University of Alberta

Solving Multi-Agent Pathfinding Problems in Polynomial Time Using Tree Decompositions

by

Mokhtar Mohamed Khorshid

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

©Mokhtar Mohamed Khorshid
Fall 2011
Edmonton, Alberta

Abstract

Multi-agent pathfinding problems involve finding plans for agents that must travel from their start locations to their targets without colliding. Recent work produced a number of algorithms to solve the problem as well as an ample supply of related theory. This work is based on a related work that studied the feasibility of multi-agent pathfinding problems on trees. Our work takes this further to actually provide a constructive proof of how to solve multi-agent pathfinding problems in a tree and culminates in a novel approach that called tree-based agent swapping strategy (TASS). I also provide a family of algorithms that decompose graphs to trees. Experimental results showed that TASS can find solutions to multi-agent pathfinding problems on a highly crowded tree with 1000 nodes and 996 agents in less than 3 seconds. Further experiments compared TASS with other modern contending algorithms and the results were very favorable.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this section I will give a brief introduction to the problem I am attempting to solve, my approach to solving it, and an overview of how well my approach performed.

1.1 Problem Definition

In this thesis I study the multi-agent pathfinding problem. In this problem agents occupy locations in an environment, each must move to its target location, and no two agents can occupy the same location simultaneously. When there is only one agent, the problem becomes that of single-agent pathfinding. This involves finding a series of moves that the agent needs to make so that it arrives at its target while avoiding obstacles; usually in the least number of moves or shortest total travel time.[1] The multi-agent pathfinding problem is a generalization of the single-agent problem; one where we want to simultaneously find complete paths for each agent that avoids not only locations that have obstacles, but also the other agents in the environment. The agents cooperate together to find the best overall solution. In this thesis I assume that at any given time step only a single agent can be moving. In Chapter 3 I will define the problem in full detail. Figure 1.1 shows an example of a simple multi-agent pathfinding problem on a tree. In this small example the agents, identified by a number in the tree, need to reach the nodes indicated by the arrows. Some problems have no possible solutions, such as the one shown in Figure 1.2, and it is typically desirable to be able to detect whether a given problem has a solution or not before attempting to find one.

1.2 Motivation

The multi-agent pathfinding problem is not a purely theoretical exercise; it has many significant real-life applications. It frequently manifests itself in fields like robotics, traffic routing, transportation, and video games. In most of these domains the difficulty that faces current optimal state-of-the art algorithms is that they do not scale up to large problems, while non-optimal algorithms typically

[1]In general any criteria can be selected to favor particular paths. For example it may be desirable to find traffic routes that minimize the number of intersections.

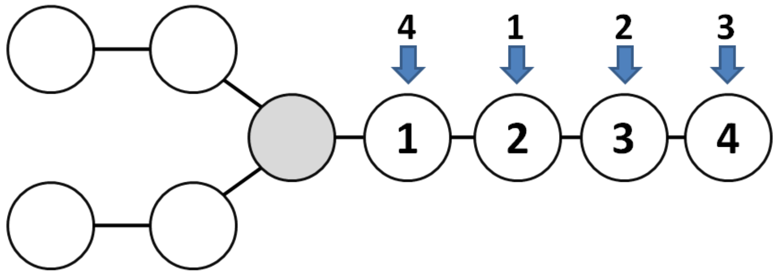

Figure 1.1: A solvable multi-agent pathfinding problem on a tree. The numbers in the nodes indicate the different agents and the arrow labels indicate the targets for each agent.

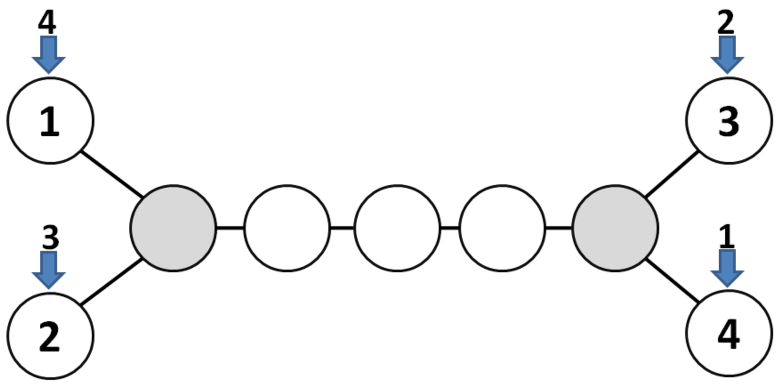

Figure 1.2: An unsolvable multi-agent pathfinding problem on a tree. The numbers in the nodes indicate the different agents and the arrow labels indicate the targets for each agent.

offer no guarantee of completeness.

The main objective of this work was to have the ability to quickly find solutions to as many hard multi-agent pathfinding problems, with a completeness guarantee, as possible. Completeness means that whenever a problem within our domain has a solution we are certain to find one in polynomial time. By solving the fundamental problem in the virtual, but representative, grid maps domain, my work can be applied to a wide range of real-life problems with little effort.

1.3 Overview of Existing Work

The single-agent pathfinding problem has been extensively studied and efficient optimal, and sub-optimal, algorithms exist for it. These same algorithms, however, were not designed to scale up to the multi-agent problem domain. In multi-agent problems there is an added difficulty that arises from the fact that the other agents in the environment are obstacles whose positions change over time and it is not known beforehand which exact nodes will be blocked at any given time step. The problem space grows exponentially because we consider the joint space resulting from the combination of all the possible decisions for each agent at each time step. Finding optimal solutions in the general multi-agent pathfinding problem is NP-Complete[2] so work in this direction has been limited to the smallest of problems.

Multi-agent pathfinding research has often focused on non-optimal solutions. A large number of algorithms were built to solve multi-agent pathfinding problems by sacrificing solution quality for speed. Most of them, however, have failed to be both complete and efficient. There has been some work to define narrower scopes within which certain algorithms can be guaranteed to be complete. The majority of fast algorithms, like [22] and [32], try to solve each agent individually with little collaboration between agents. They will typically fail in certain situations like congested bottlenecks, by reaching a deadlock where no progress can be made towards a solution. This occurs more often in the more crowded scenarios where free space is limited and simple local avoidance techniques cannot lead to solutions. This research direction is most useful for scenarios that are not highly crowded and where the number of agents have been restricted as well as in problems that provide a lot of open space where the number of possible solutions is sufficiently large.

My work builds on the work of Masehian and Nejad [14] which attempts to identify situations in which multi-agent tree problems can be solved. The details of their approach and how I extend it is covered in Chapter 3.

1.4 Contributions

My main contribution in this work is an approach that can solve all solvable multi-agent pathfinding problems on trees based on properties deduced from the work of [14], which can be checked in linear

[2]It reduces to the sliding-tile puzzle [16]. Furthermore [7] showed that coordinated movement in two dimensional space is P-SPACE hard.

time. Central to my approach is a novel tree-based agent swapping strategy (TASS) that can be used to efficiently solve any multi-agent pathfinding problem on solvable trees.[3] To solve problems on general graphs, we take the original graph along with the agents' configuration and decompose it into a tree. I have efficient tests for tree solvability based on [14] that we can apply to any tree we generate, but I have also devised algorithms that can take graphs and generate trees that can be checked for solvability in linear time. Once we get a solvable tree, we use TASS to solve the problem on the tree and then present it as a valid solution on the original graph. It has to be noted that not all problems that are solvable on general graphs will remain solvable on the induced trees.

So the contributions of my work can be enumerated as follows:

1. A simple check for solvable trees directly derived from [14].

2. A complete constructive proof that the solvability criteria are sufficient.

3. A polynomial time algorithm,[4] TASS, for solving problem instances on solvable trees.

4. A family of Graph-to-Tree Decomposition (GTD) algorithms that take a general graph as input and output a solvable tree, if possible.

1.5 Summary of Results

I was able to establish that TASS has a polynomial runtime bounded by $O(m^2n(n-m))$ where m is the number of agents and n is the number of nodes in the tree. I were also able to prove that a given GTD algorithm, GTD-SLIDEABLE, can convert any problem that belongs to the class of problems that Wang and Botea [33] call SLIDEABLE into a solvable tree, confirming that SLIDEABLE problems are a strict subset of the problems we can solve using TASS and a GTD algorithm.

Empirically, I conducted a number of experiments to test the performance of TASS and the GTD algorithms in practice. TASS was able to solve multi-agent pathfinding problems on almost fully congested ternary trees with 1,000 nodes and 996 agents in less than 3 seconds and problems of 10,000 nodes with 9,996 agents in less than 3 minutes. This by far exceeds the state-of-the-art algorithms as far as I know and shows that my approach scales up very well with an increased number of agents. Table 1.1 summarizes the result of running TASS on binary and ternary trees of various sizes. In Chapter 5 I will discuss the details of the various experiments I have conducted.

1.6 Thesis Outline

In this chapter I introduced the problem I have attempted to solve and provided a quick summary of how well my solution works. In the following chapters I will provide all the details necessary

[3]A solvable tree is a tree that meets certain fairly relaxed conditions that guarantee that any valid configuration of agents on this tree will have a solution. For more details see Chapter 3.

[4]Whenever we refer to time complexity in this work it is always assumed, unless otherwise mentioned, to be in the number of nodes in the graph which is also an upper bound for the number of agents.

Number of Nodes	Binary Trees		Ternary Trees	
	Number of Moves	Time (ms)	Number of Moves	Time (ms)
10	170	1	71	1
100	16,617	64	12,257	51
1,000	556,296	3,026	295,188	2,424
10,000	12,597,322	206,534	5,499,014	164,291

Table 1.1: Summary of running TASS of tree problems of various sizes. The number of agents in each case is equal to the total number of nodes - 4. The "Number of Moves" column refers to the total steps all agents need to take to reach their target nodes.

to understand and evaluate my approach, beginning with some essential background information in Chapter 2 that lays the foundation on which my work is based and defines the domain to which my work belongs. I will then provide in detail my constructive proof as well as the GTD algorithms in Chapter 3. I will then survey a number of important papers that are related to my work in Chapter 4. Finally, since my work is intended to be applicable to real world problems, I will show how well it performed in practice in Chapter 5 and present various ways it could be improved further as well as its limitations in Chapter 6.

Chapter 2

Essential Background

This section provides some essential background information necessary to understand our work.

2.1 Fundamental Definitions

I will start this section by giving some essential definitions starting with the most general ones. I am assuming in all that follows that we have an underlying discrete virtual environment on which virtual agents can move. One commonly used virtual environment type is a grid world. This can be easily visualized as a bounded 2D surface which is divided into a grid of tiles such as the one shown in Figure 2.1. This is a common simplification to many real life problem spaces, including video games. When I refer to grids in this document I will assume that agents can move in the four cardinal directions on the grid to travel from one tile to another if there are no obstacles in their way. Note that our work is not tied to grids in particular.

Definition An *agent* is the main entity that interacts with the environment. It could represent a robot in a physical world or some virtual unit in a video game. Agents are assumed to occupy space and as such two agents can never co-exist at the same location.

Definition A *node*, also called a *vertex*, is the smallest unit of space in which an agent can exist. At a given point in time a node can contain no agents at all, in which case I call it a vacant node, or a hole. Any node can contain at most one agent at any given time.

Definition An *edge* is a link between a pair of nodes that allows an agent to move from one of the nodes to the other. Nodes connected by an edge are said to be adjacent nodes. In our context, all edges are undirected, which means they allow motion in both directions. There is at most one edge between any given pair of nodes (*i.e.,* I do not allow multi-edges). Like nodes, in our context an edge can be traversed by a single agent at a time. This means that two agents standing at two adjacent nodes cannot swap positions directly in one move.

Definition A *graph*, in our context, is a representation of a virtual environment. It is a set of nodes and edges that make up the structure of the graph.

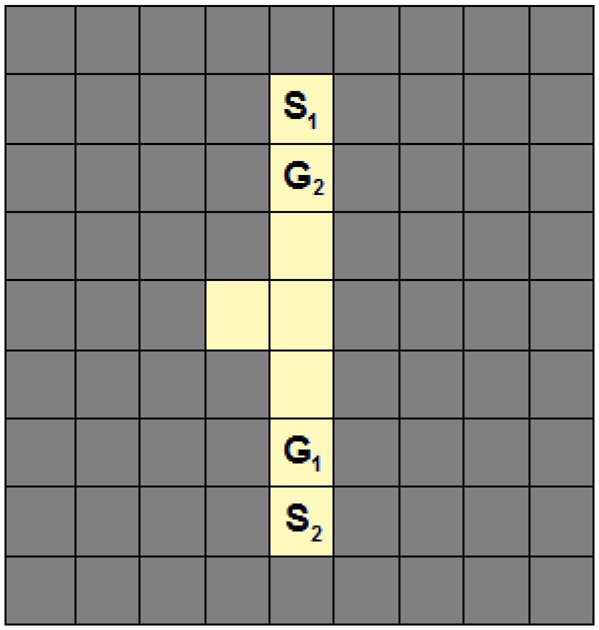

Figure 2.1: An example grid map. S is the symbol for start and G is the symbol for goal. Dark cells have obstacles while white ones do not.

Definition Given two nodes a and b a *path* is the set of nodes $a, n_1, n_2, ..., n_k, b$ where edges exist between a and n_1, between n_{i-1} and n_i for $i \in [1, k]$, and between n_k and b. In our context the graphs do not allow multiple edges between any pair of nodes so a path can be uniquely identified by only the nodes of which it is composed.

Definition A *cycle*, or *simple loop*, is a set of nodes and edges that together construct a path that an agent can take to move from one node back to itself without visiting any of the other nodes in the cycle twice.

Definition In our domain where all graphs are undirected, a tree is a connected graph that contains no cycles.

Definition A *configuration* of agents is a set of agents on a graph each having a pair of start and target nodes. A *valid configuration* is one where no two agents share a start node or an end node.

2.2 Basic Pathfinding

Pathfinding refers to the process through which a path between two nodes is found on a graph.

2.2.1 Breadth First Search (BFS)

BFS works by expanding[1] one node at a time, starting with the agent's start location, and adding the current node's neighbors to a queue for later expansion. The algorithm keeps running until either

[1]Expanding a node in this context means looking at its neighbors and potentially performing some action for each of them.

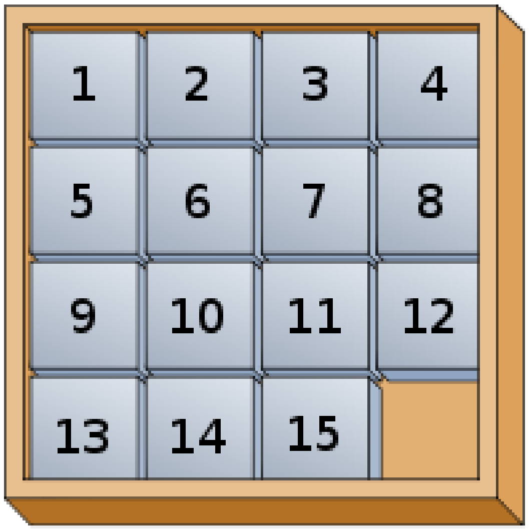

Figure 2.2: An example of a solved 15 tile puzzle. A tile can move if it is adjacent to the blank by sliding into the blank position [34].

(1) the queue is depleted, in which case it concludes that no path is possible or (2) the goal node, the agent's destination node, is generated, in which case it concludes that a path exists. BFS will find the shortest path between both nodes, provided the edges are unweighted.[2]

2.3 Multi-agent Pathfinding

The generalization of the basic pathfinding problem is that of multi-agent pathfinding. Instead of having a single agent that needs to navigate to some destination through a graph, in this generalization I have multiple agents that need to reach their respective destinations without colliding with one another. The main complexity arises from the interaction of the agents. Instead of each agent having to decide at each step which action to take in isolation, agents now have to order their moves in such a way that no two agents end up at the same location.

Multi-agent pathfinding problems can vary in difficulty mostly depending on the number of agents. On one extreme, if we have a single agent, it becomes a basic pathfinding problem that can be efficiently solved optimally. On the other extreme, we can consider the sliding puzzle problem [6], shown in Figure 2.2, a multi-agent pathfinding problem where each tile is an agent. For this reason finding optimal solutions for the multi-agent pathfinding problem is NP-complete [16]. One notable optimal approach used the idea of operator decomposition to reduce the problem space and be able to handle larger problems [24].

There are different variations on how a multi-agent pathfinding problem can be approached.

[2]Weights on edges allow them to have varying costs. In this work, I assume all edges are unweighted and always have a unit cost.

2.3.1 Centralized vs Distributed

We can classify solution approaches to solving multi-agent pathfinding problems into two classes: centralized and distributed. Centralized approaches plan a global solution taking into consideration the locations of all agents. They are typically used when optimality or completeness is needed. These approaches usually scale up poorly. Distributed approaches plan for each individual agent with minimal communication with other agents, this typically relies on agents leaving cues that other agents can use to make their decisions at a later time. Distributed approaches usually trade off completeness and optimality for performance so they scale up much better, but may end up in deadlocks.

2.3.2 Sequential vs Simultaneous Moves

There are two ways to look at how the agents move through the graph. The moves can be sequential, which means at any time step only a single agent can be traveling through some edge in the graph. The sliding puzzle problem falls into this group. At each time step there are $m \times a$ possible moves, where a is the average number of actions available to each agent and m is the number of agents.

Alternatively we could allow simultaneous moves which means at any given time multiple agents can be moving through different edges. It also means that an agent can move into an occupied node if the agent occupying it is also moving out at the same time. Figure 2.3 shows an example where agents are packed tightly with no free space and need to rotate anti-clockwise. Solving this kind of problem requires simultaneous moves. The combined search space when simultaneous moves are allowed is exponential in the number of agents as we now get a^m possible combined moves at each time step. It is easy to see that even for a moderate number of agents, and just the moves in the cardinal directions as actions, we get a huge number of possibilities per time step.

My approach in this work belongs to the centralized class, however, it scales up very well to large problems by not looking for optimal solutions. I also use sequential moves and will guarantee completeness within a specific domain of problems but will produce suboptimal solutions.

2.4 Relevant work: Solvability of Multi Robot Motion Planning Problems on Trees

The paper by Masehian and Nejad [14], on which a core part of our own work is based, focuses on multi-agent path planning problems on trees and does not attempt to solve the problem, but rather detect whether or not it is solvable. Using the number of "holes" (H) in the graph, a set of zones called "Influence Zones (IZ)" are constructed around junctions[3] containing nodes that are within a distance bounded by H. Pairs of IZ's are then merged into "Interconnected Influence Zones (IIZ)" if the distance between their junctions is not greater than H-2. Building a chain of theoretical proofs

[3]A junction is a node that has more than two neighbors

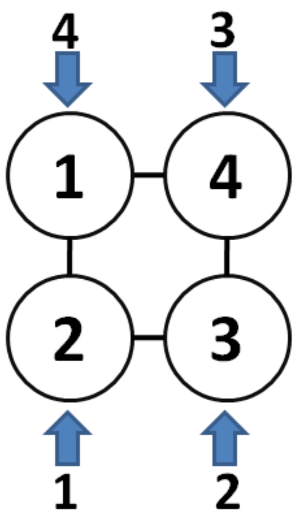

Figure 2.3: A rotation loop example that cannot be solved with sequential moves. This is easily solved by approaches that allow simultaneous moves like Operator Decomposition [24].

that reason about stars, extended stars, and finally general trees, the paper presents conditions for complete solvability for general trees and thus presents the notions of a “Solvable Tree (ST),” a “Partially Solvable Tree (PST),” and a “Minimal Solvable Tree (MST)” for a particular number of agents. The paper uses these notions to provide an instance feasibility checking algorithm on trees using what they call “Maximum Reachability Space” but we are only concerned with ST’s in this work.

Chapter 3

Theory

In this section I give a formal definition of our agent swapping algorithm, TASS, as well as proof of its polynomial bounds. I also prove that a certain class of problems, SLIDEABLE [33], is a strict subset of the problems we can solve with the help of graph-to-tree conversion algorithms. Finally, I will present a worked step-by-step example to clarify how TASS actually works.

3.1 Definitions

I will begin by providing a few key definitions, the first two of which are restated from [14]. In all of these definitions and lemmas there is some fixed (given) undirected tree T in which each node is either a "hole" or contains an "agent". All agents are distinct. An agent that is adjacent to a hole can swap places with it. The number of nodes in T is n, the number of agents is m, and the number of holes is $H = n - m$. Agents have goal locations that they are trying to reach; each goal location must be distinct for a problem to be solvable.

Definition [14] A *junction* is any node in T with 3 or more neighbors.

Definition [14] Two junctions are *near* if there is no junction on the path between them.

Definition Let v be any node in T and S any set of nodes in T. $Trees(v, S)$ is defined as the set of trees that result when v is removed from T, excluding any such tree that contains one or more nodes in S. Refer to Figure 3.1 for an example.

Definition Let v be any node in T and S any set of nodes in T. $Holes(v, S)$ is defined to be the total number of holes in $Trees(v, S)$.

Definition Let u and v be any two nodes in T. $P(u, v)$ is the set of all nodes on the path from u to v in T, including u but excluding v.

Definition Let u and v be any two nodes in T. $E(u, v)$ is the number of edges on the path from u to v in T.

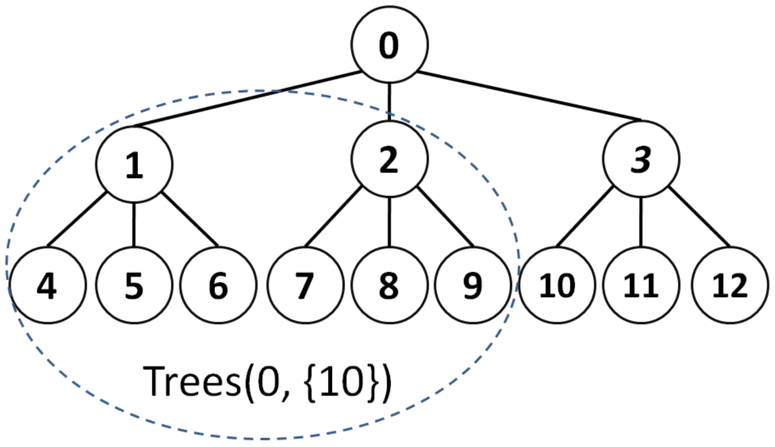

Figure 3.1: $Trees(0, \{10\})$ is the set of all trees resulting from removing vertex 0 except the tree that contains node 10.

Note: If $u \neq v$ then $E(u, v)$ is also the number of nodes on $P(u, v)$. That is how we are going to think of it in most of what follows.

Definition Let u and v be any two nodes in T. $H(u, v)$ is the number of holes on the path from u to v in T excluding u.

Note: $H(u, v)$ and $H(v, u)$ will differ by one if one of the endpoints is a hole and the other one is not.

Definition Let u and v be any two nodes in T. $A(u, v)$ is the number of agents on the path from u to v excluding u. This is equal to $E(u, v) - H(u, v)$.

3.2 Tree Solvability Conditions

Based on the work in [14] I have derived the following three simplified, but equivalent, sufficient conditions for a tree, T, to be solvable for any configuration of agents:

1. T contains at least one junction.
2. There are at most $H - 1$ edges on the path between any node and the junction nearest to it.
3. The path connecting any two junctions that are near contains at most $H - 2$ edges.

Any tree meeting all these conditions is called a "Solvable Tree". In what follows we will always assume that T is solvable.

3.3 Polynomially Solving Solvable Trees

In this section I will show that we can solve, in polynomial time as [14] has proven possible, any problem instance on a solvable tree, thus guaranteeing completeness within the domain of solvable trees.

Lemma 3.3.1 *Let x and y be any two adjacent nodes in T, and J any junction nearer to y than to x such that $Holes(J, \{x, y\}) \geq A(y, J) + 2$. Then there exists a sequence of moves that, when completed, swaps the contents of x and y and leaves the contents of all other nodes unchanged.*

Proof. Figure 3.2, ignoring the holes for now, depicts the general situation, where y and J are distinct. There are $E(y, J)$ nodes between y and J (including J but not y) and J has at least two neighbors, which I will refer to as "ports," A and B, in addition to the neighbor on the path between y and J. Let T' be the tree rooted at J excluding the subtree that contains x and y. By the premise of the lemma, T' contains at least $A(y, J) + 2$ holes. This is enough holes to clear all the nodes between y and J (including J but not y) and to clear A and B as well, since in the starting configuration there are $A(y, J)$ agents on the path from y to J (including J but not y). A sequence of moves S that establishes the configuration in Figure 3.2 is constructed as follows. Move holes within T' as necessary so that there are at least $A(y, J) + 2$ holes in the A and B subtrees combined and at least one hole in each. Now move these holes into the nodes on the path from y to J, starting with the node closest to y and finishing with J, being sure to always leave at least one hole in each of the A and B subtrees. Finally, move a hole from A's subtree into A and a hole from B's subtree into B. All this can be done without disturbing x and y since the holes are all being drawn from and moved to the part of the tree on the opposite side of y from x. This establishes the configuration shown in Figure 3.2. Let S be the sequence of moves executed thus far. Now move y into A and x into B, and then move y to the node in which x began and then x to the node in which y began. x and y have swapped positions. All that remains is to reverse sequence S to get all the nodes that have been disturbed by S back into their initial positions. S did not disturb x and y when it was executed, so reversing S will not disturb them either.

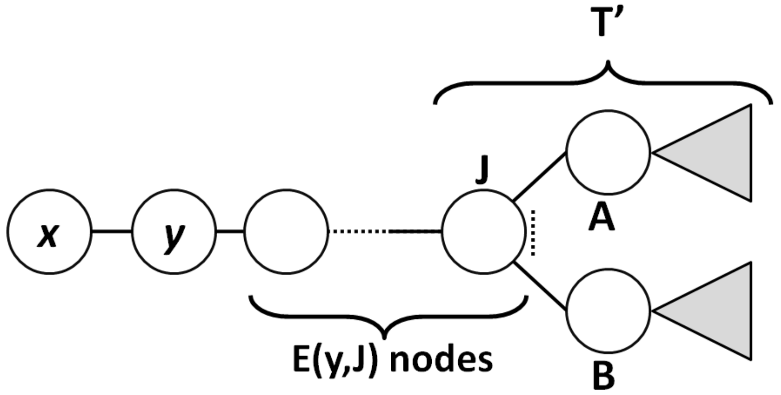

Figure 3.2: This configuration of agents allows x and y to be swapped if we can move holes to fill the blank nodes in the figure (*cf.* Lemma 3.3.1). Any node not labeled with a letter inside it should be a hole for the problem to be solvable. There may be more nodes to the left of x; they are irrelevant and so are not shown. The triangles on the right side represent the remainder of the graph beyond the ports, it is from these subtrees that we will try to bring holes as necessary.

When $y = J$ the above reasoning does not apply since the sequence S may move y out of J. This case divides into four sub-cases. In the first two there are holes in different subtrees while in the last two all holes are in a single subtree.

1. If J has two neighbors (not counting x), A and B, whose subtrees each contain at least one hole then, just as we did above, we can clear A and B using those holes without disturbing x and y (let S' be the sequence that does this), move y into A and x into B, and then move y to the node in which x began and then x into J. x and y have swapped positions, and A and B are clear. We can now reverse S' to restore the A and B subtrees to their original configuration without disturbing x and y.

2. If there is at least one hole on the opposite side of x from y, it can be moved to J. Let S' be the sequence that does this. This keeps x and y adjacent and does not decrease $Holes(J, \{x, y\})$, but y is no longer in J so we can apply the general case ($y \neq J$, above) to swap x and y. When this is finished J will be clear and when we apply S' in reverse x will move into J, y will move into the node that originally contained x and the other nodes that were disturbed to get this hole into J will be restored to their original configuration.

3. All holes are in the $Trees(x, \{y\})$. This case can never occur because, by the premise of the lemma, we have $Holes(J, \{x, y\}) \geq A(y, J)+2$. So at least 2 holes are not in $Trees(x, \{y\})$.

4. If none of the preceding cases applies it means all H holes are in one subtree rooted at a neighbor of J other than x. This subtree must contain a junction because it contains at least H nodes and T has the property that there are at most $H - 1$ edges on the path between any node and the junction nearest to it. Let J' be the junction in this subtree that is nearest to J. J' is obviously nearer to y than x and does not contain y so if we can establish that $Holes(J', \{x, y\}) \geq A(y, J')+2$, then we can use this junction and the method in the general case ($y \neq J$, above) to swap x and y. T has the property that the path connecting any two junctions that are near contains at most $H-2$ edges. J and J' are near, so this property ensures that $E(y, J') \leq H-2$. Since $A(y, J) = E(y, J)-H(y, J)$ then $A(y, J')+H(y, J') \leq H-2$ and so we get $A(y, J') + 2 \leq H - H(y, J')$. But $H - H(y, J')$ is equal to $Holes(J', \{x, y\})$ because all H holes are in this subtree, so $H - H(y, J')$ of them are in $Trees(J', \{x, y\})$. Thus we have established that $Holes(J', \{x, y\}) \geq A(y, J') + 2$ and J' can be used to swap x and y.

Definition A *safe* junction, J, for two agents u and v, where u is closer to J, is a junction where the two agents can be swapped. This means that all nodes on the path between u and J can be cleared in addition to two other nodes adjacent to J that are not on $P(u, J)$.

Definition Let u and v be any two adjacent nodes in T. $J(u, v)$ is the junction in T, if it exists, that is closest to u and closer to u than to v.

Lemma 3.3.2 *Let u and v be any two adjacent nodes in T. Then at least one of $J(u, v)$ and $J(v, u)$ always exists.*

Proof. Let J be a junction closest to u. This must exist because T contains at least one junction. If J is closer to u than to v then $J(u, v)$ exists (because J satisfies the definition of $J(u, v)$). If J is closer to v than to u then $J(u, v)$ does not exist but $J(v, u)$ does (because J satisfies the definition of $J(v, u)$). ∎

Theorem 3.3.3 *Let u and v be any two adjacent nodes in T. Then there exists a junction J such that at least one of the following conditions holds:*

1. $Holes(J, \{u, v\}) \geq A(u, J) + 2$
2. $Holes(J, \{u, v\}) \geq A(v, J) + 2$

Proof. From Lemma 3.3.2 we know that at least one of $J_u = J(u, v)$ and $J_v = J(v, u)$ exists. I will show that at least one of these satisfies the requirements for J in the Theorem statement.

I will begin with the general case, shown in Figure 3.3, where both J_u and J_v exist, $u \neq J_u$, and $v \neq J_v$.

I will simplify our notation to make the following proof simpler. K_1, H_1, K_2, H_2, W_1, and W_2 in the figure are $E(u, J_u)$, $H(u, J_u)$, $E(v, J_v)$, $H(v, J_v)$, $Holes(J_u, \{u, v\})$ and $Holes(J_v, \{v, u\})$ respectively. So to prove the theorem is to prove that either $W_1 \geq (K_1 - H_1 + 2)$ or $W_2 \geq (K_2 - H_2 + 2)$.

J_u and J_v are near junctions, so there are at most $H - 2$ edges on the path connecting them, *i.e.,* at most $H - 1$ nodes including both J_u and J_v. Hence $K_1 + K_2 + 2 \leq H - 1$, *i.e.,* $K_1 + K_2 + 3 \leq H = H_1 + H_2 + W_1 + W_2$. Hence $3 + (K_1 - H_1) + (K_2 - H_2) \leq W_1 + W_2$.

Now suppose that the theorem is false, *i.e.,* that $W_1 \leq (K_1 - H_1 + 1)$ and $W_2 \leq (K_2 - H_2 + 1)$. This implies $W_1 + W_2 \leq (K_1 - H_1 + 1) + (K_2 - H_2 + 1) = 2 + (K_1 - H_1) + (K_2 - H_2)$, contradicting the fact we just derived, $W_1 + W_2 \geq 3 + (K_1 - H_1) + (K_2 - H_2)$.

The same reasoning applies when either $u = J_u$ or $v = J_v$ (or both) since these are just the cases $K_1 = H_1 = 0$ and $K_2 = H_2 = 0$ respectively.

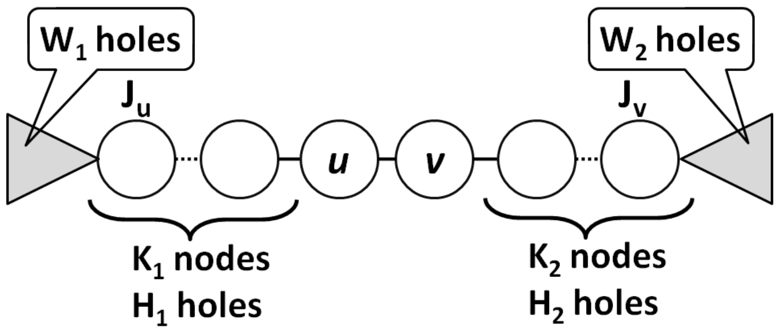

Figure 3.3: The general case analyzed in Theorem 3.3.3.

The final case to consider is when one of the junctions, say J_u, does not exist. In this case we have $W_1 = 0$ and a total of $K_1 + K_2 + 2$ nodes from the "leftmost" node on this chain, X (which can be visualized as J_u in Figure 3.3 with the understanding that the subtree to its left is empty), to J_v (including X, J_v, u and v) which means $K_1 + K_2 + 1$ edges. We know the number of edges on the path from X to J_v (the junction nearest to X in this case) cannot exceed $H - 1$ (this is one of the properties we have assumed of T), so $K_1 + K_2 + 2 \leq H = H_1 + H_2 + W_2$. This implies that $W_2 \geq 2 + (K_1 - H_1) + (K_2 - H_2)$. Since $(K_1 - H_1) \geq 0$ we are guaranteed that $W_2 \geq 2 + (K_2 - H_2)$ as required by the theorem. ∎

Corollary 3.3.4 *Let u and v be any two adjacent nodes in T. Then there exists a sequence of moves that, when completed, swaps the contents of u and v and leaves the contents of all other nodes unchanged.*

Proof. Use Theorem 3.3.3 to obtain a junction J. If condition (1) of Theorem 3.3.3 holds, invoke Lemma 3.3.1 with J, $x = v$ and $y = u$; otherwise invoke Lemma 3.3.1 with J, $x = u$ and $y = v$. ∎

Lemma 3.3.5 *The number of moves described in Corollary 3.3.4 is polynomially bounded.*

Proof. Finding the correct safe junction, as required by Theorem 3.3.3 and Lemma 3.3.1, can be performed in $O(1)$ since we need to look at no more than two junctions, and the nearest one to each node can be pre-computed.[1] The maximum distance between a node and a junction is bounded by H if the problem meets our solvability requirements. So to clear a junction for a swap we may need to move H holes (all of them) a distance of n; this can be done in $O(nH)$. Doing the actual swap is $O(n)$ and returning the agents again is $O(nH)$ so the total complexity of swapping two agents is still $O(nH)$ or, equivalently, $O(n(n - m))$. ∎

Putting together the ideas from Theorem 3.3.3, Corollary 3.3.4, and Lemma 3.3.1, I present the outline of the complete Tree-based Agent Swapping Strategy (TASS):

- Pick an agent that is not at its target node.[2]
- Move this agent to its target node by swapping it with any agents along the path to the target (See below).
- Repeat until all agents are on their target nodes.

The swapping algorithm is:

- Find the nearest junction that has enough holes to allow the agents to swap at that junction (see Theorem 3.3.3).

[1] This pre-computation can be performed in $O(n)$ by running a search algorithm like BFS with all junctions on the start queue and terminating when all nodes have been visited.

[2] In my implementation I picked agents in the order they were supplied in.

- Move all holes towards the agent until the junction can be used for the swap (see Theorem 3.3.3).
- Swap the agents as described in Lemma 3.3.1.
- Return the holes to their original positions, restoring all other agents as well.

Theorem 3.3.6 *On trees meeting our solvability conditions listed in Section 3.2, we can solve any multi-agent problem in polynomial time.*

Proof. We have m agents that may need to be swapped with all other agents to reach their final destinations for a total of m^2 swaps and from Lemma 3.3.5 we know that we can swap any two agents in $O(nH)$. In addition to all swaps moving the agent through free space takes $O(n)$, so we can solve a complete multi-agent problem in $O(m^2nH)$ or, equivalently, $O(m^2n(n-m))$. ∎

Theorem 3.3.7 *On rooted balanced trees where the branching factor of all nodes, except the leaves and possibly the root, is fixed and is greater than 2, TASS can solve multi-agent problems in $O(mlog^2(n))$ complexity.*[3]

Proof. In these trees we have a constant distance, one, from any node to its nearest junction and O(1) at most to a safe junction. Furthermore, the distance between any two nodes in the tree is bounded by $2log(n)$ so we can clear a junction along with its two ports and the path leading to it in $O(log(n))$. We also have m agents that are at most $2log(n)$ nodes away from their destinations, since the depth of the tree is $log(n)$. The agents will never need more swaps than the length of the path to their targets, so this is an upper bound on how many swaps each needs. In total we get a total complexity of $O(mlog^2(n))$. ∎

These bounds also apply to the maximum solution length, meaning that solutions will be no larger than $O(m^2nH)$, or $O(m^2n(n-m))$, moves long for general trees and $O(mlog^2(n))$ for balanced trees.

[3]The base of the log is equal to the fixed branching factor of the tree - 1.

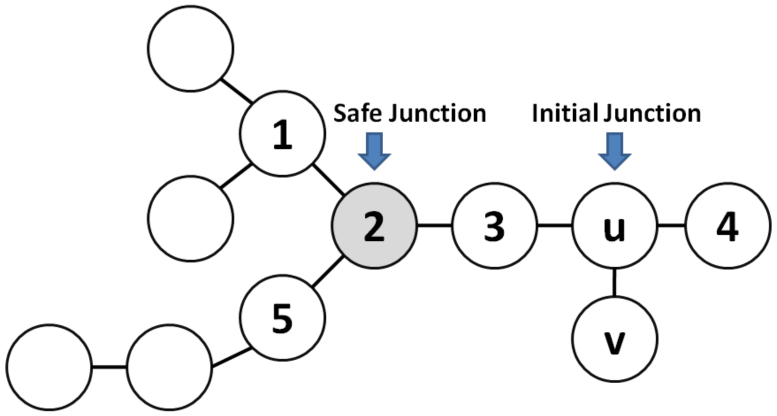

Figure 3.4: Original tree. We want to switch u and v.

3.4 Worked Example

To illustrate how TASS works, we will go over a quick example on the tree shown in Figure 3.4. Suppose we want to swap agents u and v on the shown tree. The first step would be to identify a junction that meets the conditions of Theorem 3.3.3. The first such junction we have is the one currently occupied by u. This meets the conditions of the theorem, however, it is not safe to do the swap as it falls under the third category of Lemma 3.3.1, *i.e.,* the junction does not have enough holes where they are needed. Following the reasoning in the proof, we use another junction that is safe, as labeled in Figure 3.4, and use it to swap the agents instead.

The first step of TASS is to clear the junction and two of its ports.[4] To do this TASS moves agents 1, 2, and 5 out of the way and ends up with the configuration in Figure 3.5. TASS then clears the path between u and the junction by moving the holes in the lower left[5] and we get the configuration in Figure 3.6. Now that the junction, the path to it, and two of its ports are clear we can do the swap. As shown in Figure 3.7 we swap by moving u into one port, v into the other, and then pull them out in reverse order to obtain the configuration shown in Figure 3.8. The final step is to return the other agents to their original locations at the beginning of the process by reversing the sequence of actions that displaced them. This achieves the final state in Figure 3.9.

[4]The exact order of which nodes to clear first is not significant, since holes can be easily moved around the graph.
[5]Note that when we move a hole, agents on its path are shifted one node along the path.

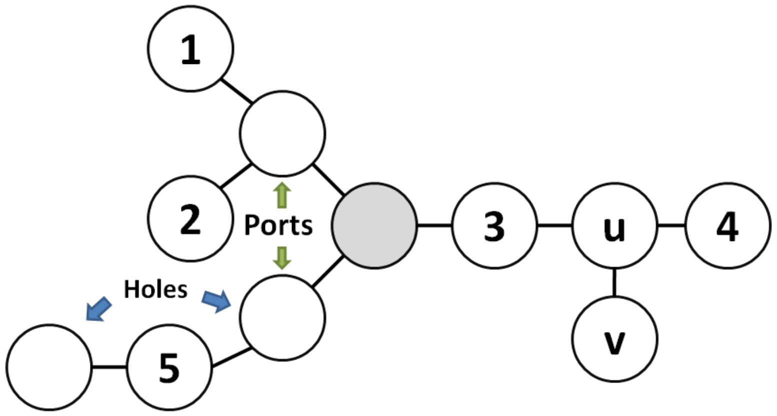

Figure 3.5: First step: Clear the junction and the two ports.

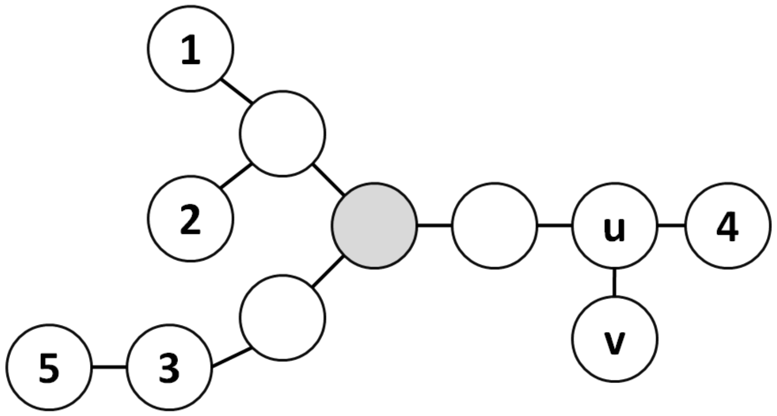

Figure 3.6: Second step: Clear the path to the junction.

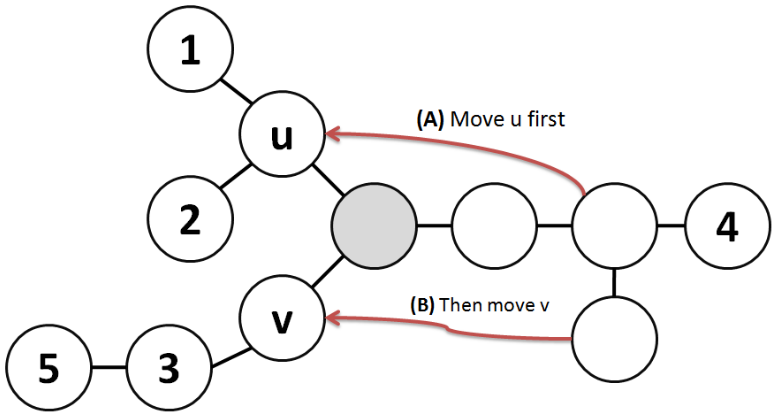

Figure 3.7: Third step: Move the two agents to the ports.

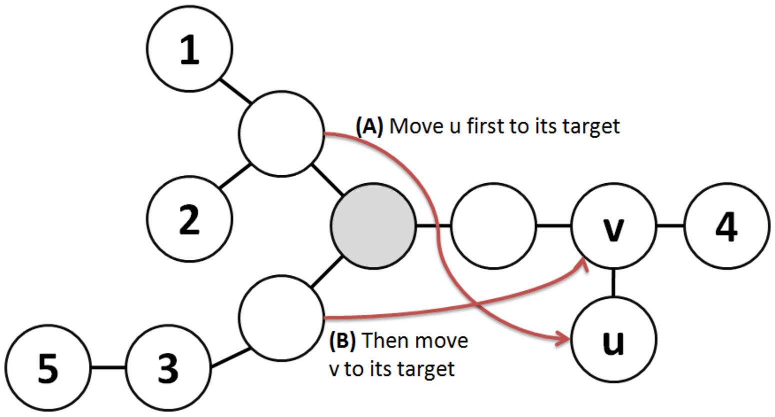

Figure 3.8: Fourth step: Move the agents to their goals in reverse order.

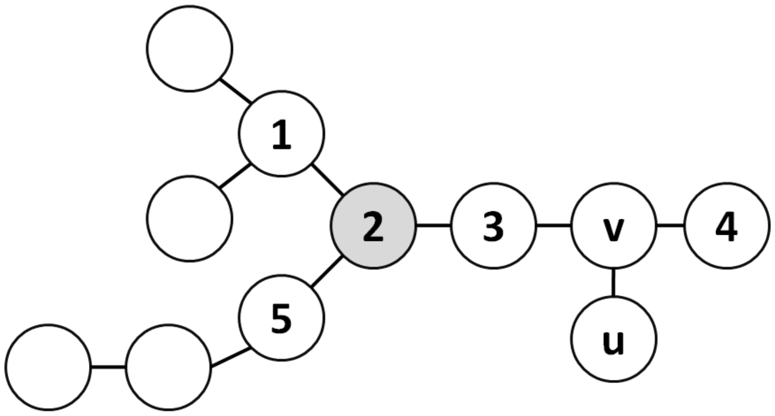

Figure 3.9: Fifth step: Return all other agents back to where they were.

3.5 Graph To Tree Decomposition

The long term objective of our work is to be able to solve multi-agent pathfinding problems on the most general problem domain. This can be achieved by finding ways to decompose graphs into solvable trees. In this section I will present two Graph-to-Tree Decomposition (GTD) algorithms that convert graphs to trees. The first is GTD-SLIDEABLE which takes as input a graph, along with a configuration of agents, that belongs to a family of problems called SLIDEABLE [33]. This class of problems was shown to be solvable in polynomial time. The original definition for SLIDEABLE problems given in [33] can be rewritten as follows:

Definition A problem is SLIDEABLE if the following conditions hold:

1. A path, call it P_i, exists for each agent i from its starting location to its target.
2. For each three consecutive steps a, b, and c on the path P_i, an alternative path, Ω_{iac}, exists from a to c that does not pass through b.
3. The first step on P_i is vacant.
4. No target for any agent, j, will be on any P_i where $i \neq j$.
5. No target for any agent will be on any of the alternative paths for any of the agents, including itself except for the alternative path leading to this target.

3.5.1 Slideable Induced Trees

GTD-SLIDEABLE converts a given SLIDEABLE problem into a set of solvable trees, which I will call SLIDEABLE Induced Trees.

GTD-SLIDEABLE

The GTD-SLIDEABLE algorithm creates a forest of trees as follows. Create the induced tree by adding all the nodes and edges on P_i for $i \in [1, m]$, breaking any cycles that are created arbitrarily.

The following lemma proves that each of the SLIDEABLE induced trees are solvable.

Lemma 3.5.1 *We can always decompose a* SLIDEABLE *problem into a set of induced trees where each is guaranteed to be solvable.*

Proof. When GTD-SLIDEABLE produces more than a single tree resulting from the conversion, we can obtain a solution for each tree separately and concatenate the solutions to reach a global plan. The global plan is valid because the final resulting trees (after potentially merging some of them) are non-intersecting. Note that we may merge some of the initial trees as described below, so in that case we consider the merged tree a single tree. In what follows I will focus on solving a single tree.

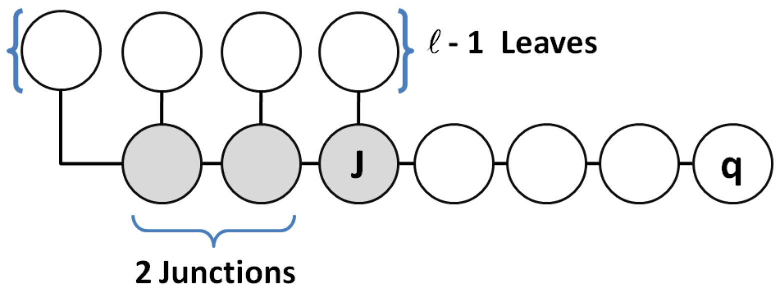

Figure 3.10: Example of a worst case scenario for Condition 2 showing a tree with three junctions where a node q is picked so that it is as far as possible from its nearest junction.

I will prove the solvability for the general case first where the induced tree has at least 3 junctions and then I will prove it for 2 or fewer junctions.

If all agents are adjacent to their target nodes, or on their target nodes, then the problem becomes trivially solvable. Otherwise, if a portion of the agents are adjacent to, or on, their target nodes, then they can be considered solved and removed from further consideration. In what follows I assume a non-trivial problem.

Observation Let ℓ be the number of leaves in T. By the definition of SLIDEABLE we know that no P_i for any i goes through a target node for any other agent except i. Tree T only contains edges from P_i for any i and thus it can never have a path going through a target node, except when it is the target for agent i in which case the path ends at the target node. This means that every target node must be on a leaf in T. The only agent that can possibly occupy the target node for agent i is agent i itself, and since we have a non-trivial problem, the target node for each agent has to be empty. This implies that $\ell \geq m$, so $H = n - m \geq n - \ell$.

The general case: Three junctions or more

Now, consider the three conditions required for tree solvability:

Condition 1: (*T contains at least one junction.*) This is satisfied trivially since we are assuming we have 3 junctions.

Condition 2: (*There are at most $H - 1$ edges on the path between any node and the junction nearest to it.*) Pick any node q and let J be a junction nearest to q. I will prove this condition by counting the number of nodes that cannot be on $P(J, q)$. We can easily see that $E(J, q) \leq n - \ell + 1 - 2 - 1$. On the right-hand side of the equation we add 1 because q can be a leaf node. We subtract 2 for the two junctions that cannot be on the path and an additional 1 because the left-hand side is expressed in edges instead of nodes. Figure 3.10 shows which nodes have to be subtracted from n in a worst case scenario. We end up with $E(J, q) \leq n - \ell - 2$. Since $H \geq n - \ell$, $E(J, q) \leq n - \ell - 2 \leq H - 2$, which is stronger than required.

Condition 3: (*The path connecting any two junctions that are near contains at most $H - 2$ edges.*) As in the proof of Condition 2, I will prove this condition by counting the number of nodes

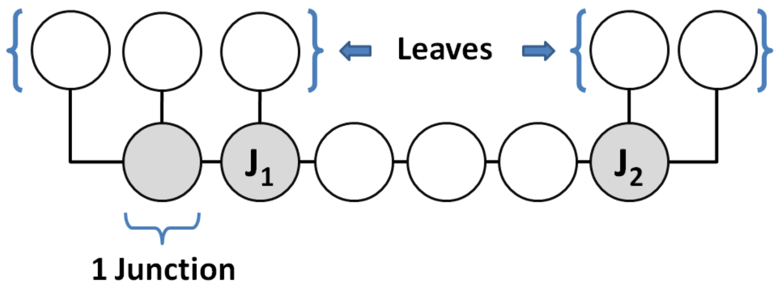

Figure 3.11: Example of a worst case scenario for Condition 3 showing a tree with three junctions where two of them are as far as possible.

that cannot be on $P(J_1, J_2)$. Because we have at least three junctions in total, $\ell \geq 5$. For all near junctions J_1 and J_2, $E(J_1, J_2) \leq n - \ell - 1 - 1$. This is because (1) leaves cannot be on the path between junctions, (2) at least one other junction will not be on the path between J_1 and J_2 and (3) $E(J_1, J_2)$ counts edges which is one less than the number of nodes on the same path. Figure 3.11 shows the nodes that cannot be on $P(J_1, J_2)$ in a worst case scenario. I have already established that $H \geq n - \ell$ so $E(J_1, J_2) \leq n - \ell - 2 \leq H - 2$. This establishes that the distance between any two near junctions is less than or equal to $H - 2$.

Special case #1: A single junction

In general we do not guarantee that the induced tree is solvable, but I will show that if it is not we can use the alternate paths to make it solvable.

Condition 1: (*T contains at least one junction.*) This is satisfied trivially since I am assuming we have a junction.

Condition 2: (*There are at most $H - 1$ edges on the path between any node and the junction nearest to it.*) Let q be a leaf node furthest from J, the only junction. Let $E(J, q) > H - 1$ because otherwise the condition is already satisfied. Figure 3.12 shows a typical example. Since all leaves except one, are not on $P(q, J)$ then $E(J, q) \leq n - \ell$. $H - 1 < E(J, q) \leq n - \ell$ then $n - m - 1 < n - \ell$ which leads to $m + 1 > \ell$ but by the observation above $m \leq \ell$ so $m = \ell$ (*i.e.,* all leaves are targets). This results in $H - 1 < E(J, q) \leq H$ which means $E(J, q) = H$ and so to satisfy the condition we need to either add one extra hole or find another path between J and q that is at least one edge shorter.

Let p be the node adjacent to J on $P(J, q)$ and let w be the node adjacent to p on $P(p, q)$.[6] Since all leaves are targets and there exists an edge between p and w in the induced tree then one of the leaves connected to J is a target for an agent that is not in J or p. This means that there must exist an alternate path from w to J that does not go through p. There are three cases for this alternate path: (1) it includes at least one node not in T, in which case we just add this node (which must be a hole) and the edge connecting it to any node in T, (2) w is connected directly to J in which case we

[6]Note that w can be the same node as q.

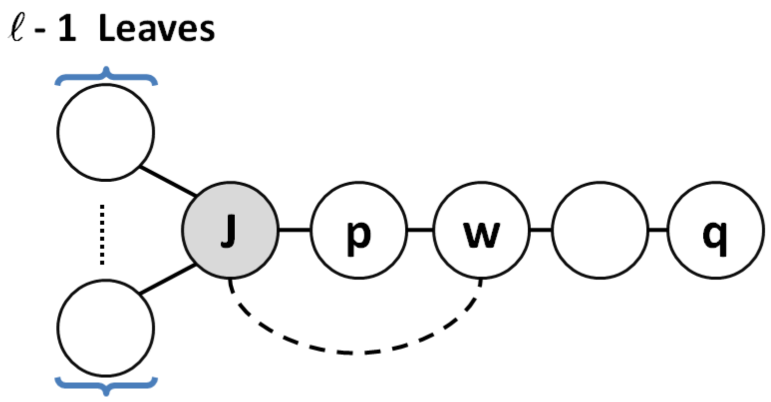

Figure 3.12: Example of a tree with a single junction.

break the edge between p and w and add the edge between w and J to shorten the distance between q and J by one, or (3) the alternative path goes through another node on $P(w, q)$, call it u, which is directly connected to J via an edge not in T and in this case we break the edge between p and J and add the edge between u and J to reduce the distance between q and J by more than one.

Condition 3: (*The path connecting any two junctions that are near contains at most $H - 2$ edges.*) This is satisfied trivially since there is only one junction.

Special case #2: Exactly 2 junctions

Condition 1: (*T contains at least one junction.*) This is satisfied trivially since I am assuming we have 2 junctions.

Condition 2: (*There are at most $H - 1$ edges on the path between any node and the junction nearest to it.*) Pick any node q and its nearest junction J, we want to show that $E(J, q) \leq H-1$. We know that at least $\ell-1$ leaves are not on the path as well as the other junction. So $E(J, q) \leq n-\ell-1$. One leaf node can be potentially on the path (that is q itself) so we subtract one for this leaf and another for the junction, and one node has to be added to the RHS of the inequality since the LHS is expressed in number of edges instead of nodes. We know that $m \leq \ell$ so $E(J, q) \leq (n - m) - 1$ or equivalently $E(J, q) \leq H - 1$.

Condition 3: (*The path connecting any two junctions that are near contains at most $H - 2$ edges.*) We only have two junctions, call them J_1 and J_2, so we want to show $E(J_1, J_2) \leq H - 2$. As illustrated in Figure 3.13, none of the leaves can be on the path and so $E(J_1, J_2) \leq n - \ell - 1$. Now assume that $E(J_1, J_2) > H - 2$, otherwise we are done. We know that $H \geq m + 1$ for any SLIDEABLE induced tree since at least one extra node is vacant because the first step of P_i is vacant for any i. An exception is if every agent is adjacent to its target node, in which case the problem is trivially solved by moving every agent directly to its target. So we have $H - 2 < E(J_1, J_2) \leq n - \ell - 1$ which leads to $n - m - 2 < n - \ell - 1$ and we get $m + 1 > \ell$. So $m = \ell$. This means that all leaves are target nodes and so no alternate paths can go through any of the leaves. This means we

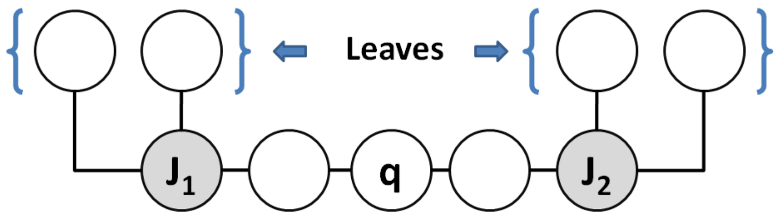

Figure 3.13: Example of a tree with 2 junctions.

are, at worst, lacking just one extra hole.[7] Let q be any node on $P(J_1, J_2)$ excluding the junctions (if such node does not exist then the junctions are adjacent and the condition holds trivially since we have at least two agents and 3 holes). Since the tree is connected, then at least one agent on one side of q has a target on the opposite side. This means there must be some alternate path, Ω, between the two nodes adjacent to q that does not include q. Ω does not go through any of the leaves, so we have two possibilities:

1. Ω uses a node that is not in T. This node must be a hole since all agents are on T and thus adding this node as a leaf and the edge leading to it to T will increase the number of holes by one and we are done.

2. Ω uses only nodes in T. This means that a node on $P(q, J_1)$ is connected to a node on $P(q, J_2)$ in the tree. If they are connected directly with an edge then adding this edge and removing one of the edges incident on q will keep T a tree but will reduce the distance between the two junctions by at least one and we are done. If they are connected through some other node not in T, then we add this extra node along with its incident edge to increase the number of holes by one.

Now that I have proven that with the aid of a GTD algorithm, TASS can solve all SLIDEABLE problems, I have shown that the SLIDEABLE class of problems is a strict subset of the problems that TASS + GTD can solve. Figure 3.14 shows an example that is not SLIDEABLE because there are no alternate paths, yet this can easily be solved by TASS. Observe that problems on trees can never belong to the SLIDEABLE class.

3.5.2 GTD-MaxJunction

The GTD-MaxJunction algorithm aims to maximize both the number of junctions and the degrees of the junctions when inducing a tree. This is a subtractive algorithm that starts with the complete graph and keeps removing edges as long as a cycle exists. For each cycle, one at a time and in no particular order, the edge with the least priority is broken first. The priorities are assigned to edges as follows (when I refer to nodes, I mean the pair of nodes that the edge connects):

- If either node has a degree of 1, then the edge has priority 4 (highest).

[7]Because $E(J_1, J_2) \leq n - \ell - 1$ leads to $E(J_1, J_2) \leq H - 1$ when $m = l$.

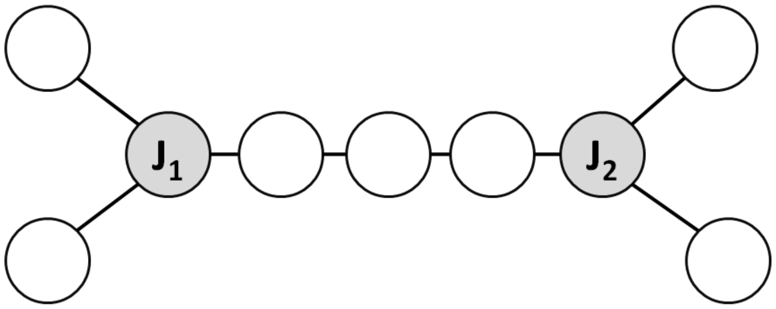

Figure 3.14: A problem which is not SLIDEABLE, but can be solved by TASS. Agents at J_1 and J_2 need to swap positions.

- If either node has a degree of 3, then the edge has priority 3.
- If either node has a degree of 2, then the edge has priority 2.
- All other edges have priority 1 (lowest).

The priorities are assigned in order, so the first condition that is met assigns the priority. For example, if an edge connects nodes whose degrees are 1 and 3, then it would get priority 4. These priorities force the algorithm to avoid isolating any nodes as well as breaking junctions unless there is no other alternative. It also tries to maintain as much connectivity as possible, so when given the choice it would favor removing an edge between two large-degree junctions than to remove an edge on a long chain. Whenever an edge is removed, its neighboring edges' priorities are updated.

The GTD-MaxJunction algorithm runs in polynomial time. Detecting cycles is $O(n)$ and we may need to break up to $n^2 - n$ cycles in a fully connected graph. However, this bound is on general graphs. On grid graphs, it is easy to see that breaking at most one edge for each tile[8] will break all the cycles, in effect bringing down the bound to $O(n)$ and in total we can run the whole algorithm in $O(n^2)$. Note that identifying the edges to break can be done while detecting the cycles instead of sorting after it is done so it has no effect on the complexity. It has to be noted that unlike GTD-SLIDEABLE, GTD-MaxJunction is not instance-based. That is, for a given graph we only need to run it once and use the induced graph for all instances. This means that it will typically be a pre-processing step done of graphs when they are created instead of part of the solver algorithm. The same is not true for GTD-SLIDEABLE as it makes use of the actual configuration of agents to induce its tree, and so it must be executed for each problem instance.

[8]Four nodes forming a unit square.

Chapter 4

Related Work

The problem of multi-agent pathfinding has been extensively studied over the years. This section surveys papers that are related to our work. Some papers present centralized algorithms for solving the problem, where a global plan is built by taking into consideration the locations of all agents in the system, like [19] and [13]. Other works present distributed algorithms, where agents cooperate not by communicating but by finding individual plans that are less likely to conflict with the plans of other agents, such as [22] and [32]. The works can also be divided into theoretical papers like [1], [12], and [33], and papers that include experiments like [29] and [24]. In addition, we can classify work as optimal (and therefore complete[1]) like [24] and suboptimal such as [22]. Finally we can classify work as complete, but not necessarily optimal, as in [19] and [33] and incomplete such as [17] and [32]. In what follows, unless otherwise mentioned, it will be assumed that simultaneous agent moves are allowed.

Table 4.1 lists the related works I cover in this chapter as well as their classification. The completeness column in the table refers to the completeness of an algorithm for the domain of problems it claims to solve, and not necessarily all problems that have solutions. The empirical column indicates whether or not the paper had an empirical section. Some works may have an empirical section that is irrelevant to our work or uses a different metric that is incomparable to ours. Finally, the last column indicates whether or not I have performed my own experiments to compare the algorithm to TASS.

[1]A complete algorithm is guaranteed to eventually find a solution if one exists, whereas an incomplete algorithm may fail to find a solution in some cases.

Paper/Algorithm	**Centralized/Distributed**	**Completeness**	**Optimality**	**Empirical**	**Compared**
Operator Decomposition [24]	Centralized	Complete	Optimal	Yes	Yes
Maximum Group Size [25]	Centralized	Complete	Variable	Yes	No
Increasing Cost Tree Search [20]	Centralized	Complete	Optimal	Yes	No
BIBOX [29]	Centralized	Complete	Suboptimal	Yes	Yes
Weak Transpositions & Critical Path Parallelism [30]	Centralized	Complete	Suboptimal	Yes	No
Subgraph Decomposition [19]	Centralized	Complete	Suboptimal	Yes	No
Constraint-based Path Planning [18]	Centralized	Complete	Suboptimal	Yes	No
Push and Swap [13]	Centralized	Complete	Suboptimal	Yes	Yes
Coordinating within Roadmaps [15]	Centralized	Complete	Suboptimal	Yes	No
MAPP [33]	Centralized	Complete	Suboptimal	No	Yes
Cooperative Pathfinding [22]	Distributed	Incomplete	Suboptimal	Yes	No
FAR [32]	Distributed	Incomplete	Suboptimal	Yes	No
Direction Maps [8] [9]	Distributed	Incomplete	Suboptimal	Yes	No
Planning by Heuristic Priority Adjustment [17]	Distributed	Incomplete	Suboptimal	Limited	No
Cellular Automata-Based Algorithm [31]	Centralized	Complete	Suboptimal	Very limited	No
Coordinating Pebble Motion [12]	Centralized	Complete	Suboptimal	No	No
Feasibility of Pebble Motion of Trees [1]	Centralized	Complete	Suboptimal	No	No
Modeling Collision Avoidance [5]	Distributed	Incomplete	Suboptimal	Yes	No
Biased Cost Pathfinding [3]	Distributed	Incomplete	Suboptimal	Yes	No
Tree-based Agent Swapping Strategy [10]	Centralized	Complete	Suboptimal	Yes	N/A

Table 4.1: Table of related works and their classification. Note that the names in the first column are not always the full names of the papers or algorithms due to space constraints.

4.1 Finding Optimal Solutions to Cooperative Pathfinding Problems

One of the efficient optimal solvers relied on what Standley calls Operator Decomposition (OD) [24]. Instead of branching for all possible moves for all agents at every search node as is typical in pathfinding search, OD proposes a different representation for the search space that is then searched using A*. The idea is that a single agent will be considered at any "intermediate" node and either assigned a move or a wait action. Once all agents have been assigned an action, the time step is incremented and a "standard" node is reached. This approach cuts down the branching factor tremendously (from 9^n to 9) while increasing the depth of the tree by a factor of n where n is the number of agents. The implementation guarantees optimality by allowing moves conflicting with agents that have not yet been assigned moves. A heuristic based on single agent distances is built using Reverse Resumable A* (RRA*) [22] or a similar exhaustive search and is used to guide the search for A*. A* with OD cannot solve problems with more than 9 agents on general grids (and even with 9 agents it solved very few problems) [24], despite its improvement over the standard approach. For this reason an Independence Detection (ID) algorithm is used to partition the agents into non-conflicting groups of minimal size. Two groups are said to be non-conflicting if none of the planned paths of either group conflicts with those of the other. The total runtime will naturally be dominated by the size of the largest group. Each time the ID algorithm adds a unit to a group it computes cooperative paths using OD for the new group (since the work will be insignificant compared to that of the same group with one more unit). This approach is both complete and optimal.

It has to be noted that the problem definition used in this paper is different from ours in that simultaneous moves are allowed. In other words, an agent can move into an occupied node if the agent in this occupied node will move out to a different node at the same time.

4.2 Complete Algorithms for Cooperative Pathfinding Problems

Standley's more recent work [25] builds on his previous work [24] and attempts to make Operator Decomposition (OD) applicable to larger problems. It presents two related algorithms: Maximum Group Size (MGS) and Optimal Anytime (OA). The key idea of MGS is to dynamically drop the optimality conditions for both OD and ID whenever the size of an independent group reaches a given constant bounding value, thus speeding up the search but keeping it complete. OA applies the concept of iterative deepening [11] to MGS by calling MGS with increasing group sizes such that whenever the algorithm is terminated, it would have a valid solution from the last iteration ready to be returned. Given sufficient time, OA would eventually reach the optimal solution. The paper shows that on grid of 32x32 with 250 agents, MGS with a bounding value of 1 could solve 94% of the instances on which it was run in under 1 second and provided suboptimal plans for these agents.

4.3 The Increasing Cost Tree Search for Optimal Multi-Agent Pathfinding

Another attempt at optimal solvers was provided in [20] in the form of a data structure, called the Increasing Cost Tree (ICT), that holds path lengths for all agents per node. The root of the tree holds a vector of the optimal single-agent distances between each agent to its destination. The tree grows by adding one step to the plan length of one of the agents and creating a new node for it. If the tree is searched in breadth-first manner, then the first node that has a valid plan is an optimal solution. For each node, a low-level test is performed to find out whether there is a global plan for the agents with the specific path lengths in the node. This lower level test makes use of a data structure called Multi-value Decision Diagram (MDD) [23] to hold all possible paths of a given cost for a given agent. The MDDs are then used to define a search space within which the low level search is performed. Obviously the ICT itself is still exponential in the difference between the heuristic optimal plan length and the real optimal plan length. This approach does not adequately handle congested scenarios where the optimal solution differs greatly from the combined optimal solution lengths for each agent individually, but is a good choice when the difference is small. Independence Detection [24] was employed for this work to allow it to solve larger problems. The paper reported that ICT search was able to solve problems with up to 9 agents in a single independent group. On an 8x8 grid, solutions for 9 agents were found in almost half a minute while on a 3x3 grid, solutions for 8 agents were found in a bit more than a minute. Further pruning techniques were presented in [21] to speed up the runtime of this approach. The paper reports that with pruning techniques problems with 11 agents in a single independent group on a 8x8 grid could be solved in less than a second while on more general large game maps the best reported was 7 agents in about half a minute.

4.4 A Novel Approach to Path Planning for Multiple Robots in Bi-connected Graphs

Surynek's BIBOX [29] is guaranteed to solve multi-agent pathfinding problems in bi-connected graphs in polynomial time ($O(n^3)$) as long as they contain 2 unoccupied nodes. According to [29], any bi-connected graph can be created from an original cycle to which additional loops are added. To solve the problem BIBOX works in two steps. First it solves all agents whose targets are on the additional loops (this excludes the original cycle and the first loop). This leaves a graph shaped like Theta (Θ), composed of the original cycle and the first loop, which is solved in the second step. For each loop, the agents whose targets are on the loop are pushed into the loop in a stack-like manner by rotating agents deeper in the loop whenever a new agent is pushed into it. When the last agent is pushed, all agents on the loop will be at the right positions and the problem will become simpler by having one less loop to worry about. The experimental section of this work shows that BIBOX outperforms a number of general planners and more importantly the algorithm presented in [12] in both runtime and solution quality. The general planners that made it to the empirical section in [29] were SGPLAN 5.1 [2] and LPG-td 1.0 [4]. In this paper, BIBOX was reported to solve instances of problems with 400 nodes in about 4 seconds.

4.5 Making Solutions of Multi-robot Path Planning Problems Shorter Using Weak Transpositions and Critical Path Parallelism

Surynek improved the work on BIBOX [29] by shortening the suboptimal solutions it produces [30]. The Weak Transpositions approach aims to relax the conditions required to re-arrange agents in a cycle with a loop, or more precisely a Theta-like graph, so that instead of keeping all agents constant and swapping only two of them at a time, only a subset of agents is kept in order while the two agents are exchanged. The Critical Path Parallelism method uses a partial ordering predicate to decide whether two moves are dependent or not. This is done using the critical path method, of operations research, where time step "slacks" can be detected and such moves are started earlier. The paper's main concepts are partially based on Ryan's work on subgraph decomposition [19] but with a focus on ring topologies in particular. The paper claims that any graph with one or two holes, depending on the underlying algorithm used, can be represented as a ring (cycle) with loops and is guaranteed to have a solution.

These approaches among others can augment our work by improving solution quality in post-processing steps after we quickly produce a valid solution.

4.6 Exploiting Subgraph Structure in Multi-robot Path Planning

Perhaps the most relevant piece of work that could augment ours is Ryan's subgraph planning [19] in the sense that both could potentially be combined to capitalize on the strengths of each.

Subgraph planning is a centralized approach that is based on the idea of dividing the original graph into disjoint induced subgraphs and then reasoning about them as units while maintaining completeness by ensuring that any abstract plan making use of the subgraphs can be resolved into a concrete plan on the original graph. Central to subgraph planning is the distinction between "arrangements" and "configurations." An arrangement is an exact representation of a complete state of agents within a subgraph. A configuration is a set of "equivalent" arrangements in a subgraph. Two arrangements within a subgraph are equivalent if there exists a plan to move the agents from one of configuration to the other without having any of the agents leave the subgraph during the plan execution.

Search on the abstract level uses two main actions. More precisely an agent can leave or enter a subgraph. These actions are identified by the leaving or entering edge and the agent doing the action. Either of these two actions results in changing the configuration of both subgraphs. The location of the entering edge as well as the current configuration dictates the resulting configuration. For example in a long hallway with multiple entrance points, the number of agents before and after the entrance point used will dictate the final configuration after a new agent enters the subgraph.

The paper presents a number of pre-designated subgraph types. For each subgraph type there are three operations that can be performed: Enter, Exit, and Terminate. For each operation we need: (1) a way to check whether the operation is possible and, if it is, a way for calculating the resulting configuration and (2) a resolution plan that algorithmically defines how to reach the resulting configuration that was calculated. A terminate operation refers to the process that moves all agents in the subgraph to their destinations in the required terminating configuration. The resolution plans are customized for each subgraph type.

If we can build an abstract plan using only the subgraph operations that are valid and we find a solution, then we are guaranteed to have at least one concrete plan to make this solution valid. We are also guaranteed that the search is complete and so there is no possibility of missing a possible concrete solution at the abstract level search. This algorithm is complete. It will also be very efficient whenever the graph topology allow for adequate partitioning. However, finding optimal partitions is hard to do automatically. The choice of the partition affects both the runtime and solution quality. In general, the larger the subgraphs are, the faster the abstract plan can be computed but the lower the quality of the solution will be. Furthermore, the original work uses only a limited number of subgraph types which are easily resolved but does not allow easy partitioning of most practical scenarios. Most notably, none of the subgraph types mentioned in the paper handles free space efficiently. Intuitively, this approach would be well suited to handle mazes, worlds with lots of obstacles, and partially segmented worlds (those without huge free space zones). This solver, like most centralized solvers, does not scale up well and the experiments presented in the paper did not go beyond 20 agents.

The concept of subgraph planning can be used as a framework for TASS where the original graph is decomposed into solvable trees instead of pre-defined subgraph types. Our work provides a quick way to identify whether a given tree is solvable and as such can grow the tree subgraphs incrementally as needed. We also guarantee polynomial solution time within each tree. In effect, the larger the size of individual trees, the faster the overall solution would be found thus allowing the algorithm to scale up much better. The added benefit is that more general graphs can be solved efficiently while guaranteeing completeness. We can easily create tiny trees to fill in gaps left by the larger trees, or we can mix different sizes to balance solution lengths vs computation speed. It is also conceivable that this combination could result in better solution quality or faster execution than using TASS alone, as there is more guidance given to the search within the trees. However, handling the propagation of holes across multiple trees would need to be studied further to verify the validity of this combination.

4.7 Constraint-based Multi-agent Path Planning

This approach [18] is based on Ryan's previous Subgraph Planning work [19]. It uses the same concept of partitioning the original graph into a number of smaller induced subgraphs which are then used in abstract planning. This time the choice of subgraph structures has been restricted to "halls" only where a hall resembles a long corridor with multiple side exits. Apparently, the reason is that the configuration of agents in a hall can be expressed numerically much easier than other types of subgraphs; two agents will always have a strict ordering inside a hall. The key component added to the subgraph planning approach is modeling the problem as a Constraint Satisfaction Problem (CSP). By presenting the problem in mathematical formulation as a CSP problem, it can be solved by a standard CSP solver. This is combined with search at the abstract level to produce an abstract plan which can then be algorithmically resolved into a concrete plan very efficiently by just following a fixed subgraph resolution procedure.

The results presented in the paper were promising. This abstract method has outperformed a priority planner, encoded as a CSP as well but whose details were not mentioned, in terms of memory, run time, and number of instances solved. In the example presented, problems of up to 40 agents were solved in 10 seconds in the median case, in an original graph of 1808 vertices and 3029 edges. It has to be mentioned, however, that the structure of the map itself plays a role in what the subgraph planning approach can achieve. In this case, the map was naturally divided into long roads which were partitioned into 40 halls by hand; producing an abstract graph with only 187 edges. While the paper claims that no optimizations were done on the partition to give the algorithm an advantage, and it is true that most of these halls could have been detected by the automatic partitioning algorithm presented in the previous paper, it still remains a challenge to apply this algorithm on general problems without human intervention.

4.8 Push and Swap: Fast Cooperative Path-Finding with Completeness Guarantees

The approach of Luna and Bekris, Push and Swap [13], attempts to solve multi-agent pathfinding problems on general graphs that have at least two holes. The approach used is sequential and very close to what we use in TASS. Agents "push" their way towards their destinations until they get stuck, at which point they attempt a "swap" using the holes. The paper does not have a separate solvability check, but declares that no solution is possible if a swap or a push operation fails. This still occurs in polynomial time. GTD and TASS have a narrower scope defined by more strict requirements on the topology of the graphs we can handle so far, however, TASS has a lower asymptotic runtime[2] and was orders of magnitude faster in practice, see Section 5.3.2 for more details about

[2]While not explicitly stated in the paper, the runtime of Push and Swap appears to be $O(|L| * m * |J| * n^2)$ where L is the solution length, m is the number of agents, J is the number of junctions, and n is the number of nodes in the graph.

our experiment. The paper reported running the Push and Swap algorithm on graphs with up to 100 agents on graphs of 600 nodes. These were solved in around 180 seconds.

4.9 A Complete and Scalable Strategy for Coordinating Multiple Robots within Roadmaps

This work by Peasgood et al. [15] attempts to solve multi-agent pathfinding problems on roadmaps by finding a spanning tree, with a heuristic which maximizes the number of leaves while minimizing the distance between them, of the network and then solving the problem sequentially on this tree. This is very similar to what our GTD algorithms do. There is a severe restriction here that the number of agents in the system be strictly smaller than the number of leaves on the spanning tree that was created. This restriction means that any reasonably crowded problem would be unsolvable. For those problems that meet the restriction, the solution is then trivially generated by moving all agents to the leaves, swapping agents where needed by using the extra vacant leaf, and then moving the agents back in order of goal depth to their final goal nodes. The paper claimed solving 100 agents in 1.5 seconds. The problems this approach can solve are a strict subset of what TASS + GTD can solve efficiently.

4.10 Tractable Multi-agent Path Planning on Grid Maps

Wang and Botea presented a family of problems called SLIDEABLE in [33] containing problems which are defined by a few strict conditions (see definition 3.5).

The definition of SLIDEABLE is of great importance to my work because it serves as an important step in applying TASS to general graphs in a way that is proven to be solvable when the conditions are met. As more classes of problems are proven to be solvable, I approach my objective of solving the problem on general graphs.

In [33], only the case for 4 directions was handled in the paper. The Multi-Agent Path Planning (MAPP) algorithm works by progressing agents on paths that are precomputed when the algorithm starts. The units will then get assigned priorities and the master unit will traverse its path, if other agents are blocking its path then a “blank” (hole) is moved all the way to the master unit’s next step to allow it to make its move. This will potentially push lower priority units off their optimal paths, so a repositioning step will have them retrace their steps back to their optimal paths to continue the process. This algorithm is complete, but not optimal, and runs in polynomial time when the problem is SLIDEABLE. Otherwise it is not complete in the general case. This work focused on the theoretical guarantees rather than on solving problems quickly or effectively and as such the quality of solutions was typically poor.

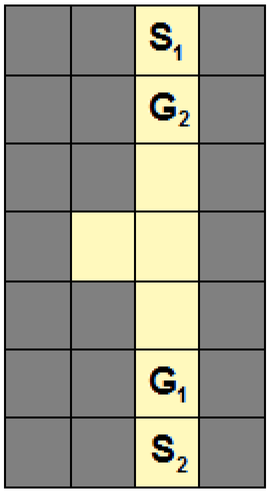

Figure 4.1: A commonly occurring situation where agents need to cross a tunnel that has a single open side gap. S_i represents the start for agent i while G_i is the target for agent i.

4.11 Cooperative Pathfinding

In his paper [22], Silver discusses a number of distributed solutions to the cooperative multi-agent pathfinding problem. He compares his work with Local Repair A* (LRA*) which he mentions is typically used in many practical situations.

4.11.1 Local Repair A* (LRA*)

LRA* is very simple to implement and is probably the most intuitive of all approaches. It simply plans complete paths for each agent from start to end and only recalculates paths when a collision is imminent, thus repairing the global solution locally. LRA* is a quick way to compute a potential solution quickly and works adequately in non-congested areas, however it is not complete and may result in deadlocks where multiple agents keep alternating between a finite number of paths on collision. This approach cannot solve situations requiring real collaboration between agents, such as the corridor with a gap example shown in Figure 4.1. The reason for failing often in examples such as this one is that regardless of the priority of agents, if one of them is to follow a shortest path to the destination, then they may all fail to find a global solution. This algorithm is obviously prone to cycles and deadlocks due to the lack of global coordination.

4.11.2 Cooperative A* (CA*)

The Cooperative A* algorithm is a simple decentralized approach that takes the 2D world space and expands it to a 3D space by adding a time dimension. Each agent then plans its path independently, reserving the spatial-temporal steps used in the path in a reservation table, while avoiding obstacles and previously reserved points in space and time. A wait move in this case is still moving through the 3D space. Naturally, the order of agent selection affects the outcome of the search. This approach is thus neither complete nor optimal and will not work for situations requiring elaborate collaboration. It is also infeasible in practice due to the huge search space resulting from adding the time component.

4.11.3 Hierarchical Cooperative A* (HCA*)

HCA* adds a heuristic function to CA* based on the space component of the search problem (*i.e.,* ignoring the time component and reservation table) to CA*. The heuristic data is built incrementally, for each agent, by running a Reverse Resumable A* (RRA*) [22] search from the goal state and proceeding until it hits the specific node we are interested in, resuming from where it left when we require a heuristic value for a new node for which a heuristic value was never computed. This speeds up search, but still suffers from the same problems of CA* and is mostly impractical. In particular agents that reach their destinations may block the paths of other lower-priority agents permanently.

4.11.4 Windowed Hierarchical Cooperative A* (WHCA*)

The main contribution of Silver is WHCA*. This adds the idea of a window abstraction to HCA*. The agents no longer search for full solutions cooperatively, rather only within a small window around their current positions. To prevent the agents from getting misled into dead ends, the agents plan a complete path but for the segment outside their window they ignore all other agents. This is achieved by considering the window itself as an abstract node in the abstract space without worrying about the details within this window. Every k steps a new window will be used to compute another partial cooperative path. This algorithm is feasible, unlike CA* and HCA*, but is neither complete nor optimal. It can very well lead to deadlocks, although less often than LRA*. Intuition says it would work well in open worlds, segmented worlds, and worlds with scattered obstacles. It can solve problems like the one in Figure 4.1, but only if the window size is large enough. It is expected to perform poorly on maze worlds and road networks where the size of the window will be small compared to the regions where the cooperation is needed. This is exactly where TASS shines since maze worlds are by nature trees. It is conceivable that WHCA* or a similar algorithm can be used beside TASS to solve simpler instances and delegate to TASS the harder problems that are expected to require a lot of computation time. On a 4-connected grid of 32x32 nodes with up to 100 agents and 20% obstacles, the paper claims that up to 98% of the agents could successfully reach their targets with an average of 0.6ms per agent at each invocation of WHCA* until the agents arrive at their targets.

4.12 Improving Collaborative Pathfinding Using Map Abstraction

Sturtevant and Buro [27] presented two ways to improve WHCA* [22] with Partial-Refinement A* (PRA*) [28]. PRA* uses a hierarchy of abstractions that start at the ground level (original graph) and abstract groups of highly connected nodes, such as cliques, into single nodes in the higher abstraction level. This process is repeated a number of times. Whenever a path needs to be found between two points A and B, an abstract search is conducted at a high level between their parent nodes, A' and B' to get an abstract path. This abstract path can then be projected down to lower levels to create a corridor of nodes to which a lower level pathfinding operation between the parents of A and B at this level, A" and B", is restricted. This process is repeated until the ground level is reached, where a simple A* search will be performed. The restriction to the corridors makes the search much faster, but sacrifices optimality. Besides abstraction, PRA* also uses the concept of refinement to incrementally refine paths as needed within a window. The first way this work combines WHCA* and PRA* is by replacing the RRA* heuristics with an abstract space search and so allowing an extra level of control over WHCA* while decreasing the memory requirements. The resulting algorithm is WHCA*(w, a) where w is the window size and a is the abstraction level that will be used to calculate the heuristic values. The second combination is Cooperative Partial-Refinement A* (CPRA*) where PRA*'s ground level A* search is replaced by WHCA* limited by the resulting corridor. WHCA* in this setting will not need to worry about long term plans because the abstract levels guarantee that it will not be misled into dead ends.

4.13 Fast and Memory-Efficient Multi-Agent Pathfinding

This paper by Wang and Botea [32] introduces a distributed multi-agent pathfinding algorithm called Flow Annotation Replanning (FAR) which overlays direction restrictions (flow annotations) on the grid map. It then finds independent solutions for the agents and merges them to form a global plan. This initial overlaying step occurs before the problem instances are solved. The solutions are then generated and cached to be followed at different time steps when possible. The overlay abstraction process maintains connectivity by adding shortcuts to restore it whenever it is lost due to the flow annotations. During execution, local mechanisms are employed to avoid collisions, like alternating between vertical and horizontal flows. A deadlock detection algorithm is run frequently to identify deadlocks and resolve them when they are about to occur by detecting the critical unit (the one causing most trouble) and moving it away from its path temporarily. This approach is very fast but is neither optimal nor complete. It will fail when real cooperation is needed in constrained areas. This method is best applied in large maps with low densities where agents have lots of room to navigate, but the number of choices makes search time for centralized algorithms prohibitive. The authors experimented with a set of huge maps with number of nodes ranging from 13,765 to 51,586. On various maps FAR could solve instances with up to 2,000 agents in less than a minute. In contrast to TASS, FAR is best suited to problems on large graphs with large numbers of agents but few congestion points. These problems are generally the ones where simpler algorithms like LRA* would be acceptable.

4.14 Direction Maps for Cooperative Pathfinding and A New Approach to Cooperative Pathfinding

The idea of direction maps [8] [9] by Jansen and Sturtevant is quite simple: record the direction vectors of agents as they move through tiles and encourage them to follow the directions used before. As more agents go through a tile, its direction vector gets updated to represent the total traffic so far. The direction vector is then used as a negative weight when a search is running. The result is that agents will adhere more to common paths without any direct communication between them. This algorithm is very efficient since it plans for agents independently, but cannot handle complex scenarios requiring real cooperation; as in the case where an agent reaches its destination and blocks other agents causing a global solution to never be found. This is expected to work very well in open space worlds regardless of the density. It should perform reasonably well in scattered obstacle worlds and worlds with few large obstacles as there is a lot of room for manoeuvring and forming virtual lanes on the fly. In this sense it is similar to [32] but with learned annotations. The papers report experiments with up to 50 agents.

4.15 Modeling Collision Avoidance Behavior for Virtual Humans

The paper by Guy et al. [5] focuses mostly on the realistic simulation of crowd movement using reciprocal cooperation between agents that are on collision courses. The emphasis is on adjusting agent speeds and movement angles to achieve smooth transitions. This emphasis puts this paper in a domain that is different from ours. However, a large number of crowd motion and flocking techniques can be used to augment complete planners, like TASS, to make the motion more realistic in open spaces.

4.16 A Cellular Automata Based Algorithm for Agents with Common Goal

This paper by Tavakoli et al. [31] focused on a different scenario, one where all agents share a single goal. This approach attempts to solve the problem by representing the problem in cellular automata formulation. It divides time into discrete steps in increments of 10 or 14 for cardinal and diagonal moves respectively. It then gives the definitions and equations that constitute the system, and then builds a table of distances to goal, the V-scheme, in terms of time steps.[3] It also builds another table with the direction of the optimal path at each cell, the D-scheme. These tables are built using dynamic programming on the table and the paper implicitly claims that this is much more efficient than running A* for each agent's position. It then uses hill climbing to find the paths. An interesting idea introduced in [31] was using the D-scheme to calculate the multi-agent heuristics based on the number of expected collisions if all agents followed their shortest paths.

[3]The V-scheme is of the same size as the map.

4.17 Cooperative Multi-robot Path Planning by Heuristic Priority Adjustment

This work by Regele and Levi [17] uses the idea of adding a time dimension to the spatial search space proposed in [22]. It works by maintaining two maps per agent. The first is a distance map which maintains the distance from each cell to the desired goal, this is similar to the V-Scheme of [31], and performs a function close to what RRA* does for CA*. These distances take into consideration only the static obstacles and ignore the time dimension. The second map is the environment map which includes the time dimension and the obstacles in a small subset of the map. This is similar to the window in WHCA* [22]. The agents use this environment map to encode knowledge about both the static obstacles and the paths of the other agents. The agents start with a priority and take their decisions based on mutual expectations of what other agents would do with minimal communication. This was desirable since this work was meant for the motion of robots. The environment map is the accumulation of all plans of other agents of higher priority. When an agent is planning, it ignores the lower priority agents and merges the plans of the higher ones into its environment map. Using the distance map information the current agent will select the most promising node leading to its target and plan a path for it, then broadcast the updated environment map to the other agents. When conflicts arise, priorities are manipulated to solve the problem. If an agent is blocked,[4] then its priority will be increased. If it is not blocked then its priority gradually decreases. This approach tries to minimize communication, which is not a concern in our case, and presents a reasonable solution to the problem. Of course it is not complete, as agents in a tunnel can still get stuck if the wrong agent claims the right of way first. A randomization of priorities is performed when priorities shoot too high since this means agents are in a deadlock, but this is not guaranteed to quickly solve the deadlocks. This approach would be more suited to worlds with ample free space. Initial priorities can also be assigned to agents based on other attributes like their expected destinations. For highly congested worlds, if the number of viable solutions is small, then the idea of pushing agents around until a solution is found for the higher priority agents may not work at all. The paper reports experiments with up to 100 agents, however it did not report the planning time nor the plan length for the experiments.

4.18 Coordinating Pebble Motion On Graphs, The Diameter Of Permutation Groups, And Applications

The pebble motion problem is equivalent to the multi-agent pathfinding problem with sequential moves where only a single pebble, which is equivalent to an agent, can be moved at each time step. Kornhauser et al. showed in [12] that a lower bound on the number of moves required to solve the

[4]Being blocked is defined as being forced to stay in the same tile for two consecutive time steps. This definition allows it to be pushed back/away by a higher priority agent instead of alternating priorities.

problem is $O(n^3)$. It also shows that the required number of moves is bounded above by $O(n^3)$ as well. While [12] presented an outline for a solving algorithm with no empirical experiments, [29] showed that their own algorithm, BIBOX, performs better in terms of runtime and solution quality.

4.19 A Linear-time Algorithm for the Feasibility of Pebble Motion on Trees

Auletta et al. studied the feasibility of solving specific instances of the pebble motion problem on trees [1]. The paper presents a linear time feasibility check that works by reducing the problem into an equivalent permutations problem which is then further reduced into a set of exchanges (swaps) between pairs of agents that can be verified in $O(n)$. If all pairs of exchanges are verified, then the problem is declared to be feasible. This paper also claims that an algorithm to solve problem instances in $O(n^3)$ can be derived from their work, although such algorithm is not explicitly described in the paper.

4.20 Biased Cost Pathfinding (BCP)

The core idea of the work of Geramifard et al. in [3] is to build an influence map that pushes agents away from specific points on the map. This works by planning paths for agents independently, then identifying points of intersection, in both time and space, and forming a field around such points. This field is then used as a heuristic additive and discourages all units involved, except the one with highest priority, from getting too close to these points. This process is repeated a number of iterations. In each iteration, the conflicts of the previous iteration are factored in to update the influence map. This approach is simple enough and like other prioritized planners, seems to perform efficiently. However, it is obviously neither complete nor optimal. It does seem to be an improvement over LRA* and its solutions can be tuned based on the time available. However, stopping the algorithm before all collisions are avoided does not guarantee a sound solution was necessarily found. Intuition suggests this would work in scenarios where free space is abundant. The empirical section of the paper reports results obtained from testing BCP in a tiny real time strategy scenario.

Chapter 5

Evaluation

I performed a number of experiments to better understand the practical performance of our approach. The purpose of the experiments was two-fold: firstly, I wanted to get a feel of how TASS, on its own as well as combined with GTD algorithms, would perform in practice and secondly, I wanted to compare it with the most notable contending approaches to find out how far did I push the state-of-the-art, if at all. This chapter is thus divided into two sections: one for the independent experiments and the other for comparative experiments. For each experiment when multiple algorithms were compared, all tests were done on the same machine. However, different sets of experiments were conducted on different machines, so comparing the results of one experiment with a different experiment would not be recommended unless the machine specifications mentioned are identical. Whenever time results are mentioned they are assumed to be in milliseconds unless otherwise noted and plan sizes are always measured in total number of steps for all agents.

5.1 Testing Methodology

The experiments on TASS were conducted as follows. I load a scenario[1] file along with its underlying graph file into memory and analyze the files to ensure that they represent a valid configuration of agents. It is at this stage that a GTD algorithm may be optionally executed and the problem instance is tested to see whether it meets our solvability conditions (see Section 3.2). I also collect some basic information about the problem instance. I then start running TASS and time its execution. The GTD algorithms time is not included in the solver time and will be indicated separately when it is used. The reason for the separation is that a GTD algorithm may be executed only once on a graph and the induced tree may be used many times without a need to re-run the GTD algorithm again.[2]

[1]A scenario file is a file that describes the configuration of agents on a particular graph. A scenario file along with its associated graph together fully describe a complete problem instance.

[2]This is only true for GTD algorithms that do not rely on the configuration of agents. GTD-MaxJunction, for example, relies only on the original graph's topology.

Number of Nodes	Binary Trees		Ternary Trees	
	Number of Moves	Time (ms)	Number of Moves	Time (ms)
10	170	1	71	1
100	16,617	64	12,257	51
1,000	556,296	3,026	295,188	2,424
10,000	12,597,322	206,534	5,499,014	164,291

Table 5.1: Summary of running TASS of tree problems of various sizes. The number of agents is always equal to the number of nodes - 4.

5.2 Independent Experiments

The purpose of the independent experiments I conducted was to evaluate in concrete terms how much time TASS takes when solving multi-agent pathfinding problems on trees of various sizes and topologies, as well as how long the plans that it produces are.

Our experiments were primarily performed on binary and ternary trees, as shown in Figures 5.1 and 5.2. These trees have branching factors of 3 and 4 respectively for most of their internal leaves. They grow exponentially in the depth, so they are different from the trees that could be induced from grid maps; however, 4-connected grid maps will have a branching factor similar to the ternary trees, as each state in a grid has at most four neighbors. Table 5.1 shows a quick overview of the results I obtained in some of these experiments.

5.2.1 Scaling Experiments

The first set of experiments were conducted to measure the scalability of TASS. In each experiment trees of increasing sizes were systematically populated with agents with which they are solvable.

The experiments were run on a 2.66 GHz Q8400 processor with 8 GB of RAM.

To make the experiments easily replicable, I have incrementally built binary and ternary trees starting with trees of size 6 all the way up to 1000 in increments of one node. For a tree of size n, there are $m = n - 4$ agents in the tree.

As shown in Figures 5.1 and 5.2, the tree nodes are labeled starting from 0 at the root all the way to TreeSize-1. The sequence of node labels, for consistency, goes level by level; so the first level only has the 0 node which is connected to nodes 1 and 2 on the second level. Then on the third level we have 3 and 4 which are connected to node 1, 5 and 6 are connected to node 2, and so on.

The agents start on nodes TreeSize-1 through node 4 and their destinations span the nodes from 0 to TreeSize-5 respectively. So for an agent i we have $Start_i + Target_i = TreeSize - 1$. For example, in the binary tree shown in Figure 5.1, there would be three agents that start on locations 6, 5 and 4 with respective goals of 0, 1 and 2.

In the results below, for binary and ternary trees, two graphs are presented to show the relationships between the number of nodes and the computation time and between the number of nodes and the solution length.

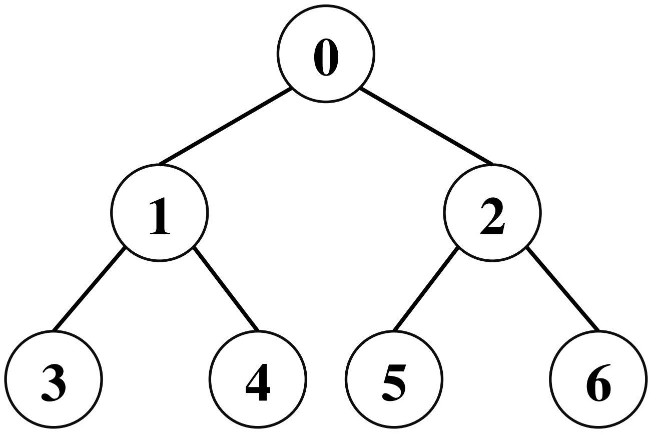

Figure 5.1: Sample of the binary trees used in the experiments.

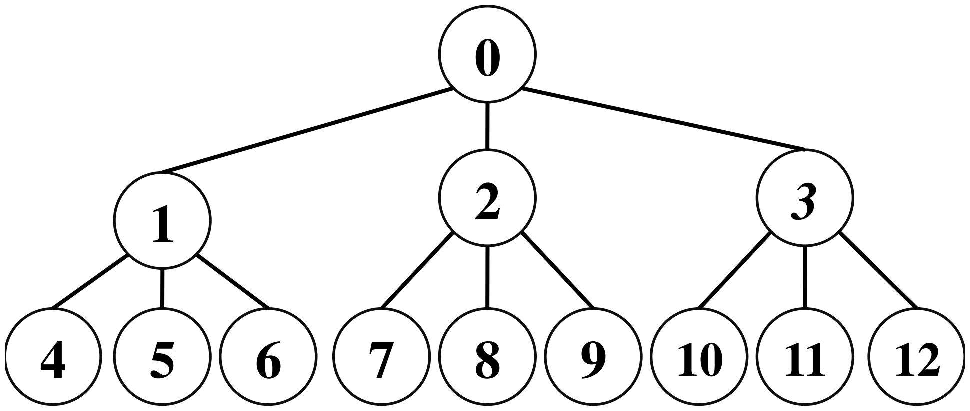

Figure 5.2: Sample of the ternary trees used in the experiments.

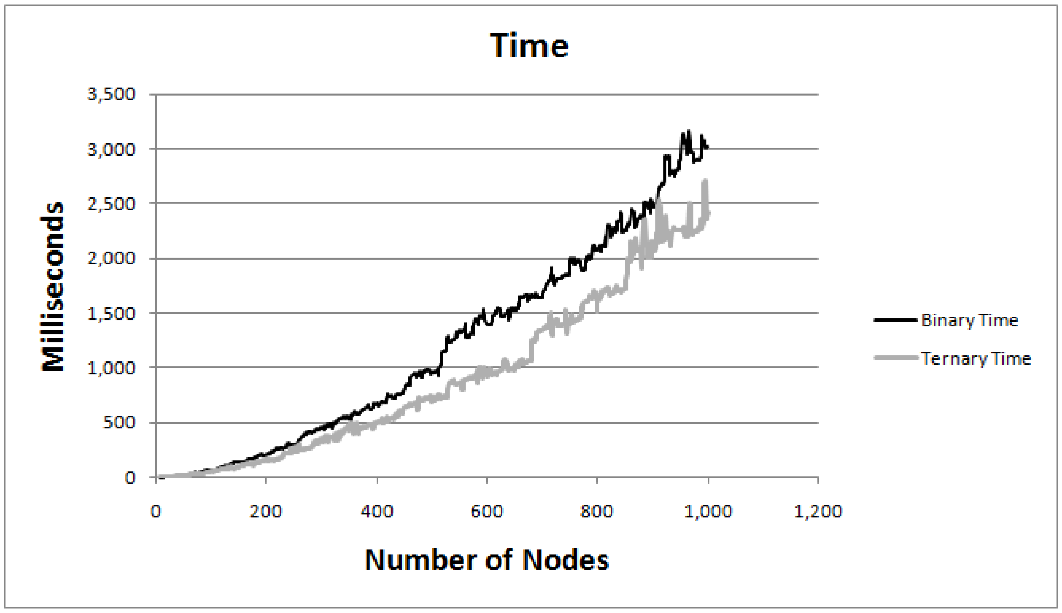

Figure 5.3: Time taken for binary and ternary tree problems.

Figure 5.3 shows the solving time for TASS on binary and ternary trees. For binary trees, I could solve a tree of 513 nodes (509 agents) in less than a second. On the other hand, a ternary tree of 1000 nodes and 996 agents took about 2 seconds and 295,188 moves to solve, which is slightly faster and half the solution length required to solve the 1000-node binary tree. I believe that the solution lengths are shorter for ternary trees because the increased branching factor causes all the nodes, and thus junctions, to be closer together. In Section 5.2.4 I run an experiment to test this conjecture.

It is important to note that these trees are almost completely filled with agents. This make the problems far more difficult by requiring much longer solutions. The solution sizes are, as expected, quite large. These problems are hard in nature and only minor attempts have been made to optimize the solution lengths.[3] In future work, I intend to further improve the suboptimality of the solutions, refer to Section 6.2 for more details.

5.2.2 Plan length distribution

Since it is possible that our experiment problems were particularly easy or hard, I wanted to get a better understanding of the distribution of the solution lengths over a fixed tree size. To do this, I fixed the size of two ternary search trees, one with 14 nodes and the other with 40 nodes. On the smaller tree I ran TASS on all permutations of destinations within the set of nodes $[0, 9]$. On the larger tree, I ran TASS on 6,499,475 problems with 36 of the 40 nodes occupied with the destination set, $[0, 35]$ was shuffled each time. Results are in Figures 5.5 and 5.6. The x-axis is the plan length, while the y-axis is the frequency of paths of that length in thousands. The arrows indicate where our chosen configurations for the scaling experiments lie. As can be seen, our chosen configurations

[3]Most of our optimizations were focused on speeding up the algorithm.

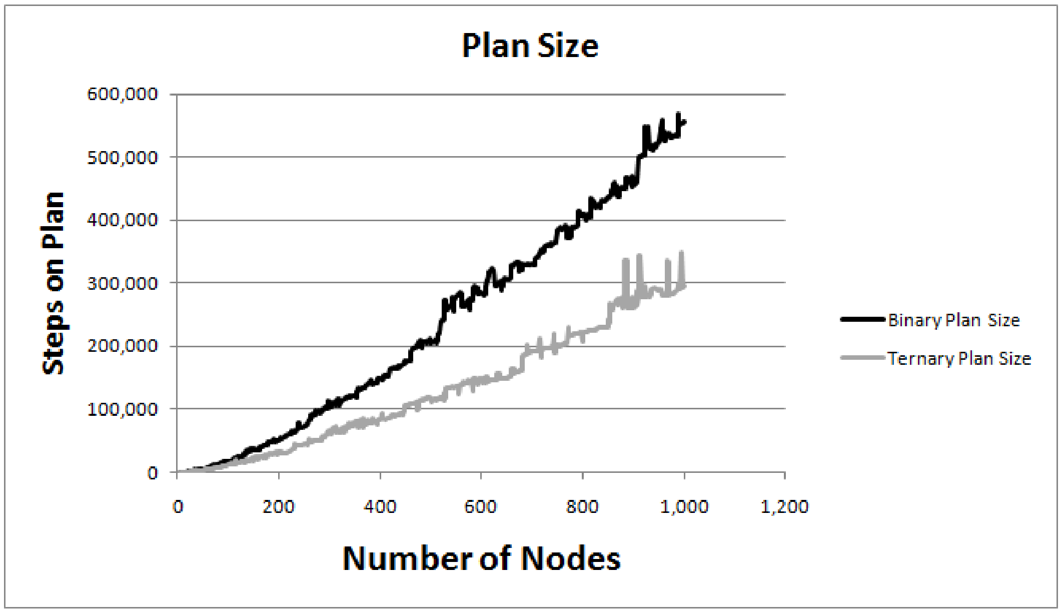

Figure 5.4: Solution lengths for binary and ternary tree problems.

were among the harder instances on both tree sizes with plan lengths among the top 25%.

The distributions appear to be heavy-tailed. It is an open question as to whether this is the result of the optimal solutions being skewed, or the result of how TASS handles different problem instances.

5.2.3 Effects of the proximity and size of Junctions

I have intuitively assumed that having junctions closer to one another as well as junctions of larger sizes would make problems easier to solve. The scalability experiments on binary and ternary trees hinted to this effect. To validate whether this is indeed the case, I conducted a small experiment on trees of the same sizes, the same number of junctions, and the same configuration of agents but with different topologies.

All trees have 18 nodes, 6 junctions and 5 agents. Tree #1, shown in Figure 5.7, has the longest distance between a pair of near junctions with the rest of the junctions on either side of the graph. Tree #2, shown in Figure 5.7, has junctions that are close to one another. Tree #3, shown in Figure 5.7, is similar to Tree #2 except that the junctions are made larger, in this tree every node in the graph is adjacent to a junction.

Our intuition suggested that the trees in the above order would have decreasing plan lengths and this appears to be in line with our findings in this experiment. Running TASS produced plans of lengths 84, 81, and 74 for trees #1, #2, and #3 respectively. It is easy to see that any solution TASS produces on Tree #2 is directly applicable on Tree #3 as well. It is also easy to see that swaps in the middle of the trees would require agents on Tree #1 to travel the most distance to the nearest safe junctions, while swaps near sides can be executed with fewer moves. Due to the small sample size

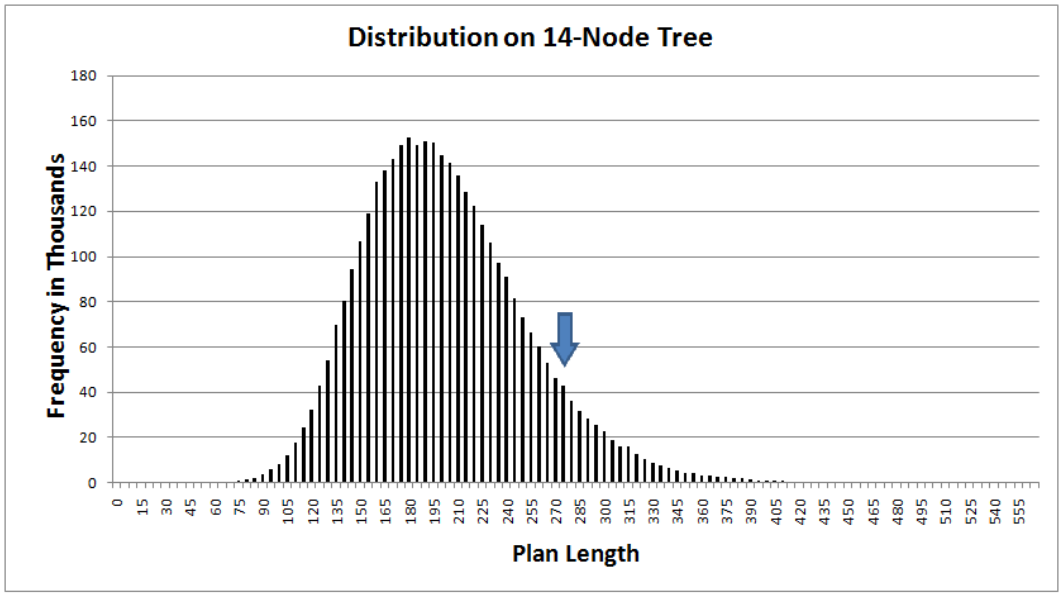

Figure 5.5: Solution length distribution for all permutations within a set of 10 nodes in a 14 node trees. The arrow shows where our configuration for the scaling experiment on this tree size goes.

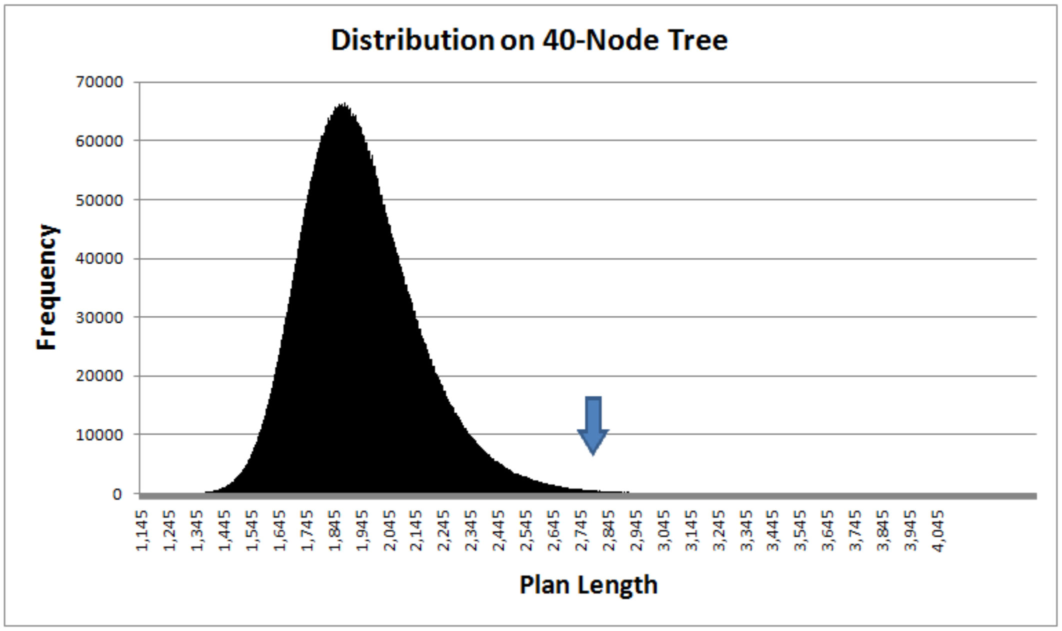

Figure 5.6: Solution length distribution over 6,499,475 random permutations within a set of 36 nodes in a 40 node tree. The arrow shows where our configuration for the scaling experiment on this tree size goes.

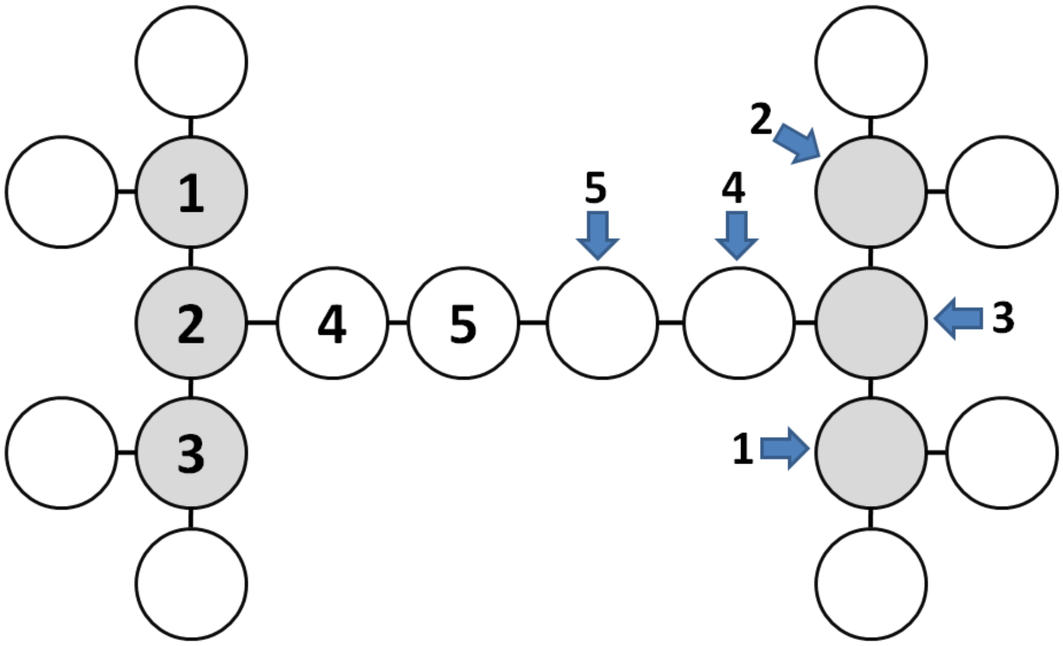

Figure 5.7: Tree #1: The junctions are clustered on either side of the graph.

of this experiment, I cannot generalize its results without further testing.

5.2.4 Congestion Analysis

Theorem 3.3.7 showed that the complexity of TASS on balanced trees is bounded by $O(mlog^2(n))$. I conducted an empirical experiment on a 720QM processor running at 1.6 GHz with 8GB of RAM to verify this result. On a fixed ternary tree of n =10,000 nodes I varied the number of agents (m) from 10 to 9,990 and recorded the solution lengths and times. Figure 5.10 shows the growth of plan lengths as the number of agents increases. The upper bound plot represents the number of moves in the worst case and the lower bound is the total of all distances from all agents to their targets. While the growth does not appear to be polynomial, it is clearly well below the theoretical upper bound which is linear in the number of agents. Keep in mind that this is as far as it could get. To increase the number of agents further the tree itself needs to grow. I am not sure why exactly problems get harder at specific congestion levels, but I have a few observations. The first is that the graph appears to be segmented, rather than being a smooth curve despite the large number of data points. This suggests that there might be different phases of complexity as the graph gets more crowded, the identification of the cause of these phase shifts is a point of future research. I also observed that on running the same experiment on a 1000-node tree, I got a similar curve which shows that the steep growth is a function of the congestion level and not the tree size.

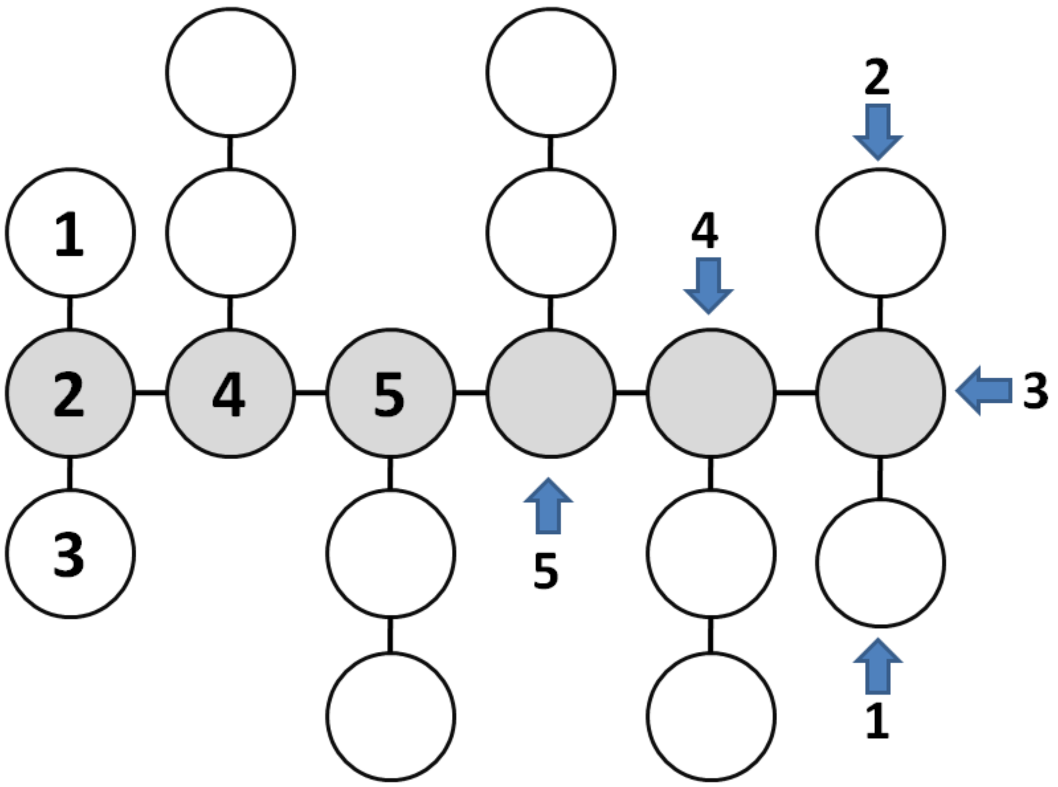

Figure 5.8: Tree #2: The junctions are evenly distributed over the tree.

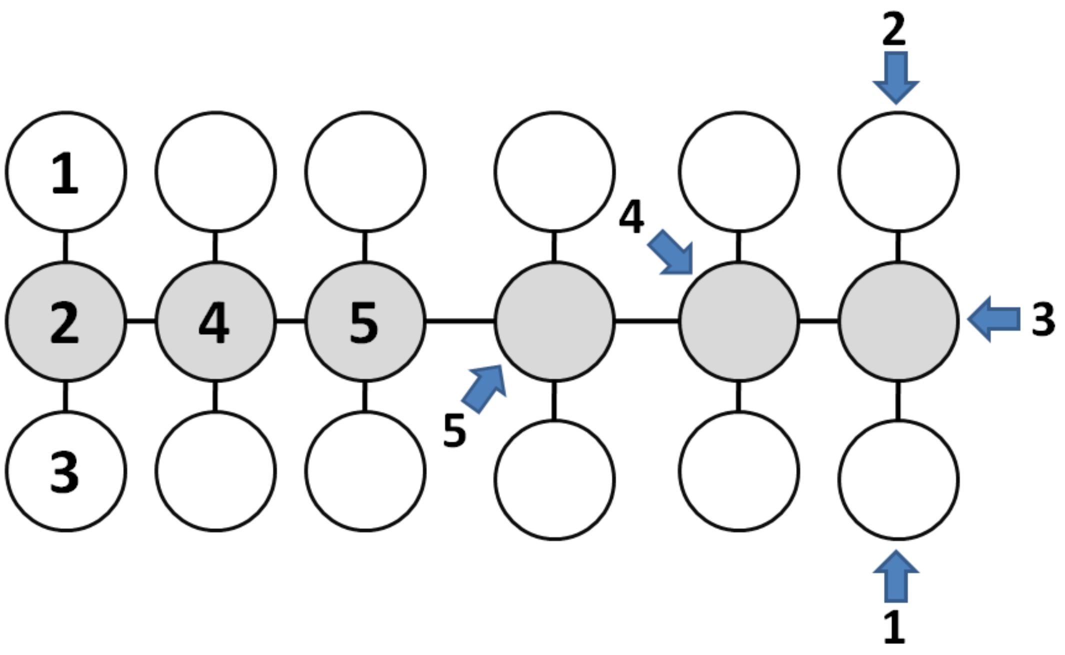

Figure 5.9: Tree #3: The junctions are evenly distributed and are as large as possible. Every node is one step away from its nearest junction.

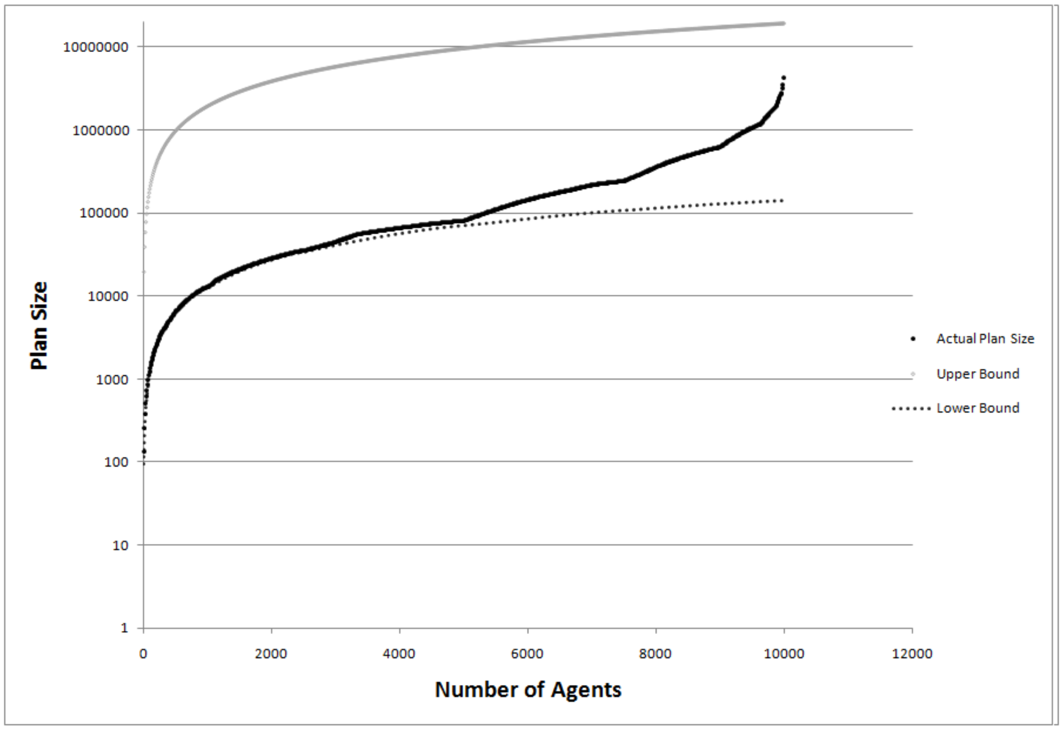

Figure 5.10: Growth of plan lengths as number of agents grow on a fixed-size tree. The graph is in logarithmic scale

5.3 Comparative Experiments

In this section I aim to put my work in perspective with other significant works in the field. While I have relied on the fact that the input graphs were trees in our independent experiments, I will now broaden the scope of the experiments to include graphs that are not trees. The experiments were done by finding common scenarios that can be tested on two or more algorithms at the same time and their runtimes and solution lengths[4] were compared.

5.3.1 Multi-Agent Path Planning (MAPP) Algorithm

I have shown in Section 3.5 that the SLIDEABLE [33] family of problems is a strict subset of the problems that can be solved with TASS with a GTD algorithm. Wang and Botea also presented a polynomial time algorithm, MAPP [33], that could solve SLIDEABLE problems efficiently. MAPP could also provide partial solutions for problems that have some agents whose configuration is SLIDEABLE along with some that are not. It does this by dealing with the non-SLIDEABLE agents as moving obstacles that will not necessarily end up at their desired destinations, but can be moved around so that other agents can reach their destinations.

Wang has provided us with the code for MAPP as well as a dataset of about 2,000 Hierarchical

[4]Solution length, also referred to as plan size, is measured as the total number of moves all agents need to make to solve the problem.

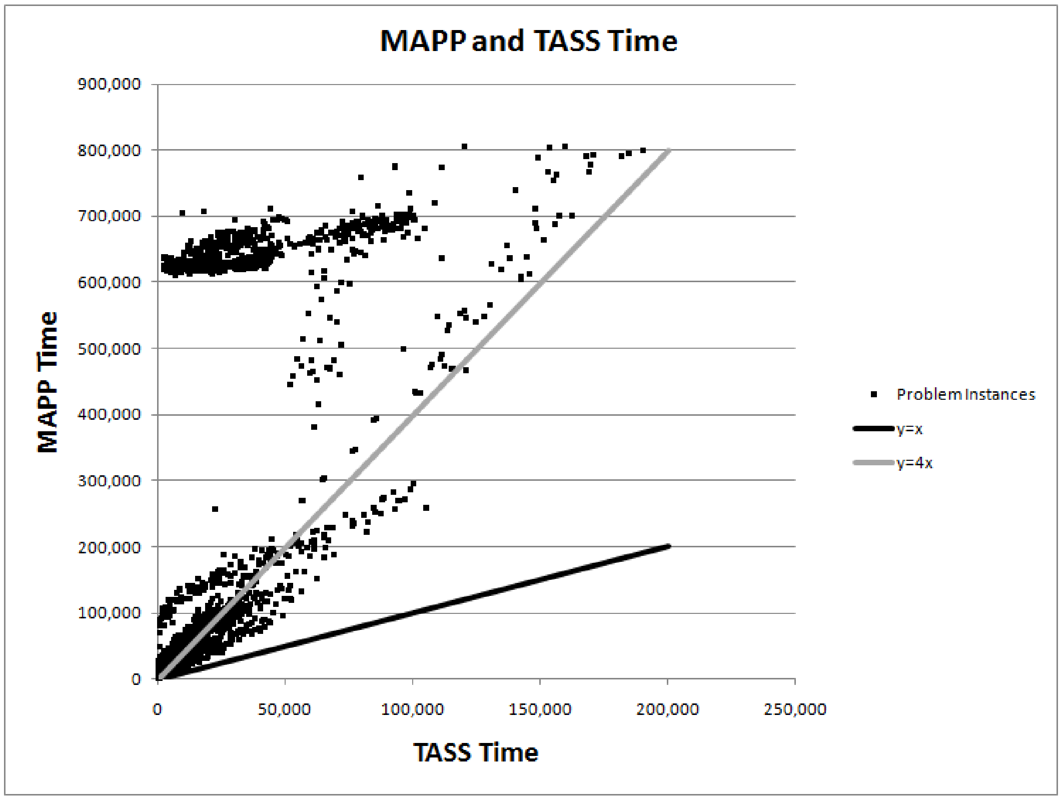

Figure 5.11: The points represent the time each scenario took to run by the two algorithms. Points above the dark y=x line indicate that TASS performed faster while points below the line indicate that MAPP was faster.

Open Graph (HOG) [26] scenarios using 10 different HOG maps with the number of agents ranging from 100 to 2,000 per scenario in increments of 100. For a given number of agents on a given map, 10 random scenarios were tested. Note that these scenarios included a number of agents that were not SLIDEABLE and as mentioned above these agents were dealt with as moving obstacles by MAPP. To run the experiment, the HOG scenarios were converted into a format compatible with the TASS code and then both MAPP and TASS were run on a machine with a 2.66 GHz Q8400 processor and 8 GB of RAM.

MAPP had a success rate of 95% over all agents in all scenarios, SLIDEABLE or not. On the same set of scenarios TASS had a success rate of 100%.

In terms of speed, TASS executed faster than MAPP on all runs as can be seen in Figure 5.11.[5] Despite the fact that both algorithms run in polynomial time, this shows that in practice TASS is a good option when speed is an important factor.

When it comes to solution lengths, though, the results were mixed. In most cases, MAPP produced better solutions than TASS. However, in the few cases, 418 cases to be exact, where TASS

[5]Note that in general the GTD, which took up to 32 seconds in the worst case and 7 on average, would also contribute to the total run time. However, since the GTD-MaxJunction algorithm used is independent of the configuration or scenario, it could easily be encoded with the map once when it was generated and thus its time does not affect the results. In this experiment, it is easy to see that adding the GTD time would not affect the conclusions made.

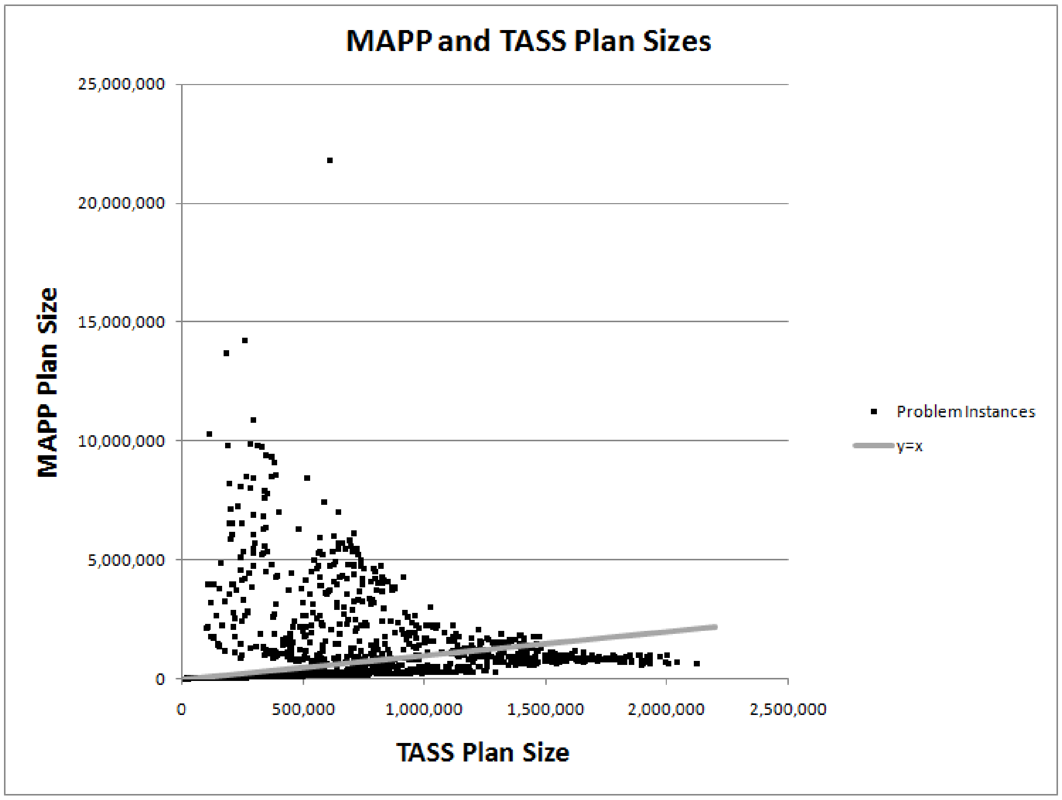

Figure 5.12: The points represent the plan size for each scenario running on either algorithm. Points above the line indicate that TASS produced longer (poorer) plans where points below the line indicate that MAPP produced longer plans.

produced a shorter solution, the difference was more profound as can be seen in Figure 5.12. In fact, when averaging the difference between plan lengths over all the runs, it turns out that TASS produced paths that were 280,226.5 steps shorter than the plans that MAPP produced on average.

5.3.2 Push & Swap

The approach Luna and Bekris presented in [13] has a core similarity to our work. Both approaches rely on agent swaps as a main building block of solutions. For this reason, I felt that this approach would yield results that are comparable to TASS. Push & Swap solves general graphs that have at least 2 holes. It solves the problem by using two basic operations the "push" and the "swap." Each agent tries to push its way to its goal, by pushing the agents on its path, as much as possible until it reaches a point where no further progress can be made by pushing alone in which case it starts swapping positions with the agents on its path.

For this experiment I obtained the original code and test data set from the authors of [13] and added a few additional trees to the list to come up with a set of 32 problems. Table 5.2 lists the scenarios in the combined dataset along with its basic information. The broader domain of Push & Swap clearly meant that it can solve graphs that are not necessarily trees. As such, TASS was not expected to be able to solve the majority of those problems. In fact only 9 of the problems were trees. So for this experiment I had to invoke GTD algorithms in addition to TASS. I tested with GTD-SLIDEABLE and GTD-MaxJunction.[6]

The experiments were conducted on a 720QM processor running at 1.6 GHz with 8GB of RAM. Table 5.3 summarizes the results of these runs. For each problem the best time and plan length is in bold. The number of edges remaining after GTD is executed on a graph is also reported in the two "edges" columns. There have been slight fluctuations of a few milliseconds when experiments were repeated on the same problems multiple times, so small differences in time should be ignored.

The first 26 problems came from the original Push & Swap data set that the authors had shared with us and were used unchanged. The last 6 were among the binary and ternary trees used in the independent TASS experiments. There was a set of problems included in the Push & Swap composed of loops with agents lining up on the nodes forming the loop and need to be rotated around the loop one step each. This set was removed from the experiment since its problems would always be unsolvable when the loops are broken.

From the table it can be seen that for non-tree graphs, Push & Swap found solutions of shorter lengths compared with TASS with either GTD algorithm. This naturally follows from the fact that TASS does not make use of all the available edges in the graph and is thus working on a far more constrained graph. Despite this fact, we can see that TASS + GTD-MaxJunction was able to solve most graph problems.

As problems start growing in size, the speed advantage of TASS becomes more prominent. This is true both on graphs and trees. On the largest graph problem, RandomBig_100, which is a large grid world with 500 nodes and 100 agents placed at random locations, TASS with GTD-MaxJunction solved the problem in less than one tenth of the time Push & Swap took. On trees, TASS in almost all cases produced much shorter solutions and was 4 orders of magnitude faster on the 1000-nodes trees.

The challenge with the large trees was not only the size, but also the high congestion. The trees had only 4 holes and were thus very difficult problems to solve. These problems emphasize the difference between the effective polynomial degrees of both TASS and Push & Swap. What I could solve in less than 4 seconds, took Push & Swap about 33 minutes and the solution TASS produced was a bit more than half the length of that produced by Push & Swap.

These results indicate that for small problems on general graphs, using Push & Swap would be a better option due to its higher solution quality. For larger graphs, where the runtime would become impractical, using TASS + GTD would be a more viable option. When the problem graph is a tree by nature, like mazes, then running TASS without GTD would produce better results in less time.

This experiment also serves as a good comparison between GTD-SLIDEABLE and GTD-MaxJunction.

[6]Note that in our experiments I only implemented a simplified version of GTD-SLIDEABLE where I do not alter the tree if it is not solvable, instead I just declare failure. The reason for this is that this algorithm was devised for theoretical purposes, where in practice GTD-MaxJunction should be used.

In almost all but the smallest of cases GTD-MaxJunction produced shorter paths in less time compared to GTD-SLIDEABLE. This is reasonable given that it maintains a larger tree in most cases and thus provides more room for the agents to maneuver. It also led to an interesting question: "Can I run TASS directly on a graph that is not a tree?" This question makes sense because adding any extra edges to the trees induced by GTD-MaxJunction would necessarily result in a cycle. I decided to answer this question empirically by running TASS, unmodified, on graphs that were not trees to observe how often it would crash or yield incorrect output. I detect wrong output by feeding the output plans into a tracer that validates every move and ensures that applying the plan to a new instance of the problem will end up with all agents at their target nodes as expected.

The result was very interesting. TASS was able to solve all but 4 of the problems in the set. On two it simply crashed, one of which was unsolvable by either GTD algorithm. On two other problems it produced wrong plans. Most interestingly, it was able to correctly solve one problem that neither GTD algorithms were able to handle. The runtime was about half that of GTD-MaxJunction (not even counting the GTD time) and the paths were around one third of the those produced by TASS + GTD-MaxJunction. While we obviously lose completeness guarantees with this approach, it is an interesting direction for future work. It may be possible to adapt TASS to work directly on graphs and detect problem graphs. As it is, however, running TASS on a non-tree graph and then tracing the solution to ensure it is correct is still a viable option.

Scenario	**Number of Agents**	**Number of Nodes**	**Number of Edges**
Tree	3	7	6
Corners	4	12	12
Tunnel	4	9	8
String	5	11	10
LoopChain	7	9	9
Connector	6	18	19
Stacks	16	24	35
RandomBig_10	10	500	788
RandomBig_20	20	500	788
RandomBig_30	30	500	788
RandomBig_40	40	500	788
RandomBig_50	50	500	788
RandomBig_60	60	500	788
RandomBig_70	70	500	788
RandomBig_75	75	500	788
RandomBig_80	80	500	788
RandomBig_90	90	500	788
RandomBig_100	100	500	788
Circle_Rotate_12	12	16	24
Circle_Rotate_16	16	24	36
Circle_Rotate_20	20	32	48
Circle_Rotate_24	24	49	84
Circle_Rotate_48	48	169	312
Circle_Rotate_72	72	361	684
Circle_Rotate_96	96	625	1200
Circle_Swap_16	16	24	36
Binary10_6	6	10	9
Binary100_96	96	100	99
Binary1000_996	996	1000	999
Ternary10_6	6	10	9
Ternary100_96	96	100	99
Ternary1000_996	996	1000	999

Table 5.2: A list of the scenario names along with their basic properties.

Scenario	Push & Swap		GTD-Slideable			GTD-MaxJunction			GTD-None	
	Moves	**Time**	**Moves**	**Time**	**Edges**	**Moves**	**Time**	**Edges**	**Moves**	**Time**
Tree	39	**0.511**	**38**	1	6	**38**	1	6	**38**	1
Corners	**60**	**0.693**	Fail	Fail	8	72	1(3)	11	72	1
Tunnel	145	1.408	**130**	**1**	8	**130**	**1**	8	**130**	**1**
String	**39**	0.358	52	**0**	10	52	**0**	10	52	**0**
LoopChain	**523**	**2.58**	Fail	Fail	6	Fail	Fail	8	Crash	Crash
Connector	**108**	2.837	173	**1(3)**	17	248	2(3)	17	Crash	Crash
Stacks	**296**	10.777	Fail	Fail	16	1032	5(2)	23	597	**4**
RandomBig_10	**224**	108.966	313	5(5)	165	564	**0(0)**	499	235	2
RandomBig_20	**426**	156.879	1279	6(7)	233	1168	16(16)	499	436	**5**
RandomBig_30	**583**	274.869	2155	11(7)	258	1822	15(16)	499	761	**9**
RandomBig_40	**804**	224.639	5130	26(7)	341	2788	16(15)	499	1025	**11**
RandomBig_50	**964**	448.024	6867	33(7)	341	3683	16(0)	499	1322	**15**
RandomBig_60	**1426**	686.013	13200	71(11)	407	5719	31(0)	499	1859	**26**
RandomBig_70	**1522**	576.675	16942	75(7)	408	6985	47(16)	499	1936	**19**
RandomBig_75	**1709**	1367.55	26906	114(9)	397	8083	31(16)	499	2747	**21**
RandomBig_80	**1888**	1286.45	19429	89(12)	402	8836	62(15)	499	3007	**45**
RandomBig_90	**2347**	1248.94	30248	147(10)	438	12030	78(15)	499	3410	**36**
RandomBig_100	**2444**	1922.06	37757	170(11)	434	14243	**109(16)**	499	Wrong	Wrong
Circle_Rotate_12	**48**	**0.318**	Fail	Fail	11	Fail	Fail	15	228	2
Circle_Rotate_16	**34**	0.488	Fail	Fail	15	590	**0(15)**	23	202	1
Circle_Rotate_20	**70**	5.783	Fail	Fail	19	1618	**0(0)**	31	306	2
Circle_Rotate_24	**74**	1.656	Fail	Fail	23	1242	16(0)	48	359	**3**
Circle_Rotate_48	**146**	10.247	Fail	Fail	47	3956	15(0)	168	719	**8**
Circle_Rotate_72	**218**	13.773	Fail	Fail	71	6804	32(0)	360	1151	**8**
Circle_Rotate_96	**290**	30.59	Fail	Fail	95	10264	62(0)	624	1655	**15**
Circle_Swap_16	**200**	16.082	2492	16	23	1262	**0(16)**	23	Wrong	Wrong
Binary10_6	237	**2.317**	**170**	4	9	**170**	4	9	**170**	4
Binary100_96	31572	2713.32	**16617**	**88**	99	**16617**	**88**	99	**16617**	**88**
Binary1000_996	925747	1975180	**556296**	**3520**	999	**556296**	**3520**	999	**556296**	**3520**
Ternary10_6	134	1.167	**77**	**1**	9	**77**	**1**	9	**77**	**1**
Ternary100_96	16220	1713.73	**12257**	**68**	99	**12257**	**68**	99	**12257**	**68**
Ternary1000_996	490205	1627140	**295188**	**3437**	999	**295188**	**3437**	999	**295188**	**3437**

Table 5.3: Push & Swap vs TASS. Time in parenthesis is GTD time. Time is in ms. Best results are bolded.

	BIBOX	TASS
Average Time	**21,944 ms**	29,562 ms
Average Plan Length	471,056,960	**29,571,558**
Median Plan Length	515,411,465	**9,180,781.5**
Instances Faster	**293**	175
Instances Shorter	24	**444**

Table 5.4: Aggregate statistics for running 468 problem instances on BIBOX and TASS. Best results are in bold.

5.3.3 BIBOX

Surynek's algorithm BIBOX [29] was designed to be a complete algorithm that solves multi-agent pathfinding problems on bi-connected graphs in polynomial time. While bi-connected graphs exclude trees by definition, the polynomial complexity of BIBOX makes it a good candidate to compare with TASS. Obviously TASS had to be coupled with a GTD algorithm, GTD-MaxJunction in this case, to be able to solve the problems on bi-connected graphs. Surynek has shared code that generates bi-connected graphs by creating one main cycle and adding loops[7] to it incrementally. The size of the individual loops was selected at random bounded by a parameter that I varied from 10 to 100. Similarly, I varied the size of the original cycle from 10 to 100 nodes. And finally I varied the number of loops from 10 to 100 as well. For the sake of time, resulting graphs with more than 1,500 nodes were removed from the experiment and I was left with 468 problem instances with randomly placed agents. The number of agents varied from 44 to 1,378.

The experiment showed that BIBOX performed slightly faster than TASS but the quality of the solutions TASS produced was higher on average. The reason BIBOX is usually faster could be attributed to the fact that it works on the original graph which has more edges while TASS works on the induced tree. This is a limitation of TASS that I hope will be addressed in future work. Table 5.4 summarizes the differences between both algorithms in terms of time and solution lengths.

Figure 5.13 shows a scatter graph of the runtime of each problem instance on both algorithms, the higher number of points above the line show that BIBOX performs faster than TASS by a small margin. On the other hand, Figure 5.14 shows the solution lengths obtained by both algorithms for each problem instance. It is clear from the graph that not only did TASS produce shorter solutions more frequently, but also the magnitude of the difference between the solution lengths of the algorithms was greatly in TASS's favor.

5.3.4 Suboptimality Experiment

Measuring the speed of TASS gives us good insights about its practical applicability, but speed was only half of the question. The second half was how suboptimal were our solutions? Since traditional optimal algorithms were too slow to run on even the small trees, I decided to use the Standley's

[7]A loop in this context refers to a chain of nodes that connect two nodes on the original main cycle.

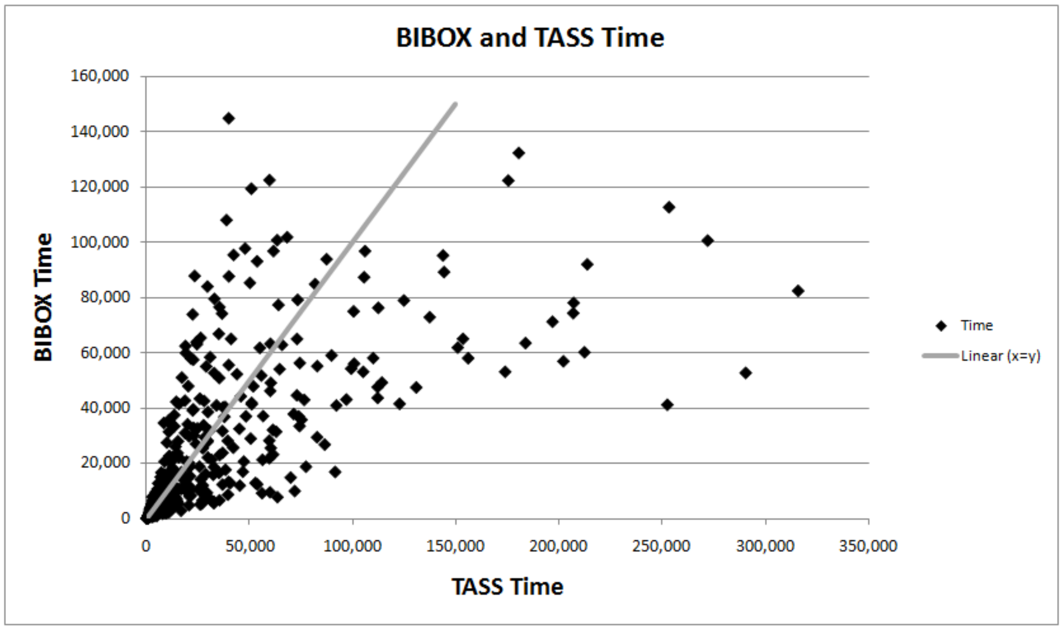

Figure 5.13: Execution times of BIBOX and TASS on the same problem instances.

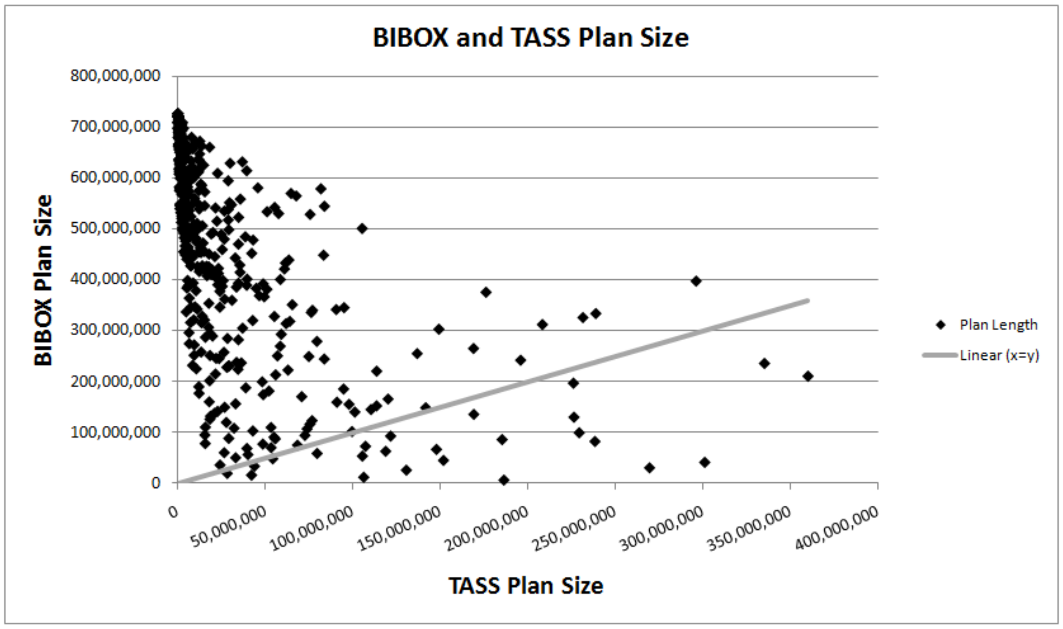

Figure 5.14: Plan sizes of BIBOX and TASS on the same problem instances.

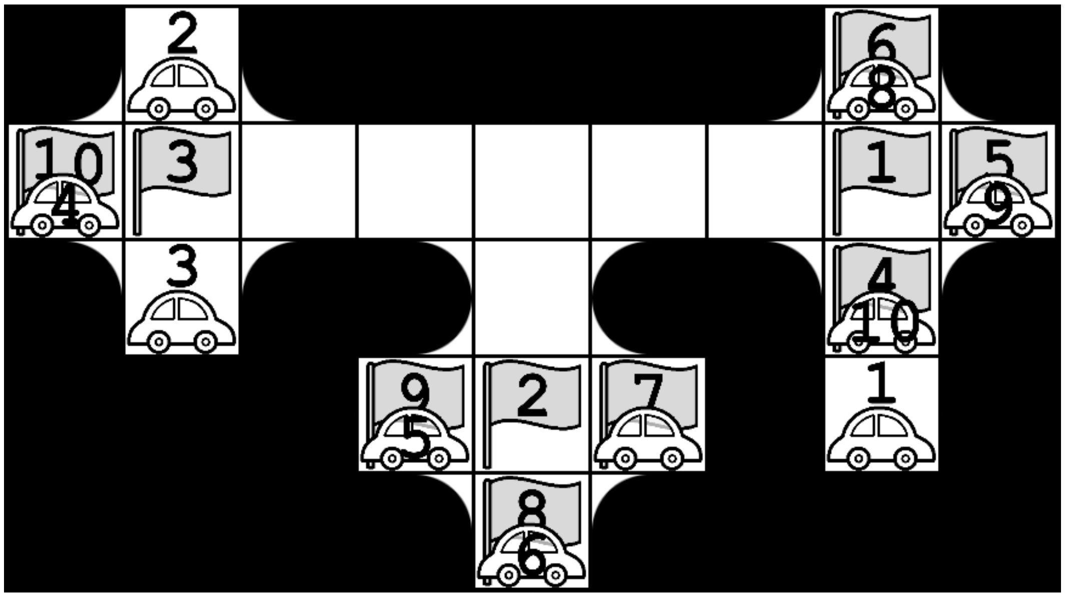

Figure 5.15: A 19-node maze that resembles a ternary tree. The car icons represent the agents and the numbered flags represent their targets.

Operator Decomposition (OD) [24] algorithm on small problems and calculate the suboptimality of TASS based on this.

The first problem I attempted was a small maze adapted from our 14-node ternary trees by adding a few extra nodes to be representable on a grid. I followed the same agent configuration convention as that used for the independent experiments and ended up with the configuration shown in Figure 5.15.

TASS solved this problem instance in 2 milliseconds and came up with a solution of length 459 moves. OD initially timed out on this experiment using the default settings, I then increased the time out period and it quickly ran out of memory. I believe that the reason for this failure is that this type of configuration makes all agents' paths overlap and thus prevents the independence detection algorithm from partitioning the problem into smaller easier subproblems. An independent experiment confirmed the difficulty of finding the optimal solution for this problem. Sharon reported that Standley's A*+OD+ID exhausted all the available memory after 20 hours while ICTS [20] was unable to solve the problem within 48 hours.[8]

I then attempted to solve a binary tree of size 15 nodes and were able to get a solution of length 234 moves in 1 millisecond using TASS. The representation for this problem on a grid did not require any additional nodes and can be seen in Figure 5.16. OD took hours on this problem before running out of memory. I was thus unable to obtain a good quantitative measure for the suboptimality of TASS on non-trivial problems. Further reducing the problem to a tree of 10 nodes as shown in Figure 5.17, OD still ran out of memory.

[8]Personal communication with Guni Sharon.

Nodes	OD		TASS		
	Moves	Time (ms)	Moves	Time (ms)	Suboptimality
7	18	7	30	1	167%
8	18	62	34	1	189%
9	32	131,358	58	1	181%
10	-	Memory Out	170	1	-

Table 5.5: Summary of running TASS and OD on tree problems of various sizes.

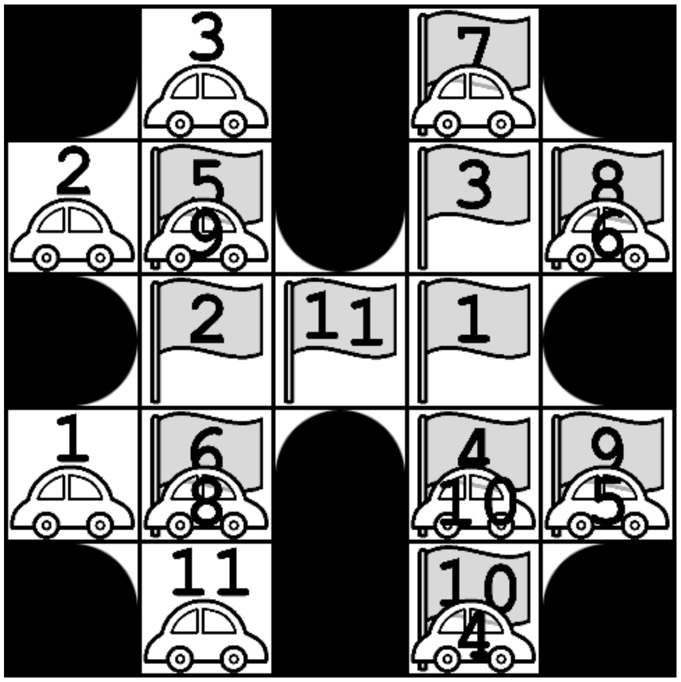

Figure 5.16: A grid representation of a 15-node binary tree. The car icons represent the agents and the numbered flags represent their targets.

Only when I scaled down to binary trees with 9 nodes or less were I able to compute optimal solutions using OD. On the largest problem OD could solve, having 9 nodes, shown in Figure 5.18, OD took 131,357.76 milliseconds to run as opposed to 1 millisecond with TASS, but was able to produce an optimal solution of 32 moves where TASS produced a solution of length 58. Table 5.5 shows a summary of the results on grids representing binary trees of sizes 7, 8, 9, and 10. Figures 5.19 and 5.20 show the 7-nodes and 8-nodes grids respectively. Note that the grids were constructed using the same methodology and agent ordering used in the independent experiments, refer to Section 5.2.1.

It is hard to generalize from the limited results I obtained in this experiment. However, we can see that TASS was always within a factor of 2 of the optimal length in this experiment, while its runtime was negligible. In contrast, we can clearly see the exponential time growth of the optimal solver even in such small grids. It is this exponential growth that makes suboptimal solutions much more practical in many of the real life problems we face today.

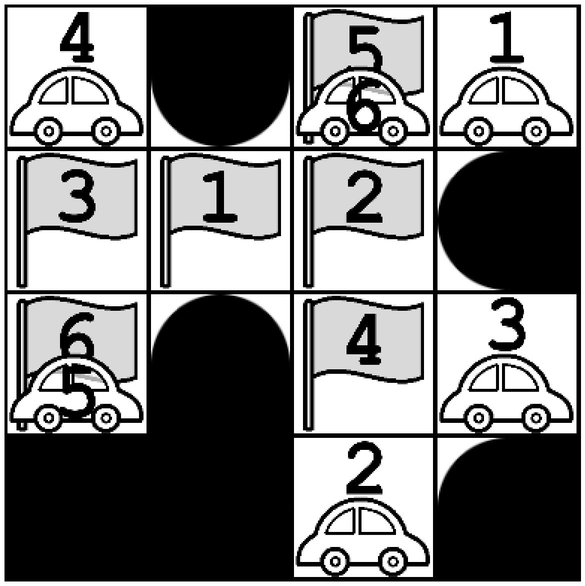

Figure 5.17: A grid representation of a 10-node binary tree. The car icons represent the agents and the numbered flags represent their targets.

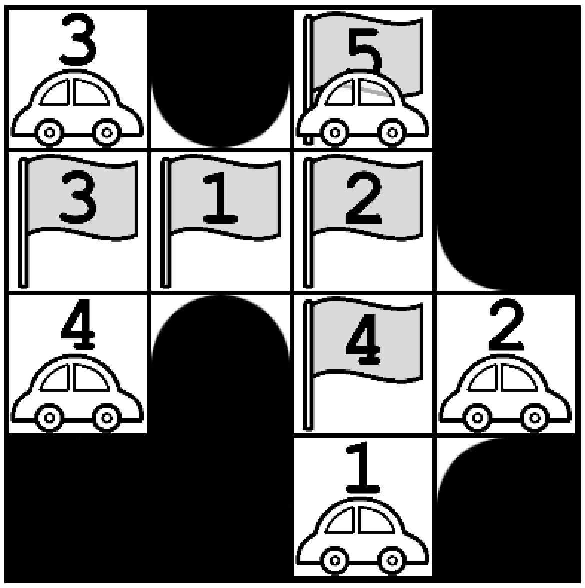

Figure 5.18: A grid representation of a 9-node binary tree. The car icons represent the agents and the numbered flags represent their targets.

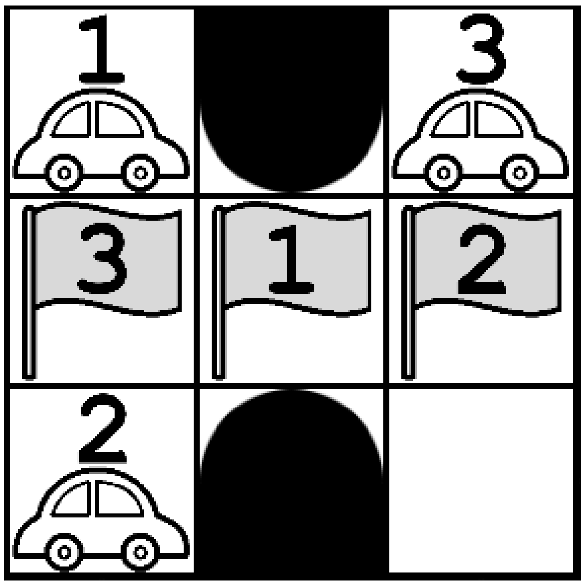

Figure 5.19: A grid representation of a 7-node binary tree. The car icons represent the agents and the numbered flags represent their targets.

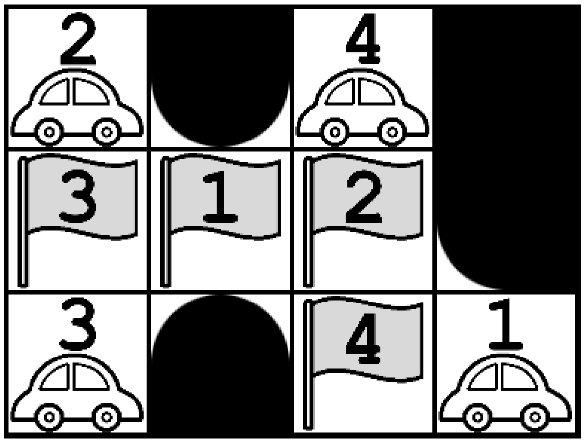

Figure 5.20: A grid representation of a 8-node binary tree. The car icons represent the agents and the numbered flags represent their targets.

Chapter 6

Conclusions

In this work I presented a new approach to solving multi-agent pathfinding problems on a subset of general graphs. Our approach relies on two components: a Tree-based Agent Swapping Strategy (TASS) and a Graph-to-Tree Decomposition (GTD) family of algorithms. Running a GTD algorithm on a general graph induces a tree on which TASS can very efficiently find solutions for multi-agent pathfinding problems. The solutions obtained on the tree are valid on the graphs because no edges are ever added to the tree that did not belong to the original graph.

The key limiting factor to the applicability of TASS is the fact that not all graphs remain solvable after being converted to trees. For this reason I also presented some solvability conditions adapted from the work of Masehian and Nejad [14] to quickly identify whether a certain tree can be solved for a given number of agents. To take our work further in terms of general application, I proved analytically, and empirically, that the family of problems on general graphs meeting the SLIDEABLE conditions defined in [33] are completely solvable using TASS with a GTD algorithm.

Finally, I conducted a number of experiments that verified that TASS was a time-efficient algorithm that could handle highly congested problems of up to 10,000 nodes in less than 3 minutes and 1,000 nodes in less than 3 seconds. A number of comparative experiments were also conducted to compare TASS with current state-of-the-art algorithms. The results were very favorable and showed that TASS was more efficient than most existing algorithms, especially on trees and maze-like graphs, and produced solutions of comparable quality.

6.1 Limitations

One of the most notable topologies that our GTD algorithms fail to handle are loop graphs. There is typically no possible way to break a loop into a solvable tree, unless there is only a single agent. This is because breaking a loop creates a long corridor where no agents can swap. It may be possible to have instance solvability on loops or tunnels, which our approach will fail to detect. These cases are, however, trivial to handle as special cases, because no agents ever swap position. If a problem is solvable it would be possible to directly move agents to their targets by sweeping from one end of the tunnel or any goal node in a loop then moving each agent in turn to its goal node.

Another limitation is that our definition excludes simultaneous moves and thus may fail to solve a range or problems where the only viable solutions require agents to move at the same time. Updating TASS to work with simultaneous agents is a future direction.

Finally, all of the optimizations performed during the course of this work were aimed at improving the runtime of the algorithm but not the solution quality. It is natural, therefore, that the next aspect to optimize would be the quality of the produced solutions. This could either be achieved by refining TASS itself, by applying post-processing to bring down the solution length possibly by executing moves in parallel, or by augmenting TASS with more optimal solvers for the smaller instances as mentioned earlier.

6.2 Future Work

Currently TASS performs very efficiently on trees in terms of runtime. For example I could solve problems with ten thousand agents in less than three minutes. However, when coupled with GTD algorithms, we usually lose a lot of maneuvering space when we induce a tree from a graph. This typically results in poor solution quality, especially in easy problems like those consisting mostly of open space. In this section I discuss ideas that can potentially improve the quality of the solutions we get as well as to speed up the computation even further.

6.2.1 Selectively Repositioning Agents

Once TASS has performed a swap, it moves back all agents disturbed during the operation back to their original positions. It is, however, only necessary to restore agents that were already at their target nodes and, perhaps, those that would block them from returning to their original positions. I believe this would further reduce the plan sizes generated by TASS.

6.2.2 Breaking Cycles Intelligently

When breaking cycles the best GTD algorithm so far, GTD-MaxJunction, orders edges by a priority value based on the degrees of their two nodes. This maintains desirable properties in the trees, but this can be further improved by intelligently deciding among edges of equal priority. One idea that is worth testing is breaking edges that are furthest from junctions first. This would potentially reduce the distances to the newly created leaves and the closest junctions.

6.2.3 Target Ordering

For some simple problems on trees that are not completely solvable, we can still solve specific instances if we sort the target nodes such that those that are unoccupied in the initial configuration are solved first. This allows us for example to solve problems like loops where agents need to rotate in some manner, or tunnels where the agents need to all move in one direction without swapping.

6.2.4 Agent Sorting

When running TASS it may be possible to swap agents more times than necessary. In particular an agent that reaches its destination may be disturbed many times as other agents move around. One way to reduce this is to sort agents by their target nodes. Those with targets on the leaves are solved first, and then we move inwards. A quick experiment showed that this indeed improves both runtime and plan sizes significantly. In particular on binary trees agent sorting brought down the plan size of a 1000-node tree from 556,296 to 508,948 moves and the time from 3,026 ms to 2,029 ms. Similar results were obtained on a 1000-node ternary tree where the plan size went down from 295,188 moves to 254,116 moves and the time from 2,424 ms to 1,583 ms. This optimization is in our final implementation, but due to time constraints was not used in our experiments.

6.2.5 Agents Push

When an agent encounters another agent on its path, we should first check whether this blocking agent is at its target node or not. If it is not, then we could attempt to push it instead of swapping with it. Of course, for the push to be valid, none of the agents on the whole chain of agents that will be pushed should be at its final target. This idea is inspired directly from the work of Luna and Bekris [13]. Figure 6.1 shows an example where this would produce shorter solutions, to move agent u to its target we can just push the chain of agents one step to the right each, then move u to its target directly. Without pushing agents, we will need to swap agents u and 1 together which involves pushing the same chain in the same way, then moving u to its target and out again, then moving agent 1 into u's original position to complete the swap before finally moving u to its final target node once more. Similarly, if we decide to move 2 to its target next, then we will have to disturb u again to swap 1 and 2 twice, where just pushing 1 to the left would quickly put both agents on their target nodes.

6.2.6 Multiple Decompositions

This concept was hinted at in our discussion but was not implemented in the algorithm. When a graph is decomposed, there is no need to come up with a single tree. In fact, it is probably more effective to break it down into separate isolated trees and solve smaller problems on these trees. The trees would only include agents whose paths overlap. GTD algorithms that act on the whole map can still be effectively used to generate a single main tree, but have slight variants of it that do not include the nodes that are occupied by agents from other subsets. This means we would still have multiple instances of trees containing different subsets of agents but share most of the nodes. We can then solve each subset in isolation, without risk of colliding with agents from other subsets. Care has to be taken to ensure that all the subtrees remain solvable and if they are not, they should be merged incrementally until they become solvable. This is expected to improve solution quality when agents are close to their destinations in the original graph but far in the induced tree.

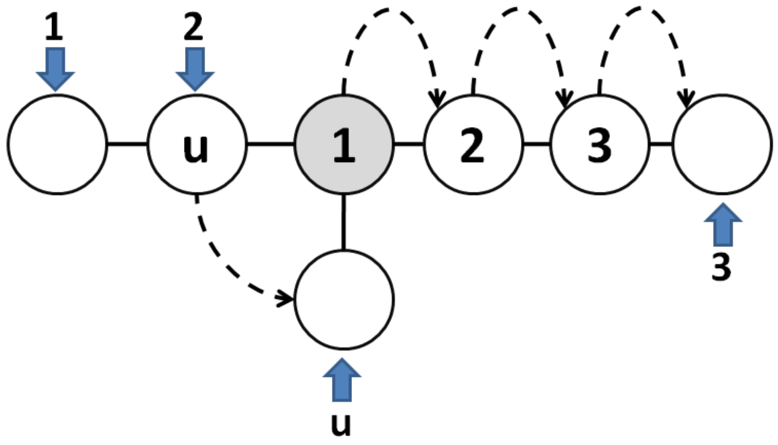

Figure 6.1: An example problem where pushing a chain of agents would require less work than doing full swaps. I assume this is part of a larger tree with enough holes to be solvable, but the rest is irrelevant.

6.2.7 Run TASS on General Graphs

During our experiments (see Section 5.3.2), I noticed that running TASS on graphs directly without converting them to trees first worked on the majority of the cases and produced better results than with a GTD algorithm. On the few (4 out of 32) where it failed, it either produced wrong results or crashed. Noting that this version of TASS had the assumption that it was running on trees, it may be possible to identify the cases where it fails and refine TASS so that it runs on graphs directly. Another possible extension to make use of this would be to run TASS on graphs first and then check the results as part of the algorithm for correctness. The total time for producing a solution and checking it would still be smaller than that of running TASS with a GTD algorithm, mainly because checking correctness is a very quick linear time operation in the number of moves.

6.2.8 Shadow Graphs

Another alternative to running TASS unmodified on general graphs is using *Shadow Graphs*. Consider a graph that represents a grid of open space where each node is connected to its four adjacent neighbors. It is clear that when this kind of graph is decomposed into a tree a large number of edges will be cut off which would cause simple problems to require convoluted solutions. Using what I call “shadow graphs” we can work around this limitation. Here is a brief overview of how this system would work.

I define a “First Complement Graph” as a graph containing all the nodes of the original graph and all edges that are not already on the induced tree. On this I take a subset, call it the “First Shadow Graph,” that keeps only sufficient edges to maintain connectivity for all the nodes on the induced tree. Connectivity is maintained by ensuring that paths exist between any pair of points but without using any edges or nodes of the induced tree, except for the pair of nodes at the two ends of each path. I also define a "Second Complement Graph" similarly to the first, but excluding all edges in the "First Shadow Graph" and I similarly create a "Second Shadow Graph" that does not use any nodes in the first shadow graphs (except for the nodes of the induced tree that it connects).

Now whenever we want to do a swap[1] between two nodes we have one of three situations:

1. There is no direct path between the two nodes in any shadow graphs. In this case we use the original TASS.

2. There is a path in the first shadow graph between the two nodes, but not in the second. One agent moves as dictated by TASS, while the other takes a shortcut through the first shadow graph. This reduces the solution length to a about half (considering that a direct path is usually insignificant compared with a busy path through the tree).

3. There are paths in both shadow graphs. In this case both agents can swap directly via both shadow graphs without any internal swaps. This almost eliminates the complexity of this one swap.

This is one of the most promising future ideas for our work as it is expected to significantly cut down the solution lengths of the plans TASS produces.

6.3 Closing Remarks

TASS was developed to be applicable in practice and for this reason its performance was our top priority. However, with our focus on speed, we sacrificed solution quality. This allowed us to solve huge problem sizes that most existing algorithms would fail to handle, making TASS ideal for hard problems but a poor choice for simpler ones. When thinking of realistically applying our work in practice, I believe that the best way would be to integrate it as a part of a larger framework that uses slower but more path size-efficient algorithms for small problems and delegate the hard ones to TASS to get the best of both worlds.

[1]In this case the agents do not need to be adjacent.

Bibliography

[1] V. Auletta, A. Monti, M. Parente, and P. Persiano. A linear-time algorithm for the feasibility of pebble motion on trees. *Algorithmica*, 23(3):223–245, 1999.

[2] Y. Chen, C.W. Hsu, and B.W. Wah. Sgplan: Subgoal partitioning and resolution in planning. *Edelkamp et al.(Edelkamp, Hoffmann, Littman, & Younes, 2004)*, 2004.

[3] Alborz Geramifard, Pirooz Chubak, and Vadim Bulitko. Biased cost pathfinding. In *AIIDE*, pages 112–114, 2006.

[4] A. Gerevini and I. Serina. Homepage of lpg. http://zeus.ing.unibs.it/lpg/, 2008.

[5] S.J. Guy, M.C. Lin, and D. Manocha. Modeling collision avoidance behavior for virtual humans. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 2-Volume 2*, pages 575–582. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[6] R. Hayes. The Sam Loyd 15-puzzle. *Puzzle*, 2001:06, 2001.

[7] J.E. Hopcroft, J.T. Schwartz, and M. Sharir. On the complexity of motion planning for multiple independent objects; pspace-hardness of the" warehouseman's problem". *The International Journal of Robotics Research*, 3(4):76, 1984.

[8] M.R. Jansen and N.R. Sturtevant. Direction maps for cooperative pathfinding. In *The Fourth Arftificial Intelligence and Interactive Digital Entertainment Conference (AIIDE poster)*, 2008.

[9] R. Jansen and N. Sturtevant. A new approach to cooperative pathfinding. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 3*, pages 1401–1404. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[10] Mokhtar M. Khorshid, Robert C. Holte, and Nathan R. Sturtevant. A polynomial-time algorithm for non-optimal multi-agent pathfinding. In *SOCS*, 2011.

[11] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[12] D. Kornhauser, G. Miller, and P. Spirakis. Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. *Annual IEEE Symposium on Foundations of Computer Science*, 0:241–250, 1984. doi: http://doi.ieeecomputersociety.org/10.1109/SFCS.1984.715921.

[13] Ryan Luna and Kostas E. Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In *IJCAI*, pages 294–300, 2011.

[14] Ellips Masehian and Azadeh Hassan Nejad. Solvability of multi robot motion planning problems on trees. In *IROS*, pages 5936–5941, 2009.

[15] M. Peasgood, C.M. Clark, and J. McPhee. A complete and scalable strategy for coordinating multiple robots within roadmaps. *IEEE Transactions on Robotics*, 24(2):283–292, 2008.

[16] D. Ratner and M. K. Warmuth. Finding a shortest solution for the N $\times$ N extension of the 15-puzzle is intractable. In *National Conference on Artificial Intelligence (AAAI-86)*, pages 168–172, 1986.

[17] R. Regele and P. Levi. Cooperative multi-robot path planning by heuristic priority adjustment. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5954–5959. IEEE, 2006.

[18] Malcolm Ryan. Constraint-based multi-robot path planning. In *ICRA*, pages 922–928, 2010.

[19] Malcolm R. K. Ryan. Exploiting subgraph structure in multi-robot path planning. *Journal of Artificial Intelligence Research*, 31:497–542, March 2008. ISSN 1076-9757. URL http://portal.acm.org/citation.cfm?id=1622655.1622670.

[20] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. pages 662–667, 2011.

[21] Guni Sharon, Roni Tzvi Stern, Meir Goldenberg, and Ariel Felner. Pruning techniques for the increasing cost tree search for optimal multi-agent pathfinding. In *SOCS*, 2011.

[22] David Silver. Cooperative pathfinding. In *AIIDE*, pages 117–122, 2005.

[23] A. Srinivasan, T. Ham, S. Malik, and R.K. Brayton. Algorithms for discrete function manipulation. In *IEEE International Conference on Computer-Aided Design. ICCAD-90. Digest of Technical Papers.*, pages 92–95. IEEE, 1990.

[24] Trevor Scott Standley. Finding optimal solutions to cooperative pathfinding problems. In *AAAI*, 2010.

[25] Trevor Scott Standley and Richard E. Korf. Complete algorithms for cooperative pathfinding problems. In *IJCAI*, pages 668–673, 2011.

[26] N. Sturtevant. Hog-hierarchical open graph. http://webdocs.cs.ualberta.ca/~nathanst/hog.html, 2005.

[27] Nathan Sturtevant and Michael Buro. Improving collaborative pathfinding using map abstraction. In *AIIDE*, pages 80–85, 2006.

[28] Nathan R. Sturtevant and Michael Buro. Partial pathfinding using map abstraction and refinement. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 1392–1397. AAAI Press / The MIT Press, 2005. ISBN 1-57735-236-X.

[29] P. Surynek. A novel approach to path planning for multiple robots in bi-connected graphs. In *IEEE International Conference on Robotics and Automation, 2009. ICRA'09.*, pages 3613–3619. IEEE, 2009.

[30] Pavel Surynek. Making solutions of multi-robot path planning problems shorter using weak transpositions and critical path parallelism. In *Proceedings of the 2009 International Symposium on Combinatorial Search*, 2009.

[31] Y. Tavakoli, H.H.S. Javadi, and S. Adabi. A cellular automata based algorithm for path planning in multi-agent systems with a common goal. *IJCSNS*, 8(7):119, 2008.

[32] Ko-Hsin Cindy Wang and Adi Botea. Fast and memory-efficient multi-agent pathfinding. In *ICAPS*, pages 380–387, 2008.

[33] Ko-Hsin Cindy Wang and Adi Botea. Tractable multi-agent path planning on grid maps. In *IJCAI'09: Proceedings of the 21st international Joint conference on Artificial intelligence*, pages 1870–1875, San Francisco, CA, USA, 2009.

[34] Wikipedia. Sliding puzzle. URL http://commons.wikimedia.org/wiki/File:15-puzzle.svg. [Online; accessed 9-August-2011].