*-ILP. However, starting from 80n128s, the DDCD-ILP runtimes dropped to much lower values (119.07s to 17.49s) and remained relatively steady at less than 50s regardless of the network size. However, the runtimes of *Grow*-ILP tests start to jump significantly since 80n128s. Therefore, it may be concluded that the DDCD-ILP approach is exceptionally robust in large-scale networks. The DDCD-ILP is the most preferred approach in solving the *p*-cycle SCA problem among *Grow*-ILP, DFS-ILP, *Grow*-CIDA, DFS-CIDA, and DDCD-CIDA.

Our experiments demonstrated that DDCD is the more robust cycle enumeration method than *Grow* or DFS. The DDCD-ILP combined algorithmic approach can provide 100% survivability with much lower spare capacity costs with much shorter runtimes than *Grow*-ILP and other methods used in this study, especially in large-scale networks.

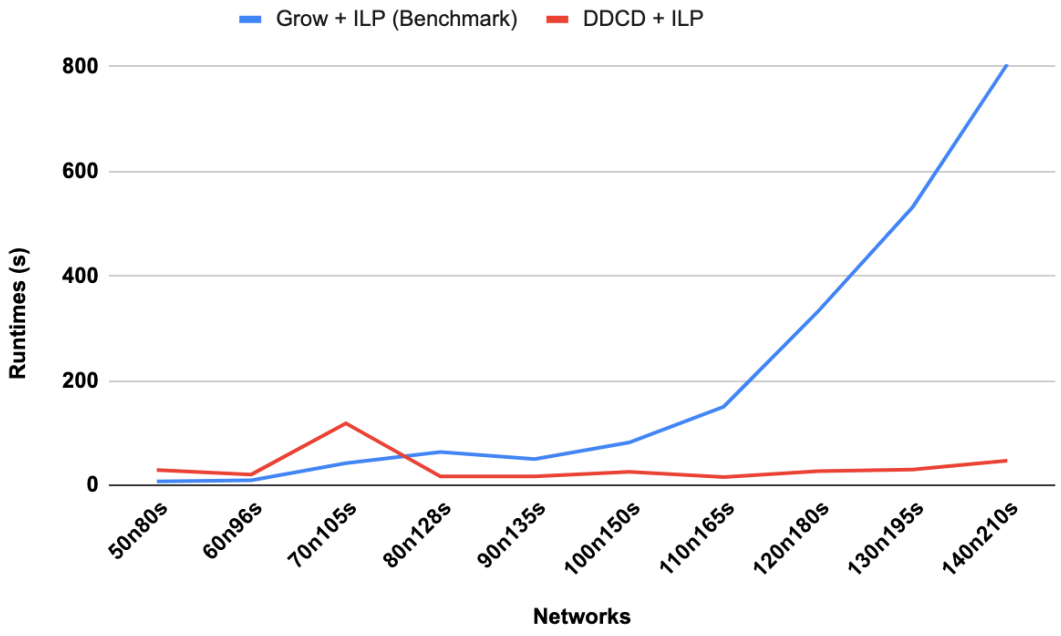


Figure 6.6 Comparing DDCD-ILP runtimes with *Grow*-ILP runtimes in 50n80s to 140n210s.

6.5 Conclusions

In this chapter, a novel heuristic algorithm for the *p*-cycle design was presented and discussed, which is the disjoint-paths Dijkstra cycle development (DDCD) algorithm. The DDCD is an iterative cycle development method capable of generating high-performance candidate *p*-cycles efficiently in small and large-scale networks. Various experiments were conducted, which demonstrated the DDCD algorithm's robust performance in solving the spare capacity allocation problem in large-scale networks compared to either *Grow* or DFS algorithm.

Based on the experimental results discussed in the previous sections, we can reach the following conclusions:

1) The DDCD algorithm outperforms either the *Grow* algorithm or conventional DFS algorithm in both small and large networks in terms of generating highly efficient candidate *p*-cycles;
2) The DDCD algorithm is most desirable for large-scale networks that are over 80 nodes, where the traditional spare capacity allocation ILP model fails to find an optimal solution, and the conventional DFS fails to provide a near-optimal solution within a satisfactory timeframe;
3) In terms of test runtimes with an increase of network sizes, the DDCD algorithm demonstrates much more robust results than the *Grow* algorithm in networks larger than 80 nodes when combined with either CIDA algorithm or the spare capacity allocation ILP model;
4) In large-scale networks over 80 nodes, DDCD-ILP is the best-performing combined algorithm in solving the *p*-cycle protection optimization problem with the best capacity costs and the least solution runtimes.

CHAPTER 7. GENETIC ALGORITHMS FOR *p*-CYCLE SPARE CAPACITY ALLOCATION

7.1 Introduction

Various real-world optimization problems in the telecommunication industry are considered *NP-hard*, which refer to complex problems that are unable to be solved and validated within polynomial time, such as network design problems, and network routing problems. Meta-heuristic methods like genetic algorithms (GAs) are preferred methods for handling highly complex *NP-hard* problems. GAs do not guarantee optimal results. However, as compared to ILP/LPs, GAs are exploration-focused search algorithms that can find near-optimal results more effectively and efficiently for the computational-intensive optimization problems.

As discussed in CHAPTER 4 of this thesis, GAs have been successfully implemented in various research and studies regarding network survivability and reliability [70]-[74]. GAs have also been adopted to solve *p*-cycle protection optimization problems like *p*-cycle selection problems and *p*-cycle placement problems [58], [83]-[84]. A *p*-cycle spare capacity allocation (SCA) problem optimizes the placement of a selective set of *p*-cycles from a given set of candidate *p*-cycles. Therefore, a well-designed GA model for SCA problems is expected to have the flexibility to be compatible with various *p*-cycle enumeration methods. A review of past literature, as presented in CHAPTER 4, indicated that there had not been a scalable GA model proposed for optimizing the *p*-cycle SCA problem with minimum allocation cost. An improved GA model was proposed in [83] that finds a combination of various candidate *p*-cycles with optimal protection ability to protect the entire network topology. This method is referred to as a *genetic p-cycle combination protection strategy* (GPCPS), which includes a working path routing sub-problem, and a protection path configuration sub-problem. The protection path configuration part of the GPCPS proposed an improved GA for allocating a combination of various *p*-cycles. However, the objective function of the GPCPS is to maximize the protection ability of a *p*-cycle combination and not to minimize the cost of allocating the cycles. Therefore, the GPCPS does not look at the cost-efficiency of placing the *p*-cycles, nor does it consider how demands may affect *p*-cycle protection effectiveness. Allocating proper candidate cycles with

minimum possible capital expenditure while fulfilling service demands are crucial factors to consider in the real-world telecommunication industry. Therefore, it is the primary purpose of this study to propose a scalable GA design to address these gaps.

This chapter will introduce a novel genetic algorithm design for optimizing the *p*-cycle spare capacity allocation optimization problem with minimal cycle allocation cost, which is referred to as a GA-SCA model. The proposed GA-SCA is designed to be used in conjunction with any *p*-cycle enumeration method, and it can be applied to any fully connected network topologies of any size. In addition to proposing a model, this study also focuses on designing and tuning problem-specific GA operators to enhance the effectiveness of this GA-SCA model.

The remainder of this chapter is organized as follows. Section 7.2 will analyze the problem formulation as well as design considerations of the GA-SCA. Section 7.3 will introduce the design of GA-SCA in extensive details, including the development of the case-specific chromosome representation and GA operators (selection, crossover, and mutation). The GA-SCA experimental set-up, network topologies, computational set-up and experiment overviews will be presented in Section 7.4. The experimental results and performance of the GA-SCA model will be presented in Section 7.5, where GA results will also be compared against those of the classic CIDA and ILP. Finally, Section 7.6 will wrap up this chapter with key research contributions and conclusions.

7.2 Statement of the Problem and Design Considerations

The formulation of a *p*-cycle spare capacity allocation (SCA) problem was discussed in CHAPTER 3 of this thesis. Let me start the GA implementation of this problem by reviewing the objective function, constraints, parameters, and variables of the SCA problem. In Section 3.4.3 of this thesis, all the relevant sets, parameters, and variables in a *p*-cycle SCA problem were presented. These will be considered again in this section when we design and implement the GA model. The set of eligible cycles (C) is an input to the GA model, which can be generated using any cycle enumeration method (e.g., the *Grow* algorithm, DFS algorithm, or the DDCD algorithm as proposed in the previous chapter). The information on the set of network spans (S)

will be collected based on the input network topology. Its data structure can be expressed as lists of on-cycle spans and straddling spans given the network topology. The relevant parameters (*ni, cj, xi,j, pi,j*) will be incorporated into the chromosome representation so that the chromosome encoding can best reflect the SCA problem formulation. *ni* is the number of copies of *p*-cycle *i* ($\forall i \in C$). *cj* is the cost or length of span *j* ($\forall j \in S$). *xi,j* is a binary parameter that equals 1 when *p*-cycle *i* traverses span *j and* zero otherwise ($\forall i \in C, \forall j \in S$). *pi,j* equals to 1 if span *j* is on *p*-cycle and is 2 if span *j* straddles *p*-cycle *i* ($\forall i \in C, \forall j \in S$). *pi,j* equals to zero if span *j* neither traverses nor straddles *p*-cycle *i*. There are two variables used in *the p*-cycle SCA formulation. The variable *wj* is not determined in the GA-SCA design because it is determined by the working capacity routing before the SCA. The working capacity routing is formulated using the shortest path routing and is not part of the GA-SCA design. Therefore, the *wj* will be treated as an input parameter in this problem. The *sj* is what the GA-SCA model optimizes and, therefore, is an output of the GA-SCA model in addition to the objective function value.

An objective function value reflects the quality of a feasible solution, thereby assessing the level of *fitness* of an individual in a population. Therefore, the objective function will be used while determining the fitness function for the GA-SCA model. The objective function of this GA-SCA problem is to minimize the total cost of placing the candidate *p*-cycles onto a network. As previously defined in the Eq. 3.7 in Section 3.4.3 of this thesis, the objective function is to minimize the function $\sum_{\forall j \in S} c_j \bullet s_j$.

7.2.1 Design Considerations

The GA-SCA model is developed based on the following key design considerations:

1) An appropriate chromosome encoding design is required for the GA-SCA, which should be accessible for calculating and assessing the fitness values of individuals.
2) The fitness function design must best reflect the problem's objective function.

3) Proper initiation of the first population must be proposed to allow access to the GA operators and to jump-start the performance of GA-SCA.
4) Choosing an appropriate selection mechanism, with sound selection pressure and proper elitism strategy, so that the most suitable individuals will be selected for subsequent reproduction processes.
5) Choosing a suitable crossover method for the chromosome encoding design and designing an effective repair mechanism for any disrupted chromosomes.
6) Designing a problem-specific mutation mechanism design that is can effectively improve the objective function value and can facilitate the exploration of solution space.
7) Determine an appropriate termination criterion for the GA-SCA.

7.3 Genetic Algorithm for *p*-Cycle Spare Capacity Allocation (GA-SCA)

This section will unfold the details regarding the novel GA-SCA design for solving the *p*-cycle spare capacity allocation problem. As emphasized previously, this algorithm is designed to minimize the cost of *p*-cycle spare capacity allocation while adequately protecting a network topology. Adopting the GA concept and schematic, the GA-SCA model is expected to generate offspring with progressively enhanced objective function values.

7.3.1 Chromosome Representation

The chromosome representation in a GA is a process of designing an artificial data structure to each chromosome, which facilitates critical information of a chromosome to be accessed and processed in a genetic algorithm model. An adequately encoded chromosome must also be accessible for fitness value calculation. The GA-SCA model in this study adopts a variable-length string encoding approach to design the chromosomes. Each chromosome represents an individual feasible solution to the objective function, which consists of a combination of various *p*-cycles that altogether protect the entire network topology. A schematic of the chromosome design (with locations of genes) is shown in Figure 7.1.

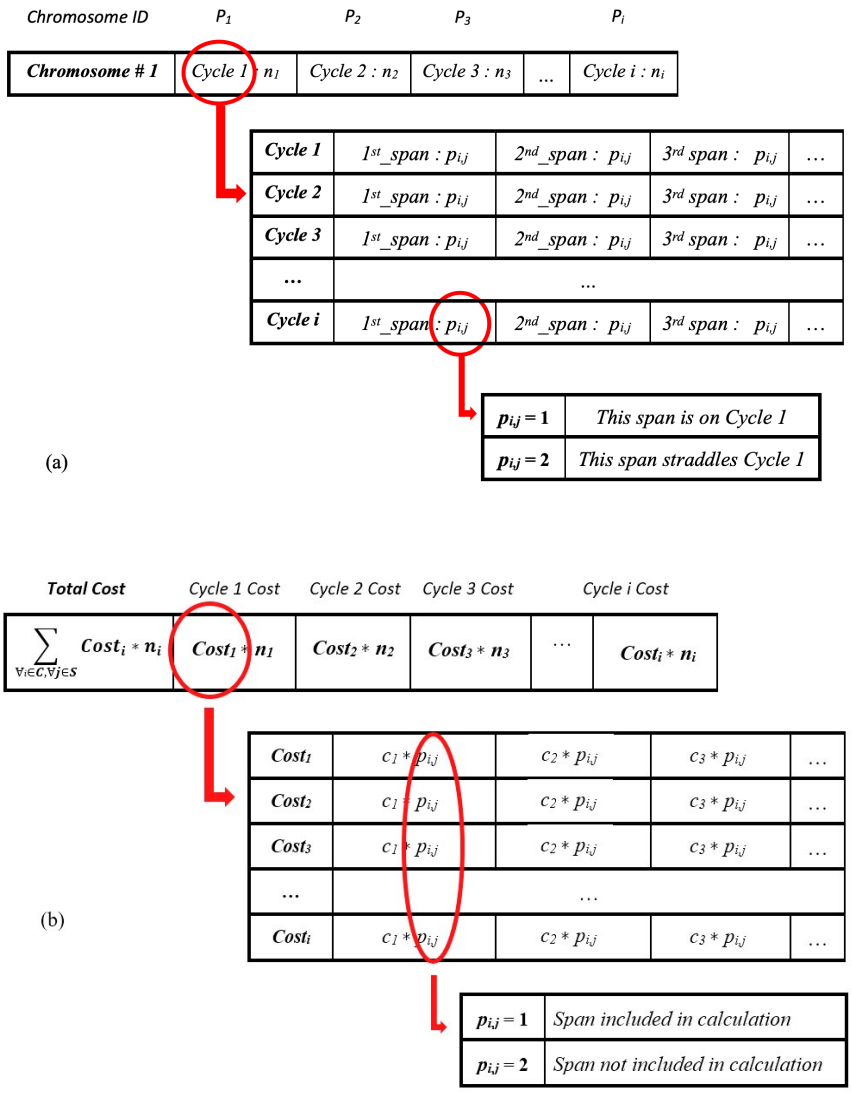


Figure 7.1 Chromosome encoding for GA-SCA. (a) shows a chromosome and (b) indicates input parameters.

Figure 7.1(a) shows a hierarchical view of the chromosome design and p_i indicates the location (locus) of each gene in a chromosome. Figure 7.1(b) illustrates how each parameter and variable is incorporated into the chromosome design. It shows how input parameters are applied in calculating the objective function values for each individual. The binary parameter $x_{i,j}$ is interpreted as whether a cycle is present in an individual's chromosome. If a cycle is not present in a chromosome ($x_{i,j}$ = 0), the cycle and its span information are excluded from the chromosome. Therefore, each chromosome does not necessarily carry all the span information of all the candidate cycles. As shown in Figure 7.1 (a), the cycles that are present in the

Chromosome #1 have $x_{i,j}$ values equal to one. The parameter n_i indicates the numbers of copies of corresponding candidate cycles used in a chromosome. The parameter $p_{i,j}$ is included as part of a cycle's span information, which indicates whether a span is on the cycle or straddles the cycle. Note that spans that are neither on a cycle or straddle a cycle ($p_{i,j} = 0$) are not present in the chromosomes. This detailed span information is reflected in the data structure of *p*-cycles and is carried over to a chromosome when a cycle is picked to form the chromosome. The c_i is used when calculating the cost of each chromosome (the objective function value), as illustrated in Figure 7.1 (b).

Pseudo-code for *p*-cycles and chromosomes can be illustrated as shown below:

```
Input_pcycle_set = {cycle_1 : {span_1 : 1, span_3 : 2, span_4 : 1, … , span_i
: 2}, cycle_2 : {span_1 : 2, span_2 : 1, span_5 : 1, span_7 : 2, … , span_i :
1}, … , cycle_n : {span_5 : 1, span_6 : 1 , span_9 : 2, … , span_i : 1} }

Chromosome1 = {cycle_1 : 5, cycle_8 : 22, cycle_9 : 10, …, cycle_n : 9}
```

Chromosome Decoding

Given the genetic code of a feasible solution, the genetic decoding of a solution chromosome will be the inverse operation of what was presented above.

7.3.2 Initial Population for GA-SCA

A group of various encoded chromosomes will form an initial population. There are two key elements to consider while initiating a GA population. Firstly, designing a suitable approach to generate an excellent initial population for the GA-SCA problem. Secondly, determining an appropriate population size for the GA-SCA problem.

Generating a good initial population is crucial for enhancing the performance of GA operations. As discussed in previous chapters, CIDA [39] is a robust *p*-cycle placement operation that selects and places a set of *p*-cycles that provide full network protection with near-

minimal spare capacity. Therefore, GA-SCA modifies the classic CIDA by adding a randomness factor, so it finds distinct relatively better-performing *p*-cycles at each execution. The randomized CIDA is referred to as a *CIDA_rand* algorithm in the pseudo-code below. In the classic CIDA, as presented in CHAPTER 3.5.2.2, the algorithm calculates current actual efficiency (*Ew*) scores for all cycles in a cycle pool, and it selects the best cycle with the highest *Ew* score. In the *CIDA_rand*, however, all cycles are assigned weight scores based on their *Ew* scores (the higher the *Ew* score, the higher weight). The cycles with higher weight scores will have better chances of being selected and vice versa. By doing so, *CIDA_rand* finds a different relatively good candidate cycle at each execution, which will eventually form a population for GA-SCA.

The pseudo-code for *CIDA_rand* and *inital_population* are as follows:

```
function CIDA_rand(CycleSet, i):
    initialize CycleEw[], work[], CycleUse[]
    while work[i] > 0 for all span i:
        CandCycle = 0
        for each cycle p in CycleSet:
            calculate Ew(p)
            save all cycle_id : cycle_ew → CycleEw
            rank CycleEw from high Ew to low Ew → assign weights
        pick p from CycleEw: higher weight more likely picked
        CandCycle = p
        if CandCycle not in CycleUse[]:
            add CandCycle to CycleUse[]
        else if CandCycle in CycleUse[]:
            CycleUse[CandCycle] = CycleUse[CandCycle] + 1
    for each on-cycle span i in CandCycle:
        work[i]= work[i] - 1
    for each straddling span i in CandCycle:
        work[i]= work[i] - 2
    return CycleUse
```

```
function initial_population(popSize, CycleSet, Spans):
    init_population = []
    for n in range(popSize):
        init_individual = CIDA_rand(CycleSet, Spans)
        if init_individual not in init_population:
            add init_population to init_pop_member
    return init_population
```

In terms of the population size for GA-SCA, various population sizes will be tested on the two calibration networks (*USA* Network and 30n45s network). The best-performing population size determined in the benchmark tests will be applied to other test case networks. It is expected that larger population sizes may increase computational runtimes; however, they can enhance a GA's search power by extending the search space. On the contrary, smaller population sizes will shorten the runtime with a higher likelihood of reaching premature convergence of the GA.

7.3.3 Fitness Function for GA-SCA

As discussed previously in CHAPTER 4, a suitable fitness function accurately evaluates the level of fitness of each individual in a population, based on its power to provide a favourable solution to a problem. It also must reflect the objective function of the problem. In the SCA problem, the objective function is to minimize the total cost of allocating the spare capacities across an entire network. Therefore, the fitness function for the GA-SCA problem is to calculate the total cost of allocating a particular set of *p*-cycles onto a network, which is defined as:

$$\sum_{\forall i \in C,\, \forall j \in S} \left(\sum c_i * p_{i,j} \right) * n_i \qquad \text{(Eq. 7.1)}$$

$\sum c_i * p_{i,j}$ calculates the cost of individual cycle i by summing up costs of all span j that traverse cycle i. This fitness function will be applied to all the individuals in a population, and the fitness values will be utilized in the GA selection operation.

7.3.4 Selection & Elitism for GA-SCA

Three selection methods are implemented and tested in this work: roulette wheel selection, tournament selection, and random selection. Preliminary tests on *USA* network indicate that the random selection is ineffective for the GA-SCA problem; therefore, only roulette wheel selection and tournament selection methods will be tested extensively on the calibration networks.

In the *roulette wheel selection* implemented in this work, the probability of an individual being selected is proportional to its fitness value as compared to the sum of all fitness values in the population. Inversion of the proportionate probabilities is used as the fitness score so that individuals with higher fineness value (higher cost) have lower probabilities to be selected. The fitness scores are then normalized and assigned to corresponding individuals. The fitness scores are then squared so that more weights are allocated to better-performing individuals. By doing so, the selection pressure of this selection method will be enhanced. As mentioned previously in CHAPTER 4, selection pressure affects how likely a better individual is favoured, and is a crucial factor that regulates the convergence rate of a GA.

For the *tournament selection* method in this study, two subset sizes are implemented to find out a more suitable selection pressure for the tournament selection. As discussed in [81], the selection pressure of a tournament selection is determined by its tournament subset size. A higher tournament subset size will increase the selection pressure, and vice versa. The first subset size applied in this work is a fixed number of ten chromosomes for any population size. Because a diverse set of population sizes are tested in this work (e.g., 250, 500, 750, and 1000), an adaptive tournament subset size is applied where 5% of population size is used as the size of a tournament subset. An adaptive tournament selection approach allows variation of tournament subset according to changes in chromosome sizes.

Finally, a *random selection* is to simply select an arbitrary chromosome from a population, where every individual has an equal opportunity to be picked for subsequent breeding. Interestingly, preliminary tests on the three selection methods in the GA-SCA model do not show satisfactory results when random selection is applied. Therefore, random selection will not be further discussed in this study or tested on any other test cases.

Elitism strategy is also implemented in the GA-SCA model, where the top-performing individuals are retained and carried over to the next generation. Details of this elitism strategy will be further explained in Section 7.3.8.

7.3.5 Crossover for GA-SCA

A pair of selected individuals (parent chromosomes) will go through crossover operation to generate a pair of offspring chromosomes. Two types of crossover operations are implemented and compared in the GA-SCA: one-point crossover, and two-point crossover.

In the one-point crossover, each parent chromosome is spliced at one arbitrary crossover point. This is followed by the two parent chromosomes exchanging portions of their genetic contents to form a pair of offspring chromosomes. Therefore, each offspring chromosome retains parts of each parent's genes. Figure 7.2 illustrates how the one-point crossover is applied in the GA-SCA problem. In a situation where duplicate cycles with different cycle usage values (n_i) occur, the one with higher n_i is retained. In contrast, the other one is removed from the offspring chromosome (as demonstrated in Figure 7.3). The two-point crossover is carried out in a similar approach, but with two crossover points on each parent chromosome.

Various crossover rates will be tested in this study, including 0.1, 0.2, 0.3, and 0.4, to find an optimal crossover rate for the GA-SCA problem. The crossover operation will result in infeasible chromosomes that will require a repair mechanism; therefore, increasing the crossover rate will extend the computational runtime drastically. Crossover rates higher than 0.4 are not preferred in this study due to extensively long runtime.

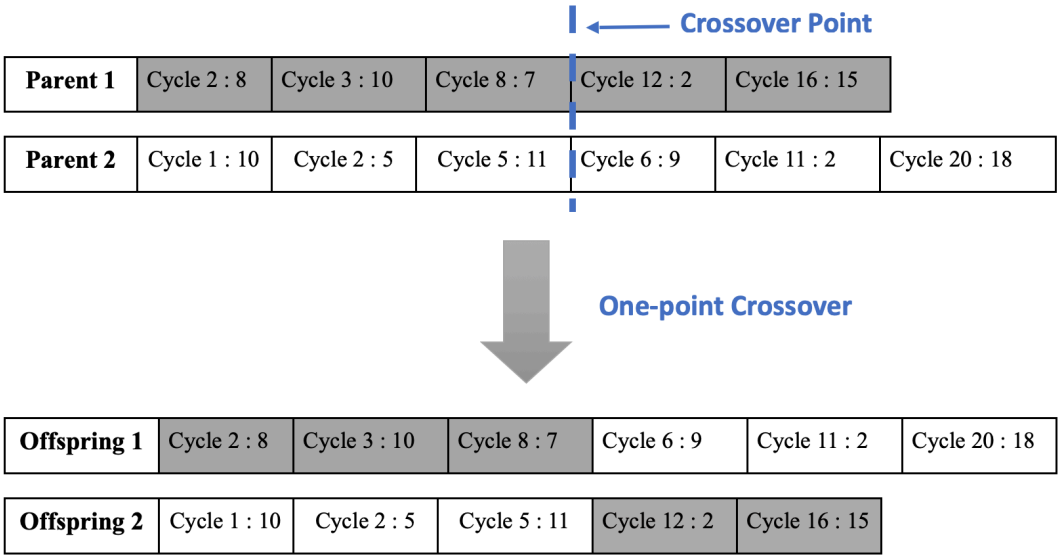


Figure 7.2 Implementation of one-point crossover in GA-SCA.

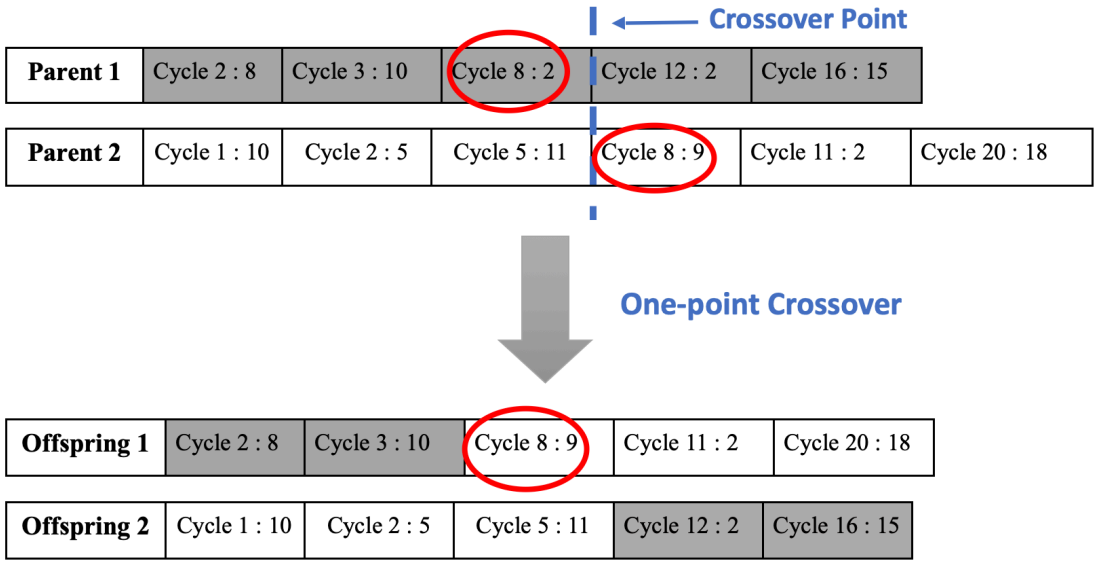


Figure 7.3 One-point crossover in GA-SCA where duplicate cycles occur in an offspring.

7.3.6 Mutation for GA-SCA

Another critical GA operator which will be extensively discussed in this study is the GA-SCA mutation operator. A mutation operation occurs on individual chromosomes in a population; it increases the diversity of a GA population and enhances GA's capability to find the global optimum. Various types of commonly used mutation operations were introduced in CHAPTER 4 of this thesis, which included one-point mutation, multi-point mutation, insertion/ deletion, and permutation mutation. In the case of a GA-SCA problem, a one-point mutation operator can be applied to mutate one of the member cycles in a chromosome. In contrast, a multi-point mutation can mutate multiple member cycles concurrently. A deletion can be implemented by completely removing a random member cycle from a chromosome. Any of these mutation operations will result in infeasible solutions (a disrupted chromosome) that must be fixed by a repair mechanism. A permutation mutation, however, will have no effect on the chromosomes in the GA-SCA problem and will not be considered in this study.

Several mutation approaches will be implemented and tested out in this GA-SCA study, which includes removing a random cycle, removing one or multiple lowest-performing cycles, removing a random cycle and add another random cycle, etc. Since these mutation approaches will result in disrupted chromosomes, therefore, problem-specific repair methods will be designed to repair the resultant chromosomes.

In addition to these conventional mutation operators, a novel problem-specific mutation operator is designed for the GA-SCA problem, which is referred to as *Cycle-Merging* mutation operator. The Cycle-Merging mutation operator is designed based on two considerations:

1) To ensure enhanced performance of the offspring over the parent chromosomes.

2) To avoid repairing the resultant chromosomes to shorten computational runtimes.

7.3.6.1 Problem-Specific Mutation Design Without Repairs - *Cycle-Merging*

To ensure enhanced operator performance and to avoid extensive runtime, a problem-specific mutation operator is designed for the GA-SCA model, which is referred to as the *cycle-merging* mutation operator. Compared to those commonly used mutation operators mentioned

previously, The cycle-merging mutation operator can benefit the GA-SCA problem in the following aspects: 1. cycle-merging fuses two candidate *p*-cycles to yield a larger but valid *p*-cycle that will not require any repair; 2. fusing two *p*-cycles ensures will eliminate one span, hence reduced cost and improved objective function value.

The cycle-merging mutation proceeds as follows: For all the genes (individual *p*-cycles and their numbers of copies) in a chromosome, the cycle-merging operator finds two cycles that share one and only one common span and merges them to form a new larger cycle. The common span is then removed and labelled as a straddling span to the new cycle. It is important to note that, other than the merged common span and its two end nodes, the two cycles must not share any other common spans or common nodes. As shown in Figure 7.4, cycle A-B-C-D-E and cycle A-B-K-H-G-F are two *p*-cycles on a 10n20s network topology that share one and only one span (span *i*). Upon applying the cycle-merging operator, the two cycles are joined at span *i* and a new cycle is formed. The common span *i* is then removed from the new cycle and is marked as a straddling span.

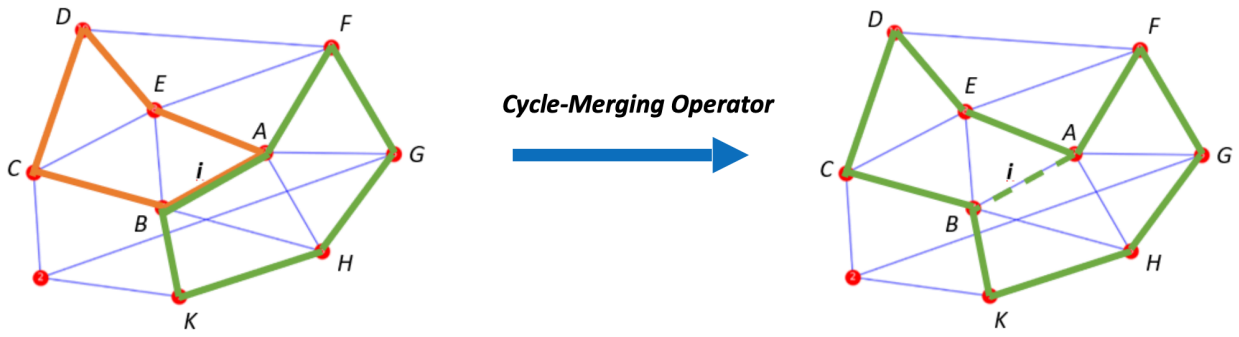


Figure 7.4 Illustration of the *cycle-merging* mutation operator for the GA-SCA on a 10n20s network.

Subsequently, the newly generated *p*-cycle is added to the chromosome, with the number of copies being the value of the two predecessor *p*-cycles with fewer copies. The two predecessor *p*-cycles are either completely removed or subtracted from the chromosome.

The cycle-merging mutation operator guarantees that the resultant new *p*-cycle is a feasible solution to the problem and that the overall cost of spare capacity placement is reduced (removal of a span reduces costs).

The pseudo-codes for the `Cycle_Merging` mutation operator is as follows:

```
function Cycle_Merging(input_genome, CycleSet):
    initiate newCycle[], newCycle_num[], mutated_ind[]
    new_individual = copy.input_genome
    new_id = 0
    for cycles in input_genome:
        pick Cycle1, Cycle2, where Cycle1 != Cycle2
        Cycle1_num = number of copies of Cycle1
        Cycle2_num = number of copies of Cycle2
        if Cycle1 and Cycle2 share one and only one span i:
            mark span i as straddling span
            assign new cycle id → Merged_CYL + str(new_id)
            assign all spans to new cycle → {Merged_CYL1:span_info}
            newCycle_num = min(Cycle1_num, Cycle2_num)
            add new cycle id and newCycle_num to newCycle[]
            updated and add Cycle1_num, Cycle2_num to newCycle[]
            new_id += 1
    mutated_ind = new_individual + newCycle
    return mutated_ind, newCycle
```

Several mutation rates will be tested in this study (0.05, 0.1, 0.2, 0.3, and 0.4) in order to find a suitable mutation rate for the GA-SCA problem. Since it is not common to find two cycles in a population that share one and only one span, the actual mutation operation is expected to occur on a chromosome at a lower rate than the applied mutation rate. Therefore, very low mutation rates (e.g., lower than 0.05) may not be beneficial to this problem and will not be considered in this study.

7.3.7 Repair Mechanism for GA-SCA

Repair mechanisms are crucial for fixing the infeasible solutions generated from crossover and mutation. As mentioned above, the one-point or two-point crossover operators and most mutation operators in this study will generate infeasible solutions that are unable to

provide full protection to the network topology. Therefore, the offspring chromosomes generated by these operators must be repaired to satisfy the desired protection efficiency while minimizing protection cost entirely.

One repair operator option for an SCA problem is CIDA. Previous chapters have discussed the effectiveness of CIDA in providing full network protection with near-minimal spare capacity. Therefore, CIDA can be used to repair an interrupted chromosome by complementing a set of cycles to protect any unprotected working capacities. However, CIDA calculates the capacity-weighted efficiencies iteratively, which will be runtime-inefficient in a GA problem with large population size and large numbers of generation. This assumption is tested and proved in preliminary tests using the *USA* network topology, where extensively long runtimes were observed due to repetitive calculation of current actual efficiency scores (E_w) with only minimal improvement on the objective function.

Apart from runtime-efficiency, another critical factor to consider while designing a suitable repair mechanism is its power to allow a GA to explore a globally optimum solution. For example, if repairing a chromosome can introduce new genes to a chromosome that does not previously exist in that chromosome. The repair operator helps expand the search space and enables the GA to explore an optimal solution globally.

To enhance the runtime-efficiency of a GA model and to boost the model's ability to explore an optimal solution globally, two problem-specific repair mechanisms are developed for the GA-SCA model. They are referred to as the *maximum-matching* repair method and the *minimum-cycle* repair method. The former one focuses on a faster way to generate results with as many straddling spans as possible (higher protection efficiency). In contrast, the latter focuses on a quick and easy way to repair with smallest possible local cycles (minimum costs).

7.3.7.1 Problem-Specific Repair Mechanism - *Maximum-Matching*

The *maximum-matching* repair mechanism starts by scanning a disrupted chromosome to collect information on all the unprotected working capacities (the spans they are located on, and the number of unprotected working capacity units). For the unprotected working capacities, maximum-matching will allocate a candidate cycle that matches the maximum possible

unprotected working capacities with the most straddling spans. If there are more than one *p*-cycles that offer the same level of protection (same amount of working capacities with the same number of straddling spans), the *p*-cycle with the least cost will be favoured. To accurately track these protection relations, two types of matrices are required: a `cycle-span relation matrix`, and a `cycle protection matrix`. The `cycle-span relation matrix` displays information regarding all the *p*-cycles from the cycle pool that can protect a particular span either as an on-cycle span or a straddling span. The `cycle protection matrix` indicates the number of straddling span protection and on-cycle span protection a particular *p*-cycle can offer. The two matrices are designed as follows:

Cycle-span relation matrix = {span_1 : [(cycle1, Ew[cycle1]),([cycle3, Ew[cycle3]),…, (cycle*p*, Ew[cycle*p*])], span_2 : [(cycle2, Ew[cycle2]), ([cycle4, Ew[cycle4]),…, (cycle*p'*, Ew[cycle*p'*])], …, span_*i* : [(cycle1, Ew[cycle1]),([cycle3, Ew[cycle3]),…, (cycle*p*, Ew[cycle*p*])]}

Cycle protection matrix = {cycle_1 : [cycle_1, straddle#, on-cycle#], cycle_2 : [cycle_2, straddle#, on-cycle#], …, cycle_*p* : [cycle_*p*, straddle#, on-cycle#]}

The `straddle#` indicates the number of straddling spans the corresponding cycle can protect, and `on-cycle#` refers to the number of on-cycle spans that the same cycle can protect.

When a *p*-cycle is placed onto the network, the network protection status is updated by subtracting one unit of each on-cycle span and two units of each straddling span (if present). This process continues iteratively until all the unprotected working capacities are fulfilled in such a way. Once the m*aximum-matching* function terminates, a complementary set of protection cycles are returned. The complementary cycles are subsequently combined with the input disrupted chromosome to form a repaired offspring chromosome.

The pseudo-codes for the *`Max-Matching`* function and the overall repair mechanism (`Repair_Genome`) are as follows:

```
function Max-Matching(CycleSet, i):
    initialize Complement_cyc[]
    max_straddle_count = 0
    min_cycle_cost = sys.maxsize
    for each unprotected span i:
        find cycle-span relation matrix for i → list(pcycles[i])
        for each pcycle in list(pcycles[i]):
            find corresponding cycle protection matrix →
                      pcycle[i] = [pcycle_id, straddle#, on-cycle#]
            if straddle# > max_straddle_count:
                max_straddle_count = straddle#
                min_cycle_cost = global_cost[pcycle_id]
                BestCycle = pcycle_id
            else if straddle# = max_straddle_count:
                if global_cost[pcycle_id] < min_cycle_cost:
                    max_straddle_count = straddle#
                    min_cycle_cost = global_cost[pcycle_id]
                    BestCycle = pcycle_id
        if BestCycle not in Complement_cyc:
            add BestCycle to Complement_cyc
        else if BestCycle in Complement_cyc:
            update BestCycle in Complement_cyc
    for each on-cycle span i in BestCycle:
        work[i]= work[i] - 1
    for each straddling span i in BestCycle:
        work[i]= work[i] - 2
    return Complement_cyc

function Repair_Genome(input_genome, Spans, CycleSet):
    initiate unprotected_spans[], protected_spans[]
    for gene in input_genome:
        collect protected_spans[]
        collect unprotected_spans[]
    complement_cycles = Max-Matching(CycleSet, unprotected_spans)
    repaired_child = input_genome + complement_cycles
    return repaired_child
```

7.3.7.2 Problem-Specific Repair Mechanism – *Minimum-Cycle*

The m*inimum-cycle* repair mechanism generates a minimum (smallest) protection cycle for each unprotected span identified. This repair process starts by identifying all the unprotected working capacities in a disrupted chromosome. Then, for each unprotected span, the *minimum-cycle* operator generates the shortest path that is node-disjoint from the unprotected span to form the smallest protection cycle for the span. If this minimum protection cycle is a duplicate cycle of an existing cycle in the chromosome, the cycle count for the existing cycle id is updated. If this is a new cycle, it will be added to the chromosome with a newly generated cycle id. This process will repeat iteratively until all spans are protected.

The pseudo-codes for the m*inimum-cycle* repair function is as follows:

```
function Min-Cycle(CycleSet, i, Graph):
    initialize Complement_cyc[]
    for each unprotected span i on Graph:
        Dijkstra(i, unmarked spans/nodes) → r1, distance(r1)
        if distance(r1) != MAX:
            add all the spans of r1 to span i → cycle a
            if cycle a not in Graph, add cycle a to Complement_cyc
            else: update BestCycle in CycleSet
        for each on-cycle span i in cycle a:
            work[i]= work[i] - 1
        for each straddling span i in cycle a:
            work[i]= work[i] - 2
    return Complement_cyc
```

7.3.8 Next Generation

The fittest will survive. At the end of a generation, the population will include original individuals (reserved for the purpose of elitism), as well as individuals generated from crossover and mutation operations, and the size of the population will enlarge to almost double of original population size. Therefore, all the individuals will be ranked based on their fitness values, and

the bottom low-performing individuals with low fitness values will be eliminated. The size of the population reduced to the original size. From generation to generation, only the top-performing individuals from predecessor generation will be retained and carried over to the next generation (an elitism strategy), which allows survival of the fittest individuals.

7.3.9 Termination Criteria for GA-SCA

Two termination criteria will be implemented in the GA-SCA model in this study:

1) Terminate when a predetermined fixed number of total generations has reached (e.g., terminate after the 1000th generation), or

2) Terminate when there is an unchanged objective function value for x iterations.

When any one of these two termination criteria is satisfied, the GA-SCA process will stop and return the best current solution.

7.4 Experimental Set-Up

The GA-SCA model takes a set of candidate p-cycles, a set of given demands, and a network topology to generate an optimized p-cycle combination and an updated set of p-cycles. The resulting p-cycle combination will provide full network protection to the given network topology with near-optimal minimal cost of allocating the spare capacities. The updated set of p-cycles excludes low-performance candidate cycles (removed by GA selection and elitism mechanism). It will include new p-cycles that are generated by the GA's reproduction mechanism. Eligible starting candidate p-cycles are generated before the GA-SCA and are brought into the GA-SCA model as input information. The candidate cycles can be generated in any method. In this study, both the *Grow* algorithm [39] and DDCD (developed in CHAPTER 6 of this thesis) are applied to generate different candidate p-cycle pools to test the GA-SCA model's consistency on different sets of cycle pools. Results of the GA-SCA will be compared to those of *Grow* + CIDA and DDCD + CIDA in the experimental results section of this chapter.

Extensive experiments will be conducted to explore various portions of the GA elements (initial population size, crossover method and repair, mutation method and repair, crossover and mutation rates, etc.). The study will also emphasize the development of an appropriate mutation method and proper repair mechanisms due to the significant impact of these elements. Once optimal GA operators are determined for the GA-SCA model, they will be applied on various test case networks with sizes ranging from 50 nodes to 100 nodes. Results of GA-SCA will be compared to those of CIDA and ILP.

The GA-SCA model is programmed using *Python 3.7.6* in the *Visual Studio Code*. AMPL is used to generate ILP *.mps* files using the ILP SCA model file (*.mod*) and specific data files (*.dat*), which are solved using IBM ILOG CPLEX Interactive Optimizer 12.6.1.0. All experiments are run on a server with 12-core ACPI multiprocessor X64-based PC with Intel Xeon® CPU E5-2430 running at 2.2 GHz with 96 GB RAM. All ILP test results include a default mipgap of 0.01% (unless specified otherwise).

7.4.1 Test Networks for GA-SCA

In this study, the *USA Long-Haul* network with 28 nodes and 45 spans (as shown in Figure 7.5 (a)) is used as the primary calibration network where an extensive number of tests are conducted to explore possible combinations of GA operators. As mentioned in previous chapters, a uniform span weight of one for *USA* network was used in [39]. We will continue adopting the same span weight in our experiments using the *USA* network to allow results comparison. To best represent the real-life examples where costs of network spans are usually different, a 30n45s network with 30 nodes and 45 spans (as illustrated in Figure 7.5 (b)) with weighted spans will be used as a secondary test network. The 30n45s network has span costs equal to their Euclidean distances and is tested using preferred GA operator combinations determined based on the *USA* test results. The most suitable option is subsequently used for tuning mutation operators for this problem. More extensive real case test networks will be used in this study to verify the suitability and scalability of the GA-SCA model, including the 50n80s, 60n96s, 70n105s, 80n128, 90n135s, 100n150s networks from CHAPTER 6 of this thesis. The topologies of these test case networks can be found in the Appendix A of this thesis.

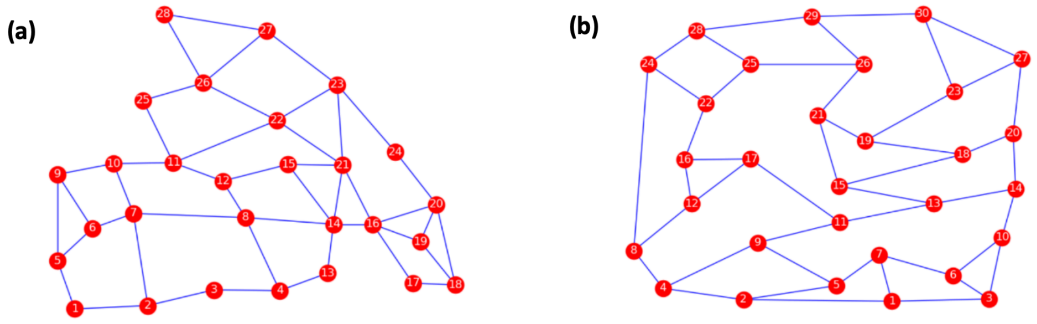


Figure 7.5 Test networks for GA-SCA: (a) *USA* network and (b) *30n45s* Network.

7.4.2 Overview of GA-SCA Tests for GA Operators Study

To obtain an optimal GA operator combination for the GA-SCA model, various tests are conducted on both the *USA* network and the 30n45s network. The GA operator combinations to be tested in this study include mixing and match of four selection methods (tournament selection, roulette wheel selection, random selection, and a customized adaptive tournament selection), three crossover methods (one-point crossover, two-point crossover, and uniform crossover), various initial population sizes, various crossover and mutation rates. Two termination criteria will be tested: a total of *X* generations, or an unchanged objective function value over *Y* generations (where *X, Y* are predetermined fixed number). The tests can be summarized into two stages: the first stage of tests is conducted on the *USA* network, and the second stage of tests is conducted on the 30n45s network. Note that a consistent mutation method is applied across all sets of tests here, which is the cycle-merging mutation method. Various mutation methods will be extensively tested at a separate study later.

A summary of GA-SCA test parameters that will initiate this study is as follows:

- Initial population sizes: 100, 200, 400, 600, 800, 1000
- Total number of generations (termination condition): 100, 200, 500, 1000
- Selection Methods: tournament selection(S1), roulette wheel selection (S1), random selection (S3), adaptive tournament selection (S4)
- Crossover Methods: one-point (C1), two-point (C2), uniform (C3)
- Crossover & Mutation Rate: 0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5

7.4.3 Overview of GA-SCA Mutation Operators Study

The study of suitable mutation operators for a GA problem has a profound impact on the performance of a model. A crossover operation primarily focuses on the exploitation of local optima, whereas a mutation operation empowers the global exploration of a search space. To study the best-performing mutation operators in the GA-SCA model, a set of best-performing combination of GA operators will be applied, which were discovered from the previous GA-SCA tests for GA operators tuning. The mutation tests are carried out with consistent initial population size, selection method, crossover method (and designated repair mechanism for a crossover), crossover rate, and termination criteria. These factors are considered input parameters for the mutation tests. Five different mutation approaches will be tested in this study (which will be denoted as M1, M1', M2, M3, M4, and M5). Details regarding these approaches are as follows:

- In M1, randomly select a member cycle from a chromosome (consists of n copies of various member cycles) and remove all copies of this cycle from the chromosome.
- In M1', randomly select a member cycle from a chromosome and only remove one copy of this cycle from the chromosome (n-1 copies remaining).
- In M2, the problem-specific cycle-merging mutation operator is applied.
- In M3, randomly select a member cycle from a chromosome, remove only one copy of this cycle from the chromosome and add one copy of a new cycle from a global p-cycle set that does not present in the chromosome.
- In M4, rank and find out the bottom 10% of worst-performing member cycles from a chromosome, remove one copy of each of these worst cycles. Then, add one copy of 3 new cycles from the global p-cycle set that does not present in the chromosome.
- In M5, randomly select a member cycle from a chromosome and remove all the copies of this cycle from the chromosome. Add one copy of a new cycle from a global p-cycle set that does not present in the chromosome.

Three repair mechanisms will be tested for repairing mutated chromosomes, which are the minimum-cycle repair (R1), maximum-matching repair (R2), and CIDA (R3). Mutation rates in the tests will range from 0.05 to 0.4.

7.5 Experimental Results and Discussion

7.5.1 GA Operators Study

Figure 7.6 depicts an overview of experimental designs and various preliminary tests for tuning the GA operators in the GA-SCA problem, mentioned briefly in Section 7.4.2. Among all selection methods, the random selection method was eliminated first due to poor performance. Adaptive tournament mutation allows variation of tournament subset according to changes in chromosome sizes. It was introduced to study how a dynamic, non-fixed tournament subset size will perform in a problem with dynamic chromosome sizes compared to a fixed small tournament size. However, test results indicate that a fixed small tournament size has better performance (better objective function value) than a dynamic tournament size. Crossover and mutation rates of 0.01, 0.05 and 0.5 are eliminated due to poor performance. A crossover rate of 0.35 (between 0.3 and 0.4) was added to the tests. For the next sets of tests, all possible combinations of selection and crossover methods are tested. The best-performing combination is found to be the tournament selection coupled with a two-point crossover. This operator pair is carried over to the next round of tests with various initial population sizes (500, 750, 1000), crossover rate (0.2, 0.3, 0.35, 0.4) and mutation rate (0.1, 0.2, 0.3, 0.4). Our case-specific cycle-merging method was used as the mutation operator for all these tests; therefore, no repair mechanism is needed at this testing phase.

Based on the results obtained from our preliminary and second round of tests (see Figure 7.6), re-test and second re-test are conducted on the *USA* network. These re-tests indicated that a larger population size (750 or 1000) is beneficial for generating better results because it enhances a GA's search power. A relatively higher crossover rate (0.35) and a relatively lower mutation rate (0.1 or 0.2) are preferred to allow extensive local search and sporadic global search. As a result, the following parameters are selected for the next sets of tests: initial population size of 750, a crossover rate of 0.35 and a mutation rate of 0.1. A population size of 1000 provides similar results on the *USA* network, but with longer runtime.

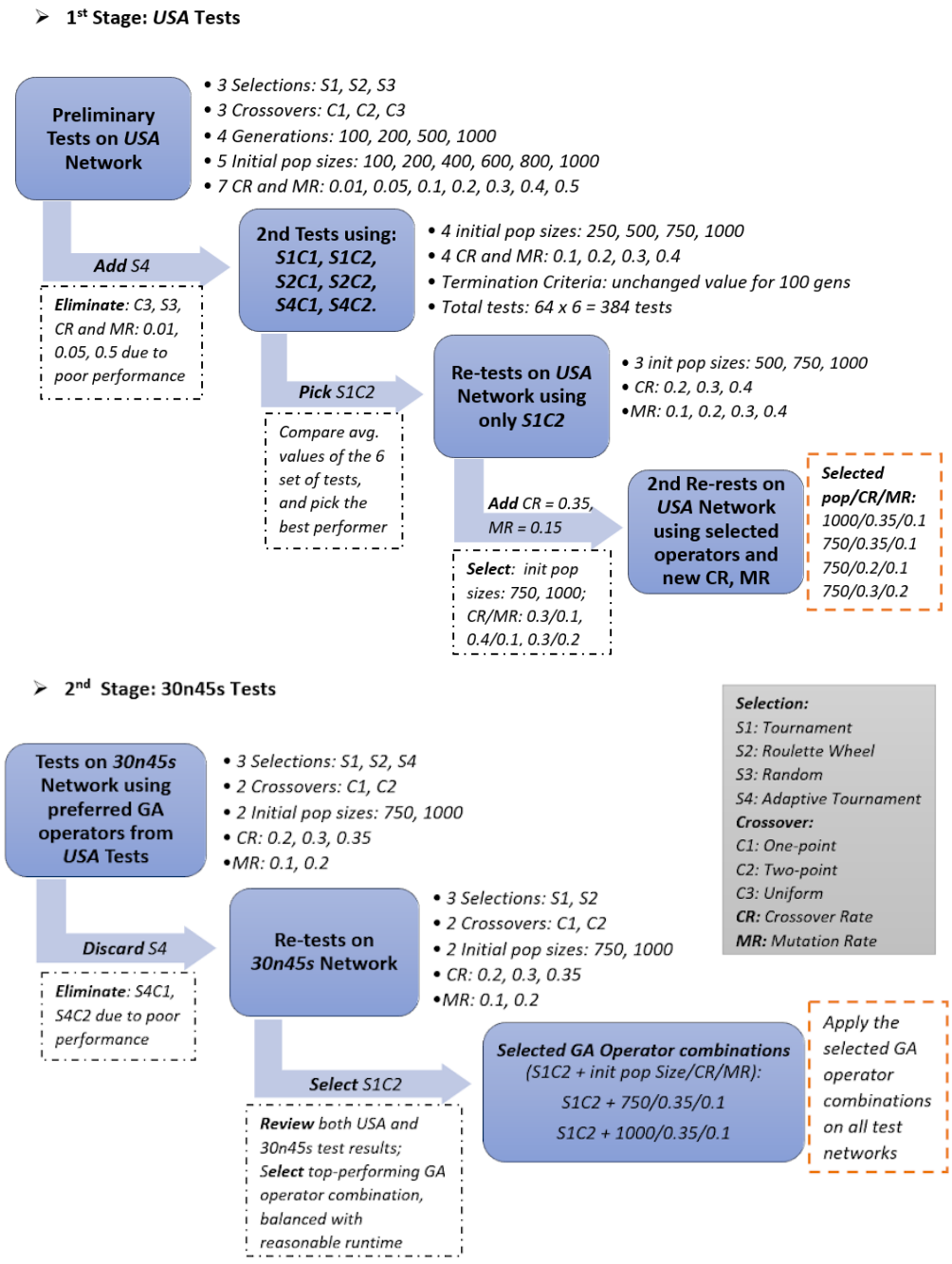


Figure 7.6 Overview of GA-SCA testing process on the *USA* and *30n45s* networks for operator tuning.

7.5.2 GA-SCA Mutation Operators Study

Based on the tests completed at the previous phase (Section 7.5.1), the following input parameters and GA operators are selected for the GA-SCA mutation operators study (conducted as outlined previously in Section 7.4.2):

- Initial population size: 750
- Selection method: tournament selection
- Crossover method: two-point crossover
- Mutation methods: M1, M1', M2, M3, M4, and M5 (details see Section 7.4.3)
- Crossover rate: 0.35
- Repair for crossover: maximum-matching repair
- Repair for mutation: minimum-cycle repair (M2 does not require any repairs)
- Termination criteria: total 1000 generations, or unchanged values for 60 generations

Figure 7.7 outlines the experimental process for the mutation operators tests in the GA-SCA problem using various mutation methods introduced previously in Section 7.4.2. Among all mutation methods, removing all copies of a random cycle showed more mediocre performance (M5 and M1') than those that only remove one copy of a cycle (M3 and M1'). Our case-specific cycle-merging method (M2) demonstrated to have the most robust performance than any other mutation operator in this study. A smaller mutation rate, 0.1 or 0,2, appeared to provide more satisfactory results with relatively shorter runtimes.

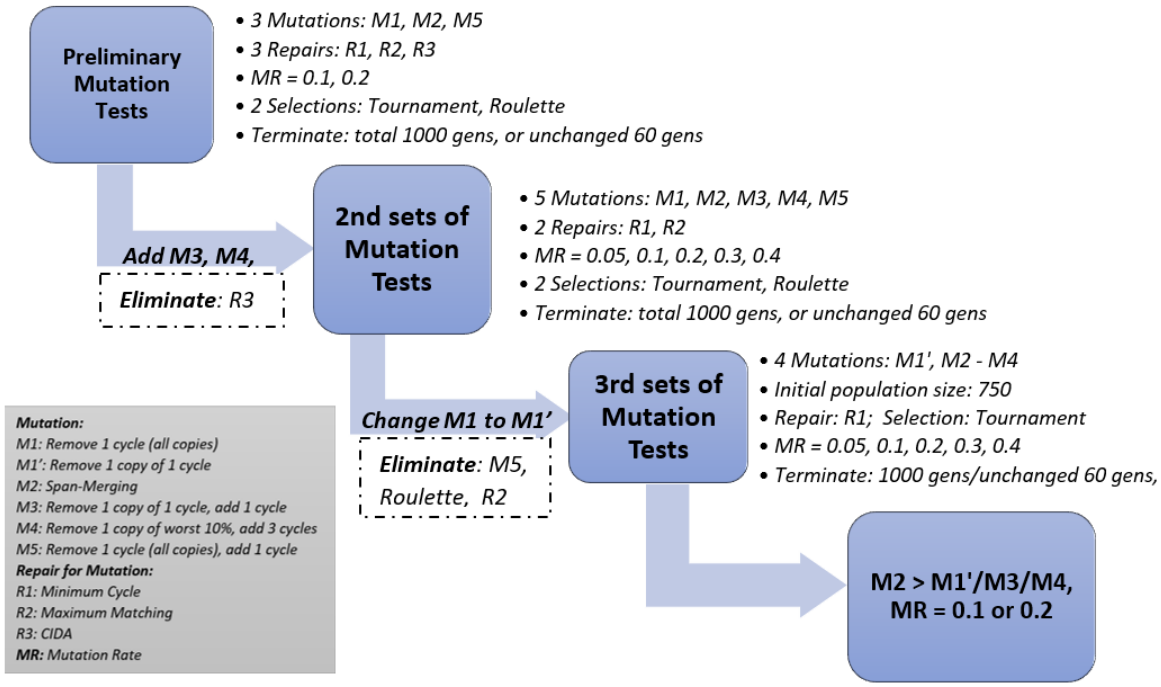


Figure 7.7 Overview of GA-SCA mutation operator testing process on *USA* Network.

Test results of the 3rd set of mutation tests using various mutation operators (M1', M2, M3, M4) and two different repair methods (R1 and R2) and various mutation rates (MR = 0.05–0.4) are presented in Table 7.1. A larger initial population of 1500 was included in the tests. The average value of each set of tests (denoted in the table as "Avg.") is calculated and compared to show each mutation method's average quality. The lower the result, the better the quality. As indicated in Table 7.1, all test results (bolded fonts) by the cycle-merging mutation (M2), regardless of mutation rate, appear to be better than those of other mutation methods. When comparing population sizes or 750 vs. 1500, it is evident that a larger population size generates relatively better results with compromised runtimes (at least 1.5 times more runtime). When comparing the performance of repair mechanisms, with the same initial population of 750, results of minimum-cycle repair (R1) are generally better than those of maximum-matching repair (R2) in the majority of the tests (except in the case of M2, which does not require any repair mechanism). Therefore, the minimum-cycle repair is generally a more suitable repair mechanism for a mutated individual. Small protection cycles generated from minimum cycle repair mechanism introduce fewer costs to an individual, hence better objective function value.

Table 7.1 GA-SCA mutation operators tests on *USA* Network using various repairs and initial population size.

Mutation	Mutation Rate	Repair: Max-Match Init.pop = 750		Repair: Min.Cycle Init.pop = 750		Repair: Max-Match Init.pop = 1500	
		Results	RT (s)	Results	RT (s)	Results	RT (s)
M1'	0.05	1206	1966.51	1198	5554.98	1200	7920.27
M1'	0.1	1197	4722.53	1200	4549.22	1201	7647.39
M1'	0.2	1203	3419.90	1203	2577.60	1199	11402.89
M1'	0.3	1200	2978.68	1200	3942.75	1203	5559.47
M1'	0.4	1202	3726.08	1201	3620.35	1200	7874.34
	Avg.	1201.6		1200.4		1200.6	
M2	0.05	1147	4793.15	1146	3436.32	1146	7377.16
M2	0.1	1146	3311.13	1147	3611.29	1145	19520.31
M2	0.2	1143	3310.56	1147	3474.66	1141	25136.73
M2	0.3	1146	3494.67	1149	2967.55	1145	19746.20
M2	0.4	1146	2853.67	1147	3052.44	1146	29206.82
	Avg.	**1145.6**		**1147.2**		**1144.6**	
M3	0.05	1206	2138.23	1204	2907.57	1198	12252.75
M3	0.1	1206	3872.17	1199	6497.77	1200	6129.27
M3	0.2	1206	2158.60	1196	4653.70	1198	9883.58
M3	0.3	1205	3015.07	1199	6183.84	1205	3556.57
M3	0.4	1201	4149.55	1202	5542.53	1196	8911.75
	Avg.	1204.8		1200.0		1199.4	
M4	0.05	1206	2363.49	1203	3883.44	1205	3644.17
M4	0.1	1200	4479.34	1201	3090.22	1201	6134.17
M4	0.2	1202	2889.05	1202	2610.04	1203	4465.00
M4	0.3	1206	2356.05	1204	3867.60	1205	3701.20
M4	0.4	1206	2485.10	1206	2359.13	1203	4799.06
	Avg.	1204.0		1203.2		1203.4	

7.5.3 *Cycle-Merging* Mutation: Benefits, Shortfalls, and Mitigation Plan

Previous tests have indicated a robust performance of the *cycle-merging* mutation operator compared to any other mutation operator. This strong performance is because whenever the cycle-merging operator is applied on a chromosome and a pair of valid cycles are found, and a guaranteed cost reduction will occur (by removing the cost of the common span). Therefore, the resultant offspring will be guaranteed to perform better (less cost) than the

parent chromosomes. A larger initial population size (750 vs. 200) and a higher mutation rate (0.2 vs. 0.05) will expedite this process and bring about more robust improvements on the objective function value. Besides, because the cycle-merging operator has been proved to generate an offspring that is a valid *p*-cycle, this operation does not require additional repair. Filling two needs with one deed. Efficiency. Efficiency. Efficiency.

In order to investigate the effect of the cycle-merging mutation on the objective function value during the process of GA-SCA (track and trend the results from generation to generation), we conducted a set of follow-up GA-SCA tests on large test case networks ranging from 50 nodes to 140 nodes. These tests use DDCD (developed from work completed in CHAPTER 6 of this thesis) to generate candidate cycles. Results of DDCD + CIDA for each test case network are used as a benchmark for the DDCD + GA-SCA results. The preferred GA operators from the previous *USA* and 30n45s benchmark tests are applied, including tournament selection, two-point crossover, an initial population of 750, a crossover rate of 0.35, and a mutation rate of 0.1. Results of these follow-up tests are presented below in Table 7.2. "Normalized Costs" for DDCD + CIDA and columns 1-4 are calculated by dividing each cost value by the minimum cost across the same row. A normalized cost of *1.000* indicates the best objective function value.

Table 7.2 Results of GA-SCA tests on large networks of 50n80s to 140n210s using DDCD to enumerate cycles.

Network	Normalized Costs of *DDCD+CIDA*	1	2	3	4	Δ (1 - 2)%	Δ (2 - 3)%	Δ (3 - 4)%
		Starting Population	Generation *G10*	Generation *G50*	Final Results	Start to *G10*	*G10* to *G50*	*G50* to Final
50n80s	1.022	1.022	1.017	1.014	1.000	0.507%	0.243%	1.395%
60n96s	1.019	1.020	1.008	1.008	1.000	1.150%	0.070%	0.770%
70n105s	1.027	1.026	1.012	1.012	1.000	1.288%	0.021%	1.206%
80n128s	1.071	1.023	1.003	1.002	1.000	1.922%	0.127%	0.199%
90n135s	1.020	1.020	1.002	1.002	1.000	1.755%	0.050%	0.170%
100n150s	1.020	1.020	1.005	1.004	1.000	1.432%	0.175%	0.365%
110n165s	1.027	1.027	1.000	1.000	1.000	2.673%	0.000%	0.000%
120n180s	1.014	1.015	1.000	1.000	1.000	1.461%	0.004%	0.000%
130n195s	1.029	1.029	1.000	1.000	1.000	2.812%	0.001%	0.000%
140n210s	1.007	1.007	1.001	1.001	1.000	0.641%	0.000%	0.069%

Upon completing these tests, we collected results at the end of the 10th (column 2) and the 50th generation (column 3) and compared them to the start (column 1) and final (column 4) objective function values. By doing so, we are able to track and trend the progress of the objective function values as the GA progresses. In Table 7.2, column *1* shows the objective function values of starting populations before applying any GA operators, and columns 2 and 3 present the results at the end of the 10th generation (dented as *G10*) and 50th generation (denoted as *G50*) for all test cases. Column 4 shows the final results after the termination criteria is satisfied in all test cases (unchanged values for 60 consecutive generations). For all test case networks presented in Table 7.2, *Δ (1 - 2)* calculates the percentage of changes in the objective function values from the starting population to the end of the 10th generation. *Δ (1 - 2)* shows the changes from the 10th to the 50th generations. Finally, *Δ (3 - 4)* records the changes from the 50th generation onwards until the termination criteria are satisfied.

As shown in Column 2 of Table 7.2, the objective function values for all test case networks appear to have surpassed those of DDCD + CIDA by the end of the 10th generation, which indicates that the cycle-merging operator can jump start improvements on the objective function values at early phase. Some smaller test cases show continuous improvements after the 50th generation, whereas most large-scale networks had no more improvement after the 50th generation (e.g., 110n165s, 120n180s, 130n195s). All test cases except the 50n80s experienced the most percentage of increase from start to the 10th generation, which slowed down significantly after that.

These observations indicate that the cycle-merging operator provides a sharp increase on the performance; however, the effectiveness of this mutation operator plateaus or declines as GA-SCA progresses. The cycle-merging process will always result in a new cycle that is almost double the size of the two constituent cycles; therefore, this mutation process will rarely introduce small new *p*-cycles. Lack of introduction of a broader variety of new cycles may hinder the GA's ability to explore global optimum thoroughly.

To utilize the robust performance of cycle-merging in the early stage of a GA process and to mitigate the shortfall of insufficient global exploration, a *coupled* mutation operator approach will be introduced and tested. A coupled mutation operator approach is to apply a conventional

mutation method first (e.g., one-point mutation, multi-point mutation, or gene insertion/ deletion) up to *X* number of generations where no changes on the objective function values happen. Then, apply the cycle-merging mutation operator to "boost" the GA performance. The entire GA process will terminate when no more improvement on the objective function values occur for *X* number of generations up to a pre-defined total number of generations.

7.5.4 The *Coupled* Mutation Operator for GA-SCA

The coupled mutation operators for our GA-SCA model is implemented by modifying the termination criteria of the previous test version. Instead of terminating the GA-SCA process entirely when an unchanged objective function value is observed for 60 consecutive generations, the coupled mutation operator approach will apply the cycle-merging mutation operator on the population to boost GA performance. Suppose the objective function is still not improving for the next 60 consecutive generations. In that case, the new GA-SCA model will then terminate and return the best current objective function value as the final result.

New tests are conducted on the *USA* network to test the performance of this coupled mutation operator approach. Three different sets of coupled mutation operators will be tested (M1'+M2, M3+M2, and M4+M2). Various mutation rates, ranging between 0.05 and 0.4, will be applied. The other GA operators remain the same as in previous tests, which include the tournament selection, two-point crossover, an initial population of 750, a crossover rate of 0.35, the maximum-matching repair for crossover and the minimum-cycle repair for mutation. Details regarding these approaches are as follows:

- **M1'+M2**: Randomly select a member cycle from a chromosome and remove one copy of this cycle from the chromosome (*M1'*). If the objective function value does not improve for 60 generations, apply the cycle-merging method (*M2*).
- **M3+M2**: Randomly select a member cycle from a chromosome, remove one copy of this cycle from the chromosome and add one copy of a new cycle from a global *p*-cycle set that does not present in the chromosome (*M3*). If the objective function value does not improve for 60 generations, apply the cycle-merging method (*M2*).

- **M4+M2**: Rank and find out the bottom 10% of worst-performing member cycles from a chromosome, remove one copy of each of these worst cycles. Then, add one copy of 3 new cycles from the global *p*-cycle set that does not present in the chromosome (*M4*). If the objective function value does not improve for 60 generations, apply the cycle-merging method (*M2*).

Results of the coupled mutation operator tests for the *USA* network are presented below in Table 7.3. We conducted two repeats ("Test #1" and "Test #2" as shown in the table) using the same testing conditions and GA operators to guide us in reaching a more reliable conclusion. The best result in each set of tests is highlighted in blue using bolded fonts. The test results are recorded under "Results" and test runtimes are converted to *minutes* and are recorded under "RT(min)". In these two sets of tests, we did not calculate an average value for any test set. This is because an average value in this case may not accurately indicate a preferred coupled mutation operator or properly evaluate the robustness of performance for any coupled mutation operators.

Table 7.3 Results of GA-SCA tests using coupled mutation operators on the *USA* network.

Mutation	Mutation Rate	Test #1		Test #2	
		Results	RT (min)	Results	RT (min)
M1'+M2	0.05	1137	179.28	1145	108.60
M1'+M2	0.1	1145	152.70	1130	160.71
M1'+M2	0.2	1147	129.96	1102	249.84
M1'+M2	0.3	1146	141.85	1107	246.81
M1'+M2	0.4	1145	163.80	**1093**	241.54
M3+M2	0.05	1150	121.50	1147	101.53
M3+M2	0.1	1149	131.30	1147	97.58
M3+M2	0.2	1143	172.06	1149	100.96
M3+M2	0.3	1139	231.47	1142	194.06
M3+M2	0.4	1136	312.93	1147	141.96
M4+M2	0.05	1144	88.23	1140	80.12
M4+M2	0.1	1133	76.71	1126	178.84
M4+M2	0.2	1151	69.24	1110	150.10
M4+M2	0.3	**1099**	225.75	1094	232.79
M4+M2	0.4	1148	131.32	1151	135.65

As shown in Table 7.3, “M4+M2” provided the best objective function value of 1099 with a mutation rate of 0.3 in Test #1. However, in Test #2 with the same test conditions and parameters, “M1'+M2” generated the best objective function value of 1093 using a mutation rate of 0.4. These observations suggest that a GA model’s performance depends on the correlation among various GA operators and factors. Adjusting the mutation method or the mutation rate alone may not effectively obtain the best near-optimal solution. More extensive tests using simulation techniques like the *Monte Carlo Simulation* [97] may be needed to find a more accurate combination of mutation operator, mutation rate, and other GA operators with the most robust performance.

Comparing the “M1'+M2” from Table 7.3 with the “M2” (cycle-merging alone) from Table 7.1 shows that the former approach generates a near-optimal result as good as 1093 with an initial population of 750 and repair mechanism of minimum-cycle. In contrast, the latter approach generated the best value of 1146 using the same GA operator conditions. Therefore, it is apparent that a one-point mutation (M1') combined with the cycle-merging mutation (M2) generates significantly better objective function values than the cycle-merging mutation alone. Besides, the coupled mutation approach avoids the disadvantage of applying cycle-merging alone, where the mutation operator’s global search power gradually diminishes after the first few generations.

Previous approaches using CIDA and ILP combined with different cycle enumeration methods (*Grow* and CCDC), in CHAPTER 6 of this thesis, generated a near-optimal solution of 1112 by the DDCD + ILP approach for the *USA* network. Compared to the solution of DDCD + ILP, some of the GA-SCA attempts using coupled mutation operators and various mutation rates surpassed the performance of DDCD + ILP (e.g., 1099 and 1093). Therefore, the coupled mutation operator approach has the potential to provide better solutions for the SCA problem than the DDCD + ILP approach. However, total runtimes for a GA-SCA approach is significantly longer (almost 100 times longer) than either the CIDA or the ILP approach.

7.5.5 Recommended GA Operators for GA-SCA on Large Networks

Based on the previous test results and discussion from Section 7.5.1 to Section 7.5.4, we will recommend the following GA operators for the final tests on large-scale networks vary from 50n80s to 100n150s. Due to considerations on extensive runtimes for these GA tests, a small initial population of 50 will be used.

Other recommended GA operators for large-scale test cases are as following:

- Selection Method: tournament selection
- Crossover Method: two-point crossover
- Repair for Crossover: maximum-matching approach
- Mutation Method: remove one copy of one random cycle + cycle-merging
- Repair for Mutation: minimum-cycle approach
- Termination Criteria:
    1. Terminate when a maximum of 1000 generation is reached, or
    2. Apply cycle-merging operator when there is an unchanged objective function value for 60 consecutive generations. After applying cycle-merging, if there is still no improvement on the objective function value over the next 60 consecutive generations, terminate the GA process and return the best current value.

7.5.6 GA-SCA vs. CIDA vs. ILP on Large-Scale Networks

Several large-scale test networks are tested using the GA-SCA model and refined GA operators, as proposed in the Section 7.5.5 of this thesis. The test case networks used in this study include 50n80s, 60n96s, 70n105s, 80n128, 90n135s and 100n150s topologies. A smaller initial population size of 50 will be used in these tests to lessen the total runtimes. A crossover rate of 0.35 and a mutation rate of 0.2 will be applied to these tests.

Results of the GA-SCA are compared to those using the CIDA algorithm and the ILP model. Results of the tests using the *Grow* algorithm to generate candidate cycles are presented in Table 7.4, and those using DDCD are presented in Table 7.5. “Normalized Costs” in both tables are calculated by dividing each cost value by the minimum cost across the same row. A

normalized cost of *1.000* indicates the best value among all three methods."# Gens" indicates the total number of generations the GA-SCA model is applied to a population before the termination criteria are satisfied. It can show the suitability of a GA model for a test case. The more suitable a GA model is, the further it can progress in exploring the search space for an optimal solution. The runtime for each test is closely associated with the "# Gens Explored" because the more generations a GA explores in a test case before the termination criteria are satisfied, the longer the runtime may be. The runtimes in these tests are converted to *minutes* and are reported under "RT(min)" in both Table 7.4 and 7.5. A total runtime that surpasses five days is considered in these experiments and is denoted by "*> 5 days*" on both tables.

Table 7.4 GA-SCA with cycles generated by *Grow* [39] on large networks ranging from 50n80s to 100n150s.

	***Grow* + CIDA**		***Grow* + ILP**		***Grow* + GA-SCA**		
	Normalized Costs	**RT (min)**	**Normalized Costs**	**RT (min)**	**Normalized Costs**	**RT (min)**	**# Gens**
50n80s	1.035	1.48	1.000	0.13	1.015	712.30	1000
60n96s	1.027	3.75	1.000	0.17	1.000	1215.79	607
70n105s	1.040	3.88	1.000	0.71	1.037	1180.81	1000
80n128s	1.033	26.90	1.001	1.07	1.000	*> 5 days*	1000
90n135s	1.026	29.91	1.000	0.84	1.024	7082.99	1000
100n150s	1.036	34.86	1.000	1.17	1.025	*> 5 days*	1000

Table 7.5 GA-SCA with cycles generated by DDCD on large networks ranging from 50n80s to 100n150s.

	DDCD + CIDA		**DDCD + ILP**		**DDCD + GA-SCA**		
	Normalized Costs	**RT (min)**	**Normalized Costs**	**RT (min)**	**Normalized Costs**	**RT (min)**	**# Gens**
50n80s	1.054	3.51	1.000	0.50	1.073	2193.66	836
60n96s	1.042	3.11	1.000	0.35	1.028	1298.44	1000
70n105s	1.051	6.28	1.000	1.98	1.029	2927.62	581
80n128s	1.084	2.89	1.040	0.29	1.000	*> 5 days*	1000
90n135s	1.051	2.69	1.000	0.29	1.040	1858.34	1000
100n150s	1.041	0.21	1.002	0.21	1.000	3317.32	1000

As reported in Table 7.4 and Table 7.5, GA-SCA progressed through the entire 1000 generations in most test cases network topologies, indicating that the carefully designed GA operators were suitable for this problem, preventing premature convergence of this GA model. The DDCD + GA-SCA or *Grow* + GA-SCA results found better solutions to this problem than those of DDCD + CIDA and *Grow* + CIDA, even with a small population size of 50. However, the results did not compete with those of DDCD + ILP and *Grow* + ILP in most test cases (except for the 80n128s network) regardless of the method used to enumerate candidate *p*-cycles. The most concerning aspect associated with these GA tests is the extensively long runtimes. As shown in both tables, the runtimes for some GA-SCA approaches are pressingly more expensive than those of either CIDA or ILP. For example, in the case of 90n135s network using *p*-cycles generated by *Grow,* the runtime of the GA-SCA (7082.99 mins) was almost 8000 times longer than that of ILPs (0.84 mins).

Using GA-SCA to find near-optimal solutions that are better than ILPs may be achieved by adopting a larger population size, which allows the GA-SCA model to explore the search space more extensively and more effectively before reaching termination criteria. Adopting a simulation technique like *Monte Carlo Simulation* using the GA-SCA model may also find the best near-optimal solutions. However, either of these two approaches may compromise the runtime further, which may discourage using genetic algorithms for this type of problem.

7.6 Conclusions

In this chapter, a scalable GA model for optimizing the *p*-cycle spare capacity allocation problem was proposed, which was referred to as a GA-SCA model. The proposed GA-SCA has been demonstrated through extensive testing in this study that the GA-SCA model applies to any fully connected network topologies of any size, and it can be combined with any *p*-cycle enumeration method.

Based on the experimental results presented in CHAPTER 7.5, the following conclusions can be drawn:

1) The GA-SCA model provides a better optimized spare capacity allocation solution than that of CIDA. In all benchmark and large test case scenarios, GA-SCA consistently shows better performance than *Grow* + CIDA and DDCD + CIDA.
2) GA operators that are specially designed to suit the problem can enhance the performance of a GA model. In the GA-SCA model, *maximum-matching* and *minimum-cycle* are two case-specific, objective function-focused repair mechanisms developed to facilitate improvements of the objective function values.
3) Mutation operator design is pivotal for enhancing the performance of a GA model. In the case of GA-SCA, a random one-point mutation is paired up with a specially designed *cycle-merging* mutation approach, which generated a synergistic effect that significantly enhanced the GA's power to explore optimal solutions globally.
4) Although the GA-SCA is highly adaptive and flexible and has the potentials to provide high-quality near-optimal solutions, it's exceedingly long runtimes may discourage adopting this method for studying *p*-cycle spare capacity allocation problems. Especially when compared to those of DDCD + ILP, which provided better and faster near-optimal solutions to the *p*-cycle spare capacity allocation problems.

CHAPTER 8. CONCLUSION AND DISCUSSION

8.1 Summary of Thesis

The primary research objective of this thesis is to study heuristic and meta-heuristic methods in solving the *p*-cycle spare capacity allocation problem for large-scale network topologies. An SCA is a two-step approach to solving *p*-cycle network protection problems. In an SCA problem, the working routing is conducted first using a preferred shortest path method, followed by protection path routing using a spare capacity allocation approach. In this case, a set of eligible candidate *p*-cycles are enumerated before spare capacity allocation. In this thesis research, two algorithmic approaches were taken to achieve this research objective: First, a heuristic *p*-cycle enumeration algorithm was developed to generate efficient candidate cycles. This algorithm focused on optimizing the cycle enumeration part of the SCA problem; Then, a novel genetic algorithm model was developed for the *p*-cycle spare capacity allocation problem. This GA approach pitched into optimizing the protection path routing part of the SCA problem. Genetic algorithms have been implemented in solving various network survivability problems, including *p*-cycle protection problems. However, to the best of our knowledge, there has not been a problem specific GA model designed and tested for solving the SCA problems, nor has there been a problem-specific repair or mutation mechanism proposed. This thesis research addressed these gaps and provided extensive experiments on various test case networks.

CHAPTER 2 of this thesis provided fundamental background knowledge and key concepts in transport networks and graph theory. CHAPTER 3 introduced fundamental *p*-cycles concepts, metrics, and designs. It also discussed relevant past research and studies on survivable network designs and optimization using *p*-cycles. Fundamental genetic algorithm concepts, genetic operator designs, and applications of genetic algorithm in *p*-cycle network survivability problems were introduced in CHAPTER 4 of this thesis. These three chapters of this thesis provided a solid foundation for developing later chapters.

In CHAPTER 6 of this thesis, a novel heuristic *p*-cycle enumeration algorithm called the disjoint-paths Dijkstra cycle development (DDCD) algorithm was introduced, developed and

discussed. It is an iterative cycle development method based on Dijkstra's algorithm and has been proved to enumerate high-performance candidate *p*-cycles in small and large networks compared to prior studies. The design was first tested on two calibration networks (the *USA* network and the France Network) and was later applied on network topologies of various sizes ranging from 10 nodes to 140 nodes. Experimental results showed that the DDCD algorithm outperformed either the *Grow* algorithm or the conventional DFS algorithm in small and large networks. It is exceptionally favourable in large networks of over 80 nodes, where conventional ILP and DFS methods fail to obtain an optimal or near-optimal solution within satisfactory runtime. DDCD algorithm outperforms the *Grow* algorithm in large networks with shorter or similar runtimes as *Grow* + CIDA and *Grow* + ILP. Lack of scalability may hinder the ILP approach in solving large-scale network optimization problems; however, DDCD algorithm can help ILP avoid the obstacles by producing highly efficient candidate cycles that significantly lessen the search time.

In CHAPTER 7, a case-specific GA model for solving the SCA problem using *p*-cycles, referred to as a GA-SCA model, was developed to optimize the *p*-cycle spare capacity allocation while minimizing cycle allocation cost. This chapter introduced and discussed problem-specific chromosome designs and GA operator designs, followed by extensive testing on defining the most suitable GA operator using two calibration network topologies (the *USA* network and the 30n45s network). Preferred GA operators are used on various large-scale network topologies, including 50n80s, 60n96s, 70n105s, 80n128, 90n135s, and 100n150s networks. In terms of refining GA operators, this chapter particularly emphasized the process of developing a suitable mutation operator. Extensive experiments were conducted on various mutation operator designs. Experimental results showed that the final GA-SCA model provided better optimized spare capacity allocation solutions than those of CIDA. Also, problem-specific GA operators were found to enhance GA performance in all test cases. Despite the better performance of GA-SCA in all test cases as compared to that of CIDA, it's exceedingly long runtimes compared to DDCD-ILP may discourage adopting this method for studying *p*-cycle spare capacity allocation problems.

8.2 Research Contributions

The primary research contributions of this thesis are the two algorithmic approaches developed to solve the *p*-cycle spare capacity allocation (SCA) problem for large-scale network topologies. These contributions are summarized as follows:

1. CHAPTER 6: Conducted extensive literature review of previous work on solving *p*-cycle SCA problems and proposed a novel and state-of-the-art heuristic algorithm for enumerating highly efficient candidate *p*-cycles, which was demonstrated to outperform either the CIDA algorithm or conventional DFS algorithm.
2. CHAPTER 7: Developed a novel genetic algorithm model for the *p*-Cycle SCA problem, which optimized the allocation of candidate *p*-cycles to fully protect a network topology with a minimum cost of capacity allocation. Our contribution to this work also included designing suitable GA operators for the *p*-cycle SCA problem, which allows better local exploitation and global exploration of this GA model.

Besides the main research contributions mentioned above, one conference paper regarding the DDCD heuristic *p*-cycle enumeration method is ready to be submitted. The RNDM 2020 conference was cancelled due to the global pandemic; therefore, the thesis will be re-submitted later time 2021.

T. Shi, T. Nakashima-Paniagua, J. Doucette, “A Heuristic Method for *p*-cycle Design in Survivable WDM Mesh Network,” Resilient Network Design and Modeling (RNDM 2020), to be submitted: May 2021.

References

[1] N. Mohan, A. Wason, and P. S. Sandhu, “Trends in survivability techniques of optical networks,” *International Journal of Computer Science and Electronics Engineering (IJCSEE),* vol. 1, no. 2, pp. 352-355, 2013.

[2] Telecommunications Industry Association, “2016-2020 ICT market review and forecast,” *Telecommunications Industry Association*, 2017. [Online]. Available: http://tr8.tiaonline.org/resources/tias-2016-2020-ict-market-review-and-forecast. [Accessed: Nov. 20, 2019].

[3] W. D. Grover, *Mesh-based Survivable Networks: Options and Strategies for Optical, MPLS, SONET, and ATM Networking*, Prentice Hall PTR, Upper Saddle River, NJ, 2004.

[4] Amazon AWS, “Summary of the Amazon EC2 and Amazon RDS service disruption in the US east region,” *aws.amazon.com*, 2011. [Online]. Available: https://aws.amazon.com/message/65648/. [Accessed: Sep. 30, 2020].

[5] D. A. Schupke, “Guaranteeing service availability in optical network design,” *ITG Symposium on Photonic Networks*, pp. 1-3, Leipzig, Germany, May 2007.

[6] The International Working Group on Cloud Computing Resiliency (IWGCR), "Downtime statistics of current cloud solutions,” *IWGCR*, March 2014. [Online]. Available: http://iwgcr.org/wp-content/uploads/2014/03/downtime-statistics-current-1.3.pdf. [Accessed: Oct. 28, 2020].

[7] J. Doucette, “Advances on design and analysis of mesh-restorable networks,” Ph.D. Thesis, University of Alberta, Edmonton, Alberta, Canada, 2005.

[8] W. D. Grover, “Self-organizing broad-band transport networks,” *Proceedings of the IEEE*, vol. 85, no. 10, pp. 1582-1611, October 1997.

[9] R. Bhandari, *Survivable Networks: Algorithms for Diverse Routing*, Kluwer Academic Publishers, November 1998.

[10] G. D. Morley, “Analysis and design of ring-based transport networks,” Ph.D. Thesis, University of Alberta, Edmonton, Alberta, Canada, February 2001.

[11] M. Herzberg and S. Bye, “An optimal spare-capacity assignment model for survivable networks with hop limits,” *1994 IEEE GLOBECOM. Communications: The Global Bridge*, San Francisco, CA, USA, vol. 3, pp. 1601-1606, November-December 1994.

[12] R. R. Iraschko, M. H. MacGregor, and W. D. Grover, “Optimal capacity placement for path restoration in STM or ATM mesh-survivable networks,” *IEEE/ACM Transactions on Networking*, vol. 6, no. 3, pp. 325-336, June 1998.

[13] S. Sengupta and R. Ramamurthy, “Capacity efficient distributed routing of mesh-restored lightpaths in optical networks,” *IEEE Global Telecommunications Conference (GlobeCom 2001)*, San Antonio, TX, pp. 2129-2133, November 2001.

[14] D. Stamatelakis and W. D. Grover, "Theoretical underpinnings for the efficiency of restorable networks using preconfigured cycles ("*p*-cycles")", *IEEE Transactions on Communications,* vol. 48, no. 8, pp. 1262-1265, August 2000.

[15] W. D. Grover and D. Stamatelakis, "Cycle-oriented distributed preconfiguration: ring-like speed with mesh-like capacity for self-planning network restoration", *Proceedings of IEEE International Conference on Communications (ICC 1998)*, Atlanta, GA, pp. 537-543, June 1998.

[16] W. D. Grover and D. Stamatelakis, "Bridging the ring-mesh dichotomy with *p*-cycles", *Proceedings of IEEE/VDE Workshop on Design of Reliable Communication Networks (DRCN 2000)*, pp. 92-104, April 2000.

[17] J. Doucette, "Advances on design and analysis of mesh-restorable networks," Ph.D. Thesis, University of Alberta, Edmonton, Alberta, Canada, 2005.

[18] J. Clausen, "Branch and bound algorithms-principles and examples," Department of Computer Science, University of Copenhagen, pp. 1-30, March 1999.

[19] T. W. Knowles, *Management Science: Building and Using Models*. Homewood, IL: IRWIN, 1989.

[20] A. Kasam, "Heuristic approaches for survivable network optimization," Ph.D. Thesis, University of Alberta, Edmonton, Alberta, Canada, 2015.

[21] C. G. Han, "Survivable Networks," in *Encyclopedia of Optimization*, C. Floudas and P. Pardalos, Eds. Boston, MA: Springer, 2001, pp- 431-434.

[22] S.V. Kartalopoulos, *Introduction to DWDM Technology: Data In a Rainbow*. Bellingham, WA: Wiley-IEEE Press, December 1999.

[23] K. Murakami and H. S. Kim, "Comparative study on restoration schemes of survivable ATM networks," in *Proceedings of IEEE Conference on Computer Communications (INFOCOM 1997)*, vol. 1, Kobe, Japan, 7-12 April 1997, pp. 345-352.

[24] Wu, Tsong-Ho. *Fiber Network Service Survivability*. Boston: Artech House, 1992.

[25] W. Wang, "Network design and availability analysis for large-scale mesh networks," Ph.D. Thesis, University of Alberta, Edmonton, Alberta, Canada, 2018.

[26] R. Asthana, Y. N. Singh, and W. D. Grover, "*p*-Cycles: an overview", in *Communications Surveys & Tutorials IEEE*, vol. 12, no. 1, pp. 97-111, First Quarter 2010.

[27] M. S. Kiaei, C. Assi, and B. Jaumard, "A survey on the *p*-cycle protection method", in *Communications Surveys & Tutorials IEEE*, vol. 11, no. 3, pp. 53-70, 3rd Quarter 2009.

[28] B. Wu, K. L. Yeung, and P. -H. Ho, "A comparative study of fast protection schemes in WDM mesh networks," *2008 IEEE International Conference on Communications*, Beijing, 2008, pp. 5160-5164.

[29] Y. Wei, K. Xu, H. Zhao, and G. Shen, "Applying *p*-cycle technique to elastic optical networks". *2014 International Conference on Optical Network Design and Modeling*, Stockholm, 19-22 May 2014, pp. 1-6.

[30] E. W. Dijkstra, “A note on two problems in connection with graphs”, *Numerische Mathematik 1,* pp. 269 - 271, 1959.

[31] D. Jungnickel, *Graphs, Networks, and Algorithms*. Berlin, Germany: Springer, 2008.

[32] H. Zhang and O. Yang, “Finding protection cycles in DWDM networks”, in *Proceedings of IEEE International Conference on Communications (ICC 2002)*, April-May 2002, vol. 5, pp. 2756-2760.

[33] C. Liu and L. Ruan, “Finding good candidate cycles for efficient *p*-cycle network design,” *in Proc. 13th International Conference on Computer Communications and Networks (IEEE Cat. No.04EX969)*, Chicago, IL, 2004, pp. 321-326.

[34] C. Liu and L. Ruan, “*p*-Cycle design in survivable WDM networks with shared risk link groups (SRLGs),” in *Proc. 5th International Workshop on Design of Reliable Communication Networks*, Ischia Island, Naples, Italy, October 2005, pp. 207-212.

[35] H. Drid, B. Cousin, S. Lahoud, and M. Molnar, “Multi-criteria *p*-cycle network design,” *33rd IEEE Conference on Local Computer Networks (LCN)*, Montreal, Quebec, 2008, pp. 361-366.

[36] M. Ju, F. Zhou, Z. Zhu, and S. Xiao, “*p*-Cycle design without candidate cycle enumeration in mixed-line-rate optical networks,” *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*, Budapest, Hungary, 2015, pp. 1-6.

[37] D. A. Schupke, C. G. Gruber, and A. Autenrieth, “Optimal configuration of *p*-cycles in WDM networks", in *Proceedings of IEEE International Conference on Communications (ICC)*, April-May 2002, vol. 5, pp. 2761-2765.

[38] T. H. Cormen, C. E. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms, Second Edition*. Cambridge, MA: McGraw-Hill Higher Education, 2001.

[39] J. Doucette, D. He, W. D. Grover, and O. Yang, “Algorithmic approaches for efficient enumeration of candidate *p*-cycles and capacitated *p*-cycle network design,” in *Proceedings of the 4th International Workshop on Design of Reliable Communication Networks (DRCN 2003)*, Banff, AB, Canada, 19-22 October 2003, pp. 212-220.

[40] Z. Zhang, W-D Zhong, and B. Mukherjee, “A heuristic method for design of survivable WDM networks with *p*-cycles,” *Communications Letters IEEE*, vol. 8, no. 7, pp. 467-469, 2004.

[41] L. S. Lasdon, *Optimization Theory for Large Systems*, Mineola, New York, USA: Dover Publications, Inc., 2002.

[42] K. Lo, D. Habibi, A. Rassan, Q. V. Phung, H. N. Nguyen, and B. Kang, "A hybrid *p*-cycle search algorithm for protection in WDM mesh networks," *2006 14th IEEE International Conference on Networks*, Singapore, 2006, pp. 1-6.

[43] K. Lo, D. Habibi, A. Rassan, Q. V. Phung, and H. N. Nguyen, "Heuristic *p*-cycle selection design in survivable WDM mesh networks," *2006 14th IEEE International Conference on Networks*, Singapore, 2006, pp. 1-6.

[44] M. Herzberg, S. J. Bye, and A. Utano, “The hop-limit approach for spare-capacity assignment in survivable networks,” *IEEE/ACM Transactions on Networking*, vol. 3, no. 6, pp. 775- 784, December 1995.

[45] P. Zhang, J. Li, P. Luo, J. Zhang, L. Zheng, and W. Gu, “Novel heuristic algorithms of candidate *p*-cycles in mesh WDM networks,” in *Proceedings of SPIE Vol. 6354, Network Architectures, Management, and Applications IV*, September 2006.

[46] W. D. Grover and J. E. Doucette, “Advances in optical network design with p-cycles: joint optimization and pre-selection of candidate p-cycles,” in *Proc. of the IEEE-LEOS Summer Topical Meeting on All Optical Networking*, Mont Tremblant, Quebec, Canada, July 2002, pp. WA2-49-WA2-50.

[47] A. Kodian, A. Sack, and W. D. Grover, “The threshold hop-limit effect in *p*-cycles: comparing hop- and circumference-limited design,” *Optical Switching and Networking*, vol. 2, no. 2, pp. 72–85, 2005.

[48] B. Kang, D. Habibi, K. Lo, Q. V. Phung, H. N. Nguyen, and A. Rassau, “An approach to generate an efficient set of candidate *p*-cycles in WDM mesh networks,” *2006 Asia-Pacific Conference on Communications*, Busan, 2006, pp. 1-5.

[49] K. Lo, D. Habibi, Q. V. Phung, A. Rassau, and H. N. Nguyen, “Efficient *p*-cycles design by heuristic *p*-cycle selection and refinement for survivable WDM Mesh Networks,” *IEEE Globecom 2006*, San Francisco, CA, 2006, pp. 1-5.

[50] T. Zhao, L. Li, H. Yu, and X. Zhang, “A heuristic method for optimal capacity design of WDM networks with *p*-cycles,” *Optical Transmission, Switching, and Subsystems IV*, 2006.

[51] A. Smutnicki and K. Walkowiak, “A heuristic approach to working and spare capacity optimization for survivable anycast streaming protected by *p*-cycles,” *Telecommunication Systems*, vol. 56, no. 1, pp. 141–156, 2013.

[52] A. Konak, “Two-edge disjoint survivable network design problem with relays: a hybrid genetic algorithm and Lagrangian heuristic approach,” *Engineering Optimization*, vol. 46, no. 1, pp. 130–145, 2013.

[53] E. T. L. Pastor, H. A. F. Crispim, H. Abdalla, A. F. D. Rocha, A. J. M. Soares, and J. Prat, “A new heuristics/GA-based algorithm for the management of the S-DRWA in IP/WDM networks,” *Managing Next Generation Networks and Services Lecture Notes in Computer Science*, pp. 265–275, 2007.

[54] O. Bozorg-Haddad, M. Solgi, and H. A. Loaiciga, *Meta-Heuristic and Evolutionary Algorithms for Engineering Optimization*. Hoboken, New Jersey: Wiley Blackwell, 2017.

[55] S. A. Fernandez, A. A. Juan, J. D. A. Adrian, D. G. E. Silva, and D. R. Terren, “Metaheuristics in telecommunication systems: network design, routing, and allocation problems,” *IEEE Systems Journal*, vol. 12, no. 4, pp. 3948–3957, 2018.

[56] P. D. Choudhury, S. Bhadra, and T. De, “A brief review of protection based routing and spectrum assignment in elastic optical networks and a novel *p*-cycle based protection approach for multicast traffic demands,” *Optical Switching and Networking*, vol. 32,

pp. 67–79, 2019.

[57] A.S. Deshpande and R. B. Kelkar, “Advanced genetic operators and techniques: an analysis of dominance & diploidy, reordering operator in genetic search,” *9th WSEAS International Conference on Evolutionary Computing (EC’08)*, Sofia, Bulgaria, May 2-4 2008, pp. 27-33.

[58] J. V. Paez, A. Talia, E. D. Gimenez, and D. P. Roa, “Optimal selection of *p*-cycles on WDM optical networks with shared risk link group independent restorability using genetic algorithm,” *IEEE Latin America Transactions*, vol. 10, no. 1, pp. 1385–1390, 2012.

[59] T. Cai, S. Huang, X. Li, S. Yin, J. Zhang, and W. Gu, “Dynamic survivable mapping algorithm based on ant colony optimization in IP over WDM networks,” *Acta Photonica Sinica*, vol. 41, no. 12, pp. 1400–1404, 2012.

[60] E. Kaldirim, F. C. Ergin, S. Uyar, and A. Yayimli, “Ant colony optimization for survivable virtual topology mapping in optical WDM networks,” *2009 24th International Symposium on Computer and Information Sciences*, pp. 334–339, 2009.

[61] T. Weise, “Global Optimization Algorithms - Theory and Application,” Second. Self-Published, 2009. [Online]. Available: . [Accessed: Jan. 18, 2020].

[62] A. Kodian and W.D. Grover, “Failure-independent path-protecting pcycles: efficient and simple fully preconnected optical-path protection,” *J. Lightwave Technology, vol. 23,* no. 10, pp. 3241–3259, 2005

[63] G. Shen, and W.D. Grover, “Extending the *p*-cycle concept to path segment protection for span and node failure recovery,” *IEEE J. Sel. Areas Commun.*, vol. 21, pp. 1306-1319, 2003

[64] B. Jaumard, H. Li, and S. Sebbah, “Design of path-segment-protecting *p*-cycles in survivable WDM mesh networks,” *2010 14th International Telecommunications Network Strategy and Planning Symposium (NETWORKS)*, 2010.

[65] B. Jaumard and H. Li, “Segment *p*-cycle design with full node protection in WDM mesh networks,” *2011 18th IEEE Workshop on Local & Metropolitan Area Networks (LANMAN)*, 2011.

[66] B. Jaumard and H. Li, “Minimum CAPEX design of segment *p*-cycles with full node protection”. *16th International Conference on Optical Networking Design and Modelling*,2012.

[67] J. Doucette, P. A. Giese, and W. D. Grover, “Combined node and span protection strategies with node-encircling *p*-cycles,” in *Proc. 5th International Workshop on Design of Reliable Communication Networks,* Ischia Island, Naples, Italy, October 2005, pp. 213–221.

[68] C. C. Meixner, L. Campuzano, D. P. Roa, and E. Davalos, “A new *p*-cycle selection approach based on efficient restoration measure for WDM optical networks,” *2010 Sixth Advanced International Conference on Telecommunications*, 2010, pp. 542-548.

[69] B. Wu, K. Yeung, and P.-H. Ho, “ILP formulations for *p*-cycle design without candidate cycle enumeration,” *IEEE/ACM Transactions on Networking*, vol. 18, no. 1, pp. 284–295, 2010.

[70] R. Morais, C. Pavan, A. Pinto, and C. Requejo, "Genetic algorithm for the topological design of survivable optical transport networks," in *IEEE/OSA Journal of Optical Communications and Networking*, vol. 3, no. 1, pp. 17-26, 2011.

[71] R. Morais and A. Pinto, “Topological design using genetic algorithms,” in *Intelligent Systems for Optical Networks Design: Advancing Techniques*, Y. Kavian and Z. Ghassemlooy, Eds. Hershey, PA: IGI Global, 2013, pp. 153-173.

[72] B. Dengiz, F. Altiparmak, and A. E. Smith, "Local search genetic algorithm for optimal design of reliable networks," in *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 3, pp. 179-188, 1997.

[73] H. T. T. Binh and N. T. Duong, “Heuristic and genetic algorithms for solving survivability problem in the design of last mile communication networks,” *Soft Computing*, vol. 19, no. 9, pp. 2619–2632, 2014.

[74] C. W. Ahn and R. Ramakrishna, “A genetic algorithm for shortest path routing problem and the sizing of populations,” *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 6, pp. 566–579, 2002.

[75] J. H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, MI, 1975.

[76] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Boston, MA: Addison-Wesley, 1989.

[77] K. F. Man, K. S. Tang, and S. Kwong, “Genetic algorithms: concepts and applications [in engineering design],” in *IEEE Transactions on Industrial Electronics*, vol. 43, no. 5, pp. 519-534, Oct. 1996.

[78] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*. Hoboken, NJ: John Wiley, 2004.

[79] D. E. Goldberg and K. Deb, “A comparative analysis of selection schemes used in genetic algorithms,” *Foundations of Genetic Algorithms*, pp. 69–93, 1991.

[80] B. L. Miller and D. E. Goldberg, “Genetic algorithms, selection schemes, and the varying effects of noise,” *Evolutionary Computation*, vol. 4, no. 2, pp. 113–131, 1996.

[81] B. L. Miller and D. E. Goldberg, “Genetic algorithms, tournament selection, and the effects of noise,” *Complex Systems,* vol. 9, no. E, pp.193–212, 1995.

[82] M. Leeson, Y. Kavian, W. Ren, E. Hines, and M. Naderi, “Survivable wavelength-routed optical network design using genetic algorithms,” *2007 ICTON Mediterranean Winter Conference*, pp. 247-255, 2007.

[83] X. Guo, J. Huang, H. Liu, and Y. Chen, “Efficient *p*-cycle combination protection strategy based on improved genetic algorithm in elastic optical networks,” *IET Optoelectronics*, vol. 12, no. 2, pp. 73–79, 2018.

[84] C. Colmán, D. Pinto, and B. Barán, “Optimal selection of sigma-cycle for optical networks. An approach based on genetic algorithm,” *CLEI Electronic Journal*, vol. 12, no. 3, 2009.

[85] France Telecom, “France telecom Form 6-K,” France Telecom, April 12, 2013. [Online]. Available: lft6k_registrationdocument.htm#_Toc353389444. [Accessed: Oct. 29, 2018]

[86] Python Software Foundation, *The Python Language Reference*, Version 3.7.6. [Online]. Available: http://www.python.org. [Accessed: May 01, 2019].

[87] Microsoft Corp., *Visual Studio Code User Guide*, Version 1.47.0. [Online]. Available: https://code.visualstudio.com/docs. [Accessed: Mar. 01, 2019].

[88] R. Fourer, D. M. Gay, B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming, Cengage Learning*, Boston, MA, USA, 2003.

[89] IBM Corp., *IBM ILOG CPLEX Optimization Studio CPLEX User’s Manua*l, Version 12, Release 7, 2016. [Online]. Available: https://www.ibm.com/support/knowledgecenter / SSSA5P_12.6.1/ilog.odms.studio.help/pdf/usrcplex.pdf. [Accessed: Mar. 01, 2019].

[90] D. A. Schupke, “An ILP for optimal p-cycle selection without cycle enumeration,” in *Proceedings 8th Conf. ONDM*, 2004.

[91] B. Wu, K. L. Yeung, K.-S. Lui, and S. Xu, “A new ILP-based *p*-cycle construction algorithm without candidate cycle enumeration,” *2007 IEEE International Conference on Communications*, Glasgow, 2007, pp. 2236-2241.

[92] J. Wu, Y. Liu, C. Yu, and Y. Wu, “Survivable routing and spectrum allocation algorithm based on *p*-cycle protection in elastic optical networks,” *Optik*, vol. 125, no. 16, pp. 4446–4451, 2014.

[93] D. P. Onguetou and W. D. Grover, “*p*-Cycle network design: from fewest in number to smallest in size,” *2007 6th International Workshop on Design and Reliable Communication Networks*, 2007.

[94] J. Doucette and W. D. Grover, “PCycle-SCP-Algorithm.exe - Version 1.0,” TRLabs proprietary AMPL ILP model, Edmonton, Alberta, July 2003.

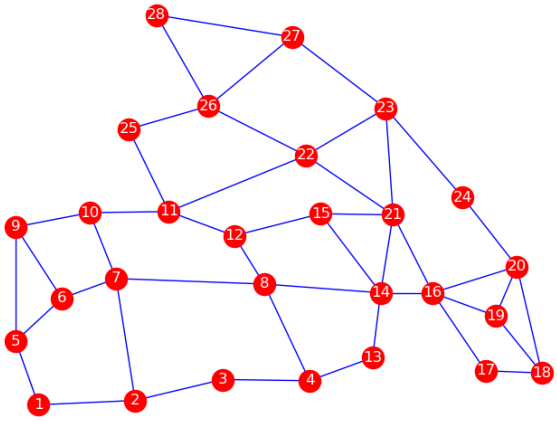
[95] J. Doucette and W. D. Grover, “PCycle-SCP-DatPrep.exe: *p*-Cycle Spare Capacity Placement AMPL Data File Preparation Software Version 1.0,” TRLabs proprietary software, Edmonton, Alberta, December 2001.

[96] J. Doucette and W. D. Grover, “pcycle-SCP.mod: *p*-Cycle SCP IP Model for AMPL - Version 1.0,” TRLabs proprietary AMPL ILP model, Edmonton, Alberta, December 2001.

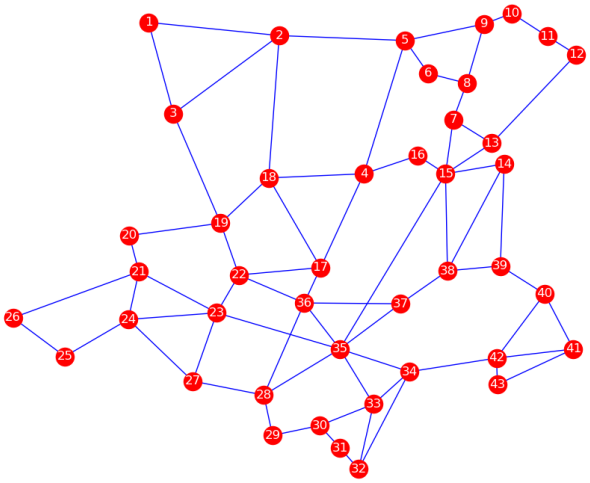
[97] A. M. Law, *Simulation Modeling and Analysis*. New York, NY: McGraw-Hill, 2015.

APPENDIX A

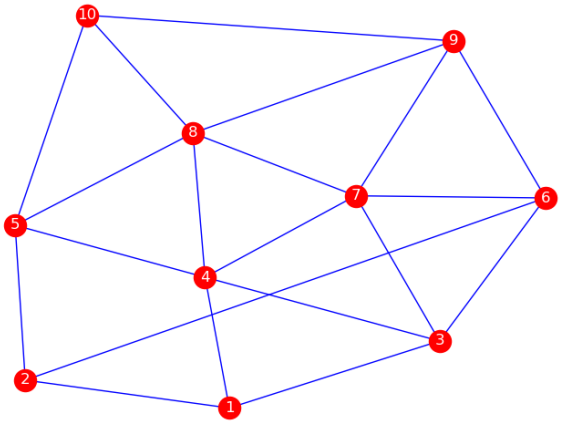
Network Topology Graphs



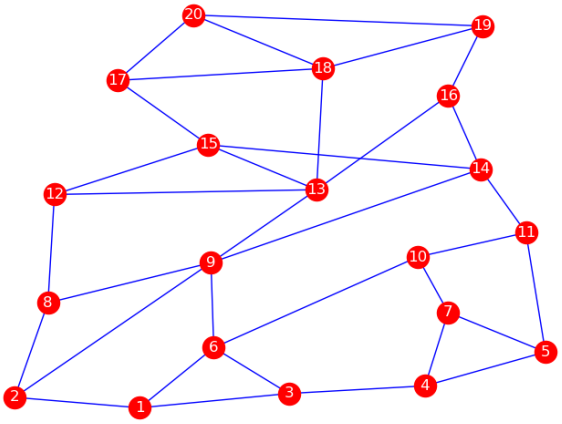
USA Long-Haul Network



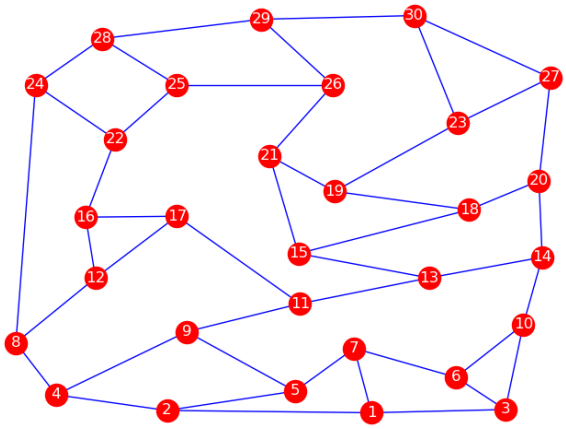
France Network



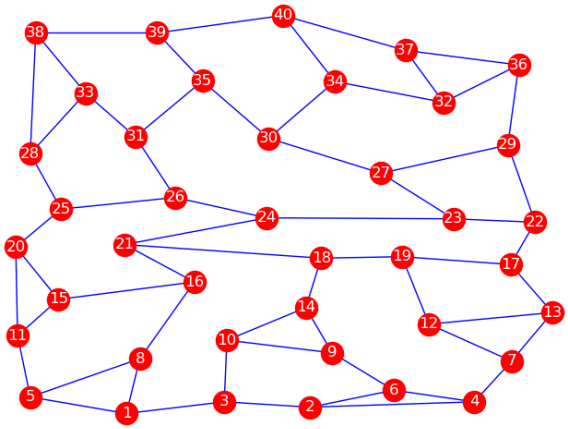
10n20s



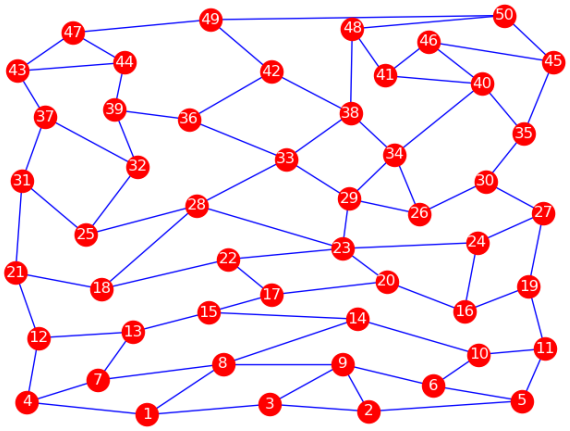
20n34s



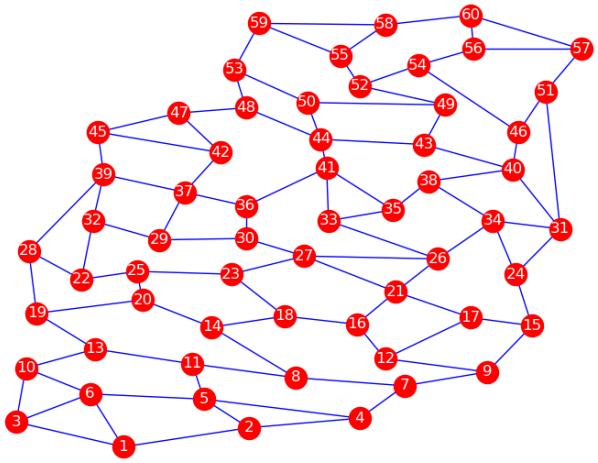
30n45s



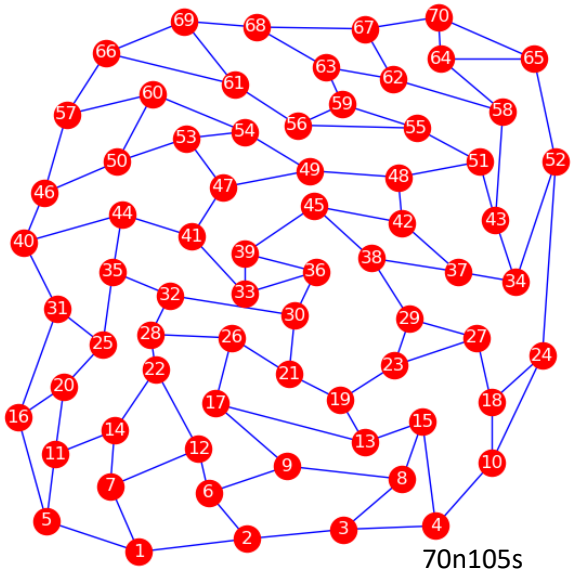
40n60s



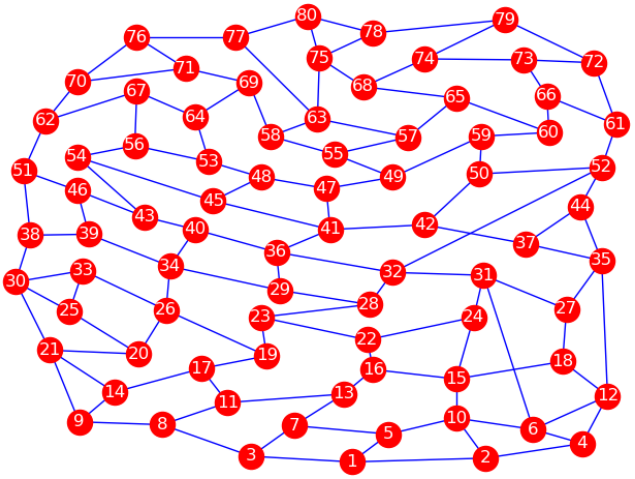
50n80s



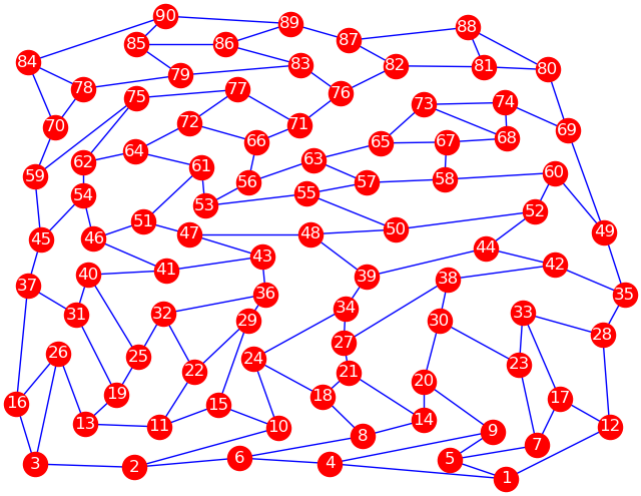
60n96s



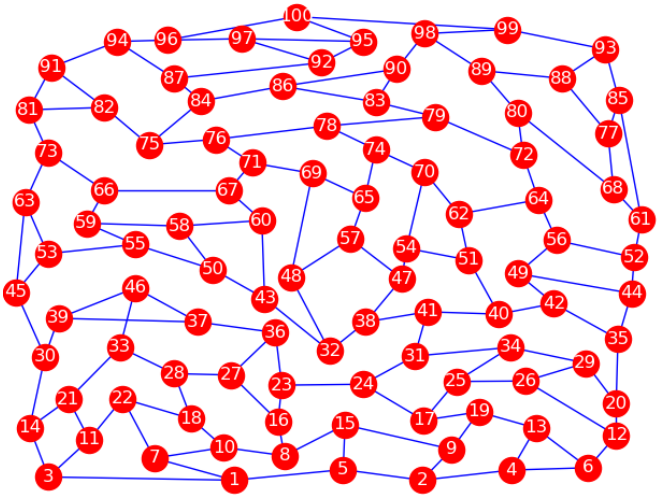
70n105s



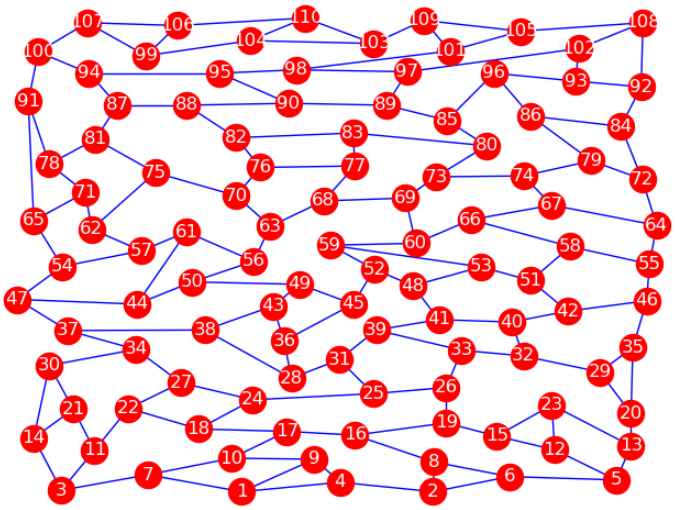
80n128s



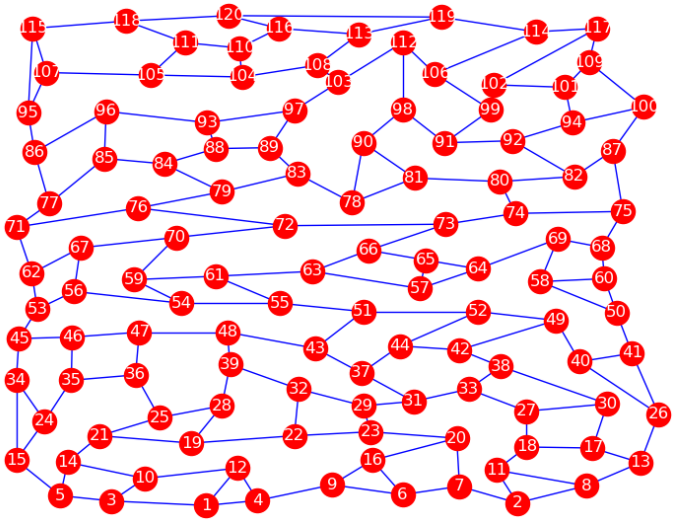
90n135s



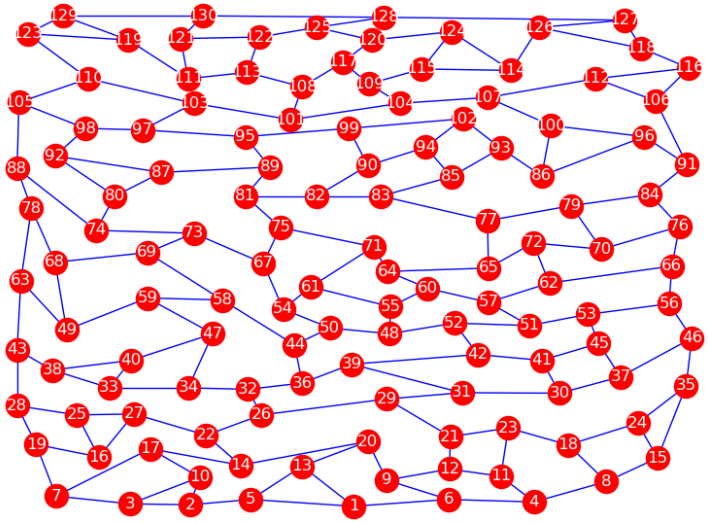
100n150s



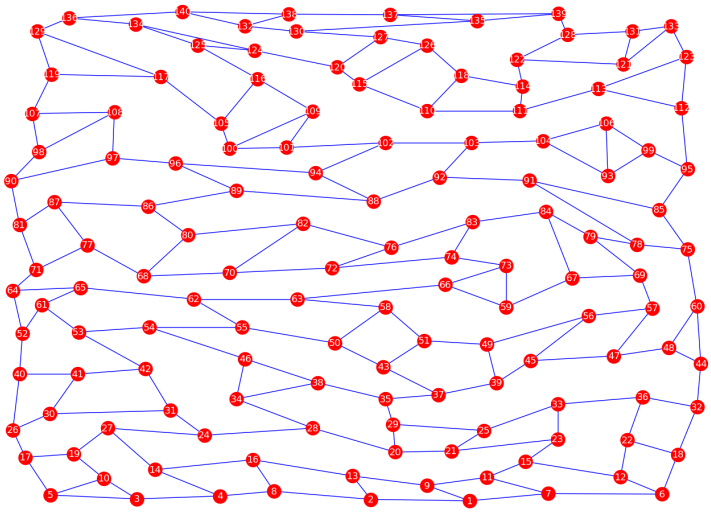
110n165s



120n180s



130n195s



140n210s

APPENDIX B

Network Topology Files (Nodes and Spans)

USA Long-Haul Backbone Network

Nodes (.node): ***NODE/ X/ Y***

N1 113.369 21.39	N8 442.781 139.037	N15 525.134 207.487	N22 503.743 264.171
N2 254.545 25.668	N9 80.214 194.652	N16 688.77 130.481	N23 620.321 310.16
N3 381.818 45.989	N10 188.235 208.556	N17 765.775 54.545	N24 732.62 223.529
N4 509.091 44.92	N11 303.743 209.626	N18 848.128 52.406	N25 244.92 289.84
N5 80.214 83.422	N12 398.93 186.096	N19 780.749 108.021	N26 361.497 312.299
N6 147.594 125.134	N13 601.07 67.38	N20 811.765 156.15	N27 483.422 379.679
N7 226.738 144.385	N14 612.834 130.481	N21 631.016 206.417	N28 285.561 401.07

Spans (.spans): ***SPAN/ SOURCE/ SINK***

S1	N1	N2	S24	N14	N16	S2	N1	N5	S25	N14	N21
S2	N1	N5	S25	N14	N21	S3	N2	N3	S26	N15	N21
S3	N2	N3	S26	N15	N21	S4	N2	N7	S27	N16	N17
S4	N2	N7	S27	N16	N17	S5	N3	N4	S28	N16	N19
S5	N3	N4	S28	N16	N19	S6	N4	N8	S29	N16	N20
S6	N4	N8	S29	N16	N20	S7	N4	N13	S30	N16	N21
S7	N4	N13	S30	N16	N21	S8	N5	N6	S31	N17	N18
S8	N5	N6	S31	N17	N18	S9	N5	N9	S32	N18	N19
S9	N5	N9	S32	N18	N19	S10	N6	N7	S33	N18	N20
S10	N6	N7	S33	N18	N20	S11	N6	N9	S34	N19	N20
S11	N6	N9	S34	N19	N20	S12	N7	N8	S35	N20	N24
S12	N7	N8	S35	N20	N24	S13	N7	N10	S36	N21	N22
S13	N7	N10	S36	N21	N22	S14	N8	N12	S37	N21	N23
S14	N8	N12	S37	N21	N23	S15	N8	N14	S38	N22	N23
S15	N8	N14	S38	N22	N23	S16	N9	N10	S39	N22	N26
S16	N9	N10	S39	N22	N26	S17	N10	N11	S40	N23	N24
S17	N10	N11	S40	N23	N24	S18	N11	N12	S41	N23	N27
S18	N11	N12	S41	N23	N27	S19	N11	N22	S42	N25	N26
S19	N11	N22	S42	N25	N26	S20	N11	N25	S43	N26	N27
S20	N11	N25	S43	N26	N27	S21	N12	N15	S44	N27	N28
S21	N12	N15	S44	N27	N28	S22	N13	N14	S45	N26	N28
S22	N13	N14	S45	N26	N28	S23	N14	N15			
S23	N14	N15	S1	N1	N2	S24	N14	N16			

France Network

Nodes (.node): *NODE/ X/ Y*

N1	172.500	512.5	N12	543.333	480	N23	231.667	218.333	N34	398.667	158.667
N2	285.833	499.167	N13	470	390	N24	155	211.667	N35	338.333	181.667
N3	194.167	420	N14	480.833	369.167	N25	100	174.167	N36	307.5	228.333
N4	359.167	359.167	N15	430	359.167	N26	55	213.333	N37	390.833	227.5
N5	395	494.167	N16	405.833	377.5	N27	210.833	149.167	N38	431.667	260.833
N6	415	460.833	N17	321.667	264.167	N28	272.5	135	N39	477.5	265.833
N7	436.667	414.167	N18	276.667	355	N29	280	94.167	N40	515.833	237.5
N8	448.333	450.833	N19	235	309.167	N30	320.833	104.167	N41	540.833	181.667
N9	463.333	510.833	N20	155.833	296.667	N31	338.667	81.333	N42	474.167	172.5
N10	487.500	521.667	N21	164.167	260	N32	354.667	60	N43	475	145.833
N11	518.333	498.333	N22	250.833	256.667	N33	367.5	125.833			

Spans (.spans): *SPAN/ SOURCE/ SINK*

S1	N1	N2	S19	N13	N15	S37	N41	N42	S55	N23	N27
S2	N1	N3	S20	N14	N15	S38	N41	N43	S56	N24	N27
S3	N2	N3	S21	N15	N16	S39	N42	N43	S57	N27	N28
S4	N2	N18	S22	N16	N4	S40	N42	N34	S58	N35	N28
S5	N3	N19	S23	N17	N18	S41	N34	N35	S59	N28	N29
S6	N18	N4	S24	N18	N19	S42	N37	N35	S60	N29	N30
S7	N2	N5	S25	N19	N20	S43	N35	N36	S61	N35	N33
S8	N6	N8	S26	N19	N22	S44	N36	N22	S62	N33	N34
S9	N8	N9	S27	N17	N22	S45	N35	N23	S63	N32	N34
S10	N9	N10	S28	N17	N36	S46	N36	N28	S64	N33	N32
S11	N10	N11	S29	N15	N35	S47	N22	N23	S65	N30	N33
S12	N11	N12	S30	N36	N37	S48	N23	N21	S66	N30	N31
S13	N7	N8	S31	N15	N38	S49	N20	N21	S67	N31	N32
S14	N7	N15	S32	N14	N38	S50	N21	N26	S68	N37	N38
S15	N4	N5	S33	N14	N39	S51	N25	N26	S69	N38	N39
S16	N4	N17	S34	N39	N40	S52	N24	N25	S70	N6	N5
S17	N12	N13	S35	N40	N41	S53	N24	N21	S71	N5	N9
S18	N7	N13	S36	N40	N42	S54	N23	N24			

30n45s Network

Nodes (.node): *NODE/ X/ Y/ SIZE*

N01	445.00	58.00	N04	61.00	97.00	N07	424.00	198.00	N10	630.00	250.00
N02	196.00	63.00	N05	352.00	105.00	N08	12.00	210.00	N11	358.00	295.00
N03	608.00	65.00	N06	548.00	135.00	N09	220.00	234.00	N12	109.00	351.00

N13	515.00	351.00	N18	564.00	499.00	N23	550.00	687.00	N28	117.00	871.00
N14	653.00	396.00	N19	400.00	539.00	N24	36.00	770.00	N29	311.00	913.00
N15	356.00	404.00	N20	649.00	562.00	N25	208.00	770.00	N30	497.00	921.00
N16	97.00	483.00	N21	321.00	616.00	N26	398.00	770.00			
N17	208.00	485.00	N22	133.00	652.00	N27	663.00	786.00			

Spans (.spans): ***SPAN/ SOURCE/ SINK***

S01	N01	N02	S13	N06	N10	S25	N15	N18	S37	N23	N27
S02	N01	N03	S14	N08	N12	S26	N15	N21	S38	N23	N30
S03	N01	N07	S15	N08	N24	S27	N16	N17	S39	N24	N28
S04	N02	N04	S16	N09	N11	S28	N16	N22	S40	N25	N26
S05	N02	N05	S17	N10	N14	S29	N18	N19	S41	N25	N28
S06	N03	N06	S18	N11	N13	S30	N18	N20	S42	N26	N29
S07	N03	N10	S19	N11	N17	S31	N19	N21	S43	N27	N30
S08	N04	N08	S20	N12	N16	S32	N19	N23	S44	N28	N29
S09	N04	N09	S21	N12	N17	S33	N20	N27	S45	N29	N30
S10	N05	N07	S22	N13	N14	S34	N21	N26			
S11	N05	N09	S23	N13	N15	S35	N22	N24			
S12	N06	N07	S24	N14	N20	S36	N22	N25			

40n60s Network

Nodes (.node): ***NODE/ X/ Y***

N01	188.00	101.00	N11	68.00	261.00	N21	186.00	449.00	N31	198.00	671.00
N02	391.00	114.00	N12	522.00	285.00	N22	639.00	495.00	N32	538.00	742.00
N03	296.00	125.00	N13	658.00	309.00	N23	549.00	502.00	N33	143.00	761.00
N04	572.00	125.00	N14	386.00	319.00	N24	343.00	504.00	N34	418.00	785.00
N05	82.00	135.00	N15	113.00	336.00	N25	116.00	522.00	N35	272.00	786.00
N06	483.00	149.00	N16	263.00	372.00	N26	242.00	546.00	N36	621.00	819.00
N07	613.00	209.00	N17	612.00	408.00	N27	469.00	596.00	N37	496.00	849.00
N08	203.00	214.00	N18	403.00	421.00	N28	82.00	636.00	N38	88.00	885.00
N09	415.00	226.00	N19	492.00	424.00	N29	609.00	654.00	N39	222.00	885.00
N10	299.00	252.00	N20	66.00	444.00	N30	345.00	668.00	N40	361.00	921.00

Spans (.spans): ***SPAN/ SOURCE/ SINK***

S01	N01	N03	S04	N02	N03	S07	N03	N10	S10	N05	N08
S02	N01	N05	S05	N02	N04	S08	N04	N06	S11	N05	N11
S03	N01	N08	S06	N02	N06	S09	N04	N07	S12	N06	N09

S13	N07	N12	S25	N15	N16	S37	N23	N27	S49	N31	N33
S14	N07	N13	S26	N15	N20	S38	N24	N26	S50	N31	N35
S15	N08	N16	S27	N16	N21	S39	N25	N26	S51	N32	N34
S16	N09	N10	S28	N17	N19	S40	N25	N28	S52	N32	N36
S17	N09	N14	S29	N17	N22	S41	N26	N31	S53	N32	N37
S18	N10	N14	S30	N18	N19	S42	N27	N29	S54	N33	N38
S19	N11	N15	S31	N18	N21	S43	N27	N30	S55	N34	N40
S20	N11	N20	S32	N20	N25	S44	N28	N33	S56	N35	N39
S21	N12	N13	S33	N21	N24	S45	N28	N38	S57	N36	N37
S22	N12	N19	S34	N22	N23	S46	N29	N36	S58	N37	N40
S23	N13	N17	S35	N22	N29	S47	N30	N34	S59	N38	N39
S24	N14	N18	S36	N23	N24	S48	N30	N35	S60	N39	N40

50n80s Network

Nodes (.node): *NODE/ X/ Y*

N01	202.00	38.00	N14	459.00	267.00	N27	687.00	521.00	N40	612.00	830.00
N02	473.00	46.00	N15	277.00	283.00	N28	263.00	539.00	N41	493.00	850.00
N03	352.00	63.00	N16	590.00	285.00	N29	449.00	555.00	N42	354.00	859.00
N04	55.00	68.00	N17	354.00	325.00	N30	616.00	596.00	N43	43.00	861.00
N05	661.00	71.00	N18	146.00	339.00	N31	49.00	598.00	N44	174.00	881.00
N06	552.00	107.00	N19	669.00	345.00	N32	190.00	632.00	N45	699.00	883.00
N07	142.00	121.00	N20	495.00	357.00	N33	372.00	650.00	N46	546.00	931.00
N08	295.00	159.00	N21	41.00	378.00	N34	505.00	661.00	N47	111.00	953.00
N09	441.00	159.00	N22	301.00	410.00	N35	663.00	711.00	N48	453.00	962.00
N10	608.00	182.00	N23	441.00	436.00	N36	253.00	745.00	N49	279.00	984.00
N11	689.00	196.00	N24	606.00	450.00	N37	77.00	751.00	N50	639.00	994.00
N12	69.00	222.00	N25	127.00	469.00	N38	451.00	760.00			
N13	184.00	236.00	N26	535.00	519.00	N39	162.00	768.00			

Spans (.spans): *SPAN/ SOURCE/ SINK*

S01	N01	N03	S10	N05	N06	S19	N10	N14	S28	N16	N24
S02	N01	N04	S11	N05	N11	S20	N11	N19	S29	N17	N20
S03	N01	N08	S12	N06	N09	S21	N12	N13	S30	N17	N22
S04	N02	N03	S13	N06	N10	S22	N12	N21	S31	N18	N21
S05	N02	N05	S14	N07	N08	S23	N13	N15	S32	N18	N22
S06	N02	N09	S15	N07	N13	S24	N14	N15	S33	N18	N28
S07	N03	N09	S16	N08	N09	S25	N15	N17	S34	N19	N27
S08	N04	N07	S17	N08	N14	S26	N16	N19	S35	N20	N23
S09	N04	N12	S18	N10	N11	S27	N16	N20	S36	N21	N31

S37	N22	N23	S48	N27	N30	S59	N34	N40	S70	N41	N46
S38	N23	N24	S49	N28	N33	S60	N35	N40	S71	N41	N48
S39	N23	N28	S50	N29	N33	S61	N35	N45	S72	N42	N49
S40	N23	N29	S51	N29	N34	S62	N36	N39	S73	N43	N44
S41	N24	N27	S52	N30	N35	S63	N36	N42	S74	N43	N47
S42	N25	N28	S53	N31	N37	S64	N37	N43	S75	N44	N47
S43	N25	N31	S54	N32	N37	S65	N38	N42	S76	N45	N46
S44	N25	N32	S55	N32	N39	S66	N38	N48	S77	N45	N50
S45	N26	N29	S56	N33	N36	S67	N39	N44	S78	N47	N49
S46	N26	N30	S57	N33	N38	S68	N40	N41	S79	N48	N50
S47	N26	N34	S58	N34	N38	S69	N40	N46	S80	N49	N50

60n96s Network

Nodes (.node): ***NODE/ X/ Y***

N01	170.00	40.00	N16	410.00	297.00	N31	619.00	497.00	N46	576.00	700.00
N02	299.00	80.00	N17	527.00	311.00	N32	139.00	514.00	N47	227.00	740.00
N03	60.00	92.00	N18	336.00	314.00	N33	381.00	514.00	N48	296.00	751.00
N04	413.00	100.00	N19	81.00	320.00	N34	550.00	514.00	N49	501.00	757.00
N05	253.00	140.00	N20	190.00	348.00	N35	447.00	537.00	N50	359.00	762.00
N06	136.00	151.00	N21	450.00	365.00	N36	296.00	545.00	N51	604.00	785.00
N07	459.00	168.00	N22	127.00	391.00	N37	233.00	574.00	N52	413.00	797.00
N08	347.00	185.00	N23	281.00	402.00	N38	484.00	597.00	N53	284.00	831.00
N09	544.00	197.00	N24	573.00	402.00	N39	150.00	611.00	N54	473.00	840.00
N10	70.00	205.00	N25	184.00	408.00	N40	570.00	622.00	N55	393.00	860.00
N11	241.00	214.00	N26	493.00	434.00	N41	379.00	625.00	N56	530.00	874.00
N12	439.00	225.00	N27	356.00	440.00	N42	270.00	657.00	N57	641.00	874.00
N13	141.00	245.00	N28	73.00	451.00	N43	479.00	674.00	N58	439.00	925.00
N14	261.00	291.00	N29	207.00	474.00	N44	373.00	685.00	N59	310.00	928.00
N15	590.00	294.00	N30	296.00	477.00	N45	144.00	700.00	N60	527.00	945.00

Spans (.spans): ***SPAN/ SOURCE/ SINK***

S01	N01	N02	S08	N04	N05	S15	N08	N11	S22	N12	N17
S02	N01	N03	S09	N04	N07	S16	N08	N14	S23	N13	N19
S03	N01	N06	S10	N05	N06	S17	N09	N12	S24	N14	N18
S04	N02	N04	S11	N05	N11	S18	N09	N15	S25	N14	N20
S05	N02	N05	S12	N06	N10	S19	N10	N13	S26	N15	N17
S06	N03	N06	S13	N07	N08	S20	N11	N13	S27	N15	N24
S07	N03	N10	S14	N07	N09	S21	N12	N16	S28	N16	N18

S29	N16	N21	S46	N26	N34	S63	N36	N41	S80	N47	N48
S30	N17	N21	S47	N27	N30	S64	N37	N39	S81	N48	N53
S31	N18	N23	S48	N28	N39	S65	N37	N42	S82	N49	N50
S32	N19	N20	S49	N29	N30	S66	N38	N40	S83	N49	N52
S33	N19	N28	S50	N29	N32	S67	N39	N45	S84	N50	N53
S34	N20	N25	S51	N29	N37	S68	N40	N43	S85	N51	N57
S35	N21	N26	S52	N30	N36	S69	N40	N46	S86	N52	N54
S36	N21	N27	S53	N31	N34	S70	N41	N44	S87	N52	N55
S37	N22	N25	S54	N31	N40	S71	N42	N45	S88	N53	N59
S38	N22	N28	S55	N31	N51	S72	N42	N47	S89	N54	N56
S39	N22	N32	S56	N32	N39	S73	N43	N44	S90	N55	N58
S40	N23	N25	S57	N33	N35	S74	N43	N49	S91	N55	N59
S41	N23	N27	S58	N33	N41	S75	N44	N48	S92	N56	N57
S42	N24	N31	S59	N34	N38	S76	N44	N50	S93	N56	N60
S43	N24	N34	S60	N35	N38	S77	N45	N47	S94	N57	N60
S44	N26	N27	S61	N35	N41	S78	N46	N51	S95	N58	N59
S45	N26	N33	S62	N36	N37	S79	N46	N54	S96	N58	N60

70n105s Network

Nodes (.node): ***NODE/ X/ Y***

N01	176.00	52.00	N19	416.00	309.00	N37	556.00	526.00	N55	507.00	769.00
N02	305.00	74.00	N20	87.00	332.00	N38	453.00	549.00	N56	365.00	774.00
N03	420.00	90.00	N21	355.00	353.00	N39	302.00	557.00	N57	90.00	792.00
N04	530.00	96.00	N22	196.00	360.00	N40	39.00	576.00	N58	610.00	797.00
N05	66.00	104.00	N23	480.00	367.00	N41	239.00	586.00	N59	419.00	809.00
N06	259.00	152.00	N24	657.00	384.00	N42	490.00	609.00	N60	193.00	827.00
N07	142.00	163.00	N25	133.00	403.00	N43	601.00	613.00	N61	290.00	843.00
N08	490.00	175.00	N26	287.00	414.00	N44	156.00	623.00	N62	479.00	852.00
N09	353.00	197.00	N27	579.00	414.00	N45	385.00	637.00	N63	399.00	872.00
N10	596.00	202.00	N28	190.00	420.00	N46	63.00	659.00	N64	536.00	886.00
N11	76.00	217.00	N29	499.00	446.00	N47	276.00	669.00	N65	647.00	886.00
N12	247.00	226.00	N30	362.00	452.00	N48	485.00	686.00	N66	137.00	896.00
N13	445.00	237.00	N31	79.00	463.00	N49	379.00	697.00	N67	445.00	937.00
N14	147.00	257.00	N32	213.00	486.00	N50	150.00	712.00	N68	316.00	940.00
N15	514.00	272.00	N33	302.00	489.00	N51	582.00	712.00	N69	230.00	950.00
N16	32.00	279.00	N34	625.00	509.00	N52	673.00	713.00	N70	533.00	957.00
N17	267.00	303.00	N35	145.00	526.00	N53	233.00	752.00			
N18	596.00	306.00	N36	387.00	526.00	N54	302.00	763.00			

Spans (.spans): ***SPAN/ SOURCE/ SINK***

S001	N01	N02	S028	N16	N20	S055	N34	N37	S082	N52	N65
S002	N01	N05	S029	N16	N31	S056	N34	N43	S083	N53	N54
S003	N01	N07	S030	N17	N26	S057	N34	N52	S084	N54	N60
S004	N02	N03	S031	N18	N24	S058	N35	N44	S085	N55	N56
S005	N02	N06	S032	N18	N27	S059	N36	N39	S086	N55	N59
S006	N03	N04	S033	N19	N21	S060	N37	N38	S087	N56	N59
S007	N03	N08	S034	N19	N23	S061	N37	N42	S088	N56	N61
S008	N04	N10	S035	N20	N25	S062	N38	N45	S089	N57	N60
S009	N04	N15	S036	N21	N26	S063	N39	N45	S090	N57	N66
S010	N05	N11	S037	N21	N30	S064	N40	N44	S091	N58	N62
S011	N05	N16	S038	N22	N28	S065	N40	N46	S092	N58	N64
S012	N06	N09	S039	N23	N27	S066	N41	N44	S093	N59	N63
S013	N06	N12	S040	N23	N29	S067	N41	N47	S094	N61	N66
S014	N07	N12	S041	N24	N52	S068	N42	N45	S095	N61	N69
S015	N07	N14	S042	N25	N31	S069	N42	N48	S096	N62	N63
S016	N08	N09	S043	N25	N35	S070	N43	N51	S097	N62	N67
S017	N08	N15	S044	N26	N28	S071	N43	N58	S098	N63	N68
S018	N09	N17	S045	N27	N29	S072	N46	N50	S099	N64	N65
S019	N10	N18	S046	N28	N32	S073	N46	N57	S100	N64	N70
S020	N10	N24	S047	N29	N38	S074	N47	N49	S101	N65	N70
S021	N11	N14	S048	N30	N32	S075	N47	N53	S102	N66	N69
S022	N11	N20	S049	N30	N36	S076	N48	N49	S103	N67	N68
S023	N12	N22	S050	N31	N40	S077	N48	N51	S104	N67	N70
S024	N13	N15	S051	N32	N35	S078	N49	N54	S105	N68	N69
S025	N13	N17	S052	N33	N36	S079	N50	N53			
S026	N13	N19	S053	N33	N39	S080	N50	N60			
S027	N14	N22	S054	N33	N41	S081	N51	N55			

80n128s Network

Nodes (.node): ***NODE/ X/ Y***

N01	370.00	48.00	N09	91.00	133.00	N17	216.00	246.00	N25	81.00	368.00
N02	507.00	56.00	N10	477.00	141.00	N18	586.00	261.00	N26	180.00	370.00
N03	267.00	60.00	N11	243.00	174.00	N19	283.00	273.00	N27	590.00	372.00
N04	606.00	87.00	N12	632.00	188.00	N20	152.00	277.00	N28	388.00	382.00
N05	408.00	107.00	N13	362.00	192.00	N21	61.00	285.00	N29	297.00	412.00
N06	556.00	117.00	N14	127.00	196.00	N22	386.00	307.00	N30	26.00	430.00
N07	311.00	125.00	N15	477.00	224.00	N23	277.00	355.00	N31	505.00	444.00
N08	176.00	127.00	N16	392.00	244.00	N24	495.00	355.00	N32	412.00	450.00

N33	95.00	454.00	N45	228.00	602.00	N57	428.00	737.00	N69	265.00	852.00
N34	184.00	463.00	N46	89.00	624.00	N58	287.00	739.00	N70	89.00	854.00
N35	626.00	475.00	N47	344.00	628.00	N59	503.00	739.00	N71	200.00	883.00
N36	293.00	493.00	N48	277.00	652.00	N60	572.00	747.00	N72	618.00	895.00
N37	548.00	511.00	N49	412.00	652.00	N61	641.00	764.00	N73	546.00	901.00
N38	41.00	529.00	N50	501.00	659.00	N62	57.00	772.00	N74	445.00	903.00
N39	101.00	531.00	N51	34.00	665.00	N63	335.00	776.00	N75	337.00	905.00
N40	210.00	541.00	N52	626.00	673.00	N64	210.00	780.00	N76	150.00	949.00
N41	348.00	541.00	N53	224.00	685.00	N65	477.00	820.00	N77	251.00	949.00
N42	445.00	551.00	N54	89.00	695.00	N66	570.00	824.00	N78	392.00	958.00
N43	158.00	568.00	N55	352.00	701.00	N67	150.00	834.00	N79	527.00	986.00
N44	604.00	590.00	N56	148.00	719.00	N68	382.00	844.00	N80	325.00	994.00

Spans (.spans): ***SPAN/ SOURCE/ SINK***

S001	N01	N02	S028	N15	N16	S055	N30	N38	S082	N47	N49
S002	N01	N03	S029	N15	N18	S056	N31	N32	S083	N48	N53
S003	N01	N05	S030	N15	N24	S057	N32	N36	S084	N49	N55
S004	N02	N04	S031	N16	N22	S058	N32	N52	S085	N49	N59
S005	N02	N10	S032	N17	N19	S059	N34	N39	S086	N50	N52
S006	N03	N07	S033	N18	N27	S060	N34	N40	S087	N50	N59
S007	N03	N08	S034	N19	N23	S061	N35	N37	S088	N51	N62
S008	N04	N06	S035	N19	N26	S062	N35	N44	S089	N52	N61
S009	N04	N12	S036	N20	N21	S063	N36	N40	S090	N53	N56
S010	N05	N07	S037	N20	N25	S064	N36	N41	S091	N53	N64
S011	N05	N10	S038	N20	N26	S065	N37	N42	S092	N54	N56
S012	N06	N10	S039	N21	N30	S066	N37	N44	S093	N55	N57
S013	N06	N12	S040	N22	N23	S067	N38	N39	S094	N55	N58
S014	N06	N31	S041	N22	N24	S068	N38	N51	S095	N56	N67
S015	N07	N13	S042	N23	N28	S069	N39	N46	S096	N57	N63
S016	N08	N09	S043	N24	N31	S070	N40	N43	S097	N57	N65
S017	N08	N11	S044	N25	N30	S071	N41	N42	S098	N58	N63
S018	N09	N14	S045	N25	N33	S072	N41	N45	S099	N58	N69
S019	N09	N21	S046	N26	N33	S073	N41	N47	S100	N59	N60
S020	N10	N15	S047	N26	N34	S074	N42	N50	S101	N60	N65
S021	N11	N13	S048	N27	N31	S075	N43	N46	S102	N60	N66
S022	N11	N17	S049	N27	N35	S076	N43	N54	S103	N61	N66
S023	N12	N18	S050	N28	N29	S077	N44	N52	S104	N61	N72
S024	N12	N35	S051	N28	N32	S078	N45	N48	S105	N62	N67
S025	N13	N16	S052	N29	N34	S079	N45	N54	S106	N62	N70
S026	N14	N17	S053	N29	N36	S080	N46	N51	S107	N63	N75
S027	N14	N21	S054	N30	N33	S081	N47	N48	S108	N63	N77

S109	N64	N67	S114	N68	N75	S119	N72	N73	S124	N75	N80
S110	N64	N69	S115	N69	N71	S120	N72	N79	S125	N76	N77
S111	N65	N68	S116	N70	N71	S121	N73	N74	S126	N77	N80
S112	N66	N73	S117	N70	N76	S122	N74	N79	S127	N78	N79
S113	N68	N74	S118	N71	N76	S123	N75	N78	S128	N78	N80

90n135s Network

Nodes (.node): ***NODE/ X/ Y***

N01	571.00	15.00	N24	283.00	273.00	N47	210.00	541.00	N70	57.00	772.00
N02	147.00	40.00	N25	152.00	277.00	N48	348.00	541.00	N71	335.00	776.00
N03	34.00	47.00	N26	61.00	285.00	N49	683.00	545.00	N72	210.00	780.00
N04	370.00	48.00	N27	386.00	307.00	N50	445.00	551.00	N73	477.00	820.00
N05	507.00	56.00	N28	681.00	324.00	N51	158.00	568.00	N74	570.00	824.00
N06	267.00	60.00	N29	277.00	355.00	N52	604.00	590.00	N75	150.00	834.00
N07	606.00	87.00	N30	495.00	355.00	N53	228.00	602.00	N76	382.00	844.00
N08	408.00	107.00	N31	81.00	368.00	N54	89.00	624.00	N77	265.00	852.00
N09	556.00	117.00	N32	180.00	370.00	N55	344.00	628.00	N78	89.00	854.00
N10	311.00	125.00	N33	590.00	372.00	N56	277.00	652.00	N79	200.00	883.00
N11	176.00	127.00	N34	388.00	382.00	N57	412.00	652.00	N80	618.00	895.00
N12	689.00	127.00	N35	706.00	411.00	N58	501.00	659.00	N81	546.00	901.00
N13	91.00	133.00	N36	297.00	412.00	N59	34.00	665.00	N82	445.00	903.00
N14	477.00	141.00	N37	26.00	430.00	N60	626.00	673.00	N83	337.00	905.00
N15	243.00	174.00	N38	505.00	444.00	N61	224.00	685.00	N84	26.00	911.00
N16	13.00	177.00	N39	412.00	450.00	N62	89.00	695.00	N85	150.00	949.00
N17	632.00	188.00	N40	95.00	454.00	N63	352.00	701.00	N86	251.00	949.00
N18	362.00	192.00	N41	184.00	463.00	N64	148.00	719.00	N87	392.00	958.00
N19	127.00	196.00	N42	626.00	475.00	N65	428.00	737.00	N88	527.00	986.00
N20	477.00	224.00	N43	293.00	493.00	N66	287.00	739.00	N89	325.00	994.00
N21	392.00	244.00	N44	548.00	511.00	N67	503.00	739.00	N90	183.00	1010.0
N22	216.00	246.00	N45	41.00	529.00	N68	572.00	747.00			
N23	586.00	261.00	N46	101.00	531.00	N69	641.00	764.00			

Spans (.spans): ***SPAN/ SOURCE/ SINK***

S001	N01	N04	S006	N02	N10	S011	N05	N07	S016	N08	N14
S002	N01	N05	S007	N03	N16	S012	N05	N09	S017	N08	N18
S003	N01	N12	S008	N03	N26	S013	N06	N08	S018	N09	N20
S004	N02	N03	S009	N04	N06	S014	N07	N17	S019	N10	N15
S005	N02	N06	S010	N04	N09	S015	N07	N23	S020	N10	N24

S021	N11	N13	S050	N28	N35	S079	N50	N52	S108	N69	N74
S022	N11	N15	S051	N29	N36	S080	N50	N55	S109	N69	N80
S023	N11	N22	S052	N30	N38	S081	N51	N61	S110	N70	N78
S024	N12	N17	S053	N31	N37	S082	N52	N60	S111	N70	N84
S025	N12	N28	S054	N31	N40	S083	N53	N55	S112	N71	N76
S026	N13	N19	S055	N32	N36	S084	N53	N56	S113	N71	N77
S027	N13	N26	S056	N34	N39	S085	N53	N61	S114	N72	N77
S028	N14	N20	S057	N35	N42	S086	N54	N62	S115	N73	N74
S029	N14	N21	S058	N35	N49	S087	N55	N57	S116	N75	N77
S030	N15	N29	S059	N36	N43	S088	N56	N63	S117	N76	N82
S031	N16	N26	S060	N37	N45	S089	N56	N66	S118	N76	N83
S032	N16	N37	S061	N38	N42	S090	N57	N58	S119	N78	N79
S033	N17	N33	S062	N39	N44	S091	N57	N63	S120	N78	N84
S034	N18	N21	S063	N39	N48	S092	N58	N60	S121	N79	N83
S035	N18	N24	S064	N40	N41	S093	N58	N67	S122	N79	N85
S036	N19	N25	S065	N41	N43	S094	N59	N70	S123	N80	N81
S037	N19	N31	S066	N41	N46	S095	N59	N75	S124	N80	N88
S038	N20	N30	S067	N42	N44	S096	N61	N64	S125	N81	N82
S039	N21	N27	S068	N43	N47	S097	N62	N64	S126	N81	N88
S040	N22	N29	S069	N44	N52	S098	N62	N75	S127	N82	N87
S041	N22	N32	S070	N45	N54	S099	N63	N65	S128	N83	N86
S042	N23	N30	S071	N45	N59	S100	N64	N72	S129	N84	N90
S043	N23	N33	S072	N46	N51	S101	N65	N67	S130	N85	N86
S044	N24	N34	S073	N46	N54	S102	N65	N73	S131	N85	N90
S045	N25	N32	S074	N47	N48	S103	N66	N71	S132	N86	N89
S046	N25	N40	S075	N47	N51	S104	N66	N72	S133	N87	N88
S047	N27	N34	S076	N48	N50	S105	N67	N68	S134	N87	N89
S048	N27	N38	S077	N49	N60	S106	N68	N73	S135	N89	N90
S049	N28	N33	S078	N49	N69	S107	N68	N74			

100n150s Network

Nodes (.node): *NODE/ X/ Y*

N001	250.00	17.00	N009	491.00	84.00	N017	460.00	147.00	N025	496.00	231.00
N002	458.00	17.00	N010	239.00	88.00	N018	203.00	150.00	N026	573.00	232.00
N003	45.00	24.00	N011	90.00	105.00	N019	521.00	165.00	N027	247.00	245.00
N004	557.00	39.00	N012	673.00	114.00	N020	673.00	183.00	N028	184.00	248.00
N005	371.00	40.00	N013	584.00	130.00	N021	67.00	192.00	N029	639.00	271.00
N006	642.00	44.00	N014	24.00	131.00	N022	127.00	194.00	N030	41.00	285.00
N007	162.00	65.00	N015	373.00	137.00	N023	303.00	223.00	N031	450.00	286.00
N008	306.00	70.00	N016	299.00	145.00	N024	393.00	225.00	N032	356.00	297.00

N033	124.00	305.00	N050	227.00	474.00	N067	244.00	645.00	N084	214.00	839.00
N034	556.00	305.00	N051	510.00	494.00	N068	670.00	648.00	N085	676.00	842.00
N035	675.00	323.00	N052	693.00	500.00	N069	337.00	682.00	N086	304.00	871.00
N036	296.00	337.00	N053	44.00	508.00	N070	461.00	685.00	N087	184.00	888.00
N037	210.00	360.00	N054	441.00	520.00	N071	270.00	705.00	N088	613.00	888.00
N038	396.00	360.00	N055	141.00	528.00	N072	570.00	722.00	N089	524.00	905.00
N039	56.00	368.00	N056	607.00	537.00	N073	44.00	728.00	N090	430.00	908.00
N040	543.00	377.00	N057	379.00	540.00	N074	407.00	734.00	N091	48.00	912.00
N041	464.00	382.00	N058	190.00	568.00	N075	156.00	744.00	N092	347.00	922.00
N042	604.00	400.00	N059	87.00	574.00	N076	230.00	756.00	N093	661.00	951.00
N043	284.00	411.00	N060	281.00	580.00	N077	664.00	768.00	N094	121.00	968.00
N044	690.00	422.00	N061	701.00	582.00	N078	353.00	785.00	N095	393.00	968.00
N045	9.00	425.00	N062	499.00	594.00	N079	473.00	805.00	N096	176.00	971.00
N046	141.00	434.00	N063	19.00	620.00	N080	564.00	814.00	N097	259.00	974.00
N047	436.00	454.00	N064	587.00	625.00	N081	23.00	820.00	N098	461.00	985.00
N048	313.00	457.00	N065	395.00	629.00	N082	107.00	825.00	N099	553.00	994.00
N049	564.00	465.00	N066	107.00	645.00	N083	407.00	837.00	N100	319.00	1022.0

Spans (.spans): ***SPAN/ SOURCE/ SINK***

S001	N001	N003	S023	N011	N022	S045	N027	N028	S067	N040	N051
S002	N001	N005	S024	N012	N020	S046	N027	N036	S068	N042	N049
S003	N001	N007	S025	N012	N026	S047	N028	N033	S069	N043	N050
S004	N002	N004	S026	N013	N019	S048	N029	N034	S070	N043	N060
S005	N002	N005	S027	N014	N021	S049	N030	N039	S071	N044	N049
S006	N002	N009	S028	N014	N030	S050	N030	N045	S072	N044	N052
S007	N003	N011	S029	N016	N023	S051	N031	N034	S073	N045	N053
S008	N003	N014	S030	N016	N027	S052	N031	N041	S074	N045	N063
S009	N004	N006	S031	N017	N019	S053	N032	N038	S075	N047	N054
S010	N004	N013	S032	N017	N024	S054	N032	N043	S076	N047	N057
S011	N005	N015	S033	N017	N025	S055	N032	N048	S077	N048	N057
S012	N006	N012	S034	N018	N022	S056	N033	N046	S078	N048	N069
S013	N006	N013	S035	N018	N028	S057	N035	N042	S079	N049	N056
S014	N007	N010	S036	N020	N029	S058	N035	N044	S080	N050	N055
S015	N007	N022	S037	N020	N035	S059	N036	N037	S081	N050	N058
S016	N008	N010	S038	N021	N033	S060	N037	N039	S082	N051	N054
S017	N008	N015	S039	N023	N024	S061	N037	N046	S083	N051	N062
S018	N008	N016	S040	N023	N036	S062	N038	N041	S084	N052	N056
S019	N009	N015	S041	N024	N031	S063	N038	N047	S085	N052	N061
S020	N009	N019	S042	N025	N026	S064	N039	N046	S086	N053	N055
S021	N010	N018	S043	N025	N034	S065	N040	N041	S087	N053	N063
S022	N011	N021	S044	N026	N029	S066	N040	N042	S088	N054	N070

S089	N055	N059	S105	N066	N073	S121	N077	N088	S137	N088	N093
S090	N056	N064	S106	N067	N071	S122	N078	N079	S138	N089	N098
S091	N057	N065	S107	N068	N077	S123	N079	N083	S139	N090	N098
S092	N058	N059	S108	N068	N080	S124	N080	N089	S140	N091	N094
S093	N058	N060	S109	N069	N071	S125	N081	N082	S141	N092	N095
S094	N059	N066	S110	N070	N074	S126	N081	N091	S142	N092	N097
S095	N060	N067	S111	N071	N076	S127	N082	N091	S143	N093	N099
S096	N061	N068	S112	N072	N079	S128	N083	N086	S144	N094	N096
S097	N061	N085	S113	N072	N080	S129	N083	N090	S145	N095	N097
S098	N062	N064	S114	N073	N081	S130	N084	N086	S146	N095	N100
S099	N062	N070	S115	N074	N078	S131	N084	N087	S147	N096	N097
S100	N063	N073	S116	N075	N076	S132	N085	N093	S148	N096	N100
S101	N064	N072	S117	N075	N082	S133	N086	N090	S149	N098	N099
S102	N065	N069	S118	N075	N084	S134	N087	N092	S150	N099	N100
S103	N065	N074	S119	N076	N078	S135	N087	N094			
S104	N066	N067	S120	N077	N085	S136	N088	N089			

110n165s Network

Nodes (.node): ***NODE/ X/ Y***

N001	250.00	17.00	N022	127.00	194.00	N043	284.00	411.00	N064	701.00	582.00
N002	458.00	17.00	N023	587.00	201.00	N044	136.00	415.00	N065	24.00	592.00
N003	54.00	20.00	N024	262.00	212.00	N045	371.00	415.00	N066	499.00	594.00
N004	357.00	37.00	N025	393.00	225.00	N046	690.00	422.00	N067	587.00	625.00
N005	657.00	42.00	N026	472.00	238.00	N047	6.00	425.00	N068	339.00	634.00
N006	541.00	49.00	N027	184.00	248.00	N048	436.00	454.00	N069	428.00	640.00
N007	148.00	53.00	N028	304.00	258.00	N049	313.00	457.00	N070	244.00	645.00
N008	459.00	80.00	N029	639.00	271.00	N050	196.00	462.00	N071	80.00	652.00
N009	329.00	86.00	N030	41.00	285.00	N051	564.00	465.00	N072	685.00	680.00
N010	239.00	88.00	N031	356.00	297.00	N052	394.00	489.00	N073	461.00	685.00
N011	90.00	105.00	N032	556.00	305.00	N053	510.00	494.00	N074	556.00	687.00
N012	590.00	106.00	N033	489.00	317.00	N054	55.00	495.00	N075	156.00	694.00
N013	673.00	114.00	N034	135.00	319.00	N055	693.00	500.00	N076	270.00	705.00
N014	24.00	131.00	N035	675.00	323.00	N056	263.00	506.00	N077	373.00	708.00
N015	527.00	135.00	N036	296.00	337.00	N057	141.00	528.00	N078	41.00	712.00
N016	373.00	137.00	N037	61.00	359.00	N058	607.00	537.00	N079	630.00	719.00
N017	299.00	145.00	N038	210.00	360.00	N059	346.00	540.00	N080	516.00	751.00
N018	203.00	150.00	N039	396.00	360.00	N060	439.00	544.00	N081	91.00	765.00
N019	472.00	163.00	N040	543.00	377.00	N061	190.00	568.00	N082	244.00	768.00
N020	673.00	183.00	N041	464.00	382.00	N062	87.00	574.00	N083	371.00	778.00
N021	67.00	192.00	N042	604.00	400.00	N063	281.00	580.00	N084	662.00	792.00

N085	473.00	805.00	N092	684.00	876.00	N099	146.00	942.00	N106	180.00	1006.0
N086	564.00	814.00	N093	613.00	888.00	N100	29.00	950.00	N107	82.00	1012.0
N087	115.00	837.00	N094	84.00	905.00	N101	476.00	952.00	N108	686.00	1012.0
N088	190.00	837.00	N095	226.00	905.00	N102	616.00	957.00	N109	448.00	1017.0
N089	407.00	837.00	N096	524.00	905.00	N103	393.00	968.00	N110	319.00	1022.0
N090	301.00	841.00	N097	430.00	908.00	N104	259.00	974.00			
N091	18.00	845.00	N098	309.00	912.00	N105	553.00	994.00			

Spans (.spans): ***SPAN/ SOURCE/ SINK***

S001	N001	N004	S033	N018	N024	S065	N041	N048	S097	N062	N075
S002	N001	N007	S034	N019	N026	S066	N042	N046	S098	N063	N068
S003	N001	N009	S035	N020	N029	S067	N042	N051	S099	N063	N070
S004	N002	N004	S036	N020	N035	S068	N043	N049	S100	N064	N067
S005	N002	N006	S037	N021	N030	S069	N044	N047	S101	N064	N072
S006	N002	N008	S038	N022	N027	S070	N044	N050	S102	N065	N071
S007	N003	N007	S039	N024	N025	S071	N044	N061	S103	N065	N091
S008	N003	N011	S040	N024	N027	S072	N045	N049	S104	N066	N067
S009	N003	N014	S041	N025	N026	S073	N045	N052	S105	N067	N074
S010	N004	N009	S042	N025	N031	S074	N046	N055	S106	N068	N069
S011	N005	N006	S043	N026	N033	S075	N047	N054	S107	N068	N077
S012	N005	N012	S044	N027	N034	S076	N048	N052	S108	N069	N073
S013	N005	N013	S045	N028	N031	S077	N048	N053	S109	N070	N075
S014	N006	N008	S046	N028	N036	S078	N049	N050	S110	N070	N076
S015	N007	N010	S047	N028	N038	S079	N050	N056	S111	N071	N078
S016	N008	N016	S048	N029	N032	S080	N051	N053	S112	N072	N079
S017	N009	N010	S049	N029	N035	S081	N051	N058	S113	N072	N084
S018	N010	N017	S050	N030	N034	S082	N052	N059	S114	N073	N074
S019	N011	N021	S051	N031	N039	S083	N053	N059	S115	N073	N080
S020	N011	N022	S052	N032	N033	S084	N054	N057	S116	N074	N079
S021	N012	N015	S053	N032	N040	S085	N054	N065	S117	N075	N081
S022	N012	N023	S054	N033	N039	S086	N055	N058	S118	N076	N077
S023	N013	N020	S055	N034	N037	S087	N055	N064	S119	N076	N082
S024	N013	N023	S056	N035	N046	S088	N056	N061	S120	N077	N083
S025	N014	N021	S057	N036	N043	S089	N056	N063	S121	N078	N081
S026	N014	N030	S058	N036	N045	S090	N057	N061	S122	N078	N091
S027	N015	N019	S059	N037	N038	S091	N057	N062	S123	N079	N086
S028	N015	N023	S060	N037	N047	S092	N058	N066	S124	N080	N083
S029	N016	N017	S061	N038	N043	S093	N059	N060	S125	N080	N085
S030	N016	N019	S062	N039	N041	S094	N060	N066	S126	N081	N087
S031	N017	N018	S063	N040	N041	S095	N060	N069	S127	N082	N083
S032	N018	N022	S064	N040	N042	S096	N062	N071	S128	N082	N088

S129	N084	N086	S139	N090	N095	S149	N097	N102	S159	N103	N109
S130	N084	N092	S140	N091	N100	S150	N098	N101	S160	N103	N110
S131	N085	N089	S141	N092	N093	S151	N099	N104	S161	N104	N110
S132	N085	N096	S142	N092	N108	S152	N099	N106	S162	N105	N108
S133	N086	N096	S143	N093	N096	S153	N099	N107	S163	N105	N109
S134	N087	N088	S144	N093	N102	S154	N100	N107	S164	N106	N107
S135	N087	N094	S145	N094	N095	S155	N101	N105	S165	N106	N110
S136	N088	N090	S146	N094	N100	S156	N101	N109			
S137	N089	N090	S147	N095	N098	S157	N102	N108			
S138	N089	N097	S148	N097	N098	S158	N103	N104			

120n180s Network

Nodes (.node): *NODE/ X/ Y*

N001	220.00	15.00	N028	237.00	218.00	N055	300.00	427.00	N082	621.00	685.00
N002	559.00	20.00	N029	391.00	218.00	N056	76.00	451.00	N083	319.00	691.00
N003	117.00	24.00	N030	657.00	222.00	N057	453.00	457.00	N084	174.00	711.00
N004	276.00	25.00	N031	446.00	227.00	N058	584.00	471.00	N085	109.00	722.00
N005	61.00	35.00	N032	321.00	252.00	N059	141.00	475.00	N086	33.00	735.00
N006	434.00	37.00	N033	506.00	252.00	N060	654.00	477.00	N087	663.00	738.00
N007	496.00	54.00	N034	13.00	270.00	N061	230.00	482.00	N088	230.00	742.00
N008	634.00	55.00	N035	73.00	270.00	N062	30.00	488.00	N089	289.00	744.00
N009	357.00	57.00	N036	143.00	281.00	N063	336.00	492.00	N090	391.00	754.00
N010	153.00	70.00	N037	389.00	285.00	N064	516.00	497.00	N091	480.00	754.00
N011	536.00	87.00	N038	541.00	298.00	N065	459.00	515.00	N092	554.00	758.00
N012	254.00	91.00	N039	246.00	302.00	N066	397.00	535.00	N093	221.00	795.00
N013	693.00	102.00	N040	627.00	308.00	N067	83.00	540.00	N094	619.00	795.00
N014	69.00	103.00	N041	684.00	325.00	N068	651.00	541.00	N095	26.00	815.00
N015	13.00	107.00	N042	496.00	332.00	N069	603.00	558.00	N096	111.00	818.00
N016	401.00	107.00	N043	339.00	335.00	N070	187.00	561.00	N097	316.00	820.00
N017	641.00	132.00	N044	431.00	340.00	N071	13.00	582.00	N098	434.00	821.00
N018	569.00	134.00	N045	16.00	355.00	N072	306.00	585.00	N099	530.00	821.00
N019	204.00	142.00	N046	74.00	358.00	N073	481.00	588.00	N100	697.00	827.00
N020	493.00	152.00	N047	147.00	368.00	N074	557.00	610.00	N101	611.00	867.00
N021	104.00	157.00	N048	243.00	368.00	N075	674.00	614.00	N102	533.00	872.00
N022	316.00	157.00	N049	601.00	392.00	N076	146.00	621.00	N103	364.00	877.00
N023	399.00	165.00	N050	667.00	408.00	N077	49.00	630.00	N104	259.00	888.00
N024	44.00	187.00	N051	391.00	410.00	N078	379.00	632.00	N105	159.00	892.00
N025	169.00	194.00	N052	516.00	410.00	N079	237.00	654.00	N106	469.00	895.00
N026	713.00	201.00	N053	36.00	415.00	N080	540.00	675.00	N107	46.00	897.00
N027	569.00	207.00	N054	193.00	427.00	N081	447.00	682.00	N108	341.00	912.00

N109	637.00	918.00	N112	434.00	958.00	N115	31.00	987.00	N118	133.00	1002.0
N110	256.00	947.00	N113	386.00	975.00	N116	299.00	987.00	N119	476.00	1010.0
N111	197.00	958.00	N114	579.00	980.00	N117	646.00	987.00	N120	244.00	1013.0

Spans (.spans): ***SPAN/ SOURCE/ SINK***

S001	N001	N003	S037	N022	N023	S073	N045	N053	S109	N071	N077
S002	N001	N004	S038	N022	N032	S074	N046	N047	S110	N072	N073
S003	N001	N012	S039	N023	N029	S075	N047	N048	S111	N072	N076
S004	N002	N007	S040	N024	N034	S076	N049	N052	S112	N073	N074
S005	N002	N008	S041	N024	N035	S077	N050	N058	S113	N074	N075
S006	N002	N011	S042	N025	N028	S078	N050	N060	S114	N074	N080
S007	N003	N005	S043	N025	N036	S079	N051	N052	S115	N075	N087
S008	N003	N010	S044	N026	N040	S080	N051	N055	S116	N076	N079
S009	N004	N009	S045	N026	N041	S081	N053	N056	S117	N077	N085
S010	N004	N012	S046	N027	N030	S082	N053	N062	S118	N077	N086
S011	N005	N014	S047	N027	N033	S083	N054	N055	S119	N078	N081
S012	N005	N015	S048	N028	N039	S084	N054	N056	S120	N078	N083
S013	N006	N007	S049	N029	N031	S085	N054	N059	S121	N078	N090
S014	N006	N009	S050	N029	N032	S086	N055	N061	S122	N079	N083
S015	N006	N016	S051	N030	N038	S087	N056	N067	S123	N079	N084
S016	N007	N020	S052	N031	N033	S088	N057	N063	S124	N080	N081
S017	N008	N011	S053	N031	N037	S089	N057	N064	S125	N080	N082
S018	N008	N013	S054	N032	N039	S090	N057	N065	S126	N081	N090
S019	N009	N016	S055	N033	N038	S091	N058	N060	S127	N082	N087
S020	N010	N012	S056	N034	N045	S092	N058	N069	S128	N082	N092
S021	N010	N014	S057	N035	N036	S093	N059	N061	S129	N083	N089
S022	N011	N018	S058	N035	N046	S094	N059	N070	S130	N084	N085
S023	N013	N017	S059	N036	N047	S095	N060	N068	S131	N084	N088
S024	N013	N026	S060	N037	N043	S096	N061	N063	S132	N085	N096
S025	N014	N021	S061	N037	N044	S097	N062	N067	S133	N086	N095
S026	N015	N024	S062	N038	N042	S098	N062	N071	S134	N086	N096
S027	N015	N034	S063	N039	N048	S099	N063	N066	S135	N087	N100
S028	N016	N020	S064	N040	N041	S100	N064	N065	S136	N088	N089
S029	N017	N018	S065	N040	N049	S101	N064	N069	S137	N088	N093
S030	N017	N030	S066	N041	N050	S102	N065	N066	S138	N089	N097
S031	N018	N027	S067	N042	N044	S103	N066	N073	S139	N090	N098
S032	N019	N021	S068	N042	N049	S104	N067	N070	S140	N091	N092
S033	N019	N022	S069	N043	N048	S105	N068	N069	S141	N091	N098
S034	N019	N028	S070	N043	N051	S106	N068	N075	S142	N091	N099
S035	N020	N023	S071	N044	N052	S107	N070	N072	S143	N092	N094
S036	N021	N025	S072	N045	N046	S108	N071	N076	S144	N093	N096

S145	N093	N097	S154	N100	N109	S163	N105	N107	S172	N111	N118
S146	N094	N100	S155	N101	N102	S164	N105	N111	S173	N113	N116
S147	N094	N101	S156	N101	N109	S165	N106	N112	S174	N113	N119
S148	N095	N107	S157	N102	N117	S166	N106	N114	S175	N114	N117
S149	N095	N115	S158	N103	N108	S167	N107	N115	S176	N114	N119
S150	N097	N103	S159	N103	N112	S168	N108	N113	S177	N115	N118
S151	N098	N112	S160	N104	N105	S169	N109	N117	S178	N116	N120
S152	N099	N102	S161	N104	N108	S170	N110	N111	S179	N118	N120
S153	N099	N106	S162	N104	N110	S171	N110	N116	S180	N119	N120

130n195s Network

Nodes (.node): *NODE/ X/ Y*

N001	358.00	3.00	N029	393.00	225.00	N057	500.00	425.00	N085	461.00	685.00
N002	186.00	6.00	N030	575.00	237.00	N058	220.00	430.00	N086	556.00	687.00
N003	123.00	8.00	N031	472.00	238.00	N059	141.00	434.00	N087	156.00	694.00
N004	548.00	12.00	N032	247.00	245.00	N060	436.00	454.00	N088	4.00	702.00
N005	250.00	17.00	N033	101.00	248.00	N061	313.00	457.00	N089	270.00	705.00
N006	458.00	17.00	N034	184.00	248.00	N062	564.00	465.00	N090	373.00	708.00
N007	45.00	24.00	N035	708.00	252.00	N063	8.00	470.00	N091	709.00	710.00
N008	624.00	54.00	N036	304.00	258.00	N064	394.00	489.00	N092	44.00	728.00
N009	393.00	57.00	N037	639.00	271.00	N065	500.00	496.00	N093	513.00	744.00
N010	196.00	63.00	N038	41.00	285.00	N066	693.00	500.00	N094	434.00	746.00
N011	513.00	65.00	N039	356.00	297.00	N067	263.00	506.00	N095	244.00	768.00
N012	459.00	80.00	N040	124.00	305.00	N068	44.00	508.00	N096	664.00	768.00
N013	304.00	86.00	N041	556.00	305.00	N069	141.00	528.00	N097	136.00	782.00
N014	239.00	88.00	N042	489.00	317.00	N070	619.00	535.00	N098	76.00	785.00
N015	678.00	103.00	N043	4.00	327.00	N071	379.00	540.00	N099	353.00	785.00
N016	90.00	105.00	N044	296.00	337.00	N072	547.00	548.00	N100	566.00	791.00
N017	144.00	122.00	N045	616.00	340.00	N073	190.00	568.00	N101	292.00	803.00
N018	584.00	130.00	N046	714.00	351.00	N074	87.00	574.00	N102	473.00	805.00
N019	24.00	131.00	N047	210.00	360.00	N075	281.00	580.00	N103	190.00	837.00
N020	373.00	137.00	N048	396.00	360.00	N076	701.00	582.00	N104	407.00	837.00
N021	460.00	147.00	N049	56.00	368.00	N077	499.00	594.00	N105	6.00	839.00
N022	203.00	150.00	N050	333.00	372.00	N078	19.00	620.00	N106	676.00	842.00
N023	521.00	165.00	N051	543.00	377.00	N079	587.00	625.00	N107	499.00	849.00
N024	657.00	175.00	N052	464.00	382.00	N080	107.00	645.00	N108	304.00	871.00
N025	67.00	192.00	N053	604.00	400.00	N081	244.00	645.00	N109	374.00	877.00
N026	261.00	192.00	N054	284.00	411.00	N082	319.00	645.00	N110	79.00	888.00
N027	127.00	194.00	N055	397.00	416.00	N083	387.00	645.00	N111	184.00	888.00
N028	4.00	211.00	N056	690.00	422.00	N084	670.00	648.00	N112	613.00	888.00

N113	246.00	901.00	N118	661.00	951.00	N123	15.00	981.00	N128	390.00	1015.0
N114	524.00	905.00	N119	121.00	968.00	N124	461.00	985.00	N129	52.00	1018.0
N115	430.00	908.00	N120	377.00	968.00	N125	319.00	994.00	N130	200.00	1018.0
N116	711.00	910.00	N121	176.00	971.00	N126	553.00	994.00			
N117	347.00	922.00	N122	259.00	974.00	N127	643.00	1009.0			

Spans (.spans): ***SPAN/ SOURCE/ SINK***

S001	N001	N005	S035	N019	N028	S069	N043	N063	S103	N065	N077
S002	N001	N006	S036	N021	N023	S070	N044	N050	S104	N066	N076
S003	N001	N013	S037	N021	N029	S071	N044	N058	S105	N067	N073
S004	N002	N003	S038	N022	N026	S072	N045	N053	S106	N067	N075
S005	N002	N005	S039	N022	N027	S073	N046	N056	S107	N068	N069
S006	N002	N010	S040	N024	N035	S074	N047	N059	S108	N068	N078
S007	N003	N007	S041	N025	N027	S075	N048	N050	S109	N069	N073
S008	N003	N010	S042	N025	N028	S076	N048	N052	S110	N070	N072
S009	N004	N006	S043	N026	N029	S077	N048	N055	S111	N070	N076
S010	N004	N008	S044	N026	N032	S078	N049	N059	S112	N070	N079
S011	N004	N011	S045	N028	N043	S079	N049	N063	S113	N071	N075
S012	N005	N013	S046	N029	N031	S080	N049	N068	S114	N073	N074
S013	N006	N009	S047	N030	N031	S081	N050	N054	S115	N074	N080
S014	N007	N017	S048	N030	N037	S082	N051	N052	S116	N074	N088
S015	N007	N019	S049	N030	N041	S083	N051	N053	S117	N075	N081
S016	N008	N015	S050	N031	N039	S084	N051	N057	S118	N076	N084
S017	N008	N018	S051	N032	N034	S085	N053	N056	S119	N077	N079
S018	N009	N012	S052	N032	N036	S086	N054	N061	S120	N077	N083
S019	N009	N020	S053	N033	N034	S087	N054	N067	S121	N078	N088
S020	N010	N017	S054	N033	N038	S088	N055	N060	S122	N079	N084
S021	N011	N012	S055	N033	N040	S089	N055	N061	S123	N080	N087
S022	N011	N023	S056	N034	N047	S090	N056	N066	S124	N080	N092
S023	N012	N021	S057	N035	N046	S091	N057	N060	S125	N081	N082
S024	N013	N020	S058	N036	N039	S092	N057	N062	S126	N081	N089
S025	N014	N017	S059	N036	N044	S093	N058	N059	S127	N082	N083
S026	N014	N020	S060	N037	N045	S094	N058	N069	S128	N082	N090
S027	N014	N022	S061	N037	N046	S095	N060	N064	S129	N083	N085
S028	N015	N024	S062	N038	N040	S096	N061	N071	S130	N084	N091
S029	N015	N035	S063	N038	N043	S097	N062	N066	S131	N085	N093
S030	N016	N019	S064	N039	N042	S098	N062	N072	S132	N085	N094
S031	N016	N025	S065	N040	N047	S099	N063	N078	S133	N086	N093
S032	N016	N027	S066	N041	N042	S100	N064	N065	S134	N086	N096
S033	N018	N023	S067	N041	N045	S101	N064	N071	S135	N086	N100
S034	N018	N024	S068	N042	N052	S102	N065	N072	S136	N087	N089

S137	N087	N092	S152	N098	N105	S167	N109	N115	S182	N118	N127
S138	N088	N105	S153	N099	N102	S168	N109	N117	S183	N119	N123
S139	N089	N095	S154	N100	N107	S169	N110	N123	S184	N119	N129
S140	N090	N094	S155	N101	N103	S170	N111	N113	S185	N120	N124
S141	N090	N099	S156	N101	N104	S171	N111	N119	S186	N120	N125
S142	N091	N096	S157	N101	N108	S172	N111	N121	S187	N121	N122
S143	N091	N106	S158	N103	N110	S173	N112	N116	S188	N121	N130
S144	N092	N098	S159	N104	N107	S174	N113	N122	S189	N122	N125
S145	N093	N102	S160	N104	N109	S175	N114	N115	S190	N123	N129
S146	N094	N102	S161	N105	N110	S176	N114	N124	S191	N125	N128
S147	N095	N097	S162	N106	N112	S177	N114	N126	S192	N126	N127
S148	N095	N099	S163	N106	N116	S178	N115	N124	S193	N127	N128
S149	N096	N100	S164	N107	N112	S179	N116	N118	S194	N128	N130
S150	N097	N098	S165	N108	N113	S180	N117	N120	S195	N129	N130
S151	N097	N103	S166	N108	N117	S181	N118	N126			

140n210s Network

Nodes (.node): ***NODE/ X/ Y***

N001	478.00	5.00	N023	569.00	134.00	N045	541.00	298.00	N067	584.00	471.00
N002	376.00	8.00	N024	204.00	142.00	N046	246.00	302.00	N068	141.00	475.00
N003	134.00	9.00	N025	493.00	152.00	N047	627.00	308.00	N069	654.00	477.00
N004	220.00	15.00	N026	6.00	153.00	N048	684.00	325.00	N070	230.00	482.00
N005	45.00	17.00	N027	104.00	157.00	N049	496.00	332.00	N071	30.00	488.00
N006	677.00	19.00	N028	316.00	157.00	N050	339.00	335.00	N072	336.00	492.00
N007	559.00	20.00	N029	399.00	165.00	N051	431.00	340.00	N073	516.00	497.00
N008	276.00	25.00	N030	44.00	187.00	N052	16.00	355.00	N074	459.00	515.00
N009	434.00	37.00	N031	169.00	194.00	N053	74.00	358.00	N075	703.00	531.00
N010	100.00	52.00	N032	713.00	201.00	N054	147.00	368.00	N076	397.00	535.00
N011	496.00	54.00	N033	569.00	207.00	N055	243.00	368.00	N077	83.00	540.00
N012	634.00	55.00	N034	237.00	218.00	N056	601.00	392.00	N078	651.00	541.00
N013	357.00	57.00	N035	391.00	218.00	N057	667.00	408.00	N079	603.00	558.00
N014	153.00	70.00	N036	657.00	222.00	N058	391.00	410.00	N080	187.00	561.00
N015	536.00	87.00	N037	446.00	227.00	N059	516.00	410.00	N081	13.00	582.00
N016	254.00	91.00	N038	321.00	252.00	N060	713.00	412.00	N082	306.00	585.00
N017	19.00	96.00	N039	506.00	252.00	N061	36.00	415.00	N083	481.00	588.00
N018	693.00	102.00	N040	13.00	270.00	N062	193.00	427.00	N084	557.00	610.00
N019	69.00	103.00	N041	73.00	270.00	N063	300.00	427.00	N085	674.00	614.00
N020	401.00	107.00	N042	143.00	281.00	N064	6.00	445.00	N086	146.00	621.00
N021	459.00	109.00	N043	389.00	285.00	N065	76.00	451.00	N087	49.00	630.00
N022	641.00	132.00	N044	717.00	292.00	N066	453.00	457.00	N088	379.00	632.00

N089	237.00	654.00	N102	391.00	754.00	N115	364.00	877.00	N128	579.00	980.00
N090	4.00	674.00	N103	480.00	754.00	N116	259.00	888.00	N129	31.00	987.00
N091	540.00	675.00	N104	554.00	758.00	N117	159.00	892.00	N130	299.00	987.00
N092	447.00	682.00	N105	221.00	795.00	N118	469.00	895.00	N131	646.00	987.00
N093	621.00	685.00	N106	619.00	795.00	N119	46.00	897.00	N132	246.00	998.00
N094	319.00	691.00	N107	26.00	815.00	N120	341.00	912.00	N133	686.00	998.00
N095	703.00	699.00	N108	111.00	818.00	N121	637.00	918.00	N134	133.00	1002.0
N096	174.00	711.00	N109	316.00	820.00	N122	527.00	928.00	N135	486.00	1009.0
N097	109.00	722.00	N110	434.00	821.00	N123	702.00	934.00	N136	64.00	1015.0
N098	33.00	735.00	N111	530.00	821.00	N124	256.00	947.00	N137	396.00	1022.0
N099	663.00	738.00	N112	697.00	827.00	N125	197.00	958.00	N138	291.00	1023.0
N100	230.00	742.00	N113	611.00	867.00	N126	434.00	958.00	N139	570.00	1024.0
N101	289.00	744.00	N114	533.00	872.00	N127	386.00	975.00	N140	181.00	1028.0

Spans (.spans): ***SPAN/ SOURCE/ SINK***

S001	N001	N002	S028	N017	N019	S055	N034	N038	S082	N051	N058
S002	N001	N007	S029	N017	N026	S056	N034	N046	S083	N052	N061
S003	N001	N009	S030	N018	N022	S057	N035	N037	S084	N052	N064
S004	N002	N008	S031	N018	N032	S058	N035	N038	S085	N053	N054
S005	N002	N013	S032	N019	N027	S059	N037	N039	S086	N053	N061
S006	N003	N004	S033	N020	N021	S060	N037	N043	S087	N054	N055
S007	N003	N005	S034	N020	N028	S061	N038	N046	S088	N055	N062
S008	N003	N010	S035	N020	N029	S062	N039	N045	S089	N056	N057
S009	N004	N008	S036	N021	N023	S063	N039	N049	S090	N057	N069
S010	N004	N014	S037	N021	N025	S064	N040	N041	S091	N058	N063
S011	N005	N010	S038	N022	N036	S065	N040	N052	S092	N059	N066
S012	N005	N017	S039	N023	N033	S066	N041	N042	S093	N059	N067
S013	N006	N007	S040	N024	N027	S067	N042	N053	S094	N059	N073
S014	N006	N012	S041	N024	N028	S068	N043	N050	S095	N060	N075
S015	N006	N018	S042	N024	N031	S069	N043	N051	S096	N061	N065
S016	N007	N011	S043	N025	N029	S070	N044	N048	S097	N062	N063
S017	N008	N016	S044	N025	N033	S071	N044	N060	S098	N062	N065
S018	N009	N011	S045	N026	N030	S072	N045	N047	S099	N063	N066
S019	N009	N013	S046	N026	N040	S073	N045	N056	S100	N064	N065
S020	N010	N019	S047	N028	N034	S074	N046	N054	S101	N064	N071
S021	N011	N015	S048	N029	N035	S075	N047	N048	S102	N066	N073
S022	N012	N015	S049	N030	N031	S076	N047	N057	S103	N067	N069
S023	N012	N022	S050	N030	N041	S077	N048	N060	S104	N067	N084
S024	N013	N016	S051	N031	N042	S078	N049	N051	S105	N068	N070
S025	N014	N016	S052	N032	N036	S079	N049	N056	S106	N068	N077
S026	N014	N027	S053	N032	N044	S080	N050	N055	S107	N068	N080
S027	N015	N023	S054	N033	N036	S081	N050	N058	S108	N069	N079

S109	N070	N072	S135	N088	N092	S161	N104	N106	S187	N121	N131
S110	N070	N082	S136	N088	N094	S162	N105	N116	S188	N121	N133
S111	N071	N077	S137	N089	N096	S163	N105	N117	S189	N122	N128
S112	N071	N081	S138	N090	N097	S164	N107	N108	S190	N123	N133
S113	N072	N074	S139	N090	N098	S165	N107	N119	S191	N124	N125
S114	N072	N076	S140	N091	N092	S166	N109	N116	S192	N124	N134
S115	N073	N074	S141	N092	N103	S167	N110	N111	S193	N125	N134
S116	N074	N083	S142	N093	N099	S168	N110	N115	S194	N126	N127
S117	N075	N078	S143	N093	N104	S169	N110	N118	S195	N127	N130
S118	N075	N085	S144	N093	N106	S170	N111	N113	S196	N128	N131
S119	N076	N082	S145	N094	N096	S171	N111	N114	S197	N128	N139
S120	N076	N083	S146	N094	N102	S172	N112	N113	S198	N129	N136
S121	N077	N087	S147	N095	N099	S173	N112	N123	S199	N130	N132
S122	N078	N079	S148	N095	N112	S174	N113	N123	S200	N130	N135
S123	N078	N091	S149	N096	N097	S175	N114	N118	S201	N131	N133
S124	N079	N084	S150	N097	N108	S176	N114	N122	S202	N132	N138
S125	N080	N082	S151	N098	N107	S177	N115	N120	S203	N132	N140
S126	N080	N086	S152	N098	N108	S178	N115	N126	S204	N134	N136
S127	N081	N087	S153	N099	N106	S179	N116	N125	S205	N135	N137
S128	N081	N090	S154	N100	N101	S180	N117	N119	S206	N135	N139
S129	N083	N084	S155	N100	N105	S181	N117	N129	S207	N136	N140
S130	N085	N091	S156	N100	N109	S182	N118	N126	S208	N137	N138
S131	N085	N095	S157	N101	N102	S183	N119	N129	S209	N137	N139
S132	N086	N087	S158	N101	N109	S184	N120	N124	S210	N138	N140
S133	N086	N089	S159	N102	N103	S185	N120	N127			
S134	N088	N089	S160	N103	N104	S186	N121	N122			

APPENDIX C

Model and Data Files for Spare Capacity Allocation

Model File (.mod):

```
# p-cycle SCP IP Model for AMPL - Version 1.0
# 11-December-2001 by John Doucette
# Copyright (C) 2001 TRLabs, Inc. All Rights Reserved.

# *******************************************************
# TRLabs
# 7th Floor
# 9107 116 Street NW
# Edmonton, Alberta, Canada
# T6G 2V4
# +1 780 441-3800
# www.trlabs.ca
# *******************************************************

# This model, including any data and algorithms contained herein, is the
# exclusive property of TRLabs, held on behalf of its sponsors. Except
# as specifically authorized in writing by TRLabs, the recipient of this
# model shall keep it confidential and shall protect it in whole or
# in part from disclosure and dissemenation to all third parties, and the
# associated readme file must accompany any such disclosure or dissemenation.
# If any part of this model, including any data and algorithms contained
# herein, is used in any derivative works or publications, TRLabs shall be
# duly cited as a reference.Recommended citation is as follows:
# J. Doucette, W. D. Grover, "pcycle.SCP.mod: p-Cycle SCP IP Model for
# AMPL - Version 1.0," TRLabs proprietary AMPL ILP model, Edmonton, AB,
# December 2001.
# TRLabs makes no representation or warranties about the suitability of
# this model, either express or implied, including but not limited to
# implied warranties of merchantability, fitness for a particular purpose,
# or non-infringement. TRLabs shall not be liable for any damages suffered
# as a result of using, modifying or distributing this model or its derivatives.
#*****************************
# This is an AMPL model for determining the minimum-cost p-cycle network design.
# This model optimizes p-cycles only... working capacity is provided as inputs.
#*****************************

#*****************************
# SETS
#*****************************
set SPANS;
# Set of all spans.

set PCYCLES;
# Set of all p-cycles.

#*****************************
# PARAMETERS
#*****************************
param Cost{j in SPANS};
# Cost of each unit of capacity on span j.

param Work{j in SPANS};
# Number of working links placed on span j.

param Xpi{p in PCYCLES, i in SPANS} default 0;
# Number of paths a single copy of p-cycle p provides for restoration of failure of
```

```
# span i (2 if straddling span, 1 if on-cycle span, 0 otherwise).

param pCrossesj{p in PCYCLES, j in SPANS} := sum{i in SPANS: i = j and Xpi[p,j] = 1}
1;
# Equal to 1 if p-cycle p passes over span j, 0 otherwise.
# i.e. if Xpi[p,j] = 1, then p-cycle p crosses span j.

#*****************************
# VARIABLES
#*****************************
var p_cycle_usage{p in PCYCLES} >=0 integer, <=10000;
# Number of copies of p-cycle p used.

var spare{j in SPANS} >=0 integer, <=10000;
# Number of spare links placed on span j.

#*****************************
# OBJECTIVE FUNCTION
#*****************************
minimize sparecost: sum{j in SPANS} Cost[j] * spare[j];

#*****************************
# CONSTRAINTS
#*****************************
subject to full_restoration{i in SPANS}:
Work[i] <= sum{p in PCYCLES} Xpi[p,i] * p_cycle_usage[p];

subject to spare_capacity_placement{j in SPANS}:
spare[j] = sum{p in PCYCLES} pCrossesj[p,j] * p_cycle_usage[p];
```

Data File (.dat): *p-cycles display only 10 out of 723*

```
set SPANS :=
S01                     S13                     S25                     S37
S02                     S14                     S26                     S38
S03                     S15                     S27                     S39
S04                     S16                     S28                     S40
S05                     S17                     S29                     S41
S06                     S18                     S30                     S42
S07                     S19                     S31                     S43
S08                     S20                     S32                     S44
S09                     S21                     S33                     S45
S10                     S22                     S34                     ;
S11                     S23                     S35
S12                     S24                     S36

param Cost :=
S01     249.05          S13     141.241         S25     228.668         S37     150.233
S02     163.15          S14     171.143         S26     214.87          S38     239.927
S03     141.566         S15     560.514         S27     111.018         S39     129.468
S04     139.216         S16     150.881         S28     172.792         S40     190.0
S05     161.555         S17     147.801         S29     168.808         S41     135.949
S06     92.195          S18     166.688         S30     105.802         S42     167.386
S07     186.304         S19     242.074         S31     110.318         S43     213.965
S08     123.167         S20     132.544         S32     210.723         S44     198.494
S09     209.881         S21     166.604         S33     224.437         S45     186.172
S10     117.614         S22     145.152         S34     172.177         ;
S11     184.567         S23     167.601         S35     152.751
S12     139.086         S24     166.048         S36     139.818

#Total Working Capacity Cost = 8993.000000

param Work :=
S01     209             S05     85              S09     195             S13     151
S02     189             S06     30              S10     138             S14     130
S03     43              S07     190             S11     197             S15     382
S04     278             S08     395             S12     129             S16     309
```

```
S17     397             S25     148             S33     268             S41     91
S18     450             S26     237             S34     251             S42     105
S19     317             S27     169             S35     134             S43     173
S20     42              S28     157             S36     188             S44     265
S21     112             S29     128             S37     79              S45     255
S22     342             S30     120             S38     81              ;
S23     343             S31     119             S39     290
S24     384             S32     88              S40     210

set PCYCLES :=
P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12 P13 P14 P15 P16 P17 P18 P19 P20 P21 P22 P23 P24 P25 P26
P27 P28 P29 P30 P31 P32 P33 P34 P35 P36 P37 P38 P39 P40 P41 P42 P43 P44 P45 P46 P47 P48 P49 P50
P51 P52 P53 P54 P55 P56 P57 P58 P59 P60 P61 P62 P63 P64 P65 P66 P67 P68 P69 P70 P71 P72 P73 P74
P75 P76 P77 P78 P79 P80 P81 P82 P83 P84 P85 P86 P87 P88 P89 P90 P91 P92 P93 P94 P95 P96 P97 P98
P99 P100 P101 P102 P103 P104 P105 P106 P107 P108 P109 P110 P111 P112 P113 P114 P115 P116 P117
P118 P119 P120 P121 P122 P123 P124 P125 P126 P127 P128 P129 P130 P131 P132 P133 P134 P135 P136
P137 P138 P139 P140 P141 P142 P143 P144 P145 P146 P147 P148 P149 P150  P151 P152 P153 P154 P155
P156 P157 P158 P159 P160 P161 P162 P163 P164 P165 P166 P167 P168 P169 P170 P171 P172 P173 P174
P175 P176 P177 P178 P179 P180 P181 P182 P183 P184 P185 P186 P187 P188 P189 P190 P191 P192 P193
P194 P195 P196 P197 P198 P199 P200 P201 P202 P203 P204 P205 P206 P207 P208 P209 P210 P211 P212
P213 P214 P215 P216 P217 P218 P219 P220 P221 P222 P223 P224 P225 P226 P227 P228 P229 P230 P231
P232 P233 P234 P235 P236 P237 P238 P239 P240 P241 P242 P243 P244 P245 P246 P247 P248 P249 P250
P251 P252 P253 P254 P255 P256 P257 P258 P259 P260 P261 P262 P263 P264 P265 P266 P267 P268 P269
P270 P271 P272 P273 P274 P275 P276 P277 P278 P279 P280 P281 P282 P283 P284 P285 P286 P287 P288
P289 P290 P291 P292 P293 P294 P295 P296 P297 P298 P299 P300 P301 P302 P303 P304 P305 P306 P307
P308 P309 P310 P311 P312 P313 P314 P315 P316 P317 P318 P319 P320 P321 P322 P323 P324 P325 P326
P327 P328 P329 P330 P331 P332 P333 P334 P335 P336 P337 P338 P339 P340 P341 P342 P343 P344 P345
P346 P347 P348 P349 P350 P351 P352 P353 P354 P355 P356 P357 P358 P359 P360 P361 P362 P363 P364
P365 P366 P367 P368 P369 P370 P371 P372 P373 P374 P375 P376 P377 P378 P379 P380 P381 P382 P383
P384 P385 P386 P387 P388 P389 P390 P391 P392 P393 P394 P395 P396 P397 P398 P399 P400 P401 P402
P403 P404 P405 P406 P407 P408 P409 P410 P411 P412 P413 P414 P415 P416 P417 P418 P419 P420 P421
P422 P423 P424 P425 P426 P427 P428 P429 P430 P431 P432 P433 P434 P435 P436 P437 P438 P439 P440
P441 P442 P443 P444 P445 P446 P447 P448 P449 P450 P451 P452 P453 P454 P455 P456 P457 P458 P459
P460 P461 P462 P463 P464 P465 P466 P467 P468 P469 P470 P471 P472 P473 P474 P475 P476 P477 P478
P479 P480 P481 P482 P483 P484 P485 P486 P487 P488 P489 P490 P491 P492 P493 P494 P495 P496 P497
P498 P499 P500 P501 P502 P503 P504 P505 P506 P507 P508 P509 P510 P511 P512 P513 P514 P515 P516
P517 P518 P519 P520 P521 P522 P523 P524 P525 P526 P527 P528 P529 P530 P531 P532 P533 P534 P535
P536 P537 P538 P539 P540 P541 P542 P543 P544 P545 P546 P547 P548 P549 P550 P551 P552 P553 P554
P555 P556 P557 P558 P559 P560 P561 P562 P563 P564 P565 P566 P567 P568 P569 P570 P571 P572 P573
P574 P575 P576 P577 P578 P579 P580 P581 P582 P583 P584 P585 P586 P587 P588 P589 P590 P591 P592
P593 P594 P595 P596 P597 P598 P599 P600 P601 P602 P603 P604 P605 P606 P607 P608 P609 P610 P611
P612 P613 P614 P615 P616 P617 P618 P619 P620 P621 P622 P623 P624 P625 P626 P627 P628 P629 P630
P631 P632 P633 P634 P635 P636 P637 P638 P639 P640 P641 P642 P643 P644 P645 P646 P647 P648 P649
P650 P651 P652 P653 P654 P655 P656 P657 P658 P659 P660 P661 P662 P663 P664 P665 P666 P667 P668
P669 P670 P671 P672 P673 P674 P675 P676 P677 P678 P679 P680 P681 P682 P683 P684 P685 P686 P687
P688 P689 P690 P691 P692 P693 P694 P695 P696 P697 P698 P699 P700 P701 P702 P703 P704 P705 P706
P707 P708 P709 P710 P711 P712 P713 P714 P715 P716 P717 P718 P719 P720 P721 P722 P723;

param Xpi :=
[P1, *] S02 1 S03 1 S07 1 S12 1 S13 1 S06 2
[P2, *] S29 1 S30 1 S32 1 S33 1 S38 1 S43 1 S37 2
[P3, *] S01 1 S02 1 S05 1 S06 1 S10 1 S12 1 S03 2
[P4, *] S35 1 S36 1 S39 1 S40 1 S42 1 S44 1 S41 2
[P5, *] S01 1 S03 1 S04 1 S09 1 S10 1 S11 1 S05 2
[P6, *] S14 1 S15 1 S21 1 S27 1 S28 1 S35 1 S20 2
[P7, *] S08 1 S09 1 S14 1 S16 1 S19 1 S20 1 S27 1 S21 2 1
[P8, *] S25 1 S26 1 S30 1 S31 1 S32 1 S33 1 S37 1 S29
[P9, *] S22 1 S23 1 S24 1 S26 1 S29 1 S30 1 S31 1 S25
[P10, *] S31 1 S32 1 S34 1 S37 1 S42 1 S43 1 S45 1 S38 2
…
[P722, *] S31 1 S32 1 S34 1 S38 1 S42 1 S45 1;
```

APPENDIX D.1
Python Model for DDCD

```
###############################################################################
# This code enumerate pcycles using DDCD method (Disjoit-paths Dijkstra Cycle Development)
# Most recently revised on Oct 12, 2020
###############################################################################

import random
import copy
import operator
import math
import datetime
import time
import sys
from collections import deque, defaultdict
from itertools import groupby
from Dijkstra_CIDA import *
    #To get shortest path: shortest_path(graph, 'A', 'D')

###############################################################################
# Input parameters to be entered in the cmd line
input_topo_name = ='30n45s'
ddcd_iteration_num = 4
output_file_name = "30n45s_DDD3"
###############################################################################


start = time.time()

### Graph Initiation
graph = Graph()

# Caulcate euc distance given two nodes
def euc_distance(x, y):
   dist = math.sqrt((y[1]-x[1])**2 + (y[0]-x[0])**2)
   final_val = '%.3f' % dist
   return final_val

nodes = {} # all nodes with X, Y coordinates
with open(input_topo_name + '.node', 'r') as f:
    for line in f.readlines():
        comp = line.strip().split()
        assert len(comp) == 4
        nodes[comp[0]] = [float(comp[1]), float(comp[2])]

for node in nodes.keys():
    graph.add_node(node)

span_node = {} # span id : both end nodes
node_span = {} # both end nodes : span id
span_wt = {} # span id: value: cost
with open(input_topo_name + '.spans', 'r') as f:
    for line in f.readlines():
        comp = line.strip().split()
        assert len(comp) == 8
        # When span costs are 1:
        # dist = 1

        # When span costs are their euc distances:
        dist = euc_distance(nodes[comp[1]], nodes[comp[2]])
        graph.add_edge(comp[1], comp[2], float(dist))

        span_node[comp[0]] = [comp[1], comp[2]]
        node_span[comp[1] + " , " + comp[2]] = comp[0]
        node_span[comp[2] + " , " + comp[1]] = comp[0]
        span_wt[comp[0]] = float(dist)
```

```
### Enumerating set of pcycles/simple cycles
    # Generate shortest path and associated pcycles using Dijkstra's Algorithm
    # sindexer = {} # span id to pcycle id
def Enumerate_Start_Cycles(nodes, span_node, node_span):
    pcycles = {}
    new_id = 0
    for sid, spanner in span_node.items():
        N1 = spanner[0]
        N2 = spanner[1]
        cost_shortest, path_shortest, cost_shortest2, path_shortest2 = double_shortestPath(graph,
N1, N2)
        if cost_shortest == sys.maxsize:
            continue
        cycle_span_info = {}
        for node_start, node_end in zip(path_shortest[:-1], path_shortest[1:]):
            span_ider = node_span[node_start + " , " + node_end]
            cycle_span_info[span_ider] = 1
        cycle_span_info[sid] = 1
        cycle_id = "Cycle " + str(new_id) # new cycle id given
        # sindexer[sid] = cycle_id
        pcycles[cycle_id] = cycle_span_info
        new_id += 1

        if cost_shortest2 == sys.maxsize:
            continue
        cycle_span_info2 = {}
        for node_start2, node_end2 in zip(path_shortest2[:-1], path_shortest2[1:]):
            span_ider2 = node_span[node_start2 + " , " + node_end2]
            cycle_span_info2[span_ider2] = 1
        pcycle_info = {**cycle_span_info, **cycle_span_info2}
        pcycle_info[sid] = 2
        pcycle_id = "PCycle " + str(new_id) # new pcycle id given
        # sindexer[sid] = pcycle_id
        pcycles[pcycle_id] = pcycle_info
        new_id += 1
    return pcycles

# Stronger grow function where more shortest paths are found from span i
def DDCD(input_cycle_set, graph, new_id2):
    # new_id2 = 0
    new_cycle_set = {}
    for sp in input_cycle_set.values():
        sp_p = sp
        remove_nodes = []
        for i in sp_p.keys():
            remove_nodes.append(span_node[i][0])
            remove_nodes.append(span_node[i][1])
        for ii, val in sp_p.items():
            if val == 2:
                continue
            remove_node_set = set(remove_nodes)
            Nor = span_node[ii][0]
            Ndt = span_node[ii][1]
            remove_node_set -= {Nor}
            remove_node_set -= {Ndt}
            remove_node_list = list(remove_node_set)

            if (Nor, Ndt) in graph.distances:
                graph.distances[(Nor, Ndt)] = sys.maxsize
                graph.distances[(Ndt, Nor)] = sys.maxsize
            else:
                raise ValueError("Not possible")

            while remove_node_list:
                visited2, paths2 = dijkstra_2(graph, Nor, remove_node_list)
                if not paths2 or visited2[Ndt] == sys.maxsize:
                    break
                full_path2 = deque()
                _destination2 = paths2[Ndt]

                while _destination2 != Nor:
                    full_path2.appendleft(_destination2)
```

```
                    _destination2 = paths2[_destination2]

                remove_node_list.extend(full_path2)
                remove_node_list = list(set(remove_node_list))
                full_path2.appendleft(Nor)
                full_path2.append(Ndt)
                shortPath2 = list(full_path2)

                cycle_span2 = {}
                for node_st2, node_ed2 in zip(shortPath2[:-1], shortPath2[1:]):
                    sp_ider2 = node_span[node_st2 + " , " + node_ed2]
                    cycle_span2[sp_ider2] = 1
                pcycle_inf = {**sp_p, **cycle_span2}
                pcycle_inf[ii] = 2
                pcyc_id = "PCycleGr " + str(new_id2) # new pcycle id given
                # sindexer[sid] = pcycle_id
                new_cycle_set[pcyc_id] = pcycle_inf
                new_id2 += 1

            if (Nor, Ndt) in graph.distances:
                graph.distances[(Nor, Ndt)] = 1
                graph.distances[(Ndt, Nor)] = 1
            else:
                raise ValueError("Not possible")
    return new_cycle_set

# Adding all straddling links info to all the enumerated cycles
def Straddle_Link(cycles_set_update):
    updated_cycle_set = cycles_set_update
    for cycle_key, sp_info in updated_cycle_set.items():
        all_nodes_on_this_cycle = []
        for ss, jj in sp_info.items():
            if jj == 1:
                node1_for_ss = span_node[ss][0]
                node2_for_ss = span_node[ss][1]
                all_nodes_on_this_cycle.append(node1_for_ss)
                all_nodes_on_this_cycle.append(node2_for_ss)
        nodeset = set(all_nodes_on_this_cycle)
        nodes_on_this_cycle = list(nodeset)

        for nn, dd in node_span.items():
            nn_nodes = nn.split(' , ')
            if nn_nodes[0] in nodes_on_this_cycle and nn_nodes[1] in nodes_on_this_cycle and dd
not in sp_info:
                sp_info[dd] = 2
                updated_cycle_set[cycle_key] = sp_info

    return updated_cycle_set

# Remove redundant pCycles from a collection of enumerated pcycles
def Remove_Redundancy_NEW(pcycles):
    # all_cycles_list_key = list(pcycles.keys())
    # all_cycles_list = list(pcycles.values())
    final_res = {}
    pcycle_cost_sum = defaultdict(int) #key: pcycle's key; value: cost

    for pkey, val in pcycles.items():
        for span_id, val_val in val.items():
            if val_val == 1:
                pcycle_cost_sum[pkey] += int(span_wt[span_id])

    keyfunc = lambda item: item[1]
    for k_len, same_pkey_group in groupby(sorted(pcycle_cost_sum.items(), key=keyfunc), keyfunc):
        lister = list(same_pkey_group)
        #print(lister)
        for ii, same_pkey1 in enumerate(lister):
            flag_dup = False
            item1 = set(spanid for spanid, ii in pcycles[same_pkey1[0]].items() if ii == 1)
            for same_pkey2 in lister[ii+1:]:
                item2 = set(spanid for spanid, ii in pcycles[same_pkey2[0]].items() if ii == 1)
                if item1 == item2:
                    flag_dup = True
```

```
                    break
            if not flag_dup:
                final_res[same_pkey1[0]] = pcycles[same_pkey1[0]]
    return final_res

### Main Function:

now = datetime.datetime.now()
print(now)

pcycles = Enumerate_Start_Cycles(nodes, span_node, node_span)
prev_limit = 0
# ccc = 0
for ind in range(ddcd_iteration_num):
    cy_id1 = 0
    pcycles_from_Grow = DDCD(pcycles, graph, cy_id1)
    pcycles_set = {**pcycles_from_Grow, **pcycles}
    cy_id1 = cy_id1 + 1000000
    # Check and remove redundant cycles in the cycle set
    pcycles = Remove_Redundancy_NEW(pcycles_set)
    if len(pcycles) == prev_limit:
        print(f'No More New Cycles. Exit! at Iteration: {ind} | {len(pcycles)}')
        break

    prev_limit = len(pcycles)
# Complete straddling link information on all pCycles
updated_cycle_set = Straddle_Link(pcycles)

### Verify all enumerated pCycles
# pCycle_Verify(updated_cycle_set, span_node, node_span)

### Verify and remove duplicate cycles
updated_cycle_set = Remove_Redundancy_NEW(updated_cycle_set)

the_end = time.time()
print('Total Runtime = ' + str(the_end - start))

### Print to .pcycle file
text_file = open(str(output_file_name) + '.pcycle', 'w')
for key, value in updated_cycle_set.items():
    text_file.write(key + ": ")
    for kk, tt in value.items():
        text_file.write(kk + " " + str(tt) + " ")
    text_file.write("\n")
text_file.close()

### Write a .eucost file to save all euclidean distances as span costs
txt_file = open(str(output_file_name) + '.eucost', 'w')
for kk, val in span_wt.items():
    txt_file.write(kk + "\t" + str(val))
    txt_file.write("\n")
txt_file.close()
```

Dijkstra_CIDA.py *(this is used in both DDCD & GA-SCA .py models)*

```
from collections import defaultdict, deque
import sys
import copy


class Graph(object):
    def __init__(self):
        self.nodes = set()
        self.edges = defaultdict(list)
        self.distances = {}

    def add_node(self, value):
        self.nodes.add(value)
```

```
    def add_edge(self, from_node, to_node, distance):
        self.edges[from_node].append(to_node)
        self.edges[to_node].append(from_node)
        self.distances[(from_node, to_node)] = distance
        self.distances[(to_node, from_node)] = distance

def dijkstra(graph, initial):
    visited = {initial: 0}
    path = {}

    nodes = set(graph.nodes)

    while nodes:
        min_node = None
        for node in nodes:
            if node in visited:
                if min_node is None:
                    min_node = node
                elif visited[node] < visited[min_node]:
                    min_node = node
        if min_node is None:
            break

        nodes.remove(min_node)
        current_weight = visited[min_node]

        for edge in graph.edges[min_node]:
            try:
                weight = current_weight + graph.distances[(min_node, edge)]
            except:
                raise ValueError("Sum error!")
            if edge not in visited or weight < visited[edge]:
                visited[edge] = weight
                path[edge] = min_node

    return visited, path

def dijkstra_2(graph, initial, remove_nodes):
    visited = {initial: 0}
    path = {}

    nodes = set(graph.nodes)
    for node_rm in remove_nodes:
        nodes.remove(node_rm)

    while nodes:
        min_node = None
        for node in nodes:
            if node in visited:
                if min_node is None:
                    min_node = node
                elif visited[node] < visited[min_node]:
                    min_node = node
        if min_node is None:
            break

        nodes.remove(min_node)
        current_weight = visited[min_node]

        for edge in graph.edges[min_node]:
            if edge in remove_nodes:
                continue
            try:
                weight = current_weight + graph.distances[(min_node, edge)]
            except:
                raise ValueError("Sum error!")
            if edge not in visited or weight < visited[edge]:
                visited[edge] = weight
                path[edge] = min_node

    return visited, path
```

```
def shortest_path(graph, origin, destination):
    if (origin, destination) in graph.distances:
        graph.distances[(origin, destination)] = sys.maxsize
        graph.distances[(destination, origin)] = sys.maxsize
    else:
        raise ValueError("Not possible")

    visited, paths = dijkstra(graph, origin)
    full_path = deque()
    _destination = paths[destination]

    while _destination != origin:
        full_path.appendleft(_destination)
        _destination = paths[_destination]

    full_path.appendleft(origin)
    full_path.append(destination)

    if (origin, destination) in graph.distances:
        graph.distances[(origin, destination)] = 1
        graph.distances[(destination, origin)] = 1
    else:
        raise ValueError("Not possible")

    return visited[destination], list(full_path)

def double_shortestPath(graph, origin, destination):
    if (origin, destination) in graph.distances:
        graph.distances[(origin, destination)] = sys.maxsize
        graph.distances[(destination, origin)] = sys.maxsize
    else:
        raise ValueError("Not possible")

    visited, paths = dijkstra(graph, origin)
    full_path = deque()
    _destination = paths[destination]

    while _destination != origin:
        full_path.appendleft(_destination)
        _destination = paths[_destination]

    full_path.appendleft(origin)
    full_path.append(destination)

    remove_nodes = list(full_path)[1:-1]

    visited2, paths2 = dijkstra_2(graph, origin, remove_nodes)
    full_path2 = deque()
    _destination2 = paths2[destination]

    while _destination2 != origin:
        full_path2.appendleft(_destination2)
        _destination2 = paths2[_destination2]

    full_path2.appendleft(origin)
    full_path2.append(destination)

    if (origin, destination) in graph.distances:
        graph.distances[(origin, destination)] = 1
        graph.distances[(destination, origin)] = 1
    else:
        raise ValueError("Not possible")

    return visited[destination], list(full_path), visited2[destination], list(full_path2)
```

APPENDIX D.2
Python Model for GA-SCA

```
import random
import time
import copy
import operator
import math
import datetime
import sys
from Dijkstra_CIDA import *
import cProfile

# Input parameters to be entered in the cmd line
topo_name = '30n45s'
pcycle_file = '30n45s_DDD.pcycle'
Selection_Method = 'select_tournament'
Crossover_Type = 'two_point'
Mutation_Type = 'mutate1'
input_pop = 50
input_cr = 0.35
input_mr = 0.2
output_pcycle = 'Test30n45s.pcycle'


# Graph Initiation
graph = Graph()

# global mutation ID
new_id_mut = 0

# Caulcate euc distance given two nodes
def euc_distance(x, y):
   dist = math.sqrt((y[1]-x[1])**2 + (y[0]-x[0])**2)
   return dist

# All node information
nodes = {}
with open(topo_name + '.node', 'r') as f:
    for line in f.readlines():
        comp = line.strip().split()
        assert len(comp) == 4
        nodes[comp[0]] = [float(comp[1]), float(comp[2])]
for node in nodes.keys():
    graph.add_node(node)

# All span-node correlations
span_node = {}
node_span = {}
span_dist = {}
with open(topo_name + '.spans', 'r') as f:
    for line in f.readlines():
        comp = line.strip().split()
        assert len(comp) == 8
        # When span costs are their euc distances:
        dist = euc_distance(nodes[comp[1]], nodes[comp[2]])
        span_dist[comp[0]] = float(format(dist, '.3f'))
        graph.add_edge(comp[1], comp[2], float(format(dist, '.3f'))) # comp[1] = origin, comp[2]
= destination, float(dist) = span weight or cost
        span_node[comp[0]] = [comp[1], comp[2]]
        node_span[comp[1] + " , " + comp[2]] = comp[0]
        node_span[comp[2] + " , " + comp[1]] = comp[0]

# Open SOL file with detailed span information
span_ft = {}
with open(topo_name + '.eucspanft', 'r') as f:
    for line in f.readlines():
        comp = line.split()
```

```
        assert len(comp) == 4
        # span_ft[comp[0]] = [float(comp[1]), float(comp[2])]
        span_ft[comp[0]] = [float(span_dist[comp[0]]), float(comp[2])]
spans = span_ft

# Open p-cycles file generated from SOL file
pcycles = {}
n = 0
with open(pcycle_file, 'r') as f:
    for _line in f.readlines():
        item = _line.split(':')
        assert len(item) == 2
        cycle_span_info = {}
        span_info = item[1].split()
        assert len(span_info) % 2 == 0
        for n in range(len(span_info)):
            if n % 2 == 0:
                cycle_span_info[span_info[n]] = int(span_info[n+1])
        pcycles[item[0]] = cycle_span_info

def calculate_ew(spans, cycle_value):
    # return a ew value given a pcycle
    # calcuate (wi * xpi)/cost
    ew1 = 0
    ew2 = 0
    for _k, _v in cycle_value.items():
        wi = spans[_k][1]
        spi = _v
        ew1 = ew1 + wi * spi
        if _v == 1:
            ew2 = ew2 + spans[_k][0]
    ew = ew1/ew2
    return ew

def index_sspans_from_pcycles(spans, pcycles):
    span_pcycle_indexer = {}
    # Step 1 Calculate ew score for each pcycle
    pcycle_ew_dict = {}
    for p_id, p_val in pcycles.items():
        pcycle_ew_dict[p_id] = calculate_ew(spans, p_val)
    # Step 2 Get span_id's pcycle list
    for span_id in spans.keys():
        for key, value in pcycles.items():
            if span_id in value:
                if span_id not in span_pcycle_indexer:
                    span_pcycle_indexer[span_id] = [(key, pcycle_ew_dict[key])]
                else:
                    span_pcycle_indexer[span_id].append((key, pcycle_ew_dict[key]))
    # Step 3 Sort pcycle list for each span
    for span_id in span_pcycle_indexer.keys():
        # e is a tuple: (key, pcycle_ew_dict[key])
        span_pcycle_indexer[span_id] = sorted(span_pcycle_indexer[span_id], key=lambda e: e[1],
reverse=True)
    return span_pcycle_indexer, pcycle_ew_dict
span_pcycle_indexer, pcycle_ew_dict = index_spans_from_pcycles(spans, pcycles)

def index_straddle_from_pcycles(spans, pcycles):
    straddle_pcycle_indexer = {}
    # Step 1 Calculate ew score for each pcycle
    pcycle_ew_dict = {}
    for p_id, p_val in pcycles.items():
        pcycle_ew_dict[p_id] = calculate_ew(spans, p_val)
    # Step 2 Get span_id's pcycle list
    for span_id in spans.keys():
        for key, value in pcycles.items():
            if span_id in value and value[span_id] == 2:
                if span_id not in straddle_pcycle_indexer:
                    straddle_pcycle_indexer[span_id] = [(key, pcycle_ew_dict[key])]
                else:
                    straddle_pcycle_indexer[span_id].append((key, pcycle_ew_dict[key]))
    # Step 3 Sort pcycle list for each span
    for span_id in straddle_pcycle_indexer.keys():
```

```
        straddle_pcycle_indexer[span_id] = sorted(straddle_pcycle_indexer[span_id], key=lambda e:
e[1], reverse=True)
    return straddle_pcycle_indexer, pcycle_ew_dict
straddle_pcycle_indexer, pcycle_ew2_dict = index_straddle_from_pcycles(spans, pcycles)

def calculate_p_cost(spans, pcycles):
    pcycle_costs = {}
    for k,v in pcycles.items():
        pcycle_cost = 0
        for kkk,vvv in v.items():
            if vvv == 1:
                _cost = int(spans[kkk][0])
                pcycle_cost = pcycle_cost + _cost
        pcycle_costs[k] = pcycle_cost
    return pcycle_costs
pcycle_costs_global = calculate_p_cost(spans, pcycles)

## Methods for Repair Genome: ##
# Minimum cycles for Repair Genome
def Min_Cycle_Repair(spans, pcycles, span_node, node_span):
    cycle_usage = {}
    new_id2 = 0
    for span_id in spans.keys():
        while spans[span_id][1] > 0:
            best_cycle_id = None
            N0 = span_node[span_id][0]
            N1 = span_node[span_id][1]
            cost_shortest1, path_shortest1 = shortest_path(graph, N0, N1)
            if cost_shortest1 == sys.maxsize:
                raise ValueError('Impossible, No shortest path found!')
            cycle_span_info = {}
            for node_start1, node_end1 in zip(path_shortest1[:-1], path_shortest1[1:]):
                span_ider1 = node_span[node_start1 + " , " + node_end1]
                cycle_span_info[span_ider1] = 1
            cycle_span_info[span_id] = 1

            ## Straddle checker : path=2
            all_nodes_on_this_cycle = []
            for ss, jj in cycle_span_info.items():
                if jj == 1:
                    node1_for_ss = span_node[ss][0]
                    node2_for_ss = span_node[ss][1]
                    all_nodes_on_this_cycle.append(node1_for_ss)
                    all_nodes_on_this_cycle.append(node2_for_ss)
            nodeset = set(all_nodes_on_this_cycle)
            nodes_on_this_cycle = list(nodeset)

            for nn, dd in node_span.items():
                nn_nodes = nn.split(' , ')
                if nn_nodes[0] in nodes_on_this_cycle and nn_nodes[1] in nodes_on_this_cycle and
dd not in cycle_span_info:
                    cycle_span_info[dd] = 2

            ## Check if cycle_Span_info is a duplicate of a cycle in pcycles
            flagger = False
            for key, val in pcycles.items():
                if cycle_span_info == val:
                    if key not in cycle_usage:
                        cycle_usage[key] = 1
                    else:
                        cycle_usage[key] = cycle_usage[key] + 1
                    flagger = True
                    break
            if not flagger:
                new_id2 = new_id2 + 1
                best_cycle_id = "Rcyc " + str(new_id2) # new pcycle id given
                cycle_usage[best_cycle_id] = 1
                pcycles[best_cycle_id] = cycle_span_info

            for key, value in cycle_span_info.items():
                if value == 1:
                    spans[key][1] = spans[key][1] - 1
```

```
                    if spans[key][1] < 0:
                        spans[key][1] = 0
            for key, value in cycle_span_info.items():
                if value == 2:
                    spans[key][1] = spans[key][1] - 2
                    if spans[key][1] < 0:
                        spans[key][1] = 0
    return cycle_usage, pcycles

# Max_Overlap for Repair Genome
def Max_Overlap_Cost(spans, pcycles):
    cycle_usage = {}
    # counter to collect pid
    while True:
        pcycle_hit = {}
        max_pid_cc = 0
        max_pid = "-1"
        min_cost_pcycle = sys.maxsize

        for span_id in spans.keys():
            if spans[span_id][1] > 0:
                straddle_flag = False
                if span_id in straddle_pcycle_indexer:
                    pcycle_list_with_this_spanid = straddle_pcycle_indexer[span_id]
                    straddle_flag = True
                else:
                    pcycle_list_with_this_spanid = span_pcycle_indexer[span_id]

                for pid, _ in pcycle_list_with_this_spanid:
                    if pid not in pcycle_hit and straddle_flag:
                        pcycle_hit[pid] = [pid, 1, 0]
                    elif pid not in pcycle_hit and not straddle_flag:
                        pcycle_hit[pid] = [pid, 0, 1]
                    elif pid in pcycle_hit and straddle_flag:
                        pcycle_hit[pid][1] += 1
                    else:
                        pcycle_hit[pid][2] += 1

                    if pcycle_hit[pid][1] > max_pid_cc:
                        max_pid_cc = pcycle_hit[pid][1]
                        min_cost_pcycle = pcycle_costs_global[pid]
                        max_pid = pid
                    elif pcycle_hit[pid][1] == max_pid_cc:
                        if pcycle_costs_global[pid] < min_cost_pcycle:
                            max_pid_cc = pcycle_hit[pid][1]
                            min_cost_pcycle = pcycle_costs_global[pid]
                            max_pid = pid
        if max_pid == "-1":
            break

        best_cycle_id = max_pid # deterministic

        if best_cycle_id not in cycle_usage:
            cycle_usage[best_cycle_id] = 1
        else:
            cycle_usage[best_cycle_id] = cycle_usage[best_cycle_id] + 1

        for key, value in pcycles[best_cycle_id].items():
            if value == 1:
                spans[key][1] = spans[key][1] - 1
                if spans[key][1] < 0:
                    spans[key][1] = 0
            elif value == 2:
                spans[key][1] = spans[key][1] - 2
                if spans[key][1] < 0:
                    spans[key][1] = 0

    return cycle_usage, pcycles

## Methods for generating initial population: #
# Picking a random cycle
def pick_random_cycle(pcycles):
```

```
    cycle_keys = list(pcycles.keys())
    cycle_picked = random.choices(population=cycle_keys, weights=None, k=1)
    return cycle_picked[0]

# 1. Non-deterministic CIDA: random picks by probs

# Pick a Cycle from a pool
def pick_cycle_by_probs(spans, pcycles):
    pool = {}
    for cycle_key, cycle_value in pcycles.items():
        ew = calculate_ew(spans, cycle_value)
        pool[cycle_key] = ew
    sorted_pool = sorted(pool.items(), key=lambda x: x[1], reverse=True)
    top_ew_scores = []
    top_cycle_keys = []
    # initial value
    importance_score = 10000
    for key, _ in sorted_pool:
        top_ew_scores.append(importance_score)
        top_cycle_keys.append(key)
        # discount weighting
        importance_score = importance_score / 2
    total_sum = sum(top_ew_scores)
    normalized_probs = [float(item / total_sum) for item in top_ew_scores]
    res = random.choices(population=top_cycle_keys, weights=normalized_probs, k=1)
    return res[0]

def CIDA_pick_by_probs(spans, pcycles):
    cycle_usage = {}
    for span_id in spans.keys():
        while spans[span_id][1] > 0:
            best_cycle_id = None
            # Random Select Best Cycle By Probs
            best_cycle_id = pick_cycle_by_probs(spans, pcycles)
            if best_cycle_id not in cycle_usage:
                cycle_usage[best_cycle_id] = 1
            else:
                cycle_usage[best_cycle_id] = cycle_usage[best_cycle_id] + 1
            for key, value in pcycles[best_cycle_id].items():
                if value == 1:
                    spans[key][1] = spans[key][1] - 1
                    if spans[key][1] < 0:
                        spans[key][1] = 0
            for key, value in pcycles[best_cycle_id].items():
                if value == 2:
                    spans[key][1] = spans[key][1] - 2
                    if spans[key][1] < 0:
                        spans[key][1] = 0
    return cycle_usage

##Generate Initial Population ##
# Deterministic:
    # init_pop_member = CIDA(spans, pcycles)
# Non-Deterministic - Random pick by probability:
    # init_pop_member = CIDA_pick_by_probs(spans, pcycles)

# Generate initial population: a list of dictionaries
    # each dictionary is an individual (key:value = pcycle_id : pcycle_num_usage) that provides
full pcycle protection to the network
def initial_population(popSize, pcycles):
    init_population = []
    for _ in range(popSize):
        spans_copy = copy.deepcopy(spans)
        init_pop_member = CIDA_pick_by_probs(spans_copy, pcycles)
        if init_pop_member not in init_population:
            init_population.append(init_pop_member)
    return init_population


### START OF GENETIC ALGORITHM ###
start = time.time()
```

```
## FITNESS FUNCTION:
    # [minimum] cost of each individual (init_pop_member)
def calculate_fitness_cost(pcycle_costs, individual):
    ind_cost = 0
    for p_key, _usage in individual.items():
        cost_p = pcycle_costs[p_key]
        ind_cost += cost_p * _usage
    return ind_cost

def rankCost(pcycle_costs, population):
    fitness_results = {}
    for i in range(0, len(population)):
        fitness_results[i] = int(calculate_fitness_cost(pcycle_costs, population[i]))
    rank_pop = sorted(fitness_results.items(), key = operator.itemgetter(1))
    ranked_pop = [population[item[0]] for item in rank_pop]
    ranked_cost = [item[1] for item in rank_pop]
    return ranked_pop, ranked_cost

## SELECTION METHODS:
def select_roulette(ranked_pop, pcycle_costs):
    cost_scores = []
    score_factor = 1200 # not very necessary but does not affect results
    for ccc in ranked_pop:
        cycle_cost = calculate_fitness_cost(pcycle_costs, ccc)
        cycle_score = (score_factor/cycle_cost)**2
        cost_scores.append(cycle_score)
    total_sum = sum(cost_scores)
    normalized_probs = [float(item / total_sum) for item in cost_scores]
    select_parent = random.choices(population=ranked_pop, weights=normalized_probs, k=1)
    return select_parent[0]

def select_tournament(ranked_pop, pcycle_costs):
    select_ind = random.choices(population=ranked_pop, weights=None, k=10)
    ranked_ind, _ = rankCost(pcycle_costs, select_ind)
    selected_parent = ranked_ind[0]
    return selected_parent

def select_tournament_adp(ranked_pop, pcycle_costs, popSize):
    select_ind = random.choices(population=ranked_pop, weights=None, k=int(popSize*0.05))
    ranked_ind, _ = rankCost(pcycle_costs, select_ind)
    selected_parent = ranked_ind[0]
    return selected_parent

def select_random(ranked_pop):
    random_index = random.randint(0,len(ranked_pop)-1)
    random_parent = ranked_pop[random_index]
    return random_parent

# REPAIR MECHANISM:
def repair_genome_cross(child_genome, spans, pcycles):
    unprotected_span = {}
    protected_spans = {}
    for k, usage in child_genome.items():
        for _k in pcycles[k].keys():
            if _k not in protected_spans:
                if pcycles[k][_k] == 1:
                    protected_spans[_k] = usage
                else:
                    protected_spans[_k] = 2*usage
            else:
                if pcycles[k][_k] == 1:
                    protected_spans[_k] += usage
                else:
                    protected_spans[_k] += 2*usage
    for span in spans.keys():
        if span not in protected_spans.keys():
            unprotected_span[span] = int(spans[span][1])
        else:
            if spans[span][1] > protected_spans[span]:
                unprotected_span[span] = int(spans[span][1]) - int(protected_spans[span])
    spans_merge = {}
    for kkey, vval in unprotected_span.items():
```

```
        spans_merge[kkey] = [spans[kkey][0], vval]
    for kkey, vval in spans.items():
        if kkey not in spans_merge:
            spans_merge[kkey] = [spans[kkey][0], 0]

    ## 3. Repair by Max_Match for cross-over operator:
    compensation_cycle, _ = Max_Overlap_Cost(spans_merge, pcycles)
    repaired_child = { k: child_genome.get(k, 0) + compensation_cycle.get(k, 0) for k in
set(child_genome) | set(compensation_cycle) }
    return repaired_child

def repair_genome_mut(child_genome, spans, pcycles):
    unprotected_span = {}
    protected_spans = {}
    new_pcycle_set = {}
    new_pcycle_costs_set = {}
    for k, usage in child_genome.items():
        for _k in pcycles[k].keys():
            if _k not in protected_spans:
                if pcycles[k][_k] == 1:
                    protected_spans[_k] = usage
                else:
                    protected_spans[_k] = 2*usage
            else:
                if pcycles[k][_k] == 1:
                    protected_spans[_k] += usage
                else:
                    protected_spans[_k] += 2*usage
    for span in spans.keys():
        if span not in protected_spans.keys():
            unprotected_span[span] = int(spans[span][1])
        else:
            if spans[span][1] > protected_spans[span]:
                unprotected_span[span] = int(spans[span][1]) - int(protected_spans[span])
    spans_merge = {}
    for kkey, vval in unprotected_span.items():
        spans_merge[kkey] = [spans[kkey][0], vval]
    for kkey, vval in spans.items():
        if kkey not in spans_merge:
            spans_merge[kkey] = [spans[kkey][0], 0]

    ## 2. Repair by Min_cycle:
    compensation_cycle, new_pcycle = Min_Cycle_Repair(spans_merge, pcycles, span_node, node_span)
    ## 3. Repair by Max_Match:
    #compensation_cycle, new_pcycle = Max_Overlap_Cost(spans_merge, pcycles)
    repaired_child = { k: child_genome.get(k, 0) + compensation_cycle.get(k, 0) for k in
set(child_genome) | set(compensation_cycle) }
    new_pcycle_set = {**new_pcycle_set, **new_pcycle} # useless for repair method 3
    new_pcycle_cost_dict = calculate_p_cost(spans, new_pcycle_set) # useless for repair method 3
    new_pcycle_costs_set = {**new_pcycle_costs_set, **new_pcycle_cost_dict} # useless for repair
method 3
    return repaired_child, new_pcycle_set, new_pcycle_costs_set

## CROSSOVER FUNCTIONS:
# Crossover 1: Two-point crossover
def two_point(parent1, parent2, pcycle_set):
    childP1 = {}
    childP2 = {}
    childP3 = {}
    childP4 = {}
    geneA1 = int(random.random() * len(parent1.items()))
    geneB1 = int(random.random() * len(parent1.items()))
    geneA2 = int(random.random() * len(parent2.items()))
    geneB2 = int(random.random() * len(parent2.items()))
    startGene1 = min(geneA1, geneB1)
    endGene1 = max(geneA1, geneB1)
    startGene2 = min(geneA2, geneB2)
    endGene2 = max(geneA2, geneB2)

    p1_list = []
    for key, value in parent1.items():
        temp = [key,value]
```

```
        p1_list.append(temp)

    p2_list = []
    for key, value in parent2.items():
        temp = [key,value]
        p2_list.append(temp)

    for i in range(startGene1, endGene1):
        _pick = p1_list[i]
        childP1[_pick[0]] = _pick[1]

    for k,v in parent1.items():
        if k not in childP1:
            childP2[k] = v

    for i in range(startGene2, endGene2):
        _pick = p2_list[i]
        childP3[_pick[0]] = _pick[1]

    for k,v in parent2.items():
        if k not in childP3:
            childP4[k] = v

    child1 = {**childP1, **childP4}
    child2 = {**childP2, **childP3}
    repaired_child1 = repair_genome_cross(child1, spans, pcycle_set)
    repaired_child2 = repair_genome_cross(child2, spans, pcycle_set)

    return repaired_child1, repaired_child2

# Crossover 2: one point crossover
def one_point(parent1, parent2, pcycle_set):
    childP1 = {}
    childP2 = {}
    childP3 = {}
    childP4 = {}

    p1_list = []
    for key, value in parent1.items():
        temp = [key,value]
        p1_list.append(temp)

    p2_list = []
    for key, value in parent2.items():
        temp = [key,value]
        p2_list.append(temp)

    gene1 = int(random.random() * len(parent1.items()))
    gene2 = int(random.random() * len(parent2.items()))

    for i in range(0, gene1):
        _pick = p1_list[i]
        childP1[_pick[0]] = _pick[1]

    for k,v in parent1.items():
        if k not in childP1:
            childP2[k] = v

    for i in range(0, gene2):
        _pick = p2_list[i]
        childP3[_pick[0]] = _pick[1]

    for k,v in parent2.items():
        if k not in childP3:
            childP4[k] = v

    child1 = {**childP1, **childP4}
    child2 = {**childP2, **childP3}
    repaired_child1 = repair_genome_cross(child1, spans, pcycle_set)
    repaired_child2 = repair_genome_cross(child2, spans, pcycle_set)

    return repaired_child1, repaired_child2
```

```
# Crossover 3: uniform crossover
    # Very simple and straightfoward Mask setting: chromosome-length with binary pattern of
[01010101010101...]
    # Not robust --> not in use any further

def breed_population3(selection, crossover, ranked_pop, pcycle_costs, cross_rate, pcycle_set,
popSize):
    children = []
    for _ in range(len(ranked_pop)):
        if random.random() < cross_rate:
            if selection == "select_tournament":
                pick1 = select_tournament(ranked_pop, pcycle_costs)
                pick2 = select_tournament(ranked_pop, pcycle_costs)
            elif selection == "select_roulette":
                pick1 = select_roulette(ranked_pop, pcycle_costs)
                pick2 = select_roulette(ranked_pop, pcycle_costs)
            elif selection == "select_tournament_adp":
                pick1 = select_tournament_adp(ranked_pop, pcycle_costs, popSize)
                pick2 = select_tournament_adp(ranked_pop, pcycle_costs, popSize)
            while pick1 == pick2:
                if selection == "select_tournament":
                    pick2 = select_tournament(ranked_pop, pcycle_costs)
                elif selection == "select_roulette":
                    pick2 = select_roulette(ranked_pop, pcycle_costs)
                elif selection == "select_tournament_adp":
                    pick2 = select_tournament_adp(ranked_pop, pcycle_costs, popSize)
            if crossover == "one_point":
                child1, child2 = two_point(pick1, pick2, pcycle_set)
            elif crossover == "two_point":
                child1, child2 = two_point(pick1, pick2, pcycle_set)
            children.append(child1)
            children.append(child2)
    mutation_pop = ranked_pop + children
    return children, mutation_pop

## MUTATION FUNCTION:  5 mutation methods
    # 1. Randomly remove one cycle (and all its copies used) --> repair it
    # 2. Cycle_merging:
    # 3. Randomly remove one copy of a cycle, and add one random cycle (one copy) --> repair it
    # 4. Randomly pick an ind, remove 10% cycles with largest costs, add 5 random cycle -->
repair it
    # 5. Randomly remove all copies of 1 cycle, add 1 copy 1 random cycle --> repair it

# 1. Randomly remove one copy of a cycle:
def mutate_rand_remove(individual, pcycles):
    individual_copy = copy.deepcopy(individual)
    individual_keys = list(individual_copy.keys())
    swappee = int(random.random() * len(individual_keys))
    individual_copy[individual_keys[swappee]] = int(individual_copy[individual_keys[swappee]])-1
# remove one copy of a candidate cycle
    new_individual = { k:v for k,v in individual_copy.items() if v > 0 }
    # Need repair algorithm:
    repaired_individual, new_pcycle_set, new_pcycle_costs_set = repair_genome_mut(new_individual,
spans, pcycles)
    return repaired_individual, new_pcycle_set, new_pcycle_costs_set

# 2. Cycle Merging:
def mutation_span_merge(individual, pcycles):
    new_individual = copy.deepcopy(individual)
    new_pcycle = {}
    new_pcycle_usage = {}
    final_individual = {}
    global new_id_mut
    ind_keys_list = list(individual.keys())
    for index1, cycle1_id in enumerate(ind_keys_list):
        for index2, cycle2_id in enumerate(ind_keys_list):
            if index2 > index1:
                if cycle1_id == cycle2_id:
                    continue
                cycle_1spans_dict = dict(pcycles[cycle1_id])
                cycle_2spans_dict = dict(pcycles[cycle2_id])
```

```
                cycle_1node_list = []
                cycle_2node_list = []
                cycle_1usage = new_individual[cycle1_id]
                cycle_2usage = new_individual[cycle2_id]
                if cycle_1usage == 0 or cycle_2usage == 0:
                    continue
                shared_items = [k for k in cycle_1spans_dict if k in cycle_2spans_dict and
cycle_1spans_dict[k] == 1 and cycle_2spans_dict[k] == 1]
                # check if no other nodes (other than the two nodes on the shared_items) are
ovelapping
                if len(shared_items) != 1:
                    continue
                new_individual_span = {}
                shared_items_N1 = span_node[list(shared_items)[0]][0]
                shared_items_N2 = span_node[list(shared_items)[0]][1]
                for sp1 in cycle_1spans_dict.keys():
                    sp1_N1 = span_node[sp1][0]
                    sp1_N2 = span_node[sp1][1]
                    cycle_1node_list.append(sp1_N1)
                    cycle_1node_list.append(sp1_N2)
                for sp2 in cycle_2spans_dict.keys():
                    sp2_N1 = span_node[sp2][0]
                    sp2_N2 = span_node[sp2][1]
                    cycle_2node_list.append(sp2_N1)
                    cycle_2node_list.append(sp2_N2)
                shared_node = set(cycle_1node_list) & set(cycle_2node_list)
                shared_node -= {shared_items_N1}
                shared_node -= {shared_items_N2}
                shared_node_list = list(shared_node)
                if shared_node_list:
                    continue
                # check if only share one span
                if len(shared_items) == 1:
                    new_individual_span[shared_items[0]] = 2
                    cycle_1spans_dict[shared_items[0]] = -1
                    cycle_2spans_dict[shared_items[0]] = -1
                    for kk, vv in cycle_1spans_dict.items():
                        if vv != -1:
                            new_individual_span[kk] = vv
                    for kk, vv in cycle_2spans_dict.items():
                        if vv != -1:
                            new_individual_span[kk] = vv
                else:
                    continue
                new_cycle_id = "Merged_CYL" + str(new_id_mut) # new pcycle id given
                new_pcycle[new_cycle_id] = new_individual_span
                usage_new_cycle = min(cycle_1usage, cycle_2usage)
                new_pcycle_usage[new_cycle_id] = int(usage_new_cycle)
                cycle_1usage -= int(usage_new_cycle)
                new_individual[cycle1_id] = cycle_1usage
                cycle_2usage -= int(usage_new_cycle)
                new_individual[cycle2_id] = cycle_2usage
                new_id_mut += 1
    new_individual_new_pcycles = {**new_individual, **new_pcycle_usage}
    for mkk, mvv in new_individual_new_pcycles.items():
        if mvv > 0:
            final_individual[mkk] = mvv
    return final_individual, new_pcycle

# 3. Remove one copy of a cycle , add one copy of a cycle:
def mutate_remove1_add1(individual, pcycles):
    individual_copy = copy.deepcopy(individual)
    individual_keys = list(individual_copy.keys())
    individual_added = {}
    swapper_id = None
    swappee = int(random.random() * len(individual_keys))
    individual_copy[individual_keys[swappee]] = int(individual_copy[individual_keys[swappee]])-1
# remove one copy of a candidate cycle
    swapper_id = pick_random_cycle(pcycles) # add one random cycle
    if swapper_id not in individual_copy:
        individual_added[swapper_id] = 1
    individual_copy = { k:v for k,v in individual_copy.items() if v > 0 }
```

```
    # Merge individual_copy and individual_added
    new_individual = { k: individual_copy.get(k, 0) + individual_added.get(k, 0) for k in
set(individual_added) | set(individual_copy) }
    repaired_individual, new_pcycle_set, new_pcycle_costs_set = repair_genome_mut(new_individual,
spans, pcycles)
    return repaired_individual, new_pcycle_set, new_pcycle_costs_set

# 4. Remove one copy of each worst 10% cycles, add 3 rand cycles:
def mutate_less10add3(individual, pcycles, pcycles_costs):
    individual_copy = copy.deepcopy(individual)
    individual_cost = []
    added_inds = {}
    pcycle_ids = []
    new_individual = {}
    for cycle_id, cycle_usage in individual_copy.items():
        if cycle_usage == 0:
            continue
        individual_cost.append((cycle_id, pcycles_costs[cycle_id]))  # individual_cost is
[('cycle id', cycle cost), (,), (,)...]
        pcycle_ids.append(cycle_id)
    while True:
        rand_index = int(random.random() * len(pcycles))
        pick_rand_cyc = list(pcycles.keys())[rand_index]
        if pick_rand_cyc not in pcycle_ids:
            added_inds[pick_rand_cyc] = 1
        if len(added_inds) == 3: # add 3 random cycles
            break
    # Remove one copy from each bottom 10% (selecting top 90%)
    sorted_tup = sorted(individual_cost, key=lambda e:e[1])  # sort the list from small cost to
large cost
    for ind in range(int(len(sorted_tup)* 0.9), len(sorted_tup)):
        select_id = sorted_tup[ind][0]
        if int(individual_copy[select_id]) >= 1:
            individual_copy[select_id] = int(individual_copy[select_id])-1
    # Merge added_inds and individual_copy
    new_individual = { k: added_inds.get(k, 0) + individual_copy.get(k, 0) for k in
set(added_inds) | set(individual_copy) }
    new_individual = { k:v for k,v in new_individual.items() if v > 0 }
    repaired_individual, new_pcycle_set, new_pcycle_costs_set = repair_genome_mut(new_individual,
spans, pcycles)
    return repaired_individual, new_pcycle_set, new_pcycle_costs_set

# 5. Remove one cycle (all copies), add one cycle:
def mutate_remove1all_add1(individual, pcycles):
    individual_copy = copy.deepcopy(individual)
    individual_keys = list(individual_copy.keys())
    individual_added = {}
    swapper_id = None
    swappee = int(random.random() * len(individual_keys))
    individual_copy[individual_keys[swappee]] = -100
    swapper_id = pick_random_cycle(pcycles) # Adding one random cycle to new_individual:
    if swapper_id not in individual_copy:
        individual_added[swapper_id] = 1
    individual2 = { k:v for k,v in individual_copy.items() if v != -100 }
    # Remove one cycle: Merge individual2 and individual_added
    new_individual = { k: individual2.get(k, 0) + individual_added.get(k, 0) for k in
set(individual_added) | set(individual2) }
    repaired_individual, new_pcycle_set, new_pcycle_costs_set = repair_genome_mut(new_individual,
spans, pcycles)
    return repaired_individual, new_pcycle_set, new_pcycle_costs_set

# Mutation 1. Randomly remove a cycle * copies of that cycle:
def mutate1(mutation_pop,mutation_rate, pcycle_costs, pcycles):
    mutated_children = []
    pcycle_set1 = pcycles
    pcycle_costs1 = pcycle_costs
    for _ in range(len(mutation_pop)):
        if random.random() < mutation_rate:
            pick_ii = select_random(mutation_pop)
            mutated_ind, new_pcycle_set, new_pcycle_costs_set = mutate_rand_remove(pick_ii,
pcycles)
            mutated_children.append(mutated_ind)
```

```
            pcycle_set1 = {**pcycle_set1, **new_pcycle_set}
            pcycle_costs1 = {**pcycle_costs1, **new_pcycle_costs_set}
    return mutated_children, pcycle_set1, pcycle_costs1

# Mutation 2. Cycle Merging:
def mutate2(mutation_pop,mutation_rate, pcycle_costs, pcycles):
    mutated_children = []
    new_pcycle_set = {}
    new_pcycle_costs_set = {}
    for _ind in range(len(mutation_pop)):
        if random.random() < mutation_rate:
            pick_ii = select_random(mutation_pop)
            mutated_ind, new_pcycle = mutation_span_merge(pick_ii, pcycles)
            mutated_children.append(mutated_ind.copy())
            new_pcycle_set = {**new_pcycle_set, **new_pcycle}
            new_pcycle_cost_dict = calculate_p_cost(spans, new_pcycle_set)
            new_pcycle_costs_set = {**new_pcycle_costs_set, **new_pcycle_cost_dict}
    return mutated_children, new_pcycle_set, new_pcycle_costs_set

# Mutation 3. Remove one copy of a cycle , add one copy of a cycle:
def mutate3(mutation_pop,mutation_rate, pcycle_costs, pcycles):
    mutated_children = []
    pcycle_set3 = pcycles
    pcycle_costs3 = pcycle_costs
    for _ in range(len(mutation_pop)):
        if random.random() < mutation_rate:
            pick_ii = select_random(mutation_pop)
            mutated_ind, new_pcycle_set, new_pcycle_costs_set = mutate_remove1_add1(pick_ii,
pcycles)
            mutated_children.append(mutated_ind)
            pcycle_set3 = {**pcycle_set3, **new_pcycle_set}
            pcycle_costs3 = {**pcycle_costs3, **new_pcycle_costs_set}
    return mutated_children, pcycle_set3, pcycle_costs3

# Mutation 4. Remove one copy of each worst 10% cycles, add 3 rand cycles:
def mutate4(mutation_pop,mutation_rate, pcycle_costs, pcycles):
    mutated_children = []
    pcycle_set4 = pcycles
    pcycle_costs4 = pcycle_costs
    for _ in range(len(mutation_pop)):
        if random.random() < mutation_rate:
            pick_ii = select_random(mutation_pop)
            mutated_ind, new_pcycle_set, new_pcycle_costs_set = mutate_less10add3(pick_ii,
pcycle_set4, pcycle_costs4)
            mutated_children.append(mutated_ind)
            pcycle_set4 = {**pcycle_set4, **new_pcycle_set}
            pcycle_costs4 = {**pcycle_costs4, **new_pcycle_costs_set}
    return mutated_children, pcycle_set4, pcycle_costs4

# Mutation 5. Remove one cycle completely, add one copy of a cycle:
def mutate5(mutation_pop,mutation_rate, pcycle_costs, pcycles):
    mutated_children = []
    pcycle_set5 = pcycles
    pcycle_costs5 = pcycle_costs
    for _ in range(len(mutation_pop)):
        if random.random() < mutation_rate:
            pick_ii = select_random(mutation_pop)
            mutated_ind, new_pcycle_set, new_pcycle_costs_set = mutate_remove1all_add1(pick_ii,
pcycles)
            mutated_children.append(mutated_ind)
            pcycle_set5 = {**pcycle_set5, **new_pcycle_set}
            pcycle_costs5 = {**pcycle_costs5, **new_pcycle_costs_set}
    return mutated_children, pcycle_set5, pcycle_costs5

def mutate_pop(mutation, ranked_pop, mutation_rate, pcycle_costs, pcycle_set):
    if mutation == "mutate1":
        mutated_children, new_pcycles, new_pcycle_costs = mutate1(ranked_pop,mutation_rate,
pcycle_costs, pcycle_set)
    elif mutation == "mutate2":
        mutated_children, new_pcycles, new_pcycle_costs = mutate2(ranked_pop,mutation_rate,
pcycle_costs, pcycle_set)
    elif mutation == "mutate3":
```

```
        mutated_children, new_pcycles, new_pcycle_costs = mutate3(ranked_pop,mutation_rate,
pcycle_costs, pcycle_set)
    elif mutation == "mutate4":
        mutated_children, new_pcycles, new_pcycle_costs = mutate4(ranked_pop,mutation_rate,
pcycle_costs, pcycle_set)
    elif mutation == "mutate5":
        mutated_children, new_pcycles, new_pcycle_costs = mutate5(ranked_pop,mutation_rate,
pcycle_costs, pcycle_set)
    return mutated_children, new_pcycles, new_pcycle_costs

## NEXT GENERATION ##

# Using all mutation methods:
def next_generation(selection, crossover, mutation, current_gen, cross_rate, mutation_rate,
pcycle_costs, pcycle_set, popSize):
    new_generation = []
    pop_copy = current_gen
    source_children = {}
    for nnn in current_gen:
        new_generation.append(nnn)
    children, _ = breed_population3(selection, crossover, pop_copy, pcycle_costs, cross_rate,
pcycle_set, popSize)
    for iii in children:
        if iii not in new_generation:
            new_generation.append(iii)
            source_children[frozenset(iii.items())] = 0
    # Options: mutate1, mutate2, mutate3, mutate4, mutate5
    mutated_children, new_pcycles, new_pcycle_costs = mutate_pop(mutation, pop_copy,
mutation_rate, pcycle_costs, pcycle_set)
    for ddd in mutated_children:
        if ddd not in new_generation:
            new_generation.append(ddd)
            source_children[frozenset(ddd.items())] = 1
    updated_pcycles = {**pcycle_set, **new_pcycles}
    updated_pcycle_costs = {**pcycle_costs, **new_pcycle_costs}

    new_pop_ranked, new_pop_scores = rankCost(updated_pcycle_costs, new_generation)
    return new_pop_ranked[:len(current_gen)], new_pop_scores[:len(current_gen)], updated_pcycles,
updated_pcycle_costs, source_children

# When unchanged value reach gen = 60, use span_merge as mutation method:
def next_gen_mutation2(selection, crossover, current_gen, cross_rate, mutation_rate,
pcycle_costs, pcycle_set, popSize):
    new_generation = []
    source_children = {}
    pop_copy = current_gen
    for nnn in current_gen:
        new_generation.append(nnn)
    children, _ = breed_population3(selection, crossover, pop_copy, pcycle_costs, cross_rate,
pcycle_set, popSize)
    for iii in children:
        if iii not in new_generation:
            new_generation.append(iii)
            source_children[frozenset(iii.items())] = 0
    mutated_children, new_pcycles, new_pcycle_costs = mutate2(pop_copy, mutation_rate,
pcycle_costs, pcycle_set)
    for ddd in mutated_children:
        if ddd not in new_generation:
            new_generation.append(ddd)
            source_children[frozenset(ddd.items())] = 1
    updated_pcycles = {**pcycle_set, **new_pcycles}
    updated_pcycle_costs = {**pcycle_costs, **new_pcycle_costs}

    new_pop_ranked, new_pop_scores = rankCost(updated_pcycle_costs, new_generation)
    return new_pop_ranked[:len(current_gen)], new_pop_scores[:len(current_gen)], updated_pcycles,
updated_pcycle_costs, source_children

### Initiate GA Process ###

def genetic_algorithm(selection, crossover, mutation, popSize, cross_rate, mutation_rate,
pcycle_set):
    pop = initial_population(popSize, pcycles)
```

```
    pcycle_costs = calculate_p_cost(spans, pcycles) # Calculate fitness values
    pop_rank, pop_scores = rankCost(pcycle_costs, pop) # rank all ind from initial population
based on their fitness values
    print(pop_rank[0]) # And print out best ind from init_pop and its cost to compare with GA
result
    print(pop_scores[0])

    global_min_cost = math.inf
    stats_source = [0, 0] #counts for children from either cross or mutation
    gen_counter = 0
    for gen in range(0, 1000):
        best_current_gen, rank_cost, updated_pcycles, updated_pcycle_costs, source_children =
next_generation(selection, crossover, mutation, pop_rank, cross_rate, mutation_rate,
pcycle_costs, pcycle_set, popSize)
        #print(source_children.values())
        if rank_cost[0] < global_min_cost:
            ## increase!
            global_min_cost = rank_cost[0]
            optimal_res = best_current_gen[0]
            gen_counter = 0
            if frozenset(optimal_res.items()) in source_children:
                stats_source[source_children[frozenset(optimal_res.items())]] += 1
            else:
                print('NEW IMPROVEMENT')
                # print(optimal_res)
            print(f'{gen}: Better Global Min Cost {global_min_cost}')
        else:
            gen_counter += 1
            if gen_counter == 60:
                best_current_gen, rank_cost, updated_pcycles, updated_pcycle_costs,
source_children = next_gen_mutation2(selection, crossover, pop_rank, cross_rate, mutation_rate,
pcycle_costs, pcycle_set, popSize)
                if rank_cost[0] < global_min_cost:
                    global_min_cost = rank_cost[0]
                    optimal_res = best_current_gen[0]
                    gen_counter = 0
                    if frozenset(optimal_res.items()) in source_children:
                        stats_source[source_children[frozenset(optimal_res.items())]] += 1
                    else:
                        print('NEW IMPROVEMENT')
                        # print(optimal_res)
                    print(f'{gen}: Cycle-Merge applied, Better Global Min Cost
{global_min_cost}')
                else:
                    print(f'Process terminated at {gen} with final global min cost of
{global_min_cost}')
                    break
        pop_rank = best_current_gen
        pcycle_set = updated_pcycles
        pcycle_costs = updated_pcycle_costs

    print(f'Improved children counts from {stats_source}')
    # print(optimal_res) # evaluate the result
    print('Total Cost = '+ str(global_min_cost)) # evaluate validity of this GA process -- is the
cost decreasing??
    return optimal_res, pcycle_set, global_min_cost

random.seed(666)
now = datetime.datetime.now()
print(now)

optimal_result, updated_pcycles, global_min_cost = genetic_algorithm(selection =
Selection_Method, crossover = Crossover_Type, mutation = Mutation_Type, popSize = input_pop,
cross_rate = input_cr, mutation_rate = input_mr, pcycle_set = pcycles)

the_end = time.time()
print('Total Runtime =' + str(the_end - start))


## Write new pcycles to a new .pcycle file [for verification of full protection]
text_file = open(output_pcycle, "w")
for key, value in updated_pcycles.items():
```

```
    text_file.write(key + ": ")
    for kk, tt in value.items():
        text_file.write(kk + " " + str(tt) + " ")
    text_file.write("\n")
text_file.close()

### Calculate pcycle set SC, WC, EW&AP scores ###

ind_sc = 0
def pcycle_spare_capacity(spans, cycle_val):
    pcycle_sc_score = 0
    for _, vv in cycle_val.items():
        if vv == 1:
            pcycle_sc_score += 1
    return pcycle_sc_score
for cy_id, cy_use in optimal_result.items():
    cycle_sc = pcycle_spare_capacity(spans, updated_pcycles[cy_id])
    ind_sc = ind_sc + cycle_sc * cy_use

def pcycle_work_capacity(spans):
    pcycle_wc_score = 0
    pcycle_wc_cost = 0
    for _, comp in spans.items():
        pcycle_wc_score += comp[1]
        pcycle_wc_cost += comp[0] * comp[1]
    return pcycle_wc_score, pcycle_wc_cost
ind_wc, ind_wc_cost = pcycle_work_capacity(spans)


print('Total Spare Capacity = '+ str(ind_sc))
print('Total Spare Capacity Cost = ' + str(global_min_cost))
print('Total Working Capacity = '+ str(ind_wc))
print('Total Working Capacity Cost = '+ str(ind_wc_cost))
print('Redundancy = ' "{0:.2f}%".format(ind_sc/ind_wc * 100))
```