University of Alberta

Verification of Graph Algorithms in Mizar

by

Gilbert Lee 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2004

Your file Votre référence
ISBN: 0-612-95791-8
Our file Notre référence
ISBN: 0-612-95791-8

Contents

Chapter 1

Introduction

1.1 Overview

The main objective of this work is to produce a formal environment for proving theorems in graph theory. The goal is to create a library suitable towards verifying the correctness of several well-known graph algorithms, in terms of graph operations. This paper discusses the conceptual differences between textbooks proofs and formal proofs, and difficulties in translating from the former into the latter. In particular, this paper discusses how Dijkstra's single-source shortest path algorithm, Prim's minimum weight spanning tree algorithm, and Ford/Fulkerson's maximum network flow algorithm are all formalized using the MIZAR proof system.

1.2 Formalized mathematics

One of the goals of formalized mathematics is to take mathematical knowledge and express it in such a way that automated processing is possible. Not only would such formalized proofs be verifiable by machine and completely rigorous, but would also form a searchable database. There is also hope that such a library may be mined for new results, or prove useful to automated theorem provers. Currently, there are quite a number of automated proof assistants in use around the world, such as MIZAR, HOL, and COQ.

1.3 Mizar

The proof assistant system used for this particular project is called MIZAR. Started in 1973 by Andrzej Trybulec, the MIZAR project is one of the oldest of all active proof systems. MIZAR consists of two main components – the verifier and the MIZAR Mathematical Library (MML). The verifier takes a MIZAR article – a text written according to MIZAR syntax – and checks the file for logical errors. The MML is a knowledge management system for mathematics, and is a database containing thousands of mathematical definitions and over thirty thousand facts. The MML is built on the axioms of the Tarski Grothendieck set theory. MIZAR's proof checker is based on classical first-order logic, furnished with some machinery for forming infinite schemes of statement.

One of the advantages of MIZAR is the readability of MIZAR proofs. For example, suppose we would like to prove that there are no isolated vertices (vertices with degree 0) in a non trivial connected graph. We present a MIZAR proof of this fact:

```
theorem tGCONNECT01:
 for G being non trivial connected _Graph,
     v being Vertex of G holds not (v is isolated)
 proof
    let G be non trivial connected _Graph, v be Vertex of G;
    consider v1,v2 being Vertex of G such that
A1: v1 <> v2 by GLIB_000:22; :: <> means not equal
    now per cases;
      suppose v1 = v; then
         v2 <> v by A1;
         hence ex u being Vertex of G st u <> v;
      end;
      suppose v1 <> v;
         hence ex u being Vertex of G st u <> v;
      end;
    end; then consider u being Vertex of G such that
A2: u <> v;
    consider W being Walk of G such that
A3: W is_Walk_from u,v by dGCONNECT;
A4: W.first() = u & W.last() = v by A3, GLIB_001:def 23; then
A5: v in W.vertices() by GLIB_001:93;
    W is non trivial by A2, A4, GLIB_001:131;
    hence not v is isolated by A5, GLIB_001:139;
end;
```

In this proof, we first let G be some arbitrary graph and let v be some vertex of G. We then consider two distinct vertices $v1$ and $v2$, which must exist because the graph is non trivial. At least one of these vertices must be different from v, so we consider one such vertex u. The graph is connected, therefore we can now consider a walk W which goes from u to v. W is non trivial (i.e. has at least one edge), so any vertex along it is not isolated, hence v is not isolated.

Readability is an important benefit, allowing people who may not necessarily know how to use MIZAR to still understand MIZAR articles. Recently MIZAR's syntax has gained popularity, and has been ported into other proof assistant systems such as HOL and Isar.

1.4 Related Works

In 1990, Hryniewiecki[9] formalized some basic graph structures in MIZAR, which was followed subsequently by articles from Rudnicki, Nakamura, and Chen[12, 13, 14, 15]. Chen also showed a proof of correctness for Dijkstra's algorithm [4], although his proof approach is completely different than ours. Instead of dealing with basic graph operations, Chen simulates operations on an array containing the graph information and temporary data necessary for the algorithm. A formalization of Prim's algorithm via the B event-based approach may be found in [1, 8]. A HOL-based formalization of graph search algorithms was done in [23].

1.5 Organization of the Paper

Chapter 2 introduces the graph primitives available and formal definitions of computational machinery. Chapter 3 discusses the issue of translating a textbook proof to a formal Mizar proof - the difficulties involved and the different type of reasoning required. Chapter 4 explains how formal proofs of correctness are done for Dijkstra's algorithm, Prim's algorithm and Ford/Fulkerson's algorithm. We conclude our paper in Chapter 5. In addition there are two appendices that give more technical information regarding our alternate approach to MIZAR structures, as well as details of the graph library environment prepared for MIZAR.

Chapter 2

Basic Terminology and Graph Primitives

2.1 Graphs, Subgraphs and Walks

For our purposes, a *graph* is composed of a non-empty set of vertices, a set of edges, and two functions mapping edges to source and target vertices respectively. An edge e is said to be *incident* to a vertex v if the source or target of e is v. We say that the edge e *joins* vertices v_1 and v_2 if v_1 is the source of e and v_2 is target of e or vice versa. The edge e *directly joins* vertices v_1 and v_2 if v_1 is the source of e, and v_2 is the target of e.

A *subgraph* of G is a graph whose vertices and edges are a subset of G, where edges have the same source and target vertex as in G. Given a non-empty subset of vertices V, and a subset of edges E whose incident vertices are all elements of V, we say that the *subgraph induced by V and E* is the graph formed by only considering the vertices V and the edges E.

For a graph G, there is the idea of a *walk* of G. A walk is a finite sequence of odd length composed of vertices and edges of G that alternate – $v_0, e_1, v_1, \ldots, e_n, v_n$. The first element of a walk is always is vertex, and every edge in the walk joins its two neighboring vertices. If every e_i $(1 \leq i \leq n)$ directly joins the vertices v_{i-1} and v_i, we say that the walk is *directed*. The length of a walk is the number of edges in it, and we say that a walk is *trivial* if it has no edges.

A *trail* is walk with distinct edges. A *path* is a trail that repeats no vertices other than perhaps the first and last. In MIZAR, we signify that a walk is directed by the prefix `D`, for example a *DPath* refers to a directed path.

A walk is said to be *closed* if its first vertex is the same as its last. A *Cycle* is a path that is non trivial and closed. If a graph has no cycles in it, we say that the graph is *acyclic*. A graph is *connected* if for any two vertices, there exist a walk between them. If there exists a walk from v_1 to v_2, we say that v_2 is *reachable* from v_1. A *tree* is a graph that is connected and acyclic.

2.2 Additional graph features

In this paper, we will be dealing with three additional features on graphs - weights, edge-labels and vertex-labels. Not only do these features play important roles in the

three graph algorithms we deal with, but they are also applicable to other graph topics such as graph coloring and graph traversals.

In general, weights may take on any value, but in the context of this paper, we will assume that weights are all real numbers. To distinguish which subset of features we are interested in, we will use the prefix `W` for weighted graphs, `E` for edge-labeled graphs, and `V` for vertex-labeled graphs. For example, a `WEGraph` is a weighted graph with edge-labels. For weighted graphs, each edge in the graph is associated with a weight. On the other hand, it is not necessary that every edge requires a label in an `EGraph`. Similarly, not all vertices require labels in a `VGraph`.

For the three algorithms we are concerned with, we deal with very basic graph operations. In fact, the only type of operation required is the ability to assign labels. For our work, we introduce two methods of modifying the labeling of a graph. The first is where we completely specify a labeling, and the second is setting the label for a particular element. For example, given a labeling L, we can assign it to a `EGraph` G using the function `G.set(ELabelSelector, L)`. For a `VGraph`, the function is named `G.set(VLabelSelector, L)`. If we want to assign a particular value x to edge e in the `EGraph` G, we can use the operation `G.labelEdge(e,x)`. Similarly, for a `VGraph`, we have an operation named `G.labelVertex(v,x)`. These four simple graph operations form the basis of all the algorithms which we proved correct.

We should point out here that we focus on proving the correctness of algorithms, and have left out proofs regarding complexity. There are two main reasons for this. First of all, in order to properly discuss complexity, we need to formalize the exact machine on which the algorithm runs. Designing and formalizing such a machine is a large undertaking beyond the scope of our current work. Secondly, even given the exact machine, the cost of graph operations may be different due to the method of implementation. For example, the cost of checking for the existence of an edge in a graph represented by an adjacency matrix is different than if the graph is represented by an adjacency list.

2.3 Formal notion of an algorithm

We also need to formalize the notion of an algorithm. In texts we often think of an algorithm as a sequence of operations which modify a graph. Formally, when we modify an object, it is no longer the same object, therefore we must treat algorithms slightly differently. Consider an infinite sequence of graphs $G_0, G_1, G_2, \ldots$, which we call a *computation sequence*. An *algorithm* defines a computation sequence, usually by initializing the graph G_0 and specifying the basic graph operations that create G_{n+1} from G_n for any value of n. When we say an algorithm *halts*, we mean that there exists a n where $G_n = G_{n+1}$ in its computation sequence. For an algorithm that halts, we say that the *lifespan* of the computation sequence is the first n such that $G_n = G_{n+1}$. The *result* of a halting algorithm is $G_{lifespan}$. Later in our proofs, we will often use the short-form G_n to represent the n^{th} term of a computation sequence when it is clear which algorithm we are talking about.

Chapter 3

Translation Problems

There are four main obstacles we faced when translating a textbook proof found in a book to a MIZAR proof: Definitional difficulties, conceptual differences, incomplete details, and extra assumptions.

3.1 Definitional difficulties

Definitional difficulties are problems encountered when trying to characterize a concept formally. When defining a concept in a formalized manner, there are two main concerns - accuracy and usability. In order for a formal definition of a concept to be accurate, it must capture all the necessary details regarding the concept, without leaving anything out.

Take for example the definition of a walk of a graph. From West[22]:

> 1.2.2. Definition. A Walk is a list $v_0, e_1, v_1, \ldots, e_k, v_k$ of vertices and edges such that for $1 \leq i \leq k$, the edge e_i has endpoints v_{i-1} and v_i."

Now suppose we translate this to mean: "A *Walk* is a finite sequence composed of vertices and edges where every even element is an edge that joins the element before it to the element after it." These two definitions may seem identical; however this translation is not accurate. It is true that any walk is also a *Walk*, but the converse is not true. According to the second definition, a sequence consisting of a single edge would also constitute as a *Walk*, yet this goes against West's definition. Due to this inherent difference, it may not be possible to prove facts regarding *Walks* which are true for walks or vice versa, such as "The first element of a *Walk* is a vertex".

Another facet of accuracy is dealing with textbook definitions that are inconsistent. While general concepts remain the same, different texts usually have slightly different characterizations. This is especially common in the graph literature. A case in point is the idea of a path:

In Cormen[5], a path is what we would consider a walk:

> A path of length k from a vertex u to a vertex u' in a graph $G = (V, E)$ is a sequence $< v_0, v_1, v_2, \ldots, v_k >$ of vertices such that $u = v_0, u' = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$.

According to Behzad[3], a path may not have any repeated vertices, which means that it automatically does not repeat any edges, and can not be closed:

> A $u - v$ path is a $u - v$ walk in which no vertex is repeated.

From Valiente[21] comes another different notion of a path:

> A trail is a walk with no repeated arcs, and a path is a trail with no repeated vertices (except, possibly, the initial and final vertices).

While all definitions may have their qualities, we have tried to select the most consistent definitions among various texts, which in this case is Valiente's notion of a path.

Even though a definition may be accurate, it may not be ideal in terms of usability. Sometimes it is better to include extra details simply to make the formalized concept easier to manipulate. Again, we use walks as an example. In the previous formalization of graphs done in Mizar[9], a walk (called a Chain) is characterized only as a finite sequence of edges. This definition also implicitly expresses the existence of a finite sequence of vertices, whose neighboring elements are joined by corresponding elements of the Chain. In the current work, a walk is defined as a single alternating sequence of both vertices and edges. The issue with the edge-only definition of a walk is that the implied finite sequence of vertices is not necessarily unique. When dealing with a walk consisting of a single edge, the order in which the two vertices occur in the vertex sequence can be in either position. This creates more work for the author, since both vertex sequences may need to be considered during proofs. With the current approach, the order in which the vertices occur along the walk is completely specified, thus avoiding the need for case analysis.

Too much information in a definition can also be a hindrance. Take for example the previous definition of a subgraph in MIZAR[9]:

```
:: Note: x c= y means x is a subset of y

definition let G be Graph;
  mode Subgraph of G -> Graph means            :: GRAPH_1:def 17
   the Vertices of it c= the Vertices of G &
   the Edges of it c= the Edges of G &
   for e st e in the Edges of it holds
    (the Source of it).e = (the Source of G).e &
    (the Target of it).e = (the Target of G).e &
    (the Source of G).e in the Vertices of it &
    (the Target of G).e in the Vertices of it;
end;
```

It turns out that for any subgraph, we do not need to claim that the source and target of any edge are vertices – this information can be derived from the rest of the definition. Including this fact in the definition results in extra unnecessary work when we want to show something is a subgraph.

Often a new definition is built upon previous definitions. A common pitfall is to overlook the nature of these previous definitions. As an example, we consider the notion of an edge joining two vertices. In the previous MIZAR work on graphs[9], a predicate was introduced to capture this notion:

```
definition let G; let x, y be Vertex of G; let v be set;
   pred v joins x, y means
:: GRAPH_1:def 9
       ((the Source of G).v = x & (the Target of G).v = y) or
       ((the Source of G).v = y & (the Target of G).v = x);
end;
```

Based on this definition, one might think that if e `joins` v_1,v_2, then e is an edge of G. Unfortunately, this is not true, because of the way the dot operator for a function is defined in MIZAR:

```
definition let f,x;
  func f.x -> set means
:: FUNCT_1:def 4
 [x,it] in f if x in dom f otherwise it = {};
end;
```

For a value that is not in the domain of a function, the dot operator returns the empty set. Thus if e is not an edge of G, and v_1 and v_2 are both equal to the empty set, e would still join v_1,v_2. In order to prevent this, we introduced a new join predicate which has the extra constraint that e must be an edge of G:

```
 definition let G be _Graph, x,y,e be set;
   pred e Joins x,y,G means
:: GLIB_000:def 14  ::dJOIN
     e in the_Edges_of G &
     (((the_Source_of G).e = x & (the_Target_of G).e = y) or
      ((the_Source_of G).e = y & (the_Target_of G).e = x));
end;
```

Another point to consider when defining a concept is the typing of its arguments. The `join` predicate was defined on vertices of G. Intuitively, this makes sense because we really are only dealing with vertices of G, however this is an unnecessary condition that can actually be derived from the `Join` predicate – if e `Joins` v_1,v_2,G then v_1, v_2 must be vertices of G. For this reason, we expanded the type of the `Joins` prediate to accept sets as arguments, instead of limiting ourselves to vertices. Avoiding unnecessary narrowing of argument types is generally a good idea in MIZAR to avoid casting. For example, instead of writing:

```
for G1 being _Graph, G2 being Subgraph of G1,
    v1,v2 being Vertex of G2, e being set holds
  e joins v1,v2 implies
    (for u1,u2 being Vertex of G1 st v1 = u1 & v2 = u2
       holds e joins u1,u2);
```

we can write:

```
for G1 being _Graph, G2 being Subgraph of G1,
    e,v1,v2 being set holds
  e Joins v1,v2,G2 implies e Joins v1,v2,G1;
```

which we feel is much easier to read and understand.

Unfortunately, there is no sure-fire way to avoid definitional problems. From personal experience, even following the guidelines listed, the only way to tell if a definition is truly good is through applied use in future articles.

3.2 Conceptual differences

Although textbook proofs and formal proofs share much in terms of logic, there are a number of conceptual details that need to be addressed. First is the idea of identity of a object. In textbooks, when dealing with a graph algorithm, the notion of "updating a graph" comes up frequently - e.g. labeling a vertex or coloring an edge. This suggests that the updated graph is the same object as the original graph, which is convenient when discussing the end result of the algorithm. Unfortunately, this doesn't hold true from a formalized point of view - the updated graph is no longer the same object as the original graph. Two structures are identical if and only if all their components are the same, including labels.

One example of this problem with identity occurs when we try to count graphs, such as the number of components of a graph. In textbooks this notion seems to be well defined, yet there is a fundamental issue that must be resolved. We begin with some definitions from West[22]:

> 1.1.16. Definition. A subgraph of a graph G is a graph H such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$ and the assignment of endpoint to edges in H is the same as in G.
>
> 1.2.8. Definition. The components of a graph G are its maximal connected subgraphs.

A problem occurs when we consider some extra features regarding graphs, such as labels. Let G be a graph, and H_1 be an unlabeled component of G. Now suppose we let H_2 be a copy of H_1, except with one of its vertices labeled. Although these two components have identical vertices and edges, they are no longer truly identical. However, both are still components of G, and would each be counted. In a textbook, H_1 and H_2 are considered to be equal, hence they would only be counted once.

There are several ways around this problem. One way would be to restrict the definition of subgraph to being graphs with no extra features such as labels. Using this approach, two subgraphs having the same vertices and edges must be identical, thus would be counted only once. Unfortunately, this approach means that we will have to create new definitions of subgraphs every time we have a new feature we would like to introduce, such as the weights needed for minimal spanning subtrees or shortest path subtrees. A less restrictive solution is to be more specific about what we are counting. In this example, we are counting components, which only takes vertices and edges into consideration, therefore we only count components that have no extra features other than vertices and edges. This solution is easily extendible for other structures with features that we may wish to count – for example if we were counting the number of minimal spanning trees in a graph, we could restrict our count to such minimal spanning trees with only vertices, edges and weights. Another solution to this problem is to not count structures, but to count representatives. For example, instead of counting actual components in a graph, we counted the subsets of vertices which induced components.

Another major conceptual difference between textbook proofs and formalized proofs is the amount of background knowledge required. With advanced topics, texts often do not need to define basic concepts such as walks, or prove theorems that are generally known. On the other hand, for a formal proof, all these background

concepts must be first defined and proven. It may come as a surprise as to how quickly the amount of work spirals when proving what seems to be a trivial theorem. To illustrate this point, we show only the graph-theory related background concepts required for a basic fact regarding trees:

> *Theorem:* A finite graph $G(V, E)$ is a tree if and only if it is connected and $|V| = |E| + 1$.
>
> *Background concepts:* Graphs, finite graphs, tree graphs, connected graphs, acyclic graphs, walks, order of a graph, size of a graph, cycles, non trivial walks, closed walks, paths, induced subgraphs, cut-vertices, isolated vertices, loopless graphs, degree of a vertex, components of a graph, number of components in a graph, subwalks of a walk, appending two walks, cutting a walk, removing a section from a walk.

We consider the proof of this theorem found in Behzad [3]:

> Exercise 2.6 Show that if G is a (p, q) graph for which $q < p - 1$, then G is disconnected.
>
> Theorem 5.2 A (p, q) graph is a tree if and only if it is acyclic and $p = q + 1$.
>
> Theorem 5.3 A (p, q) graph G is a tree if and only if G is connected and $p = q + 1$.
>
> Proof Let G be a (p, q) tree. By definition G is connected and by Theorem 5.2, $p = q + 1$. For the converse, we assume G is a connected (p, q) graph with $p = q + 1$. It suffices to show that G is acyclic. If G contains a cycle C and e is an edge of C, then $G - e$ is a connected graph of order p having $p - 2$ edges. This is impossible by Exercise 2.6; therefore, G is acyclic and is a tree.

While it is not easy to accurately gauge how much longer the MIZAR proof of this theorem is, a rough estimate would be about 2 to 10 times longer. For comparison, we include the MIZAR version of this proof, although it must be noted that the majority of work is done in previous theorems.

```
theorem tGTREE03: ::tGTREE03
  for G being finite _Graph holds
    G is Tree-like iff
      G is connected & G.order() = G.size() + 1 proof
    let G be finite _Graph;
    thus G is Tree-like implies G is connected &
      G.order() = G.size() + 1 by dGTREE,tGTREE02;
    assume
A1: G is connected & G.order() = G.size() + 1;
    now assume not G is acyclic; then
        consider W being Walk of G such that
    B1: W is Cycle-like by dGACYCLIC;
        set e = choose W.edges();
        consider G2 being removeEdge of G,e;
        W is non trivial by B1, GLIB_001:def 30; then
        W.edges() <> {} by GLIB_001:140; then
```

```
    B2: e in W.edges(); then
    B3: G2 is connected by A1, B1, tGCONNECT04;
    B4: G2.order() = G.order() &
          G2.size() + 1 = G.size() by B2,GLIB_000:48; then
    B5: G2.order() = G2.size() + 1 + 1 by A1; then
        G2.size() + 1 + 1 <= G2.size() + 1 + 0
          by B3,tGCONNECT_GF01; then
        1 <= 0 by REAL_1:53;
        hence contradiction;
    end;
    hence G is Tree-like by A1, dGTREE;
end;
```

With the creation of this graph library, it is hoped that the majority of basic concepts required for future proofs will be minimized and once stated will be heavily reused.

There are also difficulties introduced by our particular formalization of an algorithm. First of all, there is the issue of temporary data. In textbooks, we may encounter algorithms that require extra information that is modified and accessed dynamically, stored in a temporary data structure such as a priority queue. Our computation sequence consists only of graphs, therefore any such temporary information must be captured in the structure of the graph instead. Our solution to this is the use of labels. For example, we keep track of best costs to each vertex in Dijkstra'a algorithm using vertex labels. Not only is temporary data handled this way, but also static data such as the structure of the graph itself. Because of the way we formalizalized our algorithm machinery, we manipulate graphs differently than what is normally done in texts. As mentioned in the previous section, our graph operations consist of only routines which manipulate labels. By doing this, we preserve the information regarding the structure of the graph, keeping it available for the algorithm.

3.3 Incomplete details

In textbook proofs, authors rarely explain every single detail regarding correctness. This is often a sensible thing to do - saving the casual reader from pages and pages of what may very well be trivial details. These extra proof details are left to the more interested readers to figure out on their own. This luxury of passing along the work to the reader is not available to formal proof authors. For this reason, formal proofs tend to be longer than regular textbook proofs.

We take another example from West[22]:

> 1.2.10 Remark. Components are pairwise-disjoint; no two share a vertex. Adding an edge with endpoints in distinct components combines them into one component. Thus adding an edge decreases the number of components by 0 or 1, and deleting an edge increases the number of components by 0 or 1.
>
> 1.2.11 Proposition. Every graph with n vertices and k edges has at least $n - k$ components.

> Proof: An n-vertex graph with no edges has n components. By Remark 1.2.10, each edge added reduces this by at most 1, so when k edges have been added the number of components is still at least $n - k$.

West's proof is short and logically sound, however it gives no details regarding why a n-vertex graph with no edges has n components. This may be trivially obvious to a human reader, but take a moment to think about how one might go about proving this formally. Our MIZAR proof required roughly sixty lines, where we showed equality by showing that the number of components is less than or equal to the number of vertices, and vice versa.

3.4 Extra Assumptions

In most of the surveyed literature, proofs regarding graphs are not as strong as they could be, primarily due to unnecessary assumptions. For example, it is common practice in graph literature to only consider graphs that are simple and loopless. These two conditions make proofs easier to consider, since a pair of vertices can then uniquely distinguish an edge. However, many of the theorems thus proven about simple graphs are also true for general graphs. Similarly, many graph algorithms do perform correctly, even given non-simple graphs. Another example comes from the proof of Prim's minimum spanning tree algorithm from a text by Manber[11]. For the proof of correctness, it is assumed that the costs of edges are all distinct. While this guarantees that there is only one minimal spanning tree, this assumption is unnecessary and weakens the scope of the proof.

Ironically, these extra assumptions result in more work for formal proof authors. If lucky, proving the theorem without the extra assumption requires very little work - for example, prove the theorem as stated in the book assuming the extra assumption, and then prove it again without. If unlucky, the textbook proof falls apart, and a completely different proof is required.

One problem with textbook proofs is that these extra assumptions may be easily overlooked. As an example, consider the following claim from Baase[2]:

> Lemma 8.1 In a connected, weighted graph $G = (V, E, W)$, if T_1 and T_2 are two spanning trees that have the MST property, then they have the same total weight.
>
> Proof The proof is by induction on k, the number of edges that are in T_1 and not in T_2. (There are likewise exactly k edges in T_2 that are not in T_1. The base case is $k = 0$; in this case, T_1 and T_2 are identical, so they have the same weight.
>
> For $k > 0$ assume the lemma holds for trees that differ by j edges, where $0 \leq j < k$. Let uv be a *minimum-weight* edges that is in one of the trees T_1 or T_2 but not both. Assume $uv \in T_2$; the case when $uv \in T_1$ is symmetrical. Consider the (unique) path from u to v in $T_1 : w_0, w, \ldots, w_p$, where $w_0 = u, w_p = v$, and $p \geq 2$. This path must contain some edge that is not in T_2. (Why?) ....

The hidden assumption here is that the graph is simple. If there are multiple edges between u and v of minimal weight, the path from u to v in T_1 does not

necessarily need to have length greater than two. This assumption is not necessary, because minimal spanning trees are well-defined for non-simple graphs. As a side note, the question "Why?" in brackets serves as a good example of how textbook authors pass along work to the reader.

Chapter 4

Formalizing proofs of correctness

In this section, we will outline proofs of correctness for three well-known graph algorithms and discuss some of the issues we encountered while doing so. To give the reader a taste of the type of work required behind a formal MIZAR article, we will provide a large amount of detail for the proof of Dijkstra's algorithm, while sketching outlines for the other two.

4.1 Dijkstra's Single Source Shortest Path Algorithm

4.1.1 The Single Source Shortest Path Problem

Given a directed weighted graph and a starting vertex, it is interesting to find minimum-weight directed paths from the start vertex to all other vertices. This collection of paths forms a directed tree rooted at the start vertex. Dijkstra's algorithm finds such a directed tree, given the restriction that all weights are nonnegative. The underlying idea is a greedy approach - building the tree one edge at a time, with preference to the next closest vertex. In the following section we will examine the process of translating a textbook version of Dijkstra's algorithm to a formalized MIZAR version.

4.1.2 Dijkstra's Algorithm - Textbook version

We quote a description of Dijkstra's algorithm from Baase [2]. Note here that $d(u, v)$ represents the shortest distance from vertex u to v, and $W(uv)$ represents the weight of the edge uv.

dijkstraSSSP(G,n) // OUTLINE

Initialize all vertices as *unseen.*

Start the tree with the specified source vertex s; reclassify it as *tree*;

define $d(s, s) = 0$.

Reclassify all vertices adjacent to s as *fringe.*

While there are fringe vertices:

* Select an edge between a tree vertex t and a fringe vertex v such that $(d(s,t) + W(tv))$ is minimum;
* Reclassify v as *tree*; add edge tv to the tree;
* define $d(s,v) = ((d(s,t) + W(tv)))$.
* Reclassify all *unseen* vertices adjacent to v as *fringe*.

Basically, Dijkstra's algorithm grows a shortest-path tree from the source vertex s by adding vertices one at a time. Vertices are added in order of non-decreasing shortest distance from s.

4.1.3 Formal definitions

In order to begin formalizing Dijkstra's algorithm, we first need to formally define the desired result - a shortest-path tree:

Definition 4.1.1 *Given a weighted graph G and a vertex v of G, a directed shortest path tree of G rooted at v is a weighted subtree of G such that for any vertex x in G, there exists a directed path W_2 from v to x in G, such that for any directed path W_1 from v to x in G, the cost of $W_2 \leq$ the cost of W_1.*

The text written in typewriter font is the actual MIZAR definition, which the reader should be able to read with the usual effort.

```
definition let G1 be real-weighted WGraph, G2 be WSubgraph of G1,
                 v be set;
  pred G2 is_mincost_DTree_rooted_at v means
    G2 is Tree-like &
    for x being Vertex of G2 holds
      ex W2 being DPath of G2 st W2 is_Walk_from v,x &
      for W1 being DPath of G1 st W1 is_Walk_from v,x holds
        W2.cost() <= W1.cost();
end;
```

Note that this definition does not state that a directed shortest-path tree is spanning. This is to handle the unnecessary assumption being made in the textbook algorithm that the input graph is connected. Even on a disconnected graph, we can show that Dijkstra's algorithm produces a shortest-path tree, covering all vertices that are reachable in a directed sense from the source vertex.

We now make some conceptual changes to the textbook algorithm. The graphs that we will work with will have vertex-labels and edge-labels in addition to weights. Instead of labeling vertices *unseen*, we will simply not give them a label, and instead of labeling vertices by *tree*, we will label them with their shortest distance from v, (what the textbook version calls d). This is an example of using labels to store temporary information vital to the algorithm. Also, it is not necessary to label vertices *fringe*, because we simply look for edges between labeled vertices and unlabeled vertices.

We completely specify the Dijkstra computation sequence by supplying an initial graph and a transition function. Our initial `WEVGraph` is identical to the weighted input graph with respect to vertices, edges, and weights on edges, but has no labels on the edges, and just a single label of 0 on v. We give the function mapping a weighted input graph to our desired initial graph a name:

Definition 4.1.2 *Given a weighted graph G and a vertex v of G, the function* `DIJK:INIT(`G, v`)` *returns a WEVGraph whose structure is identical to G, but has no labels on any edges, and a label of* 0 *on v.*

```
definition let G be real-weighted WGraph, src be Vertex of G;
  func DIJK:Init(G, src) -> real-WEV WEVGraph equals
    G.set(ELabelSelector,{}).set(VLabelSelector,src.-->0);
```

We must force that the edge-labeling be cleared, otherwise we inherit whatever edge-labeling the original input graph G may have had.

The transition function is conditional, based on the existence of an edge going from a labeled vertex to an unlabeled vertex. Note that this condition is identical to the one tested in the while loop of the textbook algorithm. To simplify matters, we define a function which returns a set of candidate edges:

Definition 4.1.3 *Given a weighted vertex-labeled graph G, we define the function* `DIJK:NextBestEdges(`G`)` *to return a subset of edges of G. An edge e is in this subset if and only if the source of e is labeled and the target of e is unlabeled, and the sum of the vertex-label at the source of e along with the weight of e is minimal among all edges going from a labeled vertex to an unlabeled one.*

```
definition let G be real-weighted real-vlabeled WVGraph;
  func DIJK:NextBestEdges(G) -> Subset of the_Edges_of G means
    for e1 being set holds e1 in it iff
      e1 DSJoins G.labeledV(),the_Vertices_of G \ G.labeledV(),G &
      for e2 being set st e2 DSJoins G.labeledV(),
                          the_Vertices_of G \ G.labeledV(),G holds
        (the_VLabel_of G).((the_Source_of G).e1) +
          (the_Weight_of G).e1 <=
        (the_VLabel_of G).((the_Source_of G).e2) +
          (the_Weight_of G).e2;
```

We can now define the transition function as follows:

Definition 4.1.4 *Given a weighted edge-labeled vertex-labeled graph G, the function* `DIJK:Step(`G`)` *returns G if* `DIJK:NextBestEdges(`G`)` *is empty, otherwise it selects some edge e from* `DIJK:NextBestEdges(G)` *and returns a graph which inherits the structure of G, but with an extra label of 1 on e, and the sum of the vertex-label on the source of e and the weight of e on the label of the target of e.*

```
definition let G be real-WEV WEVGraph;
  set e = choose DIJK:NextBestEdges(G);
  func DIJK:Step(G) -> real-WEV WEVGraph equals
    G if DIJK:NextBestEdges(G) = {} otherwise
    G.labelEdge(e,1).labelVertex((the_Target_of G).e,
                    (the_VLabel_of G).((the_Source_of G).e) +
                    (the_Weight_of G).e);
end;
```

Here we set a value of 1 on the label of edges although this value is completely arbitrary. It must be defined in order to show that `DIJK:Step(`G`)` is truly a function. In addition, to show that `DIJK:Step(`G`)` is a function, we need that the selection

process used to choose the arbitrary edge e from `DIJK:NextBestEdges(G)` also be a function (i.e, given identical candidate sets, the edge selected would be the same). (This is what the `choose` function does in Mizar)[20].

With both the initial graph and transition function formally defined, we are now ready to define the Dijkstra computation sequence:

Definition 4.1.5 *Given a weighted graph G and a vertex v of G, we define the function* `DIJK:CompSeq(`G, v`)` *to return the sequence of WEVGraphs where the first element of the sequence is* `DIJK:Init(`G, v`)`*, and every other element in the sequence is attained by applying the function* `DIJK:Step` *to the element before it.*

```
:: Note: C.->x refers to the x-th element of the computation sequence C
  definition let G be real-weighted WGraph, src be Vertex of G;
    func DIJK:CompSeq(G,src) -> real-WEV WEVGraphSeq means
      it.->0 = DIJK:Init(G,src) &
      for n being Nat holds it.->(n+1) = DIJK:Step(it.->n);
  end;
```

Our transition function returns its argument if there are no candidate edges, therefore if there are no such edges in G_n, then $G_{n+1} = G_n$, effectively halting our algorithm. Recall that with our formal approach, the final graph we are interested in is the one at which the sequence begins to repeat, assuming the algorithm terminates. For convenience, we define another function to return this final graph:

Definition 4.1.6 *Given a weighted graph G and a vertex v of G, the function* `DIJK:SSSP(`G, v`)` *returns the WEVGraph that is the result of the computation sequence* `DIJK:CompSeq(`G, v`)`.

```
  definition let G be real-weighted WGraph, src be Vertex of G;
    func DIJK:SSSP(G,src) -> real-WEV WEVGraph equals
      DIJK:CompSeq(G,src).Result();
  end;
```

This concludes the formal definitions required for Dijkstra's algorithm. A key point to notice about these definitions is that they are as general as possible - the machinery is not restricted to non-negatively weighted graphs, or even finite graphs. These extra conditions are only necessary when we try to prove theorems.

4.1.4 Proving correctness

The first thing we will need to show is that the computation sequence we defined is halting. We now build towards this by proving some simple lemmas.

Lemma 4.1.1 *For any finite WEVGraph G, the number of labeled vertices in the graph* `DIJK:Step(`G`)` *is one more than the number of labeled vertices in G if and only if* `DIJK:NextBestEdges(`G`)` *is non-empty.*

```
  theorem tDSTEP01: ::tDSTEP01
    for G being finite real-WEV WEVGraph holds
      card DIJK:Step(G).labeledV() = card G.labeledV() + 1 iff
      DIJK:NextBestEdges(G) <> {}
```

Proof: $\Rightarrow$ Assume that the number of labeled vertices in the graph `DIJK:Step(`G`)` is exactly one more than the number of labeled vertices in the graph G. Suppose that `DIJK:NextBestEdges(`G`)` is empty. In this case, by definition 4.1.4, `DIJK:Step(G)` $= G$, which leads to a contradiction. Thus `DIJK:NextBestEdges(`G`)` is not empty.

$\Leftarrow$ Assume that `DIJK:NextBestEdges(`G`)` is not empty. In this case, by definition of `DIJK:Step`, we label the previously unlabeled target vertex of the selected candidate edge. Hence the number of labeled vertices in `DIJK:Step(`G`)` is one greater that the number of labeled vertices in G. □

Lemma 4.1.2 *For any finite weighted graph G and vertex v, the set of labeled vertices in the n^{th} element of* `DIJK:CompSeq(`G, v`)` *is a subset of the vertices reachable from v by a directed walk in G.*

```
theorem tDCS03:
  for G being finite real-weighted WGraph, v being Vertex of G,
      n being Nat holdsa
    (DIJK:CompSeq(G,src).->n).labeledV() c= G.reachableDFrom(v)
```

Proof: Proof by induction. In the base case, the only labeled vertex is v, which is reachable from itself. For the inductive step, we only add a new label to a vertex if it is adjacent to a candidate edge and a previously labeled vertex. By the inductive assumption, the previously labeled vertex is reachable from v. By taking some directed walk from v to the previously labeled vertex, and adding the candidate edge, we get that the newly labeled vertex is also reachable from v. □

Lemma 4.1.3 *For any finite weighted graph G, and vertex v of G, the set of candidate edges returned by* `DIJK:NextBestEdges` *applied on the n^{th} element of* `DIJK:CompSeq(`G, v`)` *is empty if and only if the set of labeled vertices in the n^{th} element is equal to the set of all vertices reachable from v by a directed walk in G.*

```
theorem tDCS05:
 for G being finite real-weighted WGraph, v being
  Vertex of G, n being Nat holds
  DIJK:NextBestEdges(DIJK:CompSeq(G,src).->n) = {} iff
  (DIJK:CompSeq(G,src).->n).labeledV() = G.reachableDFrom(v);
```

Proof: $\Rightarrow$ Assume that `DIJK:NextBestEdges(`G_n`)` is empty. Now suppose the set of labeled vertices in G_n is not the same as the set of all vertices reachable from v. From Lemma 4.1.2, this means that set of labeled vertices in G_n is a strict subset of the vertices reachable from v. Because of this, there must exist some unlabeled vertex x of G that is reachable from v by some directed walk W. Let y be the first vertex along this walk that is not labeled in G_n. y can not be the first vertex of W because the first vertex is v, which has been labeled for every graph in the computation sequence. This means that there exists an edge e that comes right before vertex y, which joins a labeled vertex to an unlabeled vertex. Let E represent the set of edges which join a labeled vertex to an unlabeled vertex. E is a subset of the edges of G, which is finite, therefore E is also finite. In addition, E is non-empty, since e is in E. Because E is finite and non-empty, we can then select a candidate edge out of it - i.e. an edge that minimize the sum of the label at the source of the edge and the weight of the edge among all edges of E. However, this candidate edge

is an element of DIJK:NextBestEdges(G_n), which leads to our desired contradiction since we assumed DIJK:NextBestEdges(G_n) to be empty. Hence the set of labeled vertices in G_n is equal to the set of vertices reachable from v by a directed walk in G.

$\Leftarrow$ Assume that the set of labeled vertices in G_n is equal to the set of all vertices reachable from v by a directed walk in G. If DIJK:BestEdges(G_n) is not empty, then by Lemma 4.1.1, the number of vertices labeled in G_{n+1} will be one more than the number of vertices reachable from v. This would be impossible since by Lemma 4.1.2, we know that the set of labeled vertices in G_{n+1} is a subset of the reachable vertices. □

Lemma 4.1.4 *For any finite weighted graph G, and vertex v, the number of labeled vertices in the n^{th} element of* DIJK:CompSeq(G, v) = *min($n+1$, # of vertices reachable from v by a directed walk in G).*

```
theorem tDCS06:
  for G being finite real-weighted WGraph, v being Vertex of G,
      n being Nat
    holds Card (DIJK:CompSeq(G,v).->n).labeledV() =
      min(n+1, card(G.reachableDFrom(v)))
```

Proof: The proof is by induction. Let r be the number of vertices reachable from v by a directed walk in G. When $n = 0$, the n^{th} element is actually DIJK:Init(G, v), so there is only one labeled vertex – v. r is at least one because v is reachable from itself, therefore the $\min(0+1, r) = 1$. Thus the base case holds.

Assuming that the claim is true for some arbitrary n, we now show that it remains true for $n+1$. By definition 4.1.5, G_{n+1} = DIJK:Step(G_n). There are two cases to consider. First suppose that DIJK:NextBestEdges(G_n) is empty. In this case, $G_{n+1} = G_n$, so by the inductive assumption, the number of labeled vertices in G_{n+1} is $\min(n+1, r)$. By Lemma 4.1.3, the number of labeled vertices in $G_{n+1} = r$, therefore $r = min(n+1)$. This means that $r \leq n+1$, and by transitivity, $r \leq n+1+1$. Thus the number of labeled vertices in $G_{n+1} = r = min(n+1+1, r)$. Now suppose that DIJK:NextBestEdges(G_n) is not empty. In this case, G_{n+1} = DIJK:Step(G_n), and by Lemma 4.1.3, the set of labeled vertices in G_n is not equal to the set of all vertices reachable from v by a directed walk in G. Along with Lemma 4.1.2, this means that the set of labeled vertices in G_n is a strict subset of the vertices reachable from v by a directed walk in G, hence their cardinalities can not be the same. By the inductive assumption, the number of labeled vertices in $G_n = min(n+1, r)$, but since this number can not be r, it must be $n+1$ with $n+1 < r$. By Lemma 4.1.1, the number of labeled vertices in G_{n+1} is then $n+1+1$. $n+1+1 <= r$, hence the number of labeled vertices in $G_{n+1} = min(n+1+1, r)$. □

Theorem 4.1.5 *For any finite weighted graph G and vertex v, the computation sequence* DIJK:CompSeq(G, v) *halts.*

```
theorem tDIJK01:
  for G being finite real-weighted WGraph, src being Vertex of G
    holds DIJK:CompSeq(G,src) is halting
```

Proof: In order to show that DIJK:CompSeq(G, v) halts, we need to find an n, such that the n^{th} and $n + 1^{th}$ elements of DIJK:CompSeq(G, v) are identical. Set n to be equal to the number of vertices that are reachable from v via directed walks in G. Let G_n, G_{n+1} be the n^{th} and $n + 1^{th}$ elements of DIJK:CompSeq(G, v). We will show that $G_n = G_{n+1}$. By Definition 4.1.5, G_{n+1} = DIJK:Step(G_n). If DIJK:NextBestEdges(G_n) is empty, then G_{n+1} = DIJK:Step$(G_n) = G_n$ (by Def 4.1.4) and we are done. Otherwise, suppose DIJK:NextBestEdges(G_n) is not empty. By Lemma 4.1.1, the number of labeled vertices in G_{n+1} is then one more than the number of labeled vertices in G_n. However, by Lemma 4.1.4, the number of labeled vertices in $G_n = min(n+1, n) = n$, and the number of labeled vertices in $G_{n+1} = min((n+1), n) = n$. Therefore DIJK:NextBestEdges(G_n) must be empty and $G_n = G_{n+1}$. □

For Dijkstra's algorithm, we can even prove the exact iteration at which the algorithm terminates:

Theorem 4.1.6 *For any finite weighted graph G and vertex v, the lifespan of* DIJK:CompSeq(G, v) *is equal to one less than the number of vertices reachable from v via a directed walk in G.*

```
theorem tDIJK02:
  for G being finite real-weighted WGraph,
      src being Vertex of G holds
    DIJK:CompSeq(G,src).Lifespan()+1 = card G.reachableDFrom(src)
```

Proof: Set k = (the number of vertices reachable from v via a directed walk in G)-1. The vertex v is reachable from v, which means k is nonnegative, hence we can safely talk about the graphs G_k and G_{k+1}. From Lemma 4.1.4, there are the same number of labeled vertices in G_k as in G_{k+1}. Combined with Lemma 4.1.1, we get that DIJK:NextBestEdges(G_k) is empty, which also means that $G_k = G_{k+1}$ by Definition 4.1.4. We still need to show that k is the earliest point at which the sequence repeats. This turns out to be fairly simply to show, since for any other $n < k$, the number of vertices in G_n and G_{n+1} differ by one, according to Lemma 4.1.4. □

Now that we have shown that the algorithm halts, as well as the exact point at which it does so, we prove that the end result gives us information about a shortest-path subtree whose vertices are all those reachable from the source vertex via a directed walk. In addition, we will show that the label on each vertex in the subtree is labeled with the shortest distance between it and the source.

Theorem 4.1.7 *For any finite WGraph G and vertex v, the set of vertices labeled in* DIJK:SSSP(G, v) *is the set of vertices reachable from v via a directed walk in G.*

```
theorem tDIJK03:
  for G being finite real-weighted WGraph, src being Vertex of G
    holds DIJK:SSSP(G,src).labeledV() = G.reachableDFrom(src)
```

Proof: Let R represent the set of vertices reachable from v via a directed walk in G, and L represent the vertices labeled in DIJK:SSSP(G, v). Suppose $L \neq R$.

By Lemma 4.1.2, this means that $L \subset R$, hence $|L| < |R|$. However, from Theorem 4.1.6 and Lemma 4.1.4 we get that $|L| = |R|$, giving us our contradiction. □

Lemma 4.1.8 *For any finite WGraph G and vertex v, the edges labeled in the n^{th} element of* `DIJK:CompSeq(`G, v`)` *is a subset of the edges between the vertices labeled in the n^{th} element of* `DIJK:CompSeq(`G, v`)`.

```
for G being finite real-weighted WGraph, src being Vertex of G,
    n being Nat holds
  DIJK:CompSeq(G,src).->n).labeledE() c=
 (DIJK:CompSeq(G,src).->n).edgesBetween(
                          (DIJK:CompSeq(G,src).->n).labeledV()
```

Proof: Proof is by induction. In the base case, no edges are labeled, hence the condition holds trivially. Now assume the condition is true for some arbitrary n. If `DIJK:NextBestEdges(`G_n, v`)` is empty, then $G_{n+1} = G_n$, therefore the condition is true by our inductive assumption. If `DIJK:NextBestEdges(`G_n, v`)` is non-empty, we add a new edge e to our labeling e, as well as the unlabeled target vertex of e. Both of the vertices that e joins are labeled in G_{n+1}, therefore the condition holds for $n + 1$. □

Lemma 4.1.9 *For any finite WGraph G with nonnegative-weights, and vertex v, any weighted subgraph induced by the labeled vertices and edges in the n^{th} element of* `DIJK:CompSeq(`G_n, v`)` *forms a directed shortest-path tree of G rooted at v. In addition, every vertex labeled in the n^{th} element of* `DIJK:CompSeq(`G, v`)` *is marked with the minimum cost of a directed walk from v to it.*

```
theorem tDCS08:
  for G being finite nonnegative-weighted WGraph,
      src being Vertex of G, n being Nat,
      G2 being inducedWSubgraph of G,
                           (DIJK:CompSeq(G,src).->n).labeledV(),
                           (DIJK:CompSeq(G,src).->n).labeledE()
    holds G2 is_mincost_DTree_rooted_at src &
      for v being Vertex of G
        st v in (DIJK:CompSeq(G,src).->n).labeledV()
        holds G.min_DPath_cost(src,v) =
              (the_VLabel_of (DIJK:CompSeq(G,src).->n)).v
```

Proof: Proof is by induction. In the base case, there are no edges labeled, and v is the only vertex labeled, hence the induced weighted subgraph forms a tree. The label at v is 0, which is the absolute minimum cost of a walk in a nonnegative-weighted graph, hence the graph is a directed shortest-path tree of G rooted at v. For any given n, assume the condition holds. If `DIJK:NextBestEdges(`G_n`)` is empty, then $G_{n+1} = G_n$, hence the condition holds by the inductive assumption. Otherwise, there is one new edge e that is now labeled in G_{n+1}, as well as the target vertex of e, which we call t. Let WS_n and WS_{n+1} be some arbitrary weighted subgraphs induced by the labeled edges and vertices in G_n and G_{n+1}, respectively. By Lemma 4.1.8, we know that the labeled edges are a subset of the labeled vertices, therefore we are safe to talk about such induced subgraphs. The labeled vertices and edges of G_n are a subset of the labeled vertices and in G_{n+1}, therefore WS_n is a weighted subgraph

of WS_{n+1}. From the inductive assumption, WS_n is a directed tree, therefore it is connected. We can use this to show that WS_{n+1} is also connected. Consider any two distinct vertices in WS_{n+1}, which we call x and y. If $x \neq t$ and $y \neq t$, then x and y are vertices in WS_n, hence there exists a directed walk in WS_n that goes from one to the other. WS_n is a subgraph of WS_{n+1}, so this walk also exists in WS_{n+1}. On the other hand, without loss of generality, suppose $y = t$. We can consider a directed walk in WS_n from x to the source vertex of e, which is also a walk in WS_{n+1}. Appending the edge e to the end of this walk gives us a directed walk that goes from x to y, thus showing that WS_{n+1} is connected. The number of vertices in WS_{n+1} is one more than the number of vertices in WS_n, and the same holds true for the number of edges. Being a tree, WS_n has one more vertex than edge, and using the above fact, the same must hold for WS_{n+1}. This condition, along with WS_{n+1} being connected is enough to show that WS_{n+1} is a directed tree rooted at v.

We still need to show that WS_{n+1} forms a directed shortest-path tree and that labels are marked with minimum costs from v. Consider some vertex x in WS_{n+1}. If $x \neq t$, then x is a vertex in WS_n, and by the inductive assumption, there exists a directed path in WS_n going from v to x of minimum cost. Such a directed path is also a directed path in WS_{n+1}. Here we still need to show that the cost of such a path does not change, but this is true, since WS_n is a weighted subgraph of WS_{n+1}. If $x = t$, then consider some minimum cost directed path in WS_n from v to the source vertex of e. Such a directed path is also a path in WS_{n+1}, and we can append the edge e to get a directed walk P that runs from v to t. Neither e nor t were in G_n, therefore this P is actually a directed path. The cost of P is the sum of the minimum-cost path from v to the source vertex of e and the cost of e. By the inductive argument, this is also equal to sum of the value of the label at the source vertex of e with the cost of e. Recall that e is an element of `DIJK:NextBestEdges(`G_n`)`, and therefore the sum of the vertex-label at the source of e along with the weight of e is minimal among all edges going from a labeled vertex to an unlabeled one 4.1.3. We need to show that P has minimum cost, which is done by contradiction. Suppose there exists a directed path M from v to t with a cost strictly less than the cost of P. Let u be the first vertex of M that is unlabeled in G_n. Such a vertex must exist since t is unlabeled, and in addition, u will not be the first vertex of M, since v is labeled. By our choice of e, the cost of P is less than or equal to the subpath of M formed by considering the section from v up to u. G_n has the same nonnegative weights as G, therefore the cost of this subpath is less than or equal to the cost of M. Hence, by transitivity, the cost of P is less than or equal to the cost of M, which gives us our desired contradiction. □

Theorem 4.1.10 *For any finite WGraph G with nonnegative-weights, and vertex v, any weighted subgraph induced by the labeled vertices and edges in* `DIJK:SSSP(`G, v`)` *forms a directed shortest-path tree of G rooted at v. In addition, any vertex that is reachable via a directed walk in G is labeled in* `DIJK:SSSP(`G, v`)` *with the shortest cost of a directed path from v to it.*

```
theorem tDIJK04
  for G being finite nonnegative-weighted WGraph,
      src being Vertex of G, G2 being inducedWSubgraph of G,
                                DIJK:SSSP(G,src).labeledV(),
```

```
                                DIJK:SSSP(G,src).labeledE() holds
  G2 is_mincost_DTree_rooted_at src &
  for v being Vertex of G st v in G.reachableDFrom(src) holds
    v in the_Vertices_of G2 &
    G.min_DPath_cost(src,v) = (the_VLabel_of DIJK:SSSP(G,src)).v
```

Proof: DIJK:SSSP(G, v) is an element of DIJK:CompSeq(G, v), hence most of what we want to prove comes from 4.1.9. We get that all reachable vertices are labeled in DIJK:SSSP(G, v) from 4.1.7.

This concludes our English translation of the MIZAR proof of correctness for Dijkstra's algorithm.

4.1.5 Discussion

As this was the first graph algorithm to be verified in MIZAR, the majority of the work went towards establishing the necessary background objects needed, such as the notion of trees, walks and components. As an illustration, the total number of lines we used to define machinery specific to Dijkstra's algorithm and for proving the correctness of the algorithm was roughly 1400, while we had to write over 18000 lines to cover all the necessary background library definitions and theorems.

4.2 Prim's Minimum-Weight Spanning Tree Algorithm

Prim's algorithm for finding minimum spanning trees was the second graph algorithm we verified in MIZAR. Interestingly, none of the texts we checked had proofs that could handle graphs with multiple edges between vertices, although Prim's algorithm still works correctly on such graphs.

4.2.1 The Minimum-Weight Spanning Tree Problem

Given a graph G, we say that a subgraph H *spans* the vertices of G if the vertices of G are identical to the vertices of H. For a connected graph G, it can be shown that there exist subtrees that spans the vertices of G. Given a weighted graph, Prim's algorithm finds such a spanning subtree with minimum total edge weights. Minimum spanning trees have a variety of practical applications such as in communication and transportation networks.

4.2.2 Prim's Algorithm - Textbook version

Prim's algorithm for finding minimum spanning trees is a classic example of a greedy algorithm. Skiena [19] describes Prim's algorithm as follows:

Prim-MST(G)

Select an arbitrary vertex s to start the tree from.

While (there are still nontree vertices)

- Select the edge of minimum weight between a tree and nontree vertex
- Add the selected edge and vertex to the tree T_{prim}.

In order to prove correctness in MIZAR, we must first adjust this algorithm slightly. Instead of growing a tree, we will simply label the edges and vertices that are involved, leaving the underlying graph intact. We need this information in order to properly define a transition function for the computation sequence. Here is our modifed Prim's algorithm:

Input: A finite connected graph G

Output: A minimum-weight spanning tree of G.

Pseudo-code:

1. Label all vertices as *unseen.*
2. Arbitrarily select some vertex of G to mark as *seen*
3. While there are edges going from a *seen* vertex to an *unseen* vertex:
   (a) Select an edge e, going from a *seen* vertex u to an *unseen* vertex v such that the weight of e is minimal among all edges going from *seen* vertices to *unseen* vertices.
   (b) Mark e and v as *seen*

The machinery for Prim's algorithm is quite similar to the one created for dealing with Dijkstra's algorithm. The only main difference is which edges we select as candidates and how to label them.

4.2.3 Proving correctness

Before we begin, we present a proof found at [10]:

> Let P be a connected, weighted graph. At every iteration of Prim's algorithm, an edge must be found that connects a vertex in a subgraph to a vertex outside the subgraph. Since P is connected, there will always be a path to every vertex. The output Y of Prim's algorithm is a tree, because the edge and vertex added to Y are connected to other vertices and edges of Y and at no iteration is a circuit created since each edge added connects two vertices in two disconnected sets. Also, Y includes all vertices from P because Y is a tree with n vertices, same as P. Therefore, Y is a spanning tree for P.
>
> Let Y_1 be any minimal spanning tree for P. If $Y = Y_1$, then the proof is complete. If not, there is an edge in Y that is not in Y_1. Let e be the first edge that was added when Y was constructed. Let V be the set of vertices of $Y - e$. Then one endpoint of e is in Y and another is not. Since Y_1 is a spanning tree of P, there is a path in Y_1 joining the two endpoints. As one travels along the path, one must encounter an edge f joining a vertex in V to one that is not in V. Now, at the iteration when e was added to Y, f could also have been added and it would be added instead of e if its weight was less than e. Since f was not added, we conclude that $w(f) \geq w(e)$.

Let Y_2 be the tree obtained by removing f and adding e from Y_1. It shows that Y_2 is a tree that is more common with Y than with Y_1. If Y_2 equals Y, QED. If not, we can find a tree, Y_3 with one more edge in common with Y than Y_2 and so forth. Continuing this way produces a tree that is more in common with Y than with the preceding tree. Since there are finite number of edges in Y, the sequence is finite, so there will eventually be a tree, Y_h, which is identical to Y. This shows Y is a minimal spanning tree.

Unlike some of the other proofs we encountered, (e.g. [2]), this proof does not rely on G being a simple graph. To an unattentive reader, this proof may seem simple and logically sound. Unfortunately though, there is something seriously wrong with this proof. In some cases, the edge f in Y_1 is an edge that is also in Y. This means that adding the edge e and then removing the edge f does not increase the number of edges in common with Y. We outline a similar but correct proof below, which is the one we translated into MIZAR:

Let v_1 be the arbitrary vertex that we selected to grow our minimum spanning tree from. To prove that the algorithm halts, we can show that at the n^{th} step of the computation, the number of labeled vertices $= min(n+1,\#$ of vertices reachable from $v_1)$. This gives that the lifespan of our algorithm is exactly one less than the number of vertices reachable from v_1, and that in the final result all vertices are labeled. Our proof is done via induction, similar to the proof regarding the lifespan of Dijkstra's algorithm.

We need to show that any graph induced by the labeled edges and vertices in the final result of Prim's algorithm forms a tree. This is actually true for any graph in our computation sequence - a fact we also prove by induction. In the base case, there is only a single vertex labeled, and no edges which forms a trivial tree. For the inductive step, we are adding a new edge that is incident to a vertex in the previous tree and a new vertex, which guarantees that the new graph is also a tree.

The last thing to show is that the spanning tree is minimal. Let P be a weighted subgraph induced by the labeled vertices and edges in Prim's result. Assume that P is not minimal. Consider a minimum spanning tree M that has the highest number of edges in common with P. There is also one extra tie-breaking condition for the selection of M. Let e_n represent the edge labeled in the n^{th} step of the algorithm. We select M such that it shares the edges $e_1, e_2, \ldots, e_k$ for as large a value of k as possible. In other words, up to the k^th step, Prim's algorithm choose edges that are in M, but the edge e_{k+1} is not in M. We can safely talk about the edge e_{k+1} since there must be some edge in M that is not in P, otherwise P would share the same vertices, edges, and weights as M and therefore be minimal. Let s and t be the vertices adjacent to e_{k+1}, where t is vertex unlabeled in G_k. Appending e_{k+1} to the unique path from s to t in M forms a cycle. Because s is labeled, and t is unlabeled, we can find some edge e_M that is along the path from s to t in M that goes from an unlabeled vertex to a labeled vertex in G_k. Let M_2 be the graph formed by adding the edge e_{k+1} to M while removing e_M. M_2 still has the same number of vertices and edges as M, and is also connected, therefore is a tree. The vertices have not changed, therefore M_2 is a spanning tree. By our choice of e_{k+1}, the weight of e_M is greater than or equal to the weight of e_{k+1}, hence the weight of M_2 is less than or

equal to that M. However, M is a minimal, therefore M_2 must also be a minimal spanning tree.

We now show a contradiction by proving that M_2 should have been selected in lieu of M. There are two cases to consider. If e_M is not an edge in P, then the total number of edges that M_2 shares in common with P is one greater than the number of edges M shares with P, which contradicts with our choice of M. If e_M is an edge of P, then the total number of edges that M_2 shares in common with P is identical to the number of edges M shares with P. From our choice of M, this means that M_2 contains the edges $e_1, e_2, \ldots, e_i$ for some value $1 \leq i < k$, $i \neq k$ otherwise M_2 would contain the edges $e_1, e_2, \ldots, e_k, e_{k+1}$, which contradicts our choice of M. Only e_M was removed from M, which means that e_M is one of the edges $e_1, e_2, \ldots, e_k$. However all vertices incident to any of the edges $e_1, e_2, \ldots, e_k$ are already labeled in G_k, while t is unlabeled, giving us our desired contradiction.

4.2.4 Discussion

There were two main issues when dealing with Prim's algorithm. The first was finding a correct proof that was suitable for use for graphs that were not necessarily simple. Realistically, only a single edge of minimum cost should be considered if there are multiple edges between two vertices, while self-looping edges may be discarded, however we felt it was important to formalize a proof that handled these cases properly. The other main issue was formalizing the cost of a subgraph - namely getting the sum of all the weights of all the edges in a subgraph. Summation is a tricky obstacle in MIZAR, and it was necessary to augment the machinery from [17, 18] in order to create a proper definition. Although the MML continues to improve, there are still many concepts that have not been formalized that may be interesting towards graphs. For example, in order to properly deal with planar graphs, we would need a proof of the Jordan Curve theorem, which is still a major uncompleted project in MIZAR.

As this was our second algorithm to be proven, most of the background work was already done. In terms of writing, it took roughly 4500 lines to define and prove all the material directly relating to Prim's algorithm, while another 400 lines was used towards adding additional background material. The reason why Prim's algorithm requires so many more lines than Dijkstra's algorithm is mainly due to dealing with the summations.

On a side note, it was during our work on Prim's algorithm that we decided that MIZAR's built-in system for structures was not sufficient for graphs, hence we started to use our own attribute-based system. See Appendix A for more information regarding MIZAR structures.

4.3 Ford & Fulkerson's Maximum Network Flow Algorithm

4.3.1 The Maximum Network Flow Problem

Given a directed graph G with capacities on each edge, a source vertex s and a sink vertex t, we say that G has a *valid flow* from s to t if two conditions hold. The first is that every edge e has a value assigned between zero and the capacity

on e, which we call the *flow on e*. The second condition is that for every vertex v other than s and t, the sum of the flows on all edges directed into v is equal to the sum of the flows on all edges coming out of v. This second condition is often called *conservation of flow*. For a graph that has a valid flow, we define the *value of the flow* to be equal to the sum of flows on edges going into the sink minus the sum of the flows on edges coming out of the sink. In the maximum network flow problem, we are asked to find an assignment of values on edges that results in the maximum possible flow value.

4.3.2 Ford/Fulkerson's Maximum Network Flow Algorithm

The algorithm which we proved correct using MIZAR is credited to Ford and Fulkerson. The basic idea behind the algorithm is finding and increasing the flow along augmenting paths. Augmenting paths are paths from s to t along which flow can be increased.

The pseudo-code of the algorithm is very simple to state:[5]:
FORD-FULKERSON-METHOD(G, s, t)

1. initialize flow f to 0
2. **while** there exists an augmenting path p
3. **do** augment flow f along p
4. **return** f

The proof of the algorithm relies on the claim that a graph has maximum flow if and only if it does not have any augmenting paths. This proof is quite well documented and referred to as the Max-flow Min-cut theorem[7]. Because our MIZAR proof is very similar to the proofs found in several texts [5, 22], we focus more on the details regarding how we formalize the algorithm machinery itself.

Every edge in our input graph is assigned a capacity that does not change, thereby making them equivalent to weights. We will be modifying the flow along each edge, therefore we use edge-labels to represent the amount of flow on each edge. Hence, each element of our main computation sequence will be composed of `WEGraphs`.

The unique feature of the Ford/Fulkerson algorithm versus the two other algorithms we have formalized is the use of subroutines. We can break down the Ford/Fulkerson algorithm into three subroutines:

1. Testing for the existence of an augmenting path in a graph.
2. Extracting an augmenting path, if one exists.
3. Increasing the flow along a given augmenting path.

In order to test a finite graph G for augmenting paths from source s to sink t we introduced an algorithm that creates a computation sequence defined as follows.

Initialize G_0 as the graph having the same structure as G, but with s labeled with the value 1. The value is really arbitrary, but marks that the source vertex is in the set of labeled vertices. For any value n, the graph G_{n+1} is attained by checking

for candidate edges in G_n. An edge is a candidate if flow can be increased along it. There are two types of candidate edges - *forward labeling*, and *backwards labeling*. An edge e is forward labeling if the source vertex of e is labeled, the target vertex of e is unlabeled and the current flow on e is strictly less than the capacity on e. An edge e is backward labeling if the target vertex of e is labeled, the source vertex of e is unlabeled, and the current flow on e is strictly greater than zero. If no candidates edge exist we define $G_n = G_{n+1}$, otherwise we select an arbitrary candidate edge e, and label its currently unlabeled source or target vertex with the name of the edge.

In MIZAR, our algorithm machinery looks very similar to that of Prim's and Dijkstra's, with the notion of a set of candidate edges, a step function.

```
definition let G be real-weighted real-elabeled WEVGraph;
  func AP:NextBestEdges(G) -> Subset of the_Edges_of G means
     for e being set holds e in it iff
     (e is_forward_labeling_in G or e is_backward_labeling_in G);
end;

definition let G be real-weighted real-elabeled WEVGraph;
  set e = choose AP:NextBestEdges(G);
  func AP:Step(G) -> real-weighted real-elabeled WEVGraph equals
     G if AP:NextBestEdges(G) = {},
     G.labelVertex((the_Source_of G).e, e) if
          AP:NextBestEdges(G) <> {} &
          not (the_Source_of G).e in G.labeledV() otherwise
     G.labelVertex((the_Target_of G).e, e);
end;

definition let G be real-weighted real-elabeled WEGraph,
               source be Vertex of G;
  func AP:CompSeq(G,source) -> real-weighted real-elabeled
                                WEVGraphSeq means
    it.->0 = G.set(VLabelSelector, source .--> 1) &
    for n being Nat holds it.->(n+1) = AP:Step(it.->n);
end;

definition let G be real-weighted real-elabeled WEGraph,
               source be Vertex of G;
  func AP:FindAugPath(G,source) -> real-weighted real-elabeled
                                   WEVGraph equals
  AP:CompSeq(G,source).Result();
end;
```

We show that this algorithm halts via contradiction. If there is no n such that $G_n = G_{n+1}$, it must be that there is always some candidate edge in G_n. However, whenever there is a candidate edge, an unlabeled vertex becomes labeled and there are only a finite number of unlabeled vertices.

We can show that for any point in the computation sequence there exists an augmenting path from the source to any vertex that is labeled. This is done by induction, and showing that adding a candidate edge to an augmenting path results in a new augmenting path. In our proof, we also show that all vertices for which there exists an augmenting path from the source will be labeled in the final result of our computation sequence. Hence, in order to determine if a given graph has any

augmenting paths from source to sink, it is sufficient to check if the sink is labeled in the final result of the algorithm.

There is still the issue of extracting the augmenting path, assuming it exists. The routine that we build for this purpose relies on the labels on the vertices produced by the previous algorithm. Because each labeled vertex other than the source gives us the candidate edge that the algorithm chose in order to reach that particular vertex, we can build an augmenting path by following edges backwards from the sink. We defined a function in MIZAR that did exactly this:

```
definition let G be finite real-weighted real-elabeled WEGraph,
              source,sink be Vertex of G;
  func AP:GetAugPath(G,source,sink) -> vertex-distinct
                                       augmenting Path of G means
    it is_Walk_from source,sink &
    for n being even Nat st n in dom it holds
      it.n = (the_VLabel_of AP:FindAugPath(G,source)).(it.(n+1)) if
    (sink in AP:FindAugPath(G,source).labeledV()) otherwise
    it = G.walkOf(source);
```

This is a conditional function that depends on whether the sink has been labeled by the previous algorithm for finding augmenting paths. It returns the unique augmenting path from source to sink as found in `AP:GetAugPath(`G, *source*`)`.

The last subroutine takes the augmenting path we have extracted and increases the flow along it. For a forward labeling edge, we can increase the flow at most *(capacity - current flow)* units. For a backwards labeling edge, we increase the flow along the path by decreasing it for that edge at most *(current flow)* units. These two limits are necessary to preserve that the graph has valid flow. Given an augmenting walk W, we define the *tolerance* of W as the minimum of all possible flow increases along the edges of W. In order to increase the flow safely along the path P, we increase the flow along forward labeling edges and decrease flow along backwards labeling edges by the tolerance of P. The new edge labeling resulting from this looks as follows in MIZAR:

```
definition let G be real-weighted real-elabeled WEGraph,
              P be augmenting Path of G;
  func FF:PushFlow(G,P) -> ManySortedSet of the_Edges_of G means
   (for e being set st e in the_Edges_of G & not e in P.edges()
       holds it.e = (the_ELabel_of G).e) &
   (for n being odd Nat st n < len P holds
     (P.(n+1) DJoins P.n, P.(n+2),G implies
      it.(P.(n+1)) = (the_ELabel_of G).(P.(n+1))+P.tolerance()) &
     (not P.(n+1) DJoins P.n,P.(n+2),G implies
      it.(P.(n+1)) = (the_ELabel_of G).(P.(n+1))-P.tolerance()));
end;
```

Basically, this function does not modify the flow on edges that are not part of P, and increases or decreases the amount of flow as necessary for forwards and backwards labeling edges of P.

With these three subroutines defined, we can now build the machinery for the main Ford/Fulkerson algorithm:

```
definition let G be real-weighted real-elabeled WEGraph,
              P be augmenting Path of G;
  func FF:AugmentPath(G,P) -> real-weighted real-elabeled
                              complete-elabeled WEGraph equals
    G.set(ELabelSelector, FF:PushFlow(G,P))
end;

definition let G be finite real-weighted real-elabeled
                    complete-elabeled WEGraph,
              sink, source be Vertex of G;
  func FF:Step(G, source, sink) -> finite real-weighted
                                   real-elabeled complete-elabeled
                                   WEGraph equals
    FF:AugmentPath(G, AP:GetAugPath(G,source,sink))
    if sink in AP:FindAugPath(G,source).labeledV()
  otherwise G;
end;

definition let G be finite real-weighted WGraph,
              source,sink be Vertex of G;
  func FF:CompSeq(G,source,sink) -> finite real-weighted
                                   real-elabeled complete-elabeled
                                   WEGraphSeq means
    it.->0 = G.set(ELabelSelector, the_Edges_of G --> 0) &
    for n being Nat holds ex source',sink' being Vertex of it.->n
     st source' = source & sink' = sink &
     it.->(n+1) = FF:Step(it.->n,source',sink');
end;

definition let G be finite real-weighted WGraph,
              sink,source be Vertex of G;
  func FF:MaxFlow(G,source, sink) -> finite real-weighted
                                   real-elabeled complete-elabeled
                                   WEGraph equals
  FF:CompSeq(G,source,sink).Result();
end;
```

Initially, Ford/Fulkerson sets the flows on each of the edges to 0, thereby producing a graph with valid flow. In the step function, it assigns the new labeling produced by augmenting the path from the source to the sink as found by `AP:GetAugPath`$(G, source, sink)$, assuming that such a walk was found in by the subroutine `AP:FindAugPath`$(G, source)$. Given non-negative rational weights, we can show that the Ford/Fulkerson computation sequence halts because each iteration increases the total flow, which is limited by the capacities of the edges going in and out of the sink. By the Max-flow Min-cut theorem, we know that the flow is maximal because there are no more augmenting paths from the source to the sink when we reach the result of computation sequence.

4.3.3 Discussion

The machinery required for the Ford/Fulkerson algorithm is more intricate than the other two previously covered algorithms, however breaking down the algorithm into

simple subroutines made the proof much easer to handle. Our proof required roughly 5200 lines, and we added another 400 lines towards background library materials. The two main lemmas that help prove the Max-flow Min-cut theorem deal with showing that the flow can be measured along a source/sink cut. As seen in [6], this is done mainly by manipulating sums, hence our proof followed very closely with the books. As far as we know, this is the first formalized proof of the Ford/Fulkerson algorithm.

Chapter 5

Conclusion

Establishing an extensive library of graph definitions and facts was a focal point for our work. By proving the correctness of algorithms of Dijkstra, Prim and Ford/Fulkerson, we have begun to show that our library is robust and suitable for use towards more complex graph algorithms. Currently the prepared library is already in use by another student who is in the process of verifying an algorithm for recognizing triangulated graphs, with the intention of working in the field of perfect graphs.

There are many conceptual differences and technical difficulties when translating a proof from a textbook form to MIZAR, but at the same time, there are many benefits of having a mechanized proof, such as automated verification and the ability for cataloging. It is also feasible that with future work, actual code may be extracted mechanically from these correctness proofs. While the length of a textbook proof is almost always much shorter than a formal MIZAR proof, we would like to point out that the majority of work is often directed towards proving background materials.

Future work will continue to expand the graph library to touch upon a variety of other topics. Suitable choices that MIZAR are quite adept at handling at the moment include perfect graphs, intersection graphs, graph coloring and graph isomorphism.

Bibliography

[1] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Formal derivation of spanning trees algorithms. In D. Bert et al., editor, *ZB 2003: Formal Specification and Development in Z and B: Third International Conference of B and Z Users*, volume LNCS 2651, pages 457–476. Springer-Verlag, 2003.

[2] Sara Baase and Allen Van Gelder. *Computer Algorithms.* Addison-Wesley, third edition, 2000.

[3] Mehdi Behzad and Gary Chartrand. *Introduction to the Theory of Graphs.* Allyn and Bacon Inc. Boston, 1971.

[4] Jing-Chao Chen. Dijkstra's shortest path algorithm. *Formalized Mathematics*, 11(3):237–248, 2003.
See also `http://mizar.org/JFM/Vol15/graphsp.html`.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1999.

[6] Shimon Even. *Graph Algorithms.* Computer Science Press, 1979.

[7] Lestor R. Ford and D.R Fulkerson. Flows in networks. *Princeton University Press*, 1962.

[8] Ranan Fraer. Formal development in B of a minimum spanning tree algorithm. In E. Sekerinski and K. Sere, editors, *Program Development by Refinement Case Studies Using the B Method*, FACIT. Springer-Verlag, 1999.

[9] Krzysztof Hryniewiecki. Graphs. *Formalized Mathematics*, 2(**3**):365–370, 1990.
See also `http://mizar.org/JFM/Vol2/graph_1.html`.

[10] `http://en.wikipedia.org/wiki/Prim's_algorithm`.

[11] Udi Manber. *Intoduction to algorithms.* Addison-Wesley, 1989.

[12] Yatsuka Nakamura and Jing-Chao Chen. The underlying principle of Dijkstra's shortest path algorithm. *Formalized Mathematics*, 11(2):143–152, 2003. See also `http://mizar.org/JFM/Vol15/graph_5.html`.

[13] Yatsuka Nakamura and Piotr Rudnicki. Vertex sequences induced by chains. *Formalized Mathematics*, 5(**3**):297–304, 1996.
See also `http://mizar.org/JFM/Vol7/graph_2.html`.

[14] Yatsuka Nakamura and Piotr Rudnicki. Euler circuits and paths. *Formalized Mathematics*, 6(3):417–425, 1997.
See also `http://mizar.org/JFM/Vol9/graph_3.html`.

[15] Yatsuka Nakamura and Piotr Rudnicki. Oriented chains. *Formalized Mathematics*, 7(2):189–192, 1998.
See also `http://mizar.org/JFM/Vol10/graph_4.html`.

[16] P. Rudnicki, C. Schwarzweller, and A. Trybulec. Commutative Algebra in the Mizar System. *Journal of Symbolic Computation*, 32:143–169, 2001.

[17] Piotr Rudnicki. Little bezout theorem (factor theorem). *Formalized Mathematics*, 12(1):49–58, 2004.
See also `http://mizar.org/JFM/Vol15/uproots.html`.

[18] Piotr Rudnicki and Andrzej Trybulec. Multivariate polynomials with arbitrary number of variables. *Formalized Mathematics*, 9(**1**):95–110, 2001. See also `http://mizar.org/JFM/Vol11/polynom1.html`.

[19] Steven S. Skiena. *The Algorithm Design Manual.* Springer-Verlag, 1998.

[20] Zinaida Trybulec. Properties of subsets. *Formalized Mathematics*, 1(**1**):67–71, 1990. See also `http://mizar.org/JFM/Vol1/subset_1.html`.

[21] Gabriel Valiente. *Algorithms on Trees and Graphs.* Springer, 2002.

[22] Douglas B. West. *Introduction to Graph Theory.* Prentice-Hall, second edition, 2001.

[23] Mitsuharu Yamamoto, Koichi Takahashi, Masami Hagiya, Shin-ya Nishizaki, and Tetsuo Tamai. Formalization of graph search algorithms and its applictions. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs'98*, volume LNCS 1479, pages 479–496. Springer-Verlag, 1998.

Appendix A

Graph Structures in Mizar

Although MIZAR provides built-in support for dealing with structures, it has been decided that a different approach would be more suitable for dealing with graphs. We begin by providing some background information about system of structures currently in place.

A.1 Mizar Structures

There are situations when we wish to formalize facts regarding an object which is composed of a number of other entities. Examples of such cases include the familiar algebraic structures of groups and rings. In MIZAR, structures are internally handled as a finite collection of objects with specific types. The individual components of a structure are identified by the *selectors* of the structure. MIZAR provides special support for structures—allowing them to be defined without justifications for correctness or existence. In addition, MIZAR has built-in support for defining inheritance among structures.

The general syntax of structures as given in MIZAR grammar is

```
struct [ (Ancestors) ] Structure-Symbol [ over Loci ]
  (# Fields #);
```

MML contains quite a number of various structures, they are extensively used in developing abstract algebra. These structures form an inheritance hierarchy where e.g. double loop structure (the backbone for rings and fields) is derived from a multiplicative loop structure with unity and an additive loop structure with zero. See [16] for details of algebraic structures and the inheritance mechanism.

MML contains also some treatment of graphs where the basic structure is defined in GRAPH_1[9]:

```
definition
  struct MultiGraphStruct (#
    Vertices, Edges -> set,
    Source, Target -> Function of the Edges, the Vertices
  #);
end;
```

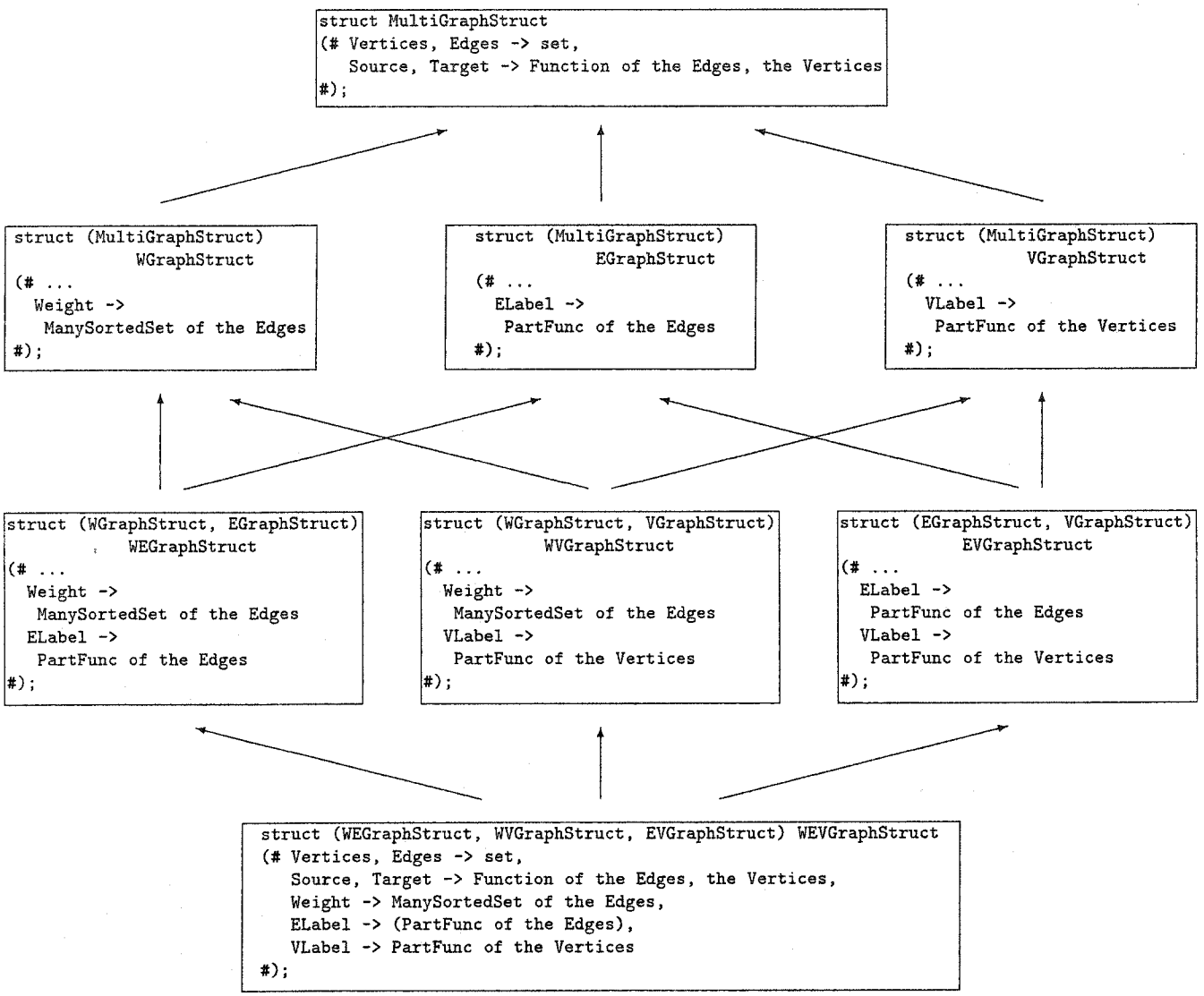

Figure A.1: Possible hierarchy of graph structures. (Note that the ... serve as space savers instead of repeating the fields from the `MultiGraphStruct`.)

The above defines `MultiGraphStruct` as an aggregate composed of two sets called `Vertices` and `Edges` and two functions `Source` and `Target`, which map edges to their endpoints. Then mode `Graph` is defined as a `MultiGraphStruct` with non empty set of `Vertices`. However, as for our needs, MML contains only the very basic facts about graphs, and we had to essentially start from scratch although originally we planned to reuse whatever was available about graphs. Unfortunately, this has led to some difficulties as we needed to perform such operations on graphs that are rarely performed on the familiar algebraic structures, e.g. updates of the values of components.

A.2 Graphs in the current model of structures

We have attempted to use the currently available MIZAR structures for our needs. The starting point was the above mentioned `MultiGraphStruct` from which we have derived 7 additional graph structures as shown in Figure A.1. In order to deal with graph algorithms on weighted graphs, there was a need to add three extra fields: weights on edges, labels on vertices, and labels on edges. They have been added one at a time as not all algorithms need all of them.

Once the graph structures were available we needed to define a number of functions and modes involving them. This led to quite a number of complications. We will illustrate them on two examples: one involves a function for labeling vertices and the other concerns a mode involving the concept of subgraph.

When dealing with graph algorithms, it is handy to have a set of helper functions which return graphs, such as functions which label a given vertex by a given value. Such a function, for any arguments, must return a unique object having a specified type. In order to guarantee uniqueness, any such function returning a structure must return a special, exact form of the structure. To achieve this, MIZAR employs the attribute `strict` which says that a unique set of fields is present in a structure. To illustrate the issue, let us consider this example: a `strict MultiGraphStruct` is a structure with only vertices, edges, source and target, and not any other structure derived from (prefixed by) `MultiGraphStruct`. However, when we have an object of type `MultiGraphStruct` (without knowing whether it is `strict`), then we really do not know whether there are any other fields in the object besides the original four: `Vertices`, `Edges`, `Source`, and `Target`. It is so, as any object derived (directly or not) from `MultiGraphStruct` can always be considered as a `MultiGraphStruct` despite it having additional fields.

Let us have a closer look at the following functor:

```
definition let G be VGraph, v be Vertex of G, x be set;
  func G.labelVertex(v,x) -> strict VGraph means
    (the GraphStruct of it) = (the GraphStruct of G) &
    (the VLabel of it) = (the VLabel of G) +* (v .--> x);
```

This function, whose name is `.labelVertex` and which is used in infix notation with one left argument and two right ones, would accept as the left parameter any structure derived from `VGraph`, e.g. `WVGraph`. However, the return type of this function is `strict VGraph` meaning that we would lose all information about the original type of the argument. Thus if the argument of this function is a `WVGraph` then the weight information is not preserved because we return a strict `VGraph`. Intuitively, labeling a vertex should not influence the weights (or any other fields besides `VLabel`) in a graph, and we would like to have this information explicit. Unfortunately, MIZAR does not allow this to be expressed easily: field selectors are not first class objects and we cannot quantify over them. Therefore we cannot say that all possible fields, other than `VLabel`, have not been changed.

There are several ways to work around this. We could introduce separate labeling functions, which return different types. For example, for `WVGraphs`:

```
definition let G be WVGraph, v be Vertex of G, x be set;
  func G.labelVertex(v,x) -> strict WVGraph means
    (the WGraphStruct of it) = (the WGraphStruct of G) &
```

```
(the VLabel of it) = (the VLabel of G) +* (v .--> x);
```

Unfortunately, this means that we may need to define many such functions, one for each combination of features we would like to see in a graph. The number of such functions grows linearly with the number of structures we derive, and the latter can grow exponentially with the number of features that we introduce. Add in the customary theorems for each function, and the amount of work required to maintain such a library of helper functions becomes immense. Due to the differing output types of the functions we would have to maintain quite a number of very similar theorems concerning these functions.

Another alternative to reclaiming the lost information is to build more helper functions which copy selectors from one structure to the next. For example:

```
definition let G be VGraph, G2 be WGraph;
           assume the Edges of G = the Edges of G2;
  func G.copyWeight_WV(G2) -> strict WVGraph means
    the VGraphStruct of it = the VGraphStruct of G &
    the Weight of it = the Weight of G2;
  correctness @proof end;
end;
```

We could then compose the two functions to get the `WVGraph` that we originally wanted: `G.labelVertex(v,x).copyWeight_WV(G)`. This is somewhat better than the previous approach, but we still need various versions of copy functions in order to deal with all the different types of graphs we would like to consider.

Both these approaches require quite a lot of extra function definitions which are essentially identical to one another. Ideally, we would like to have a function that takes in some type of a graph, modifies a vertex label, and returns the same type of graph. The function should only modify the vertex labels, and leave all the other fields of the output graph same as in the input graph. Unfortunately, there simply is no way to specify such a function using MIZAR's built-in structures.

One could consider overcoming this problem by defining all such functions just on the `WEVGraph` but in doing so we would make adding any field in the future quite unpleasant as we would just postpone the problem. Also, methodologically it seems desirable to define all functions and formulate all theorems about objects whose type is as wide as possible and not as narrow as it is convenient.

Let us now look at the concept of subgraph. The basic notion is defined in `GRAPH_1`[9]

```
definition let G be Graph;
mode Subgraph of G -> Graph means
 the Vertices of it c= the Vertices of G &
 the Edges of it c= the Edges of G &
 for v st v in the Edges of it holds
  (the Source of it).v = (the Source of G).v &
  (the Target of it).v = (the Target of G).v &
  (the Source of G).v in the Vertices of it &
  (the Target of G).v in the Vertices of it;
end;
```

When treating the Dijkstra algorithm, in order to talk about a shortest-path subtree, it was necessary to define the concept of a weighted subgraph, `WSubgraph`, whose definition could look like this:

```
definition let G be WGraph;
  mode WSubgraph of G -> WGraph means
    it is Subgraph of G &
    the Weight of it = the Weight of G | the Edges of it
```

A `WSubgraph` is clearly a subgraph, which is a narrower type, but here we can only assign it the type `WGraph` in order to access the `Weight` selector. Using this definition MIZAR automatically knows that a `WSubgraph` is a `WGraph`, but it does not automatically know that it is also a subgraph of `G`. This is unsatisfactory as the entire machinery built for subgraphs is not directly available. As an example, let us look at the following attribute:

```
definition let G be Graph, G2 be Subgraph of G;
  attr G2 is spanning means
    the Vertices of G2 = the Vertices of G;
end;
```

Now if we have an object `w` which is of type `WSubgraph of G`, we cannot directly say that

```
w is spanning
```

as `WSubgraph of G` is not automatically perceived as a `Subgraph of G`. In order to say it, we must use an additional object in order to cast the type:

```
for w1 being Subgraph of G st w = w1 holds w1 is spanning
```

This type casting is necessary as MIZAR does not see two types of an object simultaneously if one of the types is not derived from the other.

Although none of the above obstacles was crucial, they seemed sufficiently inconvenient such that we have decided to pursue an alternative treatment of graphs in MIZAR; especially that there was not much that we could have reused that was already in MML.

A.3 Implementing aggregates via attributes

The underlying idea behind our alternate approach to aggregate objects is to have selectors as first class objects and thus resigning completely from using the built-in structures. The MIZAR machinery of attributes plays the central role in our approach. Instead of fixing which collection of fields are part of an aggregate, we define what it means for an object to have some particular field. For example, a graph having some associated weight function would have the attribute `[Weighted]`, while a graph having labeled vertices would have the attribute `[VLabeled]`.

To start, we define a `GraphStruct` as a finite function whose domain is a subset of natural numbers:

```
definition
  mode GraphStruct -> finite Function means      :: GLIB_000:def 1
    dom it c= NAT;
end;
```

Although the domain could have been any fixed set, we use natural numbers for convenience. For each selector we are interested in, a functor is defined that returns a unique natural number:

```
definition
  func VertexSelector -> Nat equals 1;
  func EdgeSelector   -> Nat equals 2;
  func SourceSelector -> Nat equals 3;
  func TargetSelector -> Nat equals 4;
  func WeightSelector -> Nat equals 5;
  func ELabelSelector -> Nat equals 6;
  func VLabelSelector -> Nat equals 7;
end;
```

For each selector, another functor is introduced to simulate MIZAR's built-in version of a selector. (Note the very similar naming.):

```
definition let G be GraphStruct;
  func the_Vertices_of G equals G.VertexSelector;
  func the_Edges_of G    equals G.EdgeSelector;
  func the_Source_of G   equals G.SourceSelector;
  func the_Target_of G   equals G.TargetSelector;
end;
```

We can then define attributes which tell us the selector that is present, as well as type information:

```
definition let G be GraphStruct;
  attr G is [Graph-like] means
    VertexSelector in dom G & EdgeSelector in dom G &
    SourceSelector in dom G & TargetSelector in dom G &
    the_Vertices_of G is non empty set &
    the_Source_of G is Function of the_Edges_of G,
                                   the_Vertices_of G &
    the_Target_of G is Function of the_Edges_of G,
                                   the_Vertices_of G;

  attr G is [Weighted] means
    WeightSelector in dom G &
    G.WeightSelector is ManySortedSet of the_Edges_of G;

  attr G is [ELabeled] means
    ELabelSelector in dom G &
    ex f being Function
     st G.ELabelSelector = f & dom f c= the_Edges_of G;

  attr G is [VLabeled] means
    VLabelSelector in dom G &
    ex f being Function
     st G.VLabelSelector = f & dom f c= the_Vertices_of G;
end;
```

Unlike the built-in MIZAR structures, we have to show that objects of the new compound types exist. With our three features besides the backbone fields of each graph, we have 8 different kinds of graphs we could possibly deal with. The good

news is that we can do this in one single shot by proving the following existential cluster:

```
registration
  cluster [Graph-like] [Weighted] [ELabeled] [VLabeled]
          GraphStruct;
  existence proof ... end;
end;
```

Now that we showed that such compound objects exist, we can assign them individual modes for each subset of features:

```
definition
  mode _Graph is [Graph-like] GraphStruct;
end;

definition
  mode   WGraph is [Weighted]                          _Graph;
  mode   EGraph is            [ELabeled]               _Graph;
  mode   VGraph is                         [VLabeled] _Graph;
  mode  WEGraph is [Weighted] [ELabeled]               _Graph;
  mode  WVGraph is [Weighted]              [VLabeled] _Graph;
  mode  EVGraph is            [ELabeled] [VLabeled] _Graph;
  mode WEVGraph is [Weighted] [ELabeled] [VLabeled] _Graph;
end;
```

Thanks to the MIZAR attribute system, we get automatic inheritance. For example, MIZAR automatically knows that a `WEVGraph` is a `WGraph`, a `VGraph` and even a `EVGraph`, without us having to prove anything.

A feature found in the MIZAR implementation of structures that we have to emulate by hand is the previously automatic typing of selectors. For example, in `MultiGraphStruct`, MIZAR understands that `the Source` is a function from `the Edges` to `the Vertices`. We can do the same by redefinitions for various types of arguments.

```
definition let G be _Graph;
  redefine func the_Vertices_of G -> non empty set;

  redefine func the_Source_of G ->
    Function of the_Edges_of G, the_Vertices_of G;

  redefine func the_Target_of G ->
    Function of the_Edges_of G, the_Vertices_of G;
end;
```

A.4 How attributes solve our problems

Using our implementation of aggregates via attributes, we can address the two limitations we mentioned in Section A.2, simply because we have more control over the fields.

Given a selector, we can now replace the field associated with that particular selector, while leaving the other fields intact. An example of a function that accomplishes this task looks like this:

```
definition let G be GraphStruct, n be Nat, x be set;
  func G.set(n,x) -> GraphStruct equals
    G +* (n .--> x);
end;
```

where `G +* (n .--> x)` is MIZAR lingo for overwriting `G` at `n` by `x`. With this in hand, we return to our primitive graph helper function of labeling a vertex. Using attributes, the function now looks like:

```
definition let G be VGraph, v,x be set;
  func G.labelVertex(v,x) -> VGraph equals
    G.set(VLabelSelector, the_VLabel_of G +* (v.-->x)) if
    v in the_Vertices_of G otherwise G;
```

If we try to label a `WVGraph`, the weight information gets preserved. Nonetheless, MIZAR doesn't recognize this automatically and sees the result as only a `VGraph`. However, now that the associated weight is an attribute, we can solve this using functorial clusters:

```
registration let G be WVGraph, v,x be set;
  cluster G.labelVertex(v,x) -> [Weighted];
end;

registration let G be EVGraph, v,x be set;
  cluster G.labelVertex(v,x) -> [ELabeled];
end;
```

Now we have a more convenient labeling function. Using MIZAR's built-in treatment of structures, we would have needed many different functions, but with our implementation, we only need one, along with a couple of functorial clusters. However, we still have to prove several simple to state and prove theorems that say which selectors have not been affected by the labeling function, like the following

```
theorem                                                    :: GLIB_003:46
  for G being WVGraph, v,x being set holds
    the_Weight_of  G = the_Weight_of  G.labelVertex(v,x);

theorem                                                    :: GLIB_003:47
  for G being EVGraph, v,x being set holds
    the_ELabel_of  G = the_ELabel_of  G.labelVertex(v,x);
```

Since now we have only one update function, the number of such theorems will be quite small.

The second troubling issue we mentioned in Section A.2 deals with `WSubgraphs`. A subgraph is now defined as

```
  definition let G be _Graph;
  mode Subgraph of G -> _Graph means              :: GLIB_000:def 29
    the_Vertices_of it c= the_Vertices_of G &
    the_Edges_of it c= the_Edges_of G &
    for e being set st e in the_Edges_of it holds
      (the_Source_of it).e = (the_Source_of G).e &
      (the_Target_of it).e = (the_Target_of G).e;
end;
```

Now, that the property of having a weight is an attribute, we can talk about subgraphs that are weighted, i.e. `[Weighted] Subgraph of G`. This is not quite the same as a `WSubgraph`, because the weights are not necessarily inherited from G. What we need is another attribute which states this fact:

```
definition let G be WGraph, G2 be [Weighted] Subgraph of G;
  attr G2 is weight-inheriting means
   the_Weight_of  G2 = (the_Weight_of G) | the_Edges_of G2;
end;
```

We now define the mode `WSubgraph`:

```
definition let G be WGraph;
  mode WSubgraph of G is weight-inheriting
                         ([Weighted] Subgraph of G);
end;
```

Once again, thanks to the MIZAR attribute system, MIZAR automatically understands that a `WSubgraph of G` is a `Subgraph of G`, yet also a `WGraph` because it is both `[Weighted]` and a `_Graph`.

We should mention another feature that is built-in to MIZAR structures that we have not emulated so far is strictness. MIZAR syntax allows us to extract a strict part out of a structure. For example, for an object `G` of type `WVGraphStruct`, we can talk about `the WGraphStruct of G` which gives us a `strict WVGraphStruct`. With our attribute approach, we can do the same by specifying the selector set of a graph, using a function such as this:

```
definition let G be GraphStruct, X be set;
  func G.|X -> GraphStruct equals
    G | X;
```

As in the case of labeling a vertex, we use functional clusters to show which features are carried over:

```
definition
  func WGraphSelectors -> non empty finite Subset of NAT equals
    {VertexSelector, EdgeSelector, SourceSelector, TargetSelector,
     WeightSelector};
end;

registration let G be WGraph;
  cluster G.| WGraphSelectors -> [Graph-like] [Weighted];
end;
```

In our dealing with graphs, getting past these two limitations were of paramount importance. We feel that our implementation of aggregates via attributes has preserved all the benefits of the current MIZAR implementation of structures, yet has given us the extra flexibility to address many issues that we have faced.

Appendix B

The Mizar Graph Library

This appendix is designed to give a brief overview of the definitions we created for the MIZAR graph library. Unlike syntax currently found in MIZAR, we simulate the dot-operator method accessor syntax popular in object oriented languages such as Java.

B.1 GraphStructs

The mode `GraphStruct` is the basic structure we use to represent graphs.

B.1.1 GraphStruct Functions

- `the_Vertices_of G -> set`
  - Accesses the Vertices component of the GraphStruct G
- `the_Edges_of G -> set`
  - Accesses the Edge component of the GraphStruct G
- `the_Source_of G -> set`
  - Accesses the Source component of the GraphStruct G
- `the_Target_of G -> set`
  - Accesses the Target component of the GraphStruct G
- `G.set(n,x) -> GraphStruct`
  - Sets the component indexed by `n` to be `x`
- `G.|X -> GraphStruct`
  - Returns the GraphStruct formed by reducing the feature set to X

B.1.2 GraphStruct Attributes

- `[Graph-like]`
  - A GraphStruct G is `[Graph-like]` if it contains the features Vertices, Edges, Source and Target. In addition, `the_Vertices_of G` must be a non-empty set, and both `the_Source_of G` and `the_Target_of G` are functions mapping `the_Edges_of G` to `the_Vertices_of G`.
- `[Weighted]`
  - A GraphStruct G is `[Weighted]` if it contains the Weight feature. In addition, the Weight must be a `ManySortedSet of the_Edges_of G`.
- `[ELabeled]`
  - A GraphStruct G is `[ELabeled]` if it contains the Edge-Label feature. In addition, the Edge-Label must be a function whose domain is subset of `the_Edges_of G`.
- `[Weighted]`
  - A GraphStruct G is `[VLabeled]` if it contains the Vertex-Label feature. In addition, the Vertex-Label must be a function whose domain is a subset of `the_Vertices_of G`.

B.2 Graphs

A `_Graph` is a `[Graph-like] GraphStruct`, and is the basic unit that we deal with.

B.2.1 Graph Attributes

- `G is finite`
  - Means that both the set of vertices and edges of G are finite sets
- `G is loopless`
  - Means that there are no edges in G that have the same source and target vertex
- `G is trivial`
  - Means that G has only one vertex
- `G is non-multi`
  - Means that there is at most one undirected edge between any two vertices
- `G is non-Dmulti`
  - Means that there is at most one directed edge between any two vertices
- `G is simple`

- Means that G is loopless and non-multi

- `G is Dsimple`
  - Means that G is loopless and non-Dmulti

- `G is connected`
  - Means that there exists a walk between any two vertices of G

- `G is acyclic`
  - Means that there does not exists a Cycle-like walk in G

- `G is Tree-like`
  - Means that G is acyclic and connected

B.2.2 Graph Accessors

- `the_Vertices_of G -> non empty set`
  - The set of vertices in the graph G

- `the_Edges_of G -> set`
  - The set of edges in the graph G

- `the_Source_of G -> Function of the_Edges_of G, the_Vertices_of G`
  - The function mapping an edge to its source vertex.

- `the_Target_of G -> Function of the_Edges_of G, the_Vertices_of G`
  - The function mapping an edge to its target vertex.

B.2.3 Graph Creators

- `createGraph(V,E,S,T) -> _Graph`
  - Creates the graph using `V` as the vertices, `E` as the edges, `S` as the source function, and `T` as the target function.

B.2.4 Graph Functions

- `G.edgesBetween(X) -> Subset of the_Edges_of G`
  - Returns the set of all edges whose source and target vertex are both in the set X

- `G.edgesInOut(X) -> Subset of the_Edges_of G`
  - Returns the set of all edges whose source or target vertex is in the set X

- `G.edgesInto(X) -> Subset of the_Edges_of G`

- Returns the set of all edges whose target vertex is in the set X

- `G.edgesOutOf(X) -> Subset of the_Edges_of G`
  - Returns the set of all edges whose source vertex is in the set X
- `G.order() -> Cardinal`
  - Returns the number of vertices in G
- `G.size() -> Cardinal`
  - Returns the number of edges in G
- `G.allWalks() -> Subset of ((the_Vertices_of G)\/(the_Edges_of G))*`
  - Returns the set of all walks in G
- `G.allTrails() -> Subset of G.allWalks()`
  - Returns the set of all trails in G
- `G.allPaths() -> Subset of G.allTrails()`
  - Returns the set of all paths in G
- `G.allDWalks() -> Subset of G.allWalks()`
  - Returns the set of all directed walks in G
- `G.allDTrails() -> Subset of G.allTrails()`
  - Returns the set of all directed trails in G
- `G.allDPaths() -> Subset of G.allDTrails()`
  - Returns the set of all directed paths in G
- `G.reachableFrom(v) -> non empty Subset of the_Vertices_of G`
  - Returns the set of all vertices that are reachable via a walk from v
- `G.reachableDFrom(v) -> non empty Subset of the_Vertices_of G`
  - Returns the set of all vertices that are reachable via a directed walk from v
- `G.reachableDFrom(v) -> non empty Subset of the_Vertices_of G`
  - Returns the set of all vertices that are reachable via a directed walk from v
- `G.componentSet() -> non empty Subset of bool the_Vertices_of G`
  - Returns the set of all subsets which form a component in G
- `G.numcomponent() -> Cardinal`
  - Returns the number of components in G

B.2.5 Graph Predicates

- `e Joins x,y,G`
  - True if e is an edge that joins the vertices x and y in the graph G.
- `e DJoins x,y,G`
  - True if e is an edge that has x as its source vertex and y as its target vertex in the graph G
- `e SJoins X,Y,G`
  - True if e is an edge whose joins an element of the set X to an element of the set Y in the graph G
- `e DSJoins X,Y,G`
  - True if e is an edge whose source vertex is a element of the set X and whose target vertex is an element of the set Y with respect to the graph G
- `G1 == G2`
  - True if the $G1$ and $G2$ share the same Vertices, Edges, Source and Target
- `G1 c= G2`
  - True if the $G1$ is a subgraph of $G2$
- `G1 c< G2`
  - True if the $G1$ is a strict subgraph of $G2$
- `G is_DTree_rooted_at v`
  - True if the G is a directed tree rooted at the vertex v

B.3 Subgraph of G

B.3.1 Subgraph attributes

- `SG is spanning`
  - Means that SG has the same vertices as G
- `SG is Component-like`
  - Means that SG is maximal connected subgraph

B.4 Induced Subgraphs

- `inducedSubgraph of G,V,E -> Subgraph of G`
  - A subgraph of G whose vertices are V and edges are E
- `inducedSubgraph of G,V -> Subgraph of G`
  - A subgraph of G whose vertices are V and whose edges are all those that are between members of V
- `inducedSubgraph of G,V -> Subgraph of G`
  - A subgraph of G whose vertices are V and whose edges are all those that are between members of V
- `removeVertex of G,v -> inducedSubgraph of G, the_Vertices_of G \ {v}`
  - A graph formed by removing the vertex v from G
- `removeVertices of G,V -> inducedSubgraph of G, the_Vertices_of G \ V`
  - A graph formed by removing the vertices in V from G
- `removeEdge of G,e -> inducedSubgraph of G, the_Vertices_of G, the_Edges_of G \ {e}`
  - A graph formed by removing the edge e from G
- `removeEdge of G,E -> inducedSubgraph of G, the_Vertices_of G, the_Edges_of G \ E`
  - A graph formed by removing the edges in E from G

B.5 Vertex of G

B.5.1 Vertex Functions

- `v.adj(e) -> Vertex of G`
  - The vertex adjacent to e other than v
- `v.edgesIn() -> Subset of the Edges of G`
  - The set of edges whose target vertex is v
- `v.edgesOut() -> Subset of the Edges of G`
  - The set of edges whose source vertex is v
- `v.edgesInOut() -> Subset of the Edges of G`
  - The set of edges whose source or target vertex is v

- `v.inDegree() -> Cardinal`
  - The number of edges going into v
- `v.outDegree() -> Cardinal`
  - The number of edges going out of v
- `v.degree() -> Cardinal`
  - The sum of the number of edges going into v with the number of edges coming out of v.
- `v.inNeighbors() -> Subset of the_Vertices_of G`
  - The set of all vertices that have edges coming out of them and going into v
- `v.outNeighbors() -> Subset of the_Vertices_of G`
  - The set of all vertices that have edges going into them that come from v
- `v.allNeighbors() -> Subset of the_Vertices_of G`
  - The set of all vertices that are reachable from v via a single edge.

B.5.2 Vertex attributes

- `v is isolated`
  - Means that no edges are incident with v
- `v is endvertex`
  - Means that exactly one non-loop edge is incident to v
- `v is cut-vertex`
  - Means that removing v from G increases the number of components

B.6 ManySortedSet of NAT

A `ManySortedSet of NAT` is a function whose domain is the set of natural numbers. We can think of this as an infinite sequence, indexed by the natural numbers.

B.6.1 ManySortedSet of NAT attributes

- `F is Graph-yielding`
  - Means that every element of F is a `_Graph`
- `F is halting`
  - Means that there exists some n such that the the n^{th} element of F is identical to the $n + 1^{th}$ element of F.

B.6.2 ManySortedSet of NAT functions

- `F.Lifespan() -> Nat`
  - The minimum value n such that n^{th} element of F is identical to the $n+1^{th}$ element of F, assuming F is halting. Otherwise 0.
- `F.Result() -> Nat`
  - The `F.Lifespan()`th element of F.

B.7 GraphSeq

A `GraphSeq` is a `Graph-yielding ManySortedSet of NAT`. We use this to model the computation sequence defined by an algorithm.

B.7.1 GraphSeq attributes

Every attribute defined for graphs is repeated as an attribute for GraphSeqs, with the meaning that every graph in the sequence has that particular attribute.

- `GSq is finite`
  - Means every element of GSq is finite
- `GSq is loopless`
  - Means every element of GSq is loopless
- `GSq is trivial`
  - Means every element of GSq is trivial
- `GSq is non-trivial`
  - Means every element of GSq is non-trivial
- `GSq is non-multi`
  - Means every element of GSq is non-multi
- `GSq is simple`
  - Means every element of GSq is simple
- `GSq is connected`
  - Means every element of the GSq is connected
- `GSq is [Weighted]`
  - Means every element of GSq is weighted
- `GSq is [ELabeled]`
  - Means every element of GSq is edge-labeled

- `GSq is [VLabeled]`
  - Means every element of GSq is vertex-labeled
- `GSq is real-weighted`
  - Means every element of the `WGraphSeq` GSq is real-weighted
- `GSq is nonnegative-weighted`
  - Means every element of the `WGraphSeq` GSq is nonnegative-weighted
- `GSq is natural-weighted`
  - Means every element of the `WGraphSeq` GSq is natural-weighted
- `GSq is complete-elabeled`
  - Means every element of the `EGraphSeq` GSq is complete-elabeled
- `GSq is real-elabeled`
  - Means every element of the `EGraphSeq` GSq is real-elabeled
- `GSq is real-vlabeled`
  - Means every element of the `VGraphSeq` GSq is real-vlabeled
- `GSq is real-WEV`
  - Means every element of the `WEVGraphSeq` GSq is real-WEV

B.7.2 GraphSeq functions

- `GSq.->x -> _Graph`
  - This function returns the x^{th} element of the GraphSeq GSq. It is identical to the standard dot operator, but we needed a new one that returned a `_Graph` so that we can use it to cluster graph attributes. This would not be possible with the standard dot operator because MIZAR would see the returned object as a set instead of a `_Graph`.
  - For example, given a `finite GraphSeq GSq`, we can register the cluster `GSq.->x -> finite`, thus letting MIZAR automatically know that `GSq.->x` is a finite graph.

B.8 Walk of G

B.8.1 Walk functions

- `G.walkOf(v) -> Walk of G`
  - Returns the trivial walk consisting of the single vertex v
- `G.walkOf(x,e,y) -> Walk of G`

- Returns the walk consisting of x followed by e followed by y

- `W.first() -> Vertex of G`
  - Returns the first vertex of W
- `W.last() -> Vertex of G`
  - Returns the last vertex of W
- `W.vertexAt(n) -> Vertex of G`
  - Returns the vertex at the n^{th} position of W
- `W.reverse() -> Walk of G`
  - Returns W in reverse
- `W1.append(W2) -> Walk of G`
  - Returns the walk formed by appending the walk $W2$ to $W1$
- `W.cut(m,n) -> Walk of G`
  - Returns the walk formed by only considering from the m^{th} position to the n^{th} position of W.
- `W.remove(m,n) -> Walk of G`
  - Returns the walk formed by removing the section between the m^{th} position to the n^{th} position of W.
- `W.addEdge(e) -> Walk of G`
  - Returns the walk formed by adding the edge e to the end of W
- `W.vertexSeq() -> VertexSeq of G`
  - Returns the sequence of vertices in W
- `W.edgeSeq() -> EdgeSeq of G`
  - Returns the sequence of edges in W
- `W.vertices() -> finite Subset of the_Vertices_of G`
  - Returns the set of vertices that are in W
- `W.edges() -> finite Subset of the_Edges_of G`
  - Returns the set of edges that are in W
- `W.length() -> Nat`
  - Returns the number of edges that are in W
- `W.find(v) -> odd Nat`

- Returns the index of the first time the vertex v occurs in W

- `W.find(n) -> odd Nat`
  - Returns the index of the first time the vertex at n^{th} position occurs in W
- `W.rfind(v) -> odd Nat`
  - Returns the index of the last time the vertex v occurs in W
- `W.rfind(n) -> odd Nat`
  - Returns the index of the last time the vertex at n^{th} position occurs in W

B.8.2 Walk predicates

- `W is_Walk_from u,v`
  - True if the first element of W is u and the last element of W is v

B.8.3 Walk attributes

- `W is closed`
  - Means that the first vertex and last vertex of W are the same
- `W is open`
  - Means that W is not closed
- `W is trivial`
  - Means that the W contains no edges
- `W is Trail-like`
  - Means that no edges are repeated in W
- `W is Path-like`
  - Means that no edges are repeated in W and the only vertices that may be repeated in W are the first and last vertex
- `W is vertex-distinct`
  - Means that no vertices are repeated in W
- `W is Circuit-like`
  - Means that W is closed, Trail-like, and non trivial
- `W is Path-like`
  - Means that W is closed, Path-like, and non trivial

B.8.4 Modes of Walks

- `Subwalk of W`
  - A Subwalk of W is a walk that shares the same first and last vertex as W, whose edges are a subsequence of the edges of W.

B.9 Graphs with Features

A `WGraph` is a `[Weighted] _Graph`, while a `EGraph` is a `[ELabeled] _Graph` and a `VGraph` is a `[Weighted] _Graph`.

B.9.1 Accessors for Graphs with Features

- `the_Weight_of G -> ManySortedSet of the_Edges_of G`
  - The weight function of the WGraph G
- `the_ELabel_of G -> Function`
  - The edge-labeling function of the EGraph G
- `the_VLabel_of G -> Function`
  - The vertex-labeling function of the VGraph G

B.9.2 Functions for Graphs with Features

- `G.labeledE() -> Subset of the_Edges_of G`
  - The set of edges that are labeled in the `EGraph` G
- `G.labelEdge(e,x) -> EGraph`
  - The graph formed by labeling the edge e with the value x
- `G.labeledV() -> Subset of the_Vertices_of G`
  - The set of vertices that are labeled in the `VGraph` G
- `G.labelVertex(v,x) -> VGraph`
  - The graph formed by labeling the vertex v with the value x
- `G.min_DPath_cost(x,y) -> Real`
  - The minimum cost of a path that goes from x to y in the finite real-weighted `WGraph` G
  - The sum of all the weights on all the edges in the finite real-weighted `WGraph` G
- `G.allWSubgraphs() -> non empty set`
  - The set of all strict weighted subgraphs of the `WGraph` G

B.9.3 Attributes for Featured Graphs

- `G is real-weighted`
  - Means that the weights in the `WGraph` G have real values
- `G is nonegative-weighted`
  - Means that the weights in the `WGraph` G have non-negative values
- `G is natural-weighted`
  - Means that the weights in the `WGraph` G have natural values
- `G is real-elabeled`
  - Means that the edge-labels in the `EGraph` G have real values
- `G is complete-elabeled`
  - Means that the every edge in the `EGraph` G has a label
- `G is real-vlabeled`
  - Means that the vertex-labels in the `VGraph` G have real values
- `G is real-WEV`
  - Means that the weights, edge-labels and vertex-labels in the `WEVGraph` G are all real values

B.9.4 Attributes for Featured Subgraphs

- `G2 is weight-inheriting`
  - Means that the weights in the `[Weighted] Subgraph` $G2$ are copied from G
- `G2 is elabel-inheriting`
  - Means that the edge-labels in the `[ELabeled] Subgraph` $G2$ are copied from G
- `G2 is vlabel-inheriting`
  - Means that the vertex-labels in the `[VLabeled] Subgraph` $G2$ are copied from G

B.9.5 Functions for Walks in Featured Graphs

- `W.weightSeq() -> FinSequence`
  - Returns the sequence of weights on the edges of W in a weighted graph.
- `W.cost() -> Real`
  - Returns the sum of the weights on all the edges of W in a real-weighted `WGraph`

B.9.6 Predicates for Featured Graphs

- `W is_mincost_DPath_from x,y`
  - True if W is a path from x to y with minimum cost in the `real-weighted WGraph` G.
- `W.cost() -> Real`
  - Returns the sum of the weights on all the edges of W in a real-weighted `WGraph`