# Robotic Path Planning for High-Level Tasks in Discrete Environments

by

**Frank Imeson**

A thesis
presented to the University Of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

**Waterloo, Ontario, Canada, 2018**

**Examining Committee Membership**

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

| | |
|---|---|
| External Examiner: | Ibrahim Volkan Isler<br>Professor |
| Supervisor: | Stephen L. Smith<br>Associate Professor |
| Internal Examiner: | Dana Kulić<br>Associate Professor |
| Internal Examiner: | Wojciech Golab<br>Assistant Professor |
| Internal Examiner: | Steven Waslander<br>Associate Professor |

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

**Abstract**

This thesis proposes two techniques for solving high-level multi-robot motion planning problems with discrete environments. We focus on an important class of problems that require an allocation of spatially distributed tasks to robots, along with a set of efficient paths for the robots to visit their task locations. The first technique, SAT-TSP, models the problem with a framework that allows a natural coupling between the allocation problem and the path planning problem. The allocation problem is encoded as a Boolean Satisfiability problem (SAT) and the path planning problem is encoded as a Travelling Salesman Problem (TSP). In addition, this framework can handle complex constraints such as battery life limitations, robot carrying capacities, and robot-task incompatibilities. We propose an algorithm that leverages recent advances in Satisfiability Modulo Theory to combine state-of-the-art SAT and TSP solvers. We characterize the correctness of our algorithm and evaluate it in simulation on a series of patrolling, periodic routing, and multi-robot sample collection problems. The results show that our algorithm outperforms a state-of-the-art mathematical programming solver on a majority of the problems in our benchmark, especially the more difficult problems.

The second technique, Γ-Clustering, is used to reduce the computational effort of finding good solutions for metric discrete path planning problems. This technique can be used on the set of allocation path planning problems that do not have ordering constraints (ordering only affects the cost of the solution, not its feasibility). To obtain the computational savings, we find Γ-Clusters within the problem's environment and then restrict how feasible paths visit these clusters. We prove that solutions found using this approach are within a constant factor of the optimal. By increasing the parameter Γ we can improve the quality of the bound but we do so with less computational savings. We provide a simple polynomial-time algorithm for finding the optimal Γ-Clustering and show that for a given Γ the clustering is unique. We provide two methods for using Γ-Clusters on path planning problems, a *coupled method* and a *hierarchical method*. We demonstrate the effectiveness of these methods on travelling salesman instances, sample collection problems, and period routing problems. The results show that for many instances we obtain significant reductions in computation time with little to no reduction in solution quality. Comparing these methods to a standard integer programming approach reveals that as the problems become more difficult, the solution quality of the two methods degrade at a slower rate than the standard approach, thus for more difficult instances we can use Γ-Clustering to find higher quality solutions.

**Acknowledgements**

I would like to thank my PhD supervisor Stephen L. Smith; my Masters supervisors Siddharth Garg and Mahesh Tripunitara; and my PhD committee Ibrahim Volkan Isler, Dana Kulić, Vijay Ganesh, Wojciech Golab, and Steven Waslander.

## Dedication

I dedicate my thesis to my mother. Thank you for your guidance, encouragement, and support.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

As robots grow in sophistication, we look to them to perform complex tasks for a variety of commercial and industrial applications, such as environmental monitoring where robots are equipped with on-board sensors and tasked with selecting locations/viewpoints in the environment to monitor a set of targets [101, 93, 35, 73, 98]; or sample collection where robots seek out a set of locations in the environment to collect samples/data [26, 20, 17] (see Figure 1.1 for an example of a sample collection problem); or material transport problems that require robots to regularly deliver materials to a set of locations with different service demands [11, 6, 100]. For robots to successfully complete their tasks, they need to construct a sequence of actions (a plan) that will allow them to achieve their goals. Path/motion planning is the task of finding a set of transitions in the robot's environment that allows the robot to achieve its goals. In this thesis, we focus on solving path planning problems where a robot or group of robots is dispatched in the environment to complete a set of interdependent and spatially distributed tasks.

A common method for solving these problems is to break the problem into a high-level and low-level planning problem [50, 23, 84]. The high-level planner is responsible for finding a sequence of locations in the environment that allows the robot(s) to achieve the goals and the low-level planner is responsible for planning a collision free path between the locations. The two planners integrate into a complete motion planning framework as follows. The user provides the goals and constraints of the problem. The problem's environment is then discretized into a finite set of locations. The transition cost between locations is calculated by a low-level path planner [99, 52, 91] and used to construct a graph that captures the robot's discretized environment. The graph, goals, and constraints are used as input for the high-level path planner. The high-level planner finds a minimum cost path that visits a subset of the discretized locations that allows the robot to achieve

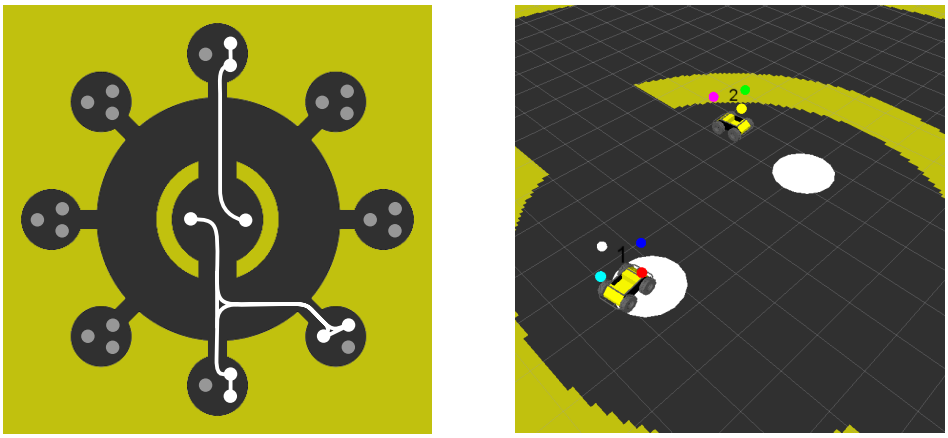



Figure 1.1: Robot Operating System (ROS) simulation for a sample collection problem. The left image shows the environment, the layout of the samples, and the robot solution paths. The right image shows the robots returning home after collecting one of each mineral type from a subset of the samples.

the goals, while obeying the constraints of the problem. The high-level path is then given to the low-level planner which finds a collision free path in the continuous space between the high-level locations. The complete solution allows the robot(s) to move in the physical world, achieve the goals, and obey the constraints while approximating optimal solution paths (the discretization may eliminate optimal solutions). Note, a more general version of this framework transitions between robot states, instead of robot locations. Adopting this strategy allows us to simplify a large path planning problem with a high density of information into one problem with sparse information (the high-level problem) and a series of small problems with dense information (the low-level problems). This thesis addresses the problem of solving high-level path planning problems where obstacle avoidance and robot to robot collisions are handled by the low-level planner. This thesis also refers to high-level problems as discrete path planning problems.

Many discrete path planning problems tend to have a strong combinatorial aspect to them and thus, many of these problems are intractable (**NP**-hard) [94]. This means the task of finding an optimal solution path, is the task of searching an exponentially large space, for which there is no known efficient method. There are a number of good solvers that we can use to search this space [32, 66, 12]. However, to utilize these solvers, we would need to express the path planning problem in a form readable by the solver. A common choice, is to use a mathematical solver which requires us to express the path planning

problem as an integer linear program (ILP). Mathematical solvers have received a lot of academic and commercial attention, as such they tend to be high performers that use a number of sophisticated techniques, some of which are proprietary and thus hidden from the public.

## 1.1 Literature Survey

In this section we give an overview of the literature related to solving high-level path planning problems. In subsequent chapters we provide a more in depth review of the literature related to the chapter's topic.

There is a set of specialized discrete path planning problems that have purpose built solvers. Two such examples of problems with purpose built solvers are the travelling salesman problem (TSP) and its generalized version, GTSP. TSP looks to find shortest tours visiting the locations and GTSP looks to find shortest tours that visit one location in each set of locations (the sets are provided as input). These problems appear in a variety of applications. In [10] and [17], the authors use TSP to find coverage paths for their environments. In [81], the authors use GTSP to find minimum length paths for a robotic welding arm. In [64], the authors use GTSP to find paths that monitor the environment. There are some very good solvers for both of these problems such as Concorde [3], LKH [32], GLKH [33] and GLNS [88]. To take advantage of the years of research that has went into these solvers, the user would need to translate their path planning problem to the specialized problem. However, due to the specialized nature of these problems the user may find this translation task challenging.

As stated, ILP [67] is a general framework used for modelling path planning problems. There are a number of good solvers, for which CPLEX [1] and Gurobi [66] are two of the more popular commercial solvers and SCIP [2] is one of the fastest non-commercial solver. In [68], the authors use MILP to plan optimal trajectories, and in [56, 44] ILP is used to solve multi-robot charging problems. In [102], the authors give an ILP solution for collision-free multi-robot problems.

Another common framework for expressing path planning problems is LTL. Here the user models their problem with a set of state transitions, e.g., if the robot is at location $x$ then the next location it will visit is $y$. Solvers developed in the model checking community can be used to compute *runs* (expressed as an automaton) that satisfy the LTL formula [36, 12]. In [89], LTL is used to express a class of persistent patrolling problems and the authors propose a method for computing optimal plans rather than just feasible plans. In [48], LTL is used to express multi-robot planning problems.

The STRIPS problem specification language [24] and its successor, PDDL [25] are like LTL in that the user models their problem as a set of state transitions (PDDL's transitions are more expressive than LTL). An annual competition [95] is held to encourage participants to develop better solvers. The problems in the competition range from emergency response scenarios to making a sandwich. There are a number of good solvers for these expressions such as FF [34] and LAMA [78].

## 1.2 Contributions

This thesis looks to solve high-level path planning problems. Below we give the contributions and organization for this thesis.

**Chapter 2** Reviews the concepts needed to solve high-level path planning problems and solves an example problem with the ILP approach.

**Chapter 3** This chapter introduces a novel alternative to ILP, called SAT-TSP. We designed SAT-TSP as a modelling language for expressing path planning problems. The language is structured in such a way to allow us to leverage recent techniques for Satisfiability Modulo Theory (SMT). We provide the SMT based solver CBTSP, for solving discrete path planning problems expressed as SAT-TSP problems. We prove its correctness and benchmark it against a commercial-grade ILP solver on a set of three important motion planning problems: patrolling, sample collection with multiple robots, and periodic routing. Our results show that CBTSP outperforms the ILP solver on a majority of the instances, especially the more difficult to solve problems.

In Appendix B we provide five additional SAT-TSP solvers and benchmark them against CBTSP. These alternatives were not as successful as CBTSP and are therefore omitted from the main body of this thesis.

Additionally, we provide a demo video[1] depicted in Figure 1.1 demonstrating one of our application problems in a ROS environment with Husky robots.

The work presented in this chapter is based on the following publications:

- Frank Imeson and Stephen L Smith. A language for robot path planning in discrete environments: The TSP with Boolean satisfiability constraints. In *IEEE International Conference on Robotics and Automation*, pages 5772–5777, 2014.

[1] `https://ece.uwaterloo.ca/~sl2smith/SAT-TSP/demo.mp4`

- Frank Imeson and Stephen L Smith. Multi-robot task planning and sequencing using the SAT-TSP language. In *IEEE International Conference on Robotics and Automation*, pages 5397–5402, 2015.

- Frank Imeson and Stephen L Smith. An SMT-based approach to motion planning for multiple robots with complex constraints. *IEEE Transactions on Robotics* (in review).

**Chapter 4** This chapter introduces a new technique for improving the efficiency of existing path planning solvers. The chapter introduces the clustering method Γ-Clustering, that it uses to prune off large portions of the solution search space (to improve solver efficiency). It provides an efficient algorithm for computing the optimal Γ-Clustering. The chapter also provides two path planning solver approaches for pruning the solution space. The first path planning solver approach, *coupled planning*, reduces the solution search space by restricting how feasible paths visit the Γ-Clusters. The second approach, *hierarchical planning*, further reduces the search space by decomposing the problem into a hierarchy of independent problems. This additional reduction of search space is at the expense of solution quality. However, in the chapter we prove that both methods find solutions within a constant factor of the optimal.

Our benchmark compares the two solver approaches to the standard approach without clustering. All simulations use an ILP solver to find path planning solutions. The benchmark problems consist of TSP, sample collection, and period routing problems. The results show that the coupled method and the hierarchical method find high quality solutions, typically within 10% of optimal. The results also show that as the problems become more difficult to solve, the coupled approach is able to maintain its performance longer than the standard approach, and the hierarchical approach is able to maintain its performance even longer.

Additionally, we compare the quality Γ-Clusters to the quality of clusters found by six different clustering methods. The benchmark shows that overall the quality of Γ-Clusters is superior to those found by the other methods.

The work presented in this chapter is based on the following publications:

- Frank Imeson and Stephen L Smith. Clustering in discrete path planning for approximating minimum length paths. In *American Control Conference*, pages 2968–2973, Seattle, WA, May 2017.

- Frank Imeson and Stephen L Smith. A hierarchical decomposition for computing approximate solutions to vehicle routing problems. Submitted to *International Conference on Automated Planning and Scheduling (ICAPS)*, 2018.

**Chapter 5** This chapter summarizes the work presented in this thesis and presents the future directions this work could take.

**Other Publications** I have also collaborated on an article that introduces a new solver for the generalized travelling salesman problem (GTSP). However, this work is not presented in this thesis.

- Stephen L Smith and Frank Imeson. GLNS: An effective large neighborhood search heuristic for the generalized traveling salesman problem. *Computers & Operations Research*, 2017.

# Chapter 2

# Preliminaries

This chapter reviews the concepts needed for solving high-level path planning problems. We walk the reader through each step of the process, starting with graphs.

## 2.1 Graphs

Graphs are used to represent the robot's discrete environment. The vertices in the graph represent the locations in the robot's environment and the set of edges capture the allowable transitions between the locations the robot is able to make. For example, an edge $\langle A, B \rangle$ with weight $w(A, B) = 4$ tells us that the robot may transition from location $A$ to location $B$ and if it does so then it will incur a cost of 4 (this cost may represent the time, distance, etc., that the robot incurs to make the transition).

A graph $G$ is defined by its tuple $\langle V, E, w \rangle$, where $V$ is the set of vertices, $E$ is the set of edges, and $w$ maps an edge $\langle v_a, v_b \rangle \in E$ to its weight/cost. In this thesis we do not allow negative weight edge transitions.

A *complete* graph contains an edge for every possible transition — there is an edge from every vertex to every other vertex ($|E| = |V|^2$). An *undirected* graph has edges that are used to capture transitions between locations, e.g., an edge $\langle A, B \rangle$ describes the transition from $A$ to $B$ and $B$ to $A$. This is not the case for *directed* graphs, where an edge is required to capture the transition from a location $A$ to a location $B$ and a separate edge is required to capture the transition from $B$ to $A$. Directed graphs allow for a greater expression of information and thus complexity (e.g., we can express one-way transitions). In this thesis, the assumption is that graphs are directed.

A *metric* graph is a complete graph that satisfies the triangle inequality. Formally, the triangle inequality states that for every set of vertices $A, B, C \in V$ the following holds:

$$w(A, C) \leq w(A, B) + w(B, C).$$

Many robotic environments satisfy the triangle inequality and thus their graph's are metric.

We use graphs to plan a sequence of transitions from one vertex to the next, which allows us to navigate the robot in its environment. We refer to these sequences as paths. In this thesis we concentrate on paths and cycles of the following form.

**Definition 2.1.1** (Paths and Cycles)**.** Given a graph $G = \langle V, E, w \rangle$, we define a *path* as a non-repeating sequence of vertices in $V$, connected by edges in $E$. A *cycle* is a path in which the first and last vertex are equal.

A Hamiltonian path is a path that visits every vertex in $V$ (exactly once) and a Hamiltonian cycle is a cycle that visits every vertex in $V$ (exactly once).

A path $p$ can also be represented by the set of edges it traverses. The cost $c$ of the path $p$ is the sum of its transition costs:

$$c = \sum_{\langle v_a, v_b \rangle \in p} w(v_a, v_b).$$

## 2.2 The Travelling Salesman Problem

The travelling salesman problem (TSP) is one of the most studied vehicle routing problems [4]. This chapter uses the TSP to demonstrate how to solve high-level path planning problems. Traditionally the TSP problem is posed as a salesman wanting to find a minimum length tour that visits a set of cities, where the length of the tour represents the total distance travelled.

**Problem 2.2.1** (Travelling Salesman Problem)**.** Given a complete and weighted graph $G = \langle V, E, w \rangle$, find a Hamiltonian cycle of $G$ with minimum cost.

## 2.3 Complexity Theory

Understanding the problem's complexity, is key to understanding the computational effort need to solve the problem. This will help us chose which solver approach to use. For

example, some problems can be solved in a polynomial amount time and thus one would likely want to choose a solver approach that guarantees the solution is found in a polynomial amount time.

### 2.3.1 Decision Problems

Many high-level path planning problems are optimization problems that look to find the lowest cost solution(s). Decision problems on the other hand are problems with a *yes* or a *no* answer. The decision version of the optimization problems that we are interested in are as follows: is there a solution to the problem of cost $c$ or lower. For example, the decision version of TSP looks to find Hamiltonian cycle of cost $c$ or less. In this way we can better classify the complexity of finding solutions, not just finding optimal solutions.

### 2.3.2 Complexity Classes

The study of complexity theory has discovered that there is a set of classes that we can use to classify a problem's complexity. These classes allow us to formally detect if problem $A$ is a variant of problem $B$, thus allowing us to use techniques/solvers developed for problem $B$ on problem $A$. This is done by reducing problem $A$ to problem $B$ and solving it with problem $B$'s solver. The algorithm used for the reduction must run in a polynomial amount of time, otherwise it is arguably a misuse of computational effort that could instead be used to find solutions for problem $A$.

Problem $A$ is said to be *complete* in its complexity class if any other problem $B$ in the same complexity class can be efficiently reduced to $A$. A problem is said to be *hard* for some complexity class $X$ if it is at least as hard as the hardest problem in $X$ (it can be harder than all of the problems in $X$). A problem *instance* is a specific realization of the problem. For example, if we were given a graph $G$ with 10 vertices and we wished to find a shortest tour visiting each vertex exactly once, then the graph $G$ is the instance and the problem is a TSP problem.

The complexity class **NP** captures the set of problems that have solutions that are efficiently verifiable. For example, a problem $A$ is in **NP** if given a solution to a problem instance of size $n$, then we can verify the solution with a polynomial amount of time ($O(n^b)$, where $b$ is some constant). Problem $A$ is **NP**-hard if it is as hard or harder than any problem in **NP**. Thus **NP**-complete problems are **NP**-hard.

As an example, the decision version of TSP problem is **NP**-complete— given a graph $G$, a tour $p$, and a budget $c$ we can efficiently verify that the path $p$ has cost $c$ or less. The

optimization version of TSP is **NP**-hard, as one cannot efficiently verify that a solution $p$ is the lowest cost solution. Therefore, to solve TSP problems we will look for a solver approach that can handle problems in this complexity class.

Some additional complexity classes are: the class **P**, which capture all problems that are solvable in polynomial time with respect to the input size of the problem; the class **PSpace**, which capture problems that are solvable with a polynomial amount of space; and the class **EXPSpace**, which are problems solvable with an exponential amount of space. The class **P** is a subset of **NP** and it is believed that it may be a strict subset. We provide the following relations to understand which classes are contained within each other and thus work towards understanding which problems are harder to solve than others:

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSpace} \subseteq \mathbf{EXPSpace}.$$

## 2.4 Using ILP to Find Solution Paths

As stated, integer linear programming is a good general approach for finding solutions for high-level path planning. This approach is suitable for finding optimal solutions for path planning problems that have their decision problem in the class **NP**-complete.

This approach uses a series of linear inequalities coupled with an objective to model the planning problem. The inequalities capture relationships between the variables, which can only take on integer values and the objective is a linear equation that is to be minimized (or maximized). The solver attempts to find an optimal assignment of the variables that satisfies the set of inequalities (minimize the objective).

Now we walk the user through how to solve TSP problems with an ILP solver. Let us use the set of variables $\{e_{i,j} | \langle v_i, v_j \rangle \in E\}$ to represent the edges and the set of variables $v_i \in V$ to represent the set of vertices for the ILP expression.

minimize

$$\sum_{v_i \in V} \sum_{v_j \in V \setminus v_i} e_{i,j} w(v_i, v_j) \tag{2.1}$$

subject to

$$\sum_{v_j \in V \setminus v_i} e_{i,j} = 1, \qquad \text{for each } v_i \in V \tag{2.2}$$

$$\sum_{v_i \in V \setminus v_j} e_{i,j} = 1, \qquad \text{for each } v_j \in V \tag{2.3}$$

$$\sum_{v_i, v_j \in S} e_{i,j} \leq |S| - 1, \qquad \text{for each } S \subset V \text{ s.t. } |S| \geq 2 \tag{2.4}$$

$$0 \leq v_i \leq 1, \qquad \text{for each } v_i \in V \tag{2.5}$$

$$0 \leq e_{i,j} \leq 1, \qquad \text{for each } \langle v_i, v_j \rangle \in E \tag{2.6}$$

The above formulation was taken from [70]. The objective (2.1) captures the cost of the solution by adding all the edge weights that are included in the tour. Constraints (2.2) and (2.3) ensure that there are exactly one incoming and one outgoing edge for each vertex. Then subtours (a solution with multiple tours instead of one tour) are eliminated in Constraint (2.4) by explicitly eliminating every possible tour. There are an exponential number of these constraints and so in practice these constraints are added to the formulation as the solver violates them. This is referred to as a lazy constraint.

Once the ILP solver has found a solution, we translate it back to a TSP solution. This is trivially done for this example since the set of edges that construct the tour $p$ are the set of included edge variables in the ILP solution.

*Remark* 2.4.1. We chose the above ILP expression over a polynomial expression because of its effectiveness. The decision version of TSP is **NP**-complete, thus there are polynomial time reductions to ILP such as the MTZ formulation [57], however in practice, better performance is achieved with the above expression [70, 69].

## 2.5 Summary

In this chapter we reviewed the steps for solving high-level path planning problems using an ILP approach. A summary of the steps are as follows:

1. Classify the path planning problem's complexity.
2. Choose an approach capable of solving problems of said complexity (for our example, we used ILP).
3. Express the problem in a readable format for the chosen approach.
4. Use an exiting solver to find solution(s).

In the next chapter we explore an alternative approach to ILP, then in Chapter 4 we explore two methods for improving solver efficiency (the chapter improves the efficiency of an ILP solver).

# Chapter 3

# An Alternative to ILP

In this chapter we propose an alternative to the ILP approach for solving high-level path planning problems. We focus on the class of problems that require the allocation of tasks to robots and a set of efficient paths that allow the robots to visit their task locations.

We start by introducing the new problem language SAT-TSP. We use this language to express/model high-level path planning problems. The problem is a combination of the satisfiability problem (SAT) and the travelling salesman problem (TSP). The SAT problem takes in as input a Boolean formula and asks the question is there an assignment of the variables that satisfies the formula. The TSP problem takes in as input a graph and searches for the minimum cost Hamiltonian cycle (see Section 2.2). The SAT-TSP problem takes in as input: one formula, one or more graphs (multiple robots use multiple graphs), and one or more cost budgets (multiple robots may use multiple cost budgets). We use the formula to express the problem's constraints and the graphs and cost budgets are used to capture the sequencing/routing problems. The constraints and the environment(s) are linked together via a set of variables. Here a location in the environment is visited if and only if its corresponding variable is assigned true. This allows SAT-TSP to express logical constraints on the robots' motion, such as task dependencies, incompatibilities, and capacity constraints. A SAT-TSP solution is a set of paths (one for each robot) that visits a subset of locations in the environment, satisfies the constraints, and has transitions that are within the cost budget(s).

The structure of SAT-TSP allows us to leverage recent developments in the Satisfiability Modulo Theory (SMT) community. Specifically, we developed the solver CBTSP to solve SAT-TSP problems using the SMT framework. In this way we were able to combine a state-of-the-art SAT solver with a state-of-the-art TSP solver.

The SAT-TSP problem inherits the strengths of its sub-problems, SAT and TSP. As such it is most suited for path planning problems with constraints that are efficiently expressible as SAT formulae and optimization objectives resembling shortest paths/tours. Conversely if a set of problems contain constraints that are not efficiently expressible in SAT or the problem has an optimization objective that has nothing to do with the robots transitions, then SAT-TSP would likely be a poor choice.

The contributions of this chapter are as follows. We formally introduce the SAT-TSP problem language and characterize its complexity. Specifically, we show that even when SAT-TSP is compromised of easy SAT and TSP problems, the SAT-TSP instance can still be hard. We provide the SAT-TSP solver, CBTSP, and prove its correctness. Then we benchmark CBTSP against an ILP approach on a series of high-level path planning problems: patrolling problems, multi robot sample collection problems, and periodic routing problems. The results show that CBTSP often outperforms the ILP approach — especially on more difficult instances.

This chapter is organized as follows. Section 3.1 reviews the related work to the SAT-TSP approach. Section 3.2 provides the necessary background needed for this chapter. Section 3.3 formally introduces the SAT-TSP problem and classifies its complexity. Section 3.4 introduces the CBTSP solver, proves its correctness for the class of TSP-monotonic instances (Definition 3.4.2), and provides a relaxation for the TSP-monotonic class to expand the set of solvable instances. Section 3.5 outlines the ILP approach used for our comparison and Section 3.6 shows how to express patrolling, collection, and period routing problems with SAT-TSP and ILP. Section 3.7 details our benchmark and presents the results. For the interested reader we provide five additional SAT-TSP solver approaches in Appendix B, which consist of SAT, TSP, GTSP, CSP, and SMT solver approaches.

## 3.1 Related Work

This section builds upon Section 1.1 to provide a more in depth review of the literature related to this chapter. As previously stated, TSP, GTSP, ILP, LTL, and STRIPS are all general frameworks used for solving high-level path planning problems.

The TSP and GTSP problems are more specialized than the SAT-TSP problem. As such, using TSP or GTSP to model complex path planning problems is more challenging than using SAT-TSP. TSP and GTSP solvers are geared towards searching for low cost paths — not solving complex logic. This is in contrast with SAT solvers, which are geared towards using the logic of the problem to eliminate infeasible solutions. As such there is no reason

to believe that TSP and GTSP solvers would be proficient at solving complex path planning constraints. Regardless, in Appendix B we explore using TSP and GTSP solvers for finding SAT-TSP solutions.

The non-optimization version (the decision version) of ILP, like SAT-TSP is **NP**-complete— both can express the same set of problems. However, in practice both approaches have their own limitations. For example, unlike ILP, it is arguably awkward to use SAT-TSP for counting constraints (e.g., visit $x$ red locations). This is due to the simple structure that SAT uses for expressing logic. However, this structure has led to the success of the DPLL [19, 18] and DPLL(T) [63] algorithms, which are used in modern SAT solvers [90, 59] and CBTSP. Furthermore, we demonstrate in this chapter that SAT-TSP can be successfully used for counting constraints. We provide a direct comparison of CBTSP to the commercial ILP solver, Gurobi, on a set of robotic path planning problems and the results show that CBTSP often outperforms the ILP solver.

Many important path planning problems lie in **NP** and our goal in this chapter is to produce a solver that is tailored to problems in this class. A drawback of LTL, is that LTL is in a higher complexity than SAT-TSP. Specifically, the decision version of LTL is in the class **PSpace**-complete [87] where the decision version of SAT-TSP is in the class **NP**-complete. The LTL language is likely more expressive than SAT-TSP, since it is believed that **PSpace** is a strict superset of **NP**. For example, LTL can be used to express planning problems that consist of infinite length solution paths (i.e., persistent problems) where SAT-TSP cannot. There are potential downfalls of using a more complex language, such as allowing the user to inadvertently increase the complexity of their problem. For example suppose the user requires a path from location $A$ to location $B$ (any path) and the user has access to a TSP solver. If they choose to express their problem as a TSP, they would be able to find a solution to their problem but they would have also increased the complexity of their problem.

STRIPS and PDDL are capable of expressing problems in **EXPSpace** [21], which is believed to be a strict superset of **PSpace** and thus **NP**. This work focuses on problems in **NP** and uses an integer programming approach as our point of comparison.

Our main solver, CBTSP is based on SMT which is an extension of SAT that allows for first order logic. SMT solvers have previously been proposed for solving path planning problems. In [86] and [38], the authors solve the high-level and low-level path planning problems simultaneously. This is unlike our approach where we leverage the SMT framework to better solve the high-level problem. In [60] and [80], the authors use an off the shelf SMT solver to find solutions for the high-level path planning problem. Our approach differs from these by using a custom SMT theory that specializes in handling the combinatorial

nature of sequencing locations. Additionally, we have explored solving SAT-TSP instances with an SMT solver and found that it does not scale well (the results are provided in Appendix B). To the best of our knowledge, we are the first to solve high-level path planning problems using a custom SMT theory to handle the combinatorial aspect of sequencing.

## 3.2 Background

This section reviews the background concepts needed for this chapter.

### 3.2.1 Boolean Satisfiability (SAT)

In this chapter we use the Boolean satisfiability problem (SAT) [18] to encode path planning constraints. A SAT formula is a propositional formula that is composed of Boolean literals and operators. A literal is either a Boolean variable ($x$) or its negation ($\neg x$). The operators are conjunction ($\wedge$, and), disjunction ($\vee$, or) and negation ($\neg$, not), which may operate on the literals or other Boolean formulae. An assignment of the variables (true or false) results in the formula being satisfied (true) or not (false). The conjunctive normal form (CNF-SAT) is the canonical form of SAT. A formula $F$ is in its canonical form if the formula is a conjunction of clauses, where each clause is a disjunction of literals. We allow formulas to be expressed in non-canonical form.

**Problem 3.2.1** (SAT)**.** Given a propositional Boolean formula $F$, determine if it is satisfiable.

### 3.2.2 Boolean Circuits

Developing SAT expressions to capture constraint logic can be difficult. To aid in this process, one can borrow from Boolean circuits. In this chapter we borrow logic from adder circuits to construct counting constraints.

Adder circuits are used in electronics to do rudimentary mathematical operations [55]. Given an input set $X$ of Boolean signals, the adder circuit counts the number of signals that are true (high). The example circuit shown in Figure 3.1 takes as input, a set of signals $X = \{x_1, x_2, \ldots, x_5\}$ and outputs the bit $b_1$. This circuit is composed of four two bit adder circuits. The circuit constrains the binary one bit $b_1$ to the equal the number of true input variables in $X$. There would be a similar circuit for the twos bit that takes

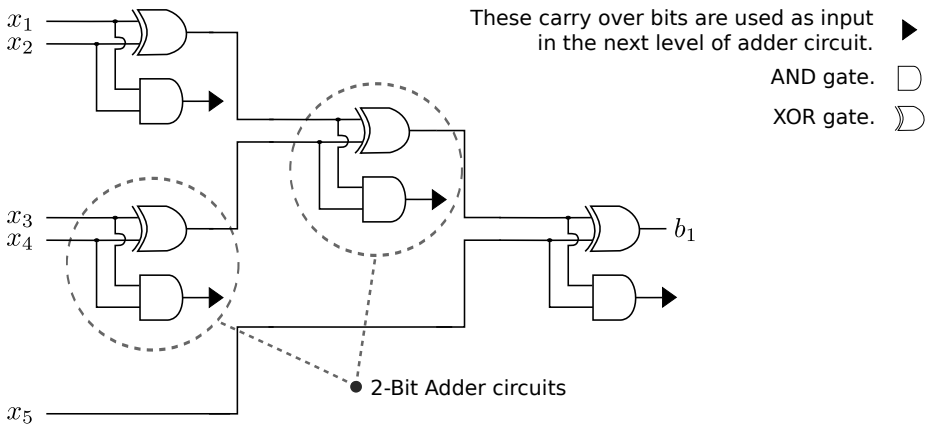



Figure 3.1: An adder circuit summing up Boolean input variables $X = \{x_1, x_2, \ldots, x_5\}$ and outputting the Boolean variable $b_1$, as well as the carryout bits.

in as input, all of the carry out bits from the example. The complete binary circuit is constructed using techniques in [55]. The circuit is then translated to a Boolean formula (SAT) in polynomial time using methods from [41].

### 3.2.3 SMT and DPLL(T)

Satisfiability Modulo Theory (SMT) is an extension of SAT that allows for first-order logic. The SMT framework extends SAT by linking non-SAT theories back to the SAT formula. Specifically, the SMT problem uses a propositional formula $F$ defined over a set of Boolean variables $X$ that contain a set of predicate variables $x_t \in X_T$ for each theory instance $t \in T$. Each theory instance belongs to a specialization of decidable first-order logic, such as arithmetic logic or quantifiable Boolean logic. The power of this approach is that a SAT solver can be used to solve the propositional formula while a set of specialized solvers can be used for the theories.

**Definition 3.2.2** (SMT)**.** An SMT formulation $\langle F, T \rangle$, is satisfiable if and only if

1. $F$ is a propositional formula defined over $X \supseteq X_T$,
2. $T$ is a set of decidable first-order logic problems defined over $Q$ such that $X \subseteq Q$,
3. and there exists an assignment of $Q$ satisfying $F$ such that for every $t \in T$ the corresponding predicate variable $x_t$ agrees with the evaluation of $x_t$ (true or false).

**DPLL(T):** The DPLL(T) algorithm for solving SMT instances is based on the DPLL algorithm [18] using solving SAT instances (propositional formulae). The DPLL algorithm solves $F$ by building a list of assignments for the Boolean variables in $F$ (a partial solution). This algorithm extends DPLL to incorporate theories by linking the predicates $x_t \in X$ to the theories $t \in T$. Informally, the algorithm works as follows: as partial solutions for $F$ are being constructed by the DPLL algorithm, the theory solvers are called to confirm that the partial SAT solutions are consistent with the theories. If a theory is consistent, then the DPLL algorithm continues. If not, the theory solver that detected the conflict constructs a learnt clause $f_{\text{conflict}}$ to capture the conflict over the set Boolean variables $X$. The learnt clause is added to $F \leftarrow F \wedge f_{\text{conflict}}$ and the algorithm backtracks some or all of its assignments until the partial solution no longer conflicts with the new $F$.

**Note 3.2.3.** The above description of DPLL(T) captures only the mechanisms of DPLL(T) that are utilized by the CBTSP solver. A full description of DPLL(T) can be found in [63].

### 3.2.4 Induced Subgraphs

An induced subgraph is a subgraph of $G$ that contains a subset of the vertices and all the edges connected to those vertices.

**Definition 3.2.4** (Induced Subgraph)**.** The *induced subgraph* of a graph $G = \langle V, E, w \rangle$ for $V' \subseteq V$ is the graph $G' = \langle V', E', w \rangle$ with $E' = \{\langle v_i, v_j \rangle \in E | v_i, v_j \in V'\}$. We say that $G'$ is induced by $V'$.

## 3.3 Problem Statement

In this section we provide a formal definition of the SAT-TSP problem and classify its complexity.

### 3.3.1 SAT-TSP Definition

Before we define the decision version of SAT-TSP and its two optimization problems, we start with some notation. A SAT-TSP instance can take as input multiple graphs with a cost budget $c_i$ for each input graph $G_i$. To simplify the notation we sometimes absorb the cost budget $c_i$ into the graph's tuple $G_i = \langle V_i, E_i, w_i, c_i \rangle$ (when there is only one graph, we do not absorb the cost budget). A formula $F$ has a solution or partial solution $M$, which

is a collection of variable assignments. A variable $x$ is assigned true if $x^T \in M$ and false if $x^F \in M$, otherwise $x$ is unassigned.

**Problem 3.3.1** (SAT-TSP)**.** The **SAT-TSP** decision problem takes as input $\langle G_1, G_2, \ldots, G_n, F, C \rangle$, where:

- $G_i = \langle V_i, E_i, w_i, c_i \rangle$ is a directed weighted graph with edge weights $w_i : E_i \to \mathbb{R}_{\geq 0}$ and cost budget $c_i$,
- $F$ is a Boolean formula defined over $X \supseteq V_1, V_2, \ldots V_n$,
- $C$ is a budget imposed on the total path cost.

Then the instance is satisfiable if and only if:

- there exists a tour of each graph $G_i$ over a subset $V_i' \subseteq V_i$ with cost $c_i' \leq c_i$,
- such that $\sum_{i=1}^{n} c_i' \leq C$,
- and there exists an assignment $M$ of $X$ satisfying $F$ such that a vertex variable $v = 1$ ($v^T \in M$) if and only if $v \in V_1' \cup V_2' \cup \ldots \cup V_n'$.

In this chapter we consider two optimization problems for SAT-TSP: 1) minimize the total cost budget and 2) minimize the maximum cost budget of any graph $G_i$. The first optimization problem minimizes $C$ and the second optimization problem minimizes $\max_i c_i$.

We refer to problem instances with metric graphs as metric SAT-TSP instances. Note that, robotic environments are typically metric as they often represent time, distance, and/or battery consumption.

**Example 3.3.2** (Modelling a problem in SAT-TSP)**.** Consider two robots: robot 1 has a battery life of 10 minutes and can travel at 2 m/s, while robot 2 has a battery life of 12 minutes and can travel at 1 m/s. The environment contains locations $L = \{1, 2, \ldots, |L|\}$. Each location must be visited by either robot 1 or 2, but not both. We encode this as a SAT-TSP instance by first creating two graphs, $G_1 = \langle V_1, E_1, w_1, c_1 \rangle$ and $G_2 = \langle V_2, E_2, w_2, c_2 \rangle$ to capture the transitions for each robot. Specifically, each graph $G_r$ contains a vertex $v_i \in V_r$ for each location $i \in L$ and an edge $\langle v_i, v_j \rangle$ in the graph represents the transition from location $i$ to $j$. The weight of the edge $\langle v_i, v_j \rangle$ in each graph is given by the travel time for the corresponding robot to travel from location $i$ to location $j$. If the distance from $i$ to $j$ is $d_{i,j}$, then the weight for robot 1 is $w_1(v_i, v_j) = d_{i,j}/2$ and the weight for robot 2 is

$w_2(v_i, v_j) = d_{i,j}$. Now the tuple $\langle V_1, E_1, w_1, 10\rangle$ captures the transition system for robot 1 and its battery budget, similarly tuple $\langle V_2, E_2, w_2, 12\rangle$ for robot 2.

Next we construct the formula $F$ using the set of variables $X = \{v_{i,r} | i \in L, r \in R\}$ that represents if vertex $v_i \in G_r$ is in the solution or not (true or false). We start by adding the set of clauses $(v_{i,1} \vee v_{i,2})$ to $F$ for each $i \in L$, to express that each location $i \in L$ must be visited by at least one robot. Then we add the clauses $(\neg v_{i,1} \vee \neg v_{i,2})$ to express that each location $i \in L$ can be visited by at most one robot.

Finally, we choose a value for the total cost budget, to be any value $C \geq 22$, since any solution satisfying the individual robot budgets will also satisfy this $C$. If we wish to find the solution with the lowest total cost, we search for feasible solutions that minimize $C$.□

### 3.3.2 Complexity of SAT-TSP

The decision version of SAT-TSP is **NP**-complete. This follows from the fact that SAT reduces to SAT-TSP (SAT is an **NP**-complete problem) and a SAT-TSP solution can be verified in polynomial time.

We also classify the complexity of SAT-TSP when its SAT and TSP problems are easy to solve. Let $\mathbf{SAT}^* \subseteq \mathbf{SAT}$ and $\mathbf{TSP}^* \subseteq \mathbf{TSP}$ be the set of problem instances that are in **P** (solvable in polynomial time). An example of a TSP instance that is easy to solve (in $\mathbf{TSP}^*$) is an instance with a graph that has all of its edge weights equal to 1. Finding an optimal solution of cost $|V|$ is accomplished in polynomial time by choosing any ordering of the vertices. We are interested in SAT-TSP instances composed of easy SAT and TSP instances because it is often the case that TSP solvers work very well on TSP problems encountered in practice, such as those in the TSP library [77]. Similarly, SAT solvers are quite efficient on practical SAT problems, such as instances in the SAT library [37]. So does this mean that if our SAT-TSP problem is composed of easy instances in $\mathbf{SAT}^*$ and $\mathbf{TSP}^*$, then it is easy to solve?

**Theorem 3.3.3.** Consider the subset of SAT-TSP problems composed of instances from $\mathbf{SAT}^*$ and $\mathbf{TSP}^*$, then this subset of SAT-TSP remains **NP**-complete.

*Proof.* We prove the above result by reducing a **NP**-complete problem to a SAT-TSP problem composed of $\mathbf{SAT}^*$ and $\mathbf{TSP}^*$ problems. Specifically, we do this for the SET-COVER problem. This problem takes in as input $\langle U, S, C\rangle$, where $U$ is a universe of finite elements (a set), $S$ is a collection of sets, for which each set $S_i \in S$ contains a subset of the elements from $U$ ($S_i \subset U$), and $C$ is a cost budget, then a solution is a subset $S' \subset S$ that covers all of the elements in $U$ with $S'$ and $|S'| \leq C$. The reduction maps the sets $S_i \in S$ to vertices

$v_i$ in the complete graph $G = \langle V, E, w \rangle$, where the edges in the graph all have a weight of 1. The inclusion/exclusion of a set $S_i$ is indicated by the SAT-TSP tour visiting the vertex $v'_i \in V$. The SAT-TSP formula

$$F = \bigwedge_{u_j \in U} \left( \bigvee_{i | u_j \in S_i} v_i \right)$$

is used to ensure that each element $u_j \in U$ is covered by at least one set $S_i$. A solution to the SAT-TSP problem $\langle G, F, C \rangle$ ($C$ is given as input to the set cover problem) is a tour of length $c' \leq C$, which translates to a set cover solution with $c'$ sets ($|S'| = c'$).

The TSP instance $G$ and sub-instances have the trivial solution of any tour (all tours have the same cost since all edges have weight 1). The SAT instance F and sub-instances also have trivial solutions since there are no negative literals in the formula (we simply assign all the literals to be true). Thus, both the SAT and TSP instances are solved in linear time (polynomial time) and since SET-COVER is **NP**-hard, then it must be the case that SAT-TSP remains **NP**-complete despite the fact that the SAT and TSP problems are in $\mathbf{SAT}^*$ and $\mathbf{TSP}^*$ respectively. □

Theorem 3.3.3 proves that SAT-TSP is **NP**-hard even when its sub-problems are easy. Therefore, we expect that an effective SAT-TSP solver will require some additional level of sophistication on top of being able to solve SAT and TSP problems.

The next Section starts by introducing the BRUTE solver as a naïve combination of a SAT and TSP solver that explores every SAT solution (there may be exponential number of solutions to explore). The CBTSP solver improves upon BRUTE by adding the ability to negate partial solutions while leveraging the sophistication of the SAT solver to choose candidate solutions. This ability of negating partial solutions allows CBTSP to more effectively prune the search space. Letting the SAT solver choose the candidate solutions allows us to take advantage the algorithm's ability to find the variables/locations in the problem that cause the most conflicts.

## 3.4 CBTSP: An SMT-based approach for SAT-TSP

In this section we provide a simple BRUTE solver as a lead-in to the CBTSP solver. We provide a high-level description of the CBTSP solver and provide the conditions under which CBTSP can be used.

### 3.4.1 The BRUTE Approach: A Lead-in to cbTSP

The BRUTE approach decouples the SAT-TSP instance by first solving the SAT instance and then the TSP instance. For simplicity, Algorithm 1 implements a SAT-TSP solver that only takes instances with one input graph. The algorithm is easily extended to take multiple graphs by replacing Line 5 with multiple calls to the TSP solver and bookkeeping the additional cost budgets.

The BRUTE solver approach is given in Algorithm 1. First it uses a SAT solver to find a feasible set of included vertices $V'$ (Lines 2 and 3). Next, it uses a TSP solver, TSP-SOLVE to find the minimum cost tour $p'$, with cost $c' \leq C$ (Line 5) of the induced subgraph $G'$ (Line 4). It negates the solution from reoccurring (Line 10) and repeats the process until it has checked every solution (Line 1). The problem with this approach is that there may be an exponential number of solutions to find and negate.

**Algorithm 1:** BRUTE-SAT APPROACH$(G, F, C)$

**Input:**
$G$: is a graph with vertices $V$.
$F$: is a Boolean formula with variables $X$.
$C$: is a $\Gamma$-Clustering.
**Output:**
$M'$: is a full assignment of $X$, that evaluates $F$ to true.
$p'$: is an optimal solution path.
$c'$: is the cost of $p'$.

```
1  while SATISFIABLE(F) do
2      M' ← SOLVE(F)
3      V' ← {v ∈ V | v^T ∈ M'}
4      G' ← SUBGRAPH(G, V')
5      ⟨p', c'⟩ ← TSP-SOLVE(G', C)
6      if c' ≤ C then
7          return ⟨M', p', c'⟩
8      else
9          f' ← (⋀_{v∈V'} v) ∧ (⋀_{v∈V\V'} ¬v)
10         F ← F ∧ ¬f'
11 return ∅
```

If we were solving metric instances, a less naïve approach would be to replace Line 9

with $f' \leftarrow (\bigwedge_{v \in V'} v)$. This would allow the algorithm to negate the solution, as well as all supersets of $V'$, thus more effectively pruning the solution space. This approach would be valid for metric instances since we cannot lower the solution cost by adding vertices to the solution tour. The CBTSP approach works in this way, but it is also able to negate partial solutions. In fact the CBTSP solver's parameters (found in Appendix A) allow it to be configured as a non-naïve brute approach (negate supersets of full solutions but ignore partial solutions). In this way, we can start to see how CBTSP is a sophisticated extension of the BRUTE approach.

### 3.4.2 The CBTSP Solver

The CBTSP solver builds on the BRUTE approach by using partial solutions to prune the search space. This allows for significant computation savings. We begin by casting the SAT-TSP problem in the SMT framework. We use the DPLL(T) algorithm (Algorithm 2) to couple a SAT and TSP solver. The CBTSP solver only works for TSP-monotonic instances (Definition 3.4.2), which essentially means that the cost of partial solutions cannot be reduced by adding vertices. TSP-monotonicity is a necessary property needed to prune partial solutions. Additionally, metric SAT-TSP instances are TSP-monotonic (Theorem 3.4.6).

**Definition 3.4.1** (Partial Solution)**.** Given a SAT-TSP instance $\langle G_1, G_2, \ldots, G_n, F, C \rangle$, a *partial solution* $M$ is a True/False assignment for a subset of the variables in $X$. The subgraph $G'_i$ *induced* by $M$ is the subgraph induced by the vertices $V'_i \subset V_i$ when corresponding variables in $M$ are true.

**Definition 3.4.2** (TSP-Monotonicity)**.** A graph $G = \langle V, E, w \rangle$ is **TSP-monotonic**, if for any vertex subsets of the following form $V_1 \subset V_2 \subseteq V$ the induced subgraphs $G_1$ and $G_2$ have TSP costs $c_1 \leq c_2$.

The TSP-monotonic property allows for the negations of partial solutions that exceed the cost budget(s) (including more vertices cannot lower the solution cost). Specifically, if the partial solution exceeds the cost budget(s), then we exclude the responsible vertices and their supersets from reoccurring (the negation details are given in Algorithm 3, Lines 6-11).

To find optimal solutions the CBTSP approach solves a series of SAT-TSP problems $\langle G_1, G_2, \ldots, G_n, F, C \rangle$ formulated as SMT problems. The SMT formulation uses a custom TSP theory (Algorithm 3) to construct the induced subgraph (Lines 2 and 3) and answer the decision problem of whether or not a TSP solution exceeds the cost budget(s). The SMT problem is as follows: the propositional formula is $F \wedge x_{tsp}$, where $F$ is the Boolean formula in the SAT-TSP instance and the predicate $x_{tsp}$ is decided by the custom TSP theory ($x_{tsp}$ is true if and only if the theory can find a TSP tour of the included vertices

within the cost budgets). The SMT theories have prior knowledge of the ground variables $X$, the graphs $G_i$, and the cost budget(s). The cost budget(s) are set by the user or a binary search algorithm used to find optimal solutions. The TSP theory is as follows:

**Problem 3.4.3.** The `TSP-Theory` takes as input the tuple $\langle G_1, G_2, \ldots, G_n, M, C\rangle$, where

- $\langle G_1, G_2, \ldots, G_n, C\rangle$ is the input to setup the theory and
- $M$ is the input when called by the SMT solver.
- Each $G_i = \langle V_i, E_i, w_i, c_i\rangle$ is a TSP-monotonic graph and has a cost budget $c_i$,
- $C$ is the total cost budget, and
- $M$ is a partial or full assignment of the ground variables in $F$.

Then the theory is satisfiable ($x_{tsp} = 1$) if and only if

- there exists a tour for each graph $G_i$, of cost $c_i$ or less over the set of vertices $\left\{v \in V_i | v^T \in M\right\}$
- and the total cost of the solution does not exceed $C$.

In the SMT formulation, the $x_{tsp}$ predicate is forced to be true, thus all solutions and partial solutions must not exceed the cost budget(s). If the TSP theory finds that there is no solution, then a learnt clause is constructed (Lines 6-10 of Algorithm 3) and added back to the formula $F$ (Line 6 of Algorithm 2). The DPLL(T) solver is subsequently tasked with solving the new formula (which includes the learnt clause).

At a high-level, CBTSP solves decision instances as follows:

1. set up the `TSP-Theory` with the graphs and budgets $\langle G_1, G_2, \ldots, G_n, C\rangle$,
2. call DPLL(T) on $F$ (Algorithm 2),
3. build partial solutions (Line 6 of Algorithm 2),
4. check the consistency of the `TSP-Theory` (Line 4),
5. negate partial solutions if there is a conflict (Line 6),
6. return the solution if satisfiable, otherwise return $\emptyset$.

**Algorithm 2:** Overview of DPLL(T) on $F$

**Input:**

$G_i$: is a graph $\langle V_i, E_i, w_i, c_i \rangle$, where $c_i$ is cost budget for $G_i$.

$F$: is a Boolean formula with variables $X$.

$C$: is a $\Gamma$-Clustering.

**Output:**

$M$: is a full assignment of $X$, that evaluates $F$ to true.

**Precondition:**

`TSP-Theory.setup`$(G_1, \ldots, G_n, C)$

$M \leftarrow \emptyset$

**while** $\exists x \in X$ *s.t.* $\left\{x^T, x^F\right\} \cap M = \emptyset$ **do**

  Add a new variable assignment to $M$

  $f_{\text{conflict}} \leftarrow$ `TSP-Theory`$(M)$

  **if** $f_{conflict} \neq \emptyset$ **then**

    $F \leftarrow F \wedge f_{\text{conflict}}$

    Backtrack $M$ to some point that does not conflict with $F$

**if** $M$ *solves* $F$ **then**

  **return** $M$

**else**

  **return** $\emptyset$

**Algorithm 3:** Overview of `Tsp-Theory` $(M)$

**Input:**

$G_i$: one of $n$ input graphs, where each $G_i = \langle V_i, E_i, w_i, c_i \rangle$.

$X$: the set of ground variables for $F$.

$C$: a budget for the total solution cost.

$M$: a partial or full assignment of the variables in $X$.

**Output:**

$f_{\text{conflict}}$: a clause that captures the conflict.

**for** *each graph* $G_i$ **do**
  $V'_i \leftarrow \{v_j \in V_i | v_j^T \in M\}$
  $G'_i \leftarrow \textsc{InducedGraph}(G_i, V'_i)$
  $\langle p'_i, c'_i \rangle \leftarrow \textsc{TspSolve}(G'_i, c_i)$

`// Construct the learnt clause`

$f_{\text{conflict}} \leftarrow \emptyset$

**if** *some* $c'_i > c_i$ **then**
  $f_{\text{conflict}} \leftarrow \neg \left( \bigwedge_{v_j \in V'_i} v_j \right)$

**else if** $\sum c'_i > C$ **then**
  $V' \leftarrow V'_1 \cup \ldots \cup V'_n$
  $f_{\text{conflict}} \leftarrow \neg \left( \bigwedge_{v_j \in V'} v_j \right)$

**return** $f_{\text{conflict}}$

**Note 3.4.4.** Algorithm 2 is a simplified version of CBTSP. The real algorithm is more sophisticated than what is depicted — it keeps track of more than $M$, such as the solution path(s) and their costs.

**Example 3.4.5** (Illustration of the CBTSP approach)**.** This example demonstrates the interaction between the SAT solver (based on DPLL) and the TSP solver (the TSP theory) in CBTSP. Suppose we have one input graph $G = \langle V, E, w, c\rangle$ and suppose the solver has constructed a partial solution $M = \left\{x_{99}^T, v_2^F, v_1^T, x_{67}^T, v_4^T, v_5^T\right\}$. Then suppose that the SAT solver extends the partial solution by assigning $v_6$ to be true. Once the assignment $v_6^T$ is added to $M$, the TSP theory solver is called to make a consistency check. The TSP theory uses $M$ to construct the TSP problem $G'$ over the set of included vertices $v_1, v_4, v_5$ and $v_6$. Note $x_{99}$ and $x_{67}$ are not vertex variables (they are auxiliary variables) and so they do not appear in this set. The TSP theory then calls the TSP solver with input $\langle G', c\rangle$ and if a tour is found within the budget the theory returns true (consistent). The DPLL solver (SAT solver) continues to build upon the partial solution. If no tour is found within the budget the check is inconsistent and the learnt clause $f_{\text{conflict}} = \neg(v_1 \wedge v_4 \wedge v_5 \wedge v_6)$ is constructed to be added to $F$ so that the SAT solver avoids this solution in the future. The SAT solver then backtracks (revert some of the partial solution) to avoid the inconsistency and looks for a new solution that satisfies $F \wedge f_{\text{conflict}}$. □

The optimization version of CBTSP uses a modified version of binary search to find a solution with minimum cost. It solves a series of SAT-TSP decision instances with different cost budgets (minimize $C$ or minimize the maximum $c_i$). Algorithm 4 shows the version of binary search that minimizes the cost $C$. The same basic algorithm is used to find optimal solutions for problems that aim to minimize the maximum cost $c_i$ (subgraph cost). This is achieved by replacing $C$ with $c_1$ in the algorithm and replace $c_i$ with $c_1$ for each TSP instance $G_i = \{V_i, E_i, w_i, c_i\}$ so that $c_1$ now constrains the maximum subgraph cost for each $G_i$. The initialization of $C^+$ and $C^*$ chooses a sufficient value that is less than the cost of any feasible solution and the parameter `bdiv` is used to configure the step size of the search algorithm. SAT solvers typically take longer to deduce that a formula is unsatisfiable and thus it is desirable to weight the search to explore satisfiable instances first (choose `bdiv` $> 2$).

### 3.4.3 Correctness

In this section we show that metric instances are TSP-monotonic (Definition 3.4.2) and prove that CBTSP yields the correct solution for TSP-monotonic instances.

**Theorem 3.4.6.** Metric graphs are TSP-monotonic.

**Algorithm 4:** Binary Search$(G_1, G_2, \ldots, G_n, F)$

**Input:**

$G_i$: is an input graph.

$F$: is the input formula.

`bdiv`: is the config parameter for the divider (nominally 2).

**Output:**

$C^*$: is the optimal cost found by the search.

$C^- \leftarrow 0$
$C^+ \leftarrow \sum_{i=1}^{n} \max\{w_i(v_a, v_b) | \langle v_a, v_b \rangle \in E_i\} |G_i|$
$C^* \leftarrow \sum_{i=1}^{n} \max\{w_i(v_a, v_b) | \langle v_a, v_b \rangle \in E_i\} |G_i|$
**while** $C^- < C^+$ **do**
    $C \leftarrow C^+ - \left\lfloor \frac{C^+ - C^-}{\texttt{bdiv}} \right\rfloor$
    $C' \leftarrow \text{CBTSP}(G_1, G_2, \ldots, G_n, F, C)$
    **if** $C' \geq 0$ **then**
        $C^+ \leftarrow C' - 1$
        $C^* \leftarrow C'$
    **else**
        $C^- \leftarrow C + 1$
**return** $C^*$

*Proof.* We use contradiction to prove the above. Assume that there is some subset $V_1 \subset V$ and $V_2 = V_1 \cup v_j$ such that the induced subgraphs $G_1$ and $G_2$ have TSP costs $c_1 > c_2$. This means that the tour of $G_1$ can be shortened if we include the vertex $v_j$. Suppose the shortest tour of $G_2$ has the edge $\langle v_i, v_j \rangle$ in the path. We construct the graph $G'_1$ to be a copy of graph $G_1$ and replace the weights of the outgoing edges for $v_i$ with $w'_1(i,j) = w_2(i,j) + w_2(j,k)$. Now the graph $G'_1$ will have the same optimal tour cost as $G_2$ but since the edge weights of $G'_1$ compared to $G_1$ are equal or larger (triangle inequality) the optimal tour cost of $G'_1$ cannot be lower than the optimal tour cost of $G_1$.

It follows that we cannot incrementally add vertices to $V_1$ to lower the TSP cost of the new graph. Consequently, the TSP costs $c_1$ and $c_2$ for $V_1$ and $V_2$ satisfies $c_1 \leq c_2$. Therefore metric graphs are TSP-monotonic. □

**Theorem 3.4.7.** The CBTSP approach is correct (i.e., sound and complete) on TSP-monotonic decision instances.

*Proof.* We first prove that the SMT formulations used by CBTSP are sound and then we prove that the SMT solver approach (DPLL(T) on $F$) used by CBTSP is sound and complete.

The soundness of the SMT formulation follows from the definition of SAT-TSP (Problem 3.3.1) and Definition 3.4.2. Specifically, the SMT formulation allows for the negation of partial solution vertex sets and supersets for induced subgraphs that exceed the TSP budget(s). This does not remove any solutions in the search space that could have TSP costs within the budget, since the graph(s) are TSP-monotonic. Furthermore, a full SMT solution consists of TSP tour(s) of the included vertices and a satisfying assignment of $F$, which is a solution to the SAT-TSP instance. Therefore, the SMT formulation is sound, since it has the same solution set as the SAT-TSP formulation.

It follows that the solver approach for SMT instances is sound and complete, since it is based on the sound and complete algorithm DPLL(T) [63]. Therefore the CBTSP approach is correct on TSP-monotonic instances. □

### 3.4.4 Relaxing TSP-monotonicity

In this section we expand on the class of instances solvable by CBTSP (TSP-monotonic instances). Specifically, we describe a relaxation of TSP-monotonicity that can be used in practice with CBTSP when we have prior knowledge of a vertex set $A \subseteq V$ that must be included in the solution. The knowledge of this set allows us to relax the TSP-monotonicity property to apply to sets $V_1 \supseteq A$. The CBTSP solver in turn avoids partial solutions that

exclude vertices in the set $A$ and thus the correctness of negating partial solutions is still valid.

To motivate this direction, let us consider the following example. Let $G$ be a metric graph with a start vertex $v_s$ and a goal vertex $v_g$. Next, remove all the incoming edges of $v_s$ and all the outgoing edges of $v_g$ other than the one edge connecting $v_g$ to $v_s$. Also, modify the formula to be $F = F \wedge v_s \wedge v_g$. This example is not TSP-monotonic. However, the addition to the formula is so simple that the CBTSP solver's preprocessing will assign $v_s$ and $v_g$ to be true (include $v_s$ and $v_g$ in the solution). Thus all partial solutions that CBTSP checks will satisfy the TSP-monotonicity property, since the graph is otherwise metric (preprocessing happens before any calls to the TSP theory).

The set of required vertices $A$, must not conflict with the formula $F$, i.e., $F$ is satisfiable if and only if

$$F \wedge \left( \bigwedge_{v \in A} v \right)$$

is satisfiable. However, for this to work in practice, the decision to include the vertices in $A$ would need to be done in the preprocessing step of CBTSP (before the DPLL solver is called, which is not discussed in this thesis). This happens for atomic clauses (clauses with only one literal), e.g., a clause $(v)$ would result in $v$ being assigned true in the preprocessing phase of CBTSP.

**Definition 3.4.8** (A-TSP-Monotonicity)**.** Given a graph $G = \langle V, E, w \rangle$ and a subset $A \subset V$, then $G$ is *A-TSP-monotonic* if for any vertex subsets $V_1 \subset V_2 \subseteq V$ such that $A \subseteq V_1$, the induced subgraphs $G_1$ and $G_2$ have TSP costs $c_1 \leq c_2$.

**Proposition 3.4.9.** The CBTSP approach is correct on A-TSP-monotonic decision instances if the decision to include the vertices in $A$ are done so in the preprocessing phase of the solver (before DPLL is called).

*Proof.* The proof follows Theorem's 3.4.7 proof — the instance behaves like a TSP-monotonic instance once DPLL is called. □

*Remark* 3.4.10 (TSP-monotonic examples). In addition to metric graphs being TSP-monotonic, disconnected metric subgraphs are also TSP-monotonic (each disconnected component of the graph is itself a complete metric subgraph).

We can connect metric subgraphs if we use the A-TSP-monotonicity relaxation. An example of an instance that satisfies the A-TSP-monotonicity is one where we have a set of metric (TSP-monotonic) subgraphs $G_1, G_2, \ldots, G_n$ that we would like the solution to visit sequentially as shown in Figure 3.2. Here we can see that if the black locations represent the

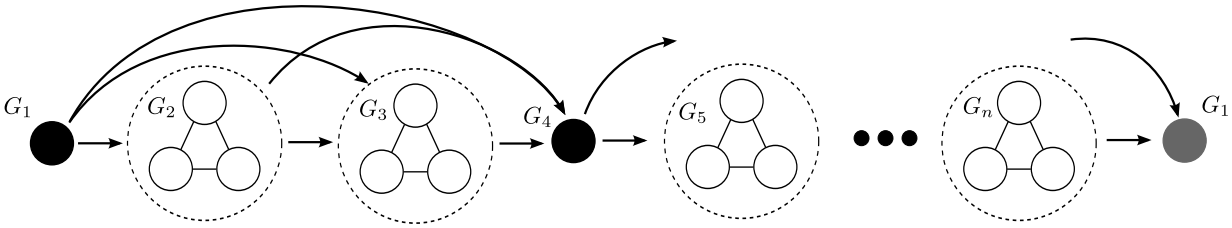



Figure 3.2: A-TSP-monotonic example. Interconnecting edges from $G_i$ to $G_j$ shown in the figure represent a collection of edges that connect every vertex in graph $G_i$ to every vertex in graph $G_j$. The edge weights of the interconnecting edges satisfy the triangle inequality.

vertices in the set $A$, then the A-TSP-monotonicity property is satisfied (the grey location represents a black location that has been duplicated). □

We now present a set of sufficient properties for achieving A-TSP-monotonicity, which we used to construct the example in Figure 3.2.

*Remark* 3.4.11 (A Sufficient Construction for Achieving TSP-Monotonicity). Given a graph $G = \langle V, E, w\rangle$ and a vertex set $A \subset V$. Let $N_{\text{out}}(v_i) = \{v | \langle v_i, v\rangle \in E\}$. Then the following is true for every edge $\langle v_i, v_j\rangle$ in $\{\langle v_i, v_j\rangle \in E | v_i \in V, v_j \in V \setminus A\}$:

1. the vertex neighbour of $v_j$ satisfies $N_{\text{out}}(v_j) \setminus \{v_i\} \subseteq N_{\text{out}}(v_i)$
2. and for every $v_k \in N_{\text{out}}(v_j) \setminus \{v_i\}$ we have $w(v_i, v_k) \leq w(v_i, v_j) + w(v_j, v_k)$.

□

## 3.5 An Integer Program Formulation

In this chapter we are interested in path planning problems that require robots to travel between multiple task locations (patrolling, sample collection, and periodic routing). Specifically, all of the path planning problems seek to find shortest tours over a subset of the vertices (not necessarily a fixed set of vertices) in each input graph. This section describes the standard method for expressing such problems as an integer linear program (ILP). The additional constraints that define the specific problem and guide the solution to choose the set of visited locations are added to the formulation in subsequent sections. The ILP formulation for the TSP aspect of the problem is drawn from the vehicle routing literature [15] and is a standard method used in the operations research community [15].

To be able to solve multiple TSP instances simultaneously, we require that each instance has a home location that must be visited. In this way we are able to eliminate sub-tours that do not visit one of the home locations.

Let the set of binary variables $e^k_{i,j} \in \{0,1\}$ represent the inclusion or exclusion of the edge $\langle v_i, v_j \rangle \in E_k$ from the solution ($e^k_{i,j} = 1$ indicates inclusion). Similarly let $v^k_i$ represent the set of Boolean variables representing the inclusion/exclusion of the vertex $v_i \in V^k$ from the solution. Additionally without loss of generality, let $v^k_1$ be the home location in each graph $G_k$. The following ILP template encodes TSP problems with multiple graphs over a non-fixed set of vertices and minimizes the max cost of the individual tours. The formulation for minimizing the total cost simply replaces the objective with $\sum_{k=1}^{n} \sum_{i=1}^{|V^k|} \sum_{j=1}^{|V^k|} w_k(v_i, v_j)\ e^k_{i,j}$.

$$\text{minimize}$$

$$c_{\max} \tag{3.1}$$

$$\text{subject to}$$

$$\sum_{k=1}^{n} \sum_{i=1}^{|V^k|} \sum_{j=1}^{|V^k|} w_k(v_i, v_j)\ e^k_{i,j} \leq C \tag{3.2}$$

$$\text{subject to, for each } k \in \{1, 2, \ldots, n\}$$

$$v^k_1 = 1, \tag{3.3}$$

$$\sum_{i=1}^{|V^k|} e^k_{i,j} = v^k_j, \qquad \text{for each } v_j \in V_k \tag{3.4}$$

$$\sum_{j=1}^{|V^k|} e^k_{i,j} = v^k_i, \qquad \text{for each } v_i \in V_k \tag{3.5}$$

$$\sum_{i \in P} \sum_{j \in P} e^k_{i,j} \leq \sum_{i \in P} v^k_i - v^k_l,$$

$$\text{for each } P \subseteq \left\{2, \ldots, |V^k|\right\}, \text{ and some } l \in P \tag{3.6}$$

$$\sum_{i=1}^{|V^k|} \sum_{j=1}^{|V^k|} w_k(v_i, v_j)\ e^k_{i,j} \leq \min(c_k, c_{\max}) \tag{3.7}$$

Constraint (3.2) ensures the total solution cost is within the budget $C$. Constraint (3.3) forces the solution to include the home vertices. Constraints (3.4) and (3.5) restrict the

incoming and outgoing degree for each location to be one only if the vertex is included. Constraint (3.6) is the sub-tour elimination constraint, where a sub-tour that visits the set $P$ but not location 1 is eliminated by ensuring that the number of edges between vertices in $P$ cannot equal $|P|$ (if each vertex in $P$ is visited then $\sum_{i \in P} v_i^k - v_l^k = |P| - 1$). Otherwise if location $l$ is not visited then the inequality is satisfied by the other constraints (each included vertex has at most one incoming and one outgoing edge). In practice the sub-tour elimination constraint is lazily implemented (if a constraint in (3.6) is violated during the construction of the solution, then the constraint is added to the formulation) to avoid expressing an exponential number of constraints. Finally, Constraint (3.7) ensures the individual tours are within the budgets $c_k$ and it ensures that the individual tour costs are equal or lower than the optimal value $c_{\max}$.

## 3.6 Robotic Applications and Expressions

In this section we describe three robotic path planning problems (patrolling, sample collection, and periodic routing) and show how they are expressed in both SAT-TSP and ILP. The SAT-TSP expressions for these problems utilize at-most-one-in-a-set constraints. We define this type of constraint here and use it throughout the rest of the section.

**Definition 3.6.1** (At-most-one-in-a-set constraint)**.** Given a set $X' \subset X$ of variables, the *at-most-one-in-a-set constraint* states that at most one variable in the set $X'$ can be assigned true. The following Boolean formula expresses the constraint:

$$\bigvee_{x,y \in X' | x \neq y} \neg(x \wedge y).$$

*Remark* 3.6.2 (Expressions). The expressions presented in this section are the expressions that we have found perform the best. As with the MTZ ILP expression for TSP [57], one can learn later that better expressions exist.

### 3.6.1 The Patrolling Problem

We consider patrolling problems that require each point of interest to be observed from multiple viewpoints. These problems were inspired by patrolling problems that allow the points of interest to be observed from multiple viewpoints [65, 7]. Our patrolling problem is defined on a metric environment that contains $m$ points of interest, $n - 1$ "observation

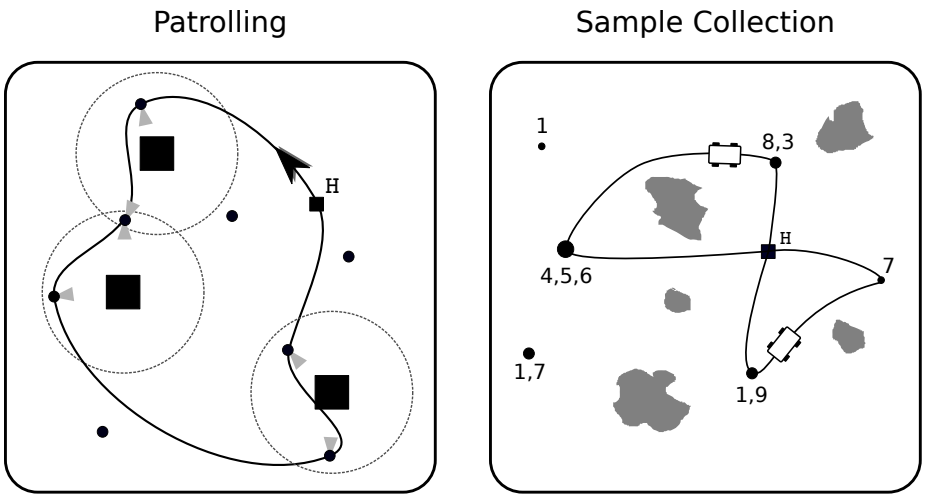



Figure 3.3: On the left is a UAV patrolling example and on the right is a UGV sample collection example. The robot's path is indicated with a solid line in both illustrations and the location marked by an "H" represents the robot's home. For the patrolling problem, points of interest are buildings represented by large squares, faint circles represent the radius that the building can be observed from, and the grey triangles indicate the different observation perspectives. In the sample collection problem, the labels above the locations represent the mineral types that are within the sample (large samples have three minerals and small samples have one). The grey contours represent obstacles.

locations" and one home location. The robot must visit a subset of the observation locations to observe all of the points of interest and return home. A point of interest $p$ is observable from a location $v$ if the distance between $p$ and $v$ is less than or equal to a threshold (provided by the user). Each point of interest $p$ must be observed by at least two complementary locations, referred to as a "complementary pair". An observation location $v_j$ is complementary to $v_i$ if both points observe $p$ from perspectives that are separated by a minimum threshold angle (provided by the user).

A solution is a tour that visits a set of observation locations with minimum length such that each point of interest is observed by at least one complementary pair. An illustrative example of this problem is given in Figure 3.3 (left illustration).

To aid in the expression of the problem, we let the set $V_p$ represent the set of observation locations that can observe the point of interest $p$. The set $V_{p,i}$ represents the set of locations that are complementary observation locations of $v_i$ for $p$ and the set $P = \{1, 2, \ldots, m\}$ represents the points of interest.

**SAT-TSP expression** This problem is encoded into SAT-TSP by constructing a formula $F$ that captures the logic of visiting the home location and at least one complementary pair for each point of interest. The clause

$$(v_h)$$

captures the home location requirement ($v_h$ is the robot's home) and the set of clauses

$$\bigvee_{v_i \in V_p} v_i \wedge \left( \bigvee_{v_j \in V_{p,i}} v_j \right), \text{ for each } p \in P$$

captures the logic of visiting at least one complementary pair for each point of interest ($p$ is observed from $v_i$ only if it is observed by at least one of its complements $v_j$). Now the tuple $\langle G, F, C \rangle$ encodes the patrolling problem as a SAT-TSP instance, where $G$ is the discrete graph representing the distances between observation locations and $C$ represents the maximum cost of the solution tour (finding the minimum $C$ solves the optimization problem).

**ILP expression** To express this problem as an ILP, we build upon the formulation given in Section 3.5. Let the set of Binary variables $v_{p,i} \in \{0, 1\}$ represent whether or not $v_i$

is part of a complementary pair observing the point of interest $p$. The following are the additional constraints in the ILP formulation.

$$\text{for each } p \in \{1, 2, \ldots, m\}$$

$$v_{p,i} \leq v_i, \qquad \text{for each } v_i \in V_p \tag{3.8}$$

$$v_{p,i} \leq \sum_{v_j \in V_{p,i}} v_j, \qquad \text{for each } v_i \in V_p \tag{3.9}$$

$$\sum_{v_i \in V_p} v_{p,i} \geq 1 \tag{3.10}$$

Constraint (3.8) restricts the indicator $v_{p,i}$ from being true if the location $v_i$ itself is not visited, Constraint (3.9) restricts the indicator from being true if none of the complementary locations of $v_i$ are visited, and Constraint (3.10) ensures that at least one complementary pair is visited.

### 3.6.2 The Sample Collection Problem

The following problem is inspired by sample collection problems that arise for science rovers [26]. In this problem, we have a set of $R$ robots, $n-1$ samples, one home location, and a set of $m$ different minerals that can appear in the $n-1$ samples. Each sample in the environment is either small, medium, or large. Small samples have within them one type of mineral, medium samples have up to two different minerals, and large samples have up to three different minerals. There are two types of robots used to collect samples, small and large. A small robot can collect an unlimited number of small samples and up to one medium sample but no large samples. A large robot cannot collect small samples, but it can collect an unlimited number of medium samples and up to one large sample. The small and large robots may have different speeds. Each robot is given the same time budget to collect samples and return home. Since each location only contains one sample, multiple robots are restricted from visiting the same location.

A solution to this problem is a tour for each robot that satisfies the time budget, starts at the home location, and visits a set of locations in the environment that allows the robots to collect a set of samples that contain one of each type of mineral found in the environment. An optimal solution minimizes the total time taken by the set of robots. An illustrative example of this problem is given in Figure 3.3 (on the right). In this example

there are no samples with mineral 2 and so feasible solutions do not collect mineral 2. Additionally, we demonstrate a sample collection solution in a demo video[1]. This solution utilizes one small and one large robot to collect seven different mineral types.

To express this problem we introduce the following. Let $V_t$ be the set of locations that contains a mineral of type $t$; let $V_S, V_M$ and $V_L$ be the set of locations that respectively have small, medium and large samples; let $R_S, R_L$ and $R = \{1, 2, \ldots, |R|\}$ respectively be the set of small, large, and all robots; and let $T = \{1, 2, \ldots, m\}$ be the set of minerals.

**SAT-TSP expression** The problem is encoded as a SAT-TSP instance by constructing a graph $G_r$ for each robot $r$ and a formula $F$ that captures the goals and capacity restrictions of the robots. Each graph $G_r$ contains all of the locations in the environment and its edge weights are given by transitions for robot $r$ to move within its environment. The set of variables $v_{i,r}$ is used to indicate if robot $r$ visits location $i$. When the location/robot pair is incompatible, such as location $i$ contains a large sample and robot $r$ is small, the variable is negated. We chose this approach instead of constructing graphs without incompatible location/robot pairs to simplify the SAT-TSP expression in this chapter.

We start our expression by forcing each robot to visit the home location using the clauses

$$\bigwedge_{r \in R} v_{h,r},$$

where $v_h \in V$ is the home location for the robots. Next, we negate all incompatible location/robot pairs (e.g., small robot large sample)

$$\bigwedge_{v_{i,r} \in X_I} \neg v_{i,r},$$

where

$$X_I = \{v_{i,r} | v_i \in V_L, r \in R_S\} \cup \{v_{i,r} | v_i \in V_S, r \in R_L\}.$$

We encode the goal of collecting at least one of each mineral (if the mineral exists), with a disjunctive clause for each mineral

$$\bigwedge_{t \in T} \left( \bigvee_{v_i \in V_t, r \in R} v_{i,r} \right).$$

[1] `https://ece.uwaterloo.ca/~sl2smith/SAT-TSP/demo.mp4`

To restrict multiple robots from visiting the same location $v_i$, we use at-most-one-in-a-set constraints (Definition 3.6.1) for the sets $\{v_{i,r} | r \in R\}$ for each $v_i \in V$,

$$\bigwedge_{x,y \in R | x \neq y} \neg(v_{i,x} \wedge v_{i,y}), \text{ for each } v_i \in V.$$

Finally, we restrict the robots' carrying capacity with at-most-one-in-a-set constraints on the sets $\{v_{i,r} | v_i \in V_M\}$ for each small robot $r \in R_S$ and $\{v_{i,r} | v_i \in V_L\}$ for each large robot $r \in R_L$:

$$\bigwedge_{v_i, v_j \in V_M | i \neq j} \neg(v_{i,r} \wedge v_{j,r}), \text{ for each } r \in R_S$$

and

$$\bigwedge_{v_i, v_j \in V_L | i \neq j} \neg(v_{i,r} \wedge v_{j,r}), \text{ for each } r \in R_L.$$

The tuple $\langle G_1, G_2, \ldots, G_{|R|}, F, C \rangle$ now encodes the problem as a SAT-TSP instance, where $G_r = \langle V_r, E_r, w_r, c_r \rangle$ is the discrete graph for robot $r$ that captures the time transitions, $c_r$ is the robot's time budget, $F$ captures the goals and constraints of the problem, and $C$ encodes the maximum total time incurred by all of the robots. The total time budget $C$ can be minimized to find the optimal solution.

**ILP expression** To express this problem as an ILP, we build upon the formulation given in Section 3.5. Similar to the SAT-TSP expression we negate incompatible location/robot pairs instead of altering the previously established ILP formulation. For example, a location $i$ would be incompatible with the small robot $r$ if it contained a large sample. The incompatibility is negated with the assignment $v_i^r = 0$ (notation defined in Section 3.5). The following is the extension of the ILP formulation:

$$\sum_{v_i \in V_t} \sum_{r \in R} v_i^r \geq 1, \qquad \text{for each } t \in \{1, 2, \ldots, m\} \tag{3.8}$$

$$\sum_{r \in R} v_i^r \leq 1, \qquad \text{for each } v_i \in V \tag{3.9}$$

$$\sum_{v_i \in V_M} v_i^r \leq 1, \qquad \text{for each } r \in R_S \tag{3.10}$$

$$\sum_{v_i \in V_L} v_i^r \leq 1, \qquad \text{for each } r \in R_L \tag{3.11}$$

$$v_i^r = 0, \qquad \text{for each } v_i^r \in I \tag{3.12}$$

where $I = \{v_i^r | v_i \in V_L, r \in R_s\} \cup \{v_i^r | v_i \in V_S, r \in R_L\}$ is used to represent the set of incompatible location/robot pairs. Constraint (3.8) encodes the goal of retrieving at least one of each mineral type (if it exists), Constraint (3.9) restricts multiple robots from retrieving the same sample, Constraint (3.10) restricts small robots from collecting more than one medium sample, Constraint (3.11) restricts small robots from collecting more than one large sample, and Constraint (3.12) negates the incompatible location/robot pairs.

### 3.6.3 The Period Routing Problem

The period routing problem [11, 100] requires a robot (vehicle) to service a set of $n-1$ locations over a set of $m$ periods. In each period the robot starts from a designated home location and travels to a subset of the service locations. Each location $v$ requires $f(v) \leq m$ out of $m$ periods of service and has restrictions on which periods or period combinations are allowed. For this chapter, locations restrict back to back service (the first and last period are also considered back to back). This problem arises in collection [100] and delivery [11] tasks. Also period routing problems often contain a capacity that limits the number of tasks that can be performed.

A solution to this problem is a set of tours, one for each period that meets the service demands of the locations, respects the capacity limits, and respects the service restrictions. The optimal solution minimizes the maximum length tour over the $m$ periods.

**SAT-TSP expression** The problem is encoded as a SAT-TSP instance by constructing a graph for each period and a formula that captures the goals and restrictions of the problem. Each graph $G_p$ represents the transition costs for the robot to move within its environment during period $p$ (all graphs are the same).

The problem goals of providing a specific number of service periods to the locations is captured with the standard technique of using adder circuits [55], which are efficiently translated to a Boolean formula [41] and added to $F$. These adder circuits take as input the set of location/period variables for each location $\{v_{i,p} | p \in P\}$ and output a set of Boolean variables $\{b_{i,1}, b_{i,2}, b_{i,4}, \ldots\}$ that capture the binary encoding of how many of the inputs are true. Following that, the outputs of the adder circuit are forced to the desired service demands, $f(v_i)$. As an example, if $v_i$ requires two visits, then we force the twos bit, $b_{i,2}$, to be true and the remaining bits to be false. This is accomplished by adding the following to $F \leftarrow F \wedge \neg b_{i,1} \wedge b_{i,2} \wedge \neg b_{i,4} \wedge \ldots$.

We force the robot to visit the home locations with the clause

$$\bigwedge_{p \in P} v_{h,p},$$

where $v_h \in V$ is the home location of the robot. The back to back period restriction is exhaustively handled by negating every possible illegal combination.

$$\neg(v_{i,p} \wedge v_{i,p^+}) \text{ for all } v_i \in V, p \in P,$$

where $p^+ = (p \mod m) + 1$.

The tuple $\langle G_1, G_2, \ldots, G_m, F, C \rangle$ now encodes the problem as a SAT-TSP instance, where: $G_p = \langle V_p, E_p, w_p, c_p \rangle$ is the discrete graph for period $p$; the cost budget $C$ is set to $\infty$ to indicate that there is no budget; $F$ captures the goals and constraints of the problem; and $c_p$ encodes the maximum cost of graph $G_p$'s path which is minimized to find the optimal solution.

**ILP expression** To express this problem as an ILP, we build upon the formulation given in Section 3.5. The following are the additional constraints in the ILP formulation,

$$\sum_{p \in P} v_i^p = f(v_i), \qquad \text{for each } v_i \in V \tag{3.8}$$

$$v_i^p + v_i^{p^+} \leq 1, \qquad \text{for each } v \in V \text{ and } p \in P \tag{3.9}$$

where $p^+ = (p \mod m) + 1$. Constraint (3.8) encodes the goal of servicing each location the proper number of times and Constraint (3.9) restricts back to back visits.

## 3.7 Experiments

In this section we present simulation results that compare CBTSP [2] to an ILP solver on the instances presented in Section 3.4. The problem instances in this section all have metric travel costs and thus CBTSP are solvable by CBTSP (metric instances are TSP-monotonic).

[2] CBTSP is available at `https://github.com/fcimeson/cbTSP`

### 3.7.1 Simulations

We ran all simulations on an Intel Core i7-4600U, 2.10GHz with 16GB of RAM. The CBTSP solver used a custom DPLL(T) solver (based on MINISAT [90]) to test partial solutions by making external callbacks to a TSP solver (a version of LKH [32] we modified to solve decision TSP problems). The CBTSP solver takes as input the SAT-TSP instance, a time budget, and a set of parameters used to configure the solver. The solver parameters are used to configure the behaviour of CBTSP's SAT and TSP solvers as well as a few CBTSP specific behaviours. The details of these parameters and their values are given in Appendix A. The ILP solver, Gurobi [66] was accessed through Python, which also takes as input a time budget. To make a fair comparison we restricted Gurobi to a single process (thread). Additionally, we seeded the solver with a random seed each time it was called to ensure each run is different from the last (set using Gurobi's parameters).

All of our simulations use a 300 second time budget and the best solution found within the budget is reported in our results. As well, we track the solvers' progress within its time budget to compare the solvers' results as the results are found. The use of a fixed time budget allows us to simulate real world conditions where the robot must make decisions while it operates as opposed to letting the solvers run to completion overnight (or longer). In the latter case of running the solvers to completion, comparing solution quality would be meaningless since both approaches would likely find the optimal solution. Thus, using a fixed time budget allows us to compare the solver's ability to find solutions quickly. For the simulations presented in this section both solvers typically consume their entire time budget, thus we do not explicitly report solver times as they would all be 300 seconds. Instead we compare the solution quality in a table that averages ten runs for each instance and track the solution quality found over time using a series of plots.

The time budget does not include the time taken to create the SAT-TSP or ILP instances (the reduction times) as we do not wish to benchmark this process. However, the reduction times are polynomial with respect to the input size.

### 3.7.2 Patrolling

The patrolling problem instances were generated as follows. All of the locations (the $n-1$ observation locations, the robot's home, and the $m$ points of interest) were uniformly randomly distributed in a $1000 \times 1000$ meter two dimensional square. A point of interest $p = (x_p, y_p)$ is observable from location $v = (x_v, y_v)$ if

$$|p - v| \leq \frac{4000}{5m\sqrt{m}},$$

| | | | Gurobi | | cbTSP | |
|---|---|---|---|---|---|---|
| No. | $n$ | $m$ | Best | Avg. Cost | Best | Avg. Cost |
| 1 | 40 | 5 | 2,268 | 2,292 | **2,260** | **2,260** |
| 2 | 40 | 10 | 1,812 | 1,812 | 1,812 | 1,812 |
| 3 | 40 | 20 | 3,171 | 3,171 | **3,111** | **3,111** |
| 4 | 60 | 7 | 1,825 | 1,925 | **1,801** | **1,801** |
| 5 | 60 | 15 | 2,615 | 2,845 | **2,430** | **2,430** |
| 6 | 60 | 30 | 3,252 | 3,346 | **3,176** | **3,176** |
| 7 | 80 | 10 | 2,020 | 2,227 | **1,903** | **1,903** |
| 8 | 80 | 20 | 2,948 | 4,681 | **2,533** | **2,602** |
| 9 | 80 | 40 | 4,640 | 6,920 | **3,244** | **3,585** |
| 10 | 100 | 12 | 2,398 | 3,265 | **1,937** | **1,939** |
| 11 | 100 | 25 | 2,825 | 4,324 | **2,431** | **2,473** |
| 12 | 100 | 50 | 7,385 | 8,273 | **3,803** | **4,104** |

Table 3.1: Experimental results for patrolling problem instances (300 second trials). The left three columns indicate the patrolling instance number, the number of observation locations ($n$), and the number of points of interest ($m$) in the environment. The best results are shown in bold.

for which the equation was designed to help ensure that most randomly generated instances will have a feasible solution (one where each point of interest can be observed by a complementary pair). A location $u$ is complementary to $v$ for $p$ if

$$(\theta_v - \theta_u) \mod 360 \geq 60,$$

where

$$\theta_v = \tan^{-1}\left(\frac{y_p - y_v}{x_p - x_v}\right) \text{ and } \theta_u = \tan^{-1}\left(\frac{y_p - y_u}{x_p - x_u}\right).$$

An illustrative example of this problem is given in Figure 3.3 (on the left).

We compared CBTSP to the ILP solver, Gurobi, on a set of 12 instances as shown in Table 3.1 and Figure 3.4. As we can see from the table, CBTSP outperforms Gurobi on almost every instance and as the instances get more difficult (larger instances) the performance gap widens. This is seen by comparing the first and last instances in the table — Gurobi has an average cost of 0.99 times more than CBTSP on the first instance and an average cost of 1.69 on the last. In the figure we see that CBTSP is able to almost instantly find high quality solutions, where Gurobi requires around 10 seconds to solve the majority of the runs and it uses its remaining time to improve its average solution quality from a factor of 2 to a factor of 1.5.

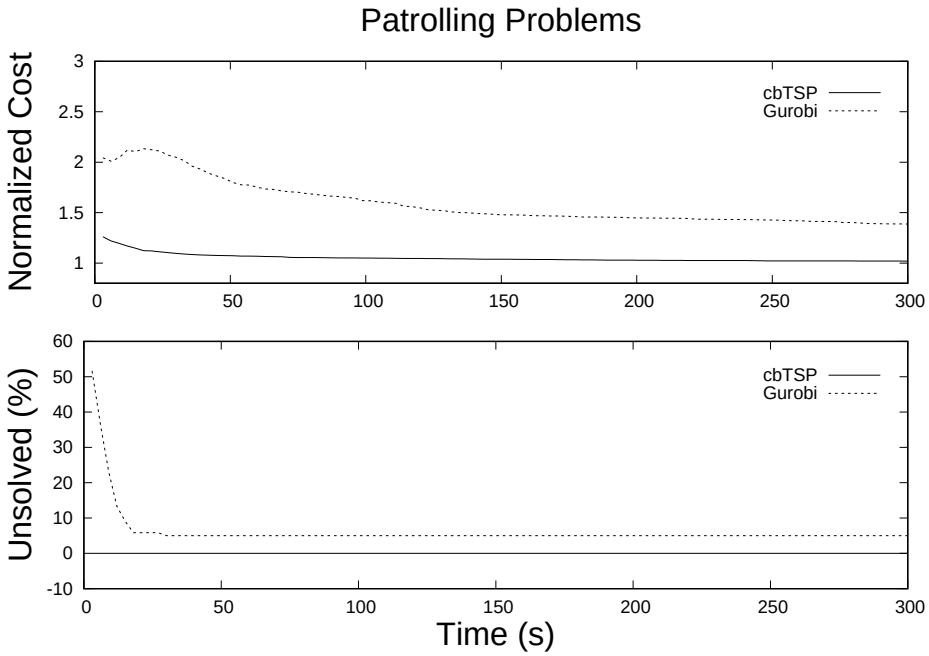



Figure 3.4: The top plot captures the average normalized solution quality obtained during the time budget for patrolling instances solved by CBTSP and Gurobi. The bottom plot captures the % of unsolved runs (10 runs per instance) over the time budget.

### 3.7.3 Sample Collection

The sample collection problem instances we tested were generated as follows. There is one home location and $n-1$ sample locations in the environment and the set of samples contain up to $m$ different mineral types. All locations (sample locations and the home location) were uniformly randomly distributed in a $1000 \times 1000$ meter two dimensional square. The distribution of sample sizes are as follows: 3:1 for small to large and 2:1 for small to medium. The types of minerals in each sample are uniformly randomly distributed (the same mineral type may reoccur in a sample). There are two types of robots used to collect samples, small and large, three of each, six in total. Each robot has a time budget of 25 minutes to collect samples and return home and the small robots travel at a speed of 2m/s, while the large robots travel at half that speed (1m/s).

**Note 3.7.1.** Although the large robots can complete the task by collecting about half the samples, it costs double to travel the same distance. Furthermore, it is equally likely for a mineral type to be found in a small, medium, or large sample.

We compared CBTSP to the ILP solver, Gurobi, on 12 sample collection instances and reported the data in Table 3.2 and Figure 3.5. The results show that CBTSP is competitive with Gurobi. Specifically, both solvers find the same quality solutions for the easy instances (instances 1-3 in the table — the smaller instances); then as the instances move into the medium difficult range (instances 4-7), CBTSP starts to outperform Gurobi; and once the instances become difficult (instances 8-12 — the largest instances) CBTSP often outperforms Gurobi by quite a bit. Here, Gurobi only outperforms CBTSP on one instance and fails to find feasible solutions for two out of five difficult instances. From Figure 3.5, we see that CBTSP is able to solve each run within the first 20 seconds, while Gurobi is not able to find feasible solutions for more than 20% of the runs. Additionally, we see that at first CBTSP finds some pretty low quality solutions (2-4.5 times the best known) and it takes CBTSP about 100 seconds to improve this quality to within 1.5 of the best known. Although, not shown here, the runs that take longer to improve are the larger/more difficult instances, some of which do not show up in Gurobi's average because Gurobi was not able to find feasible solutions. Thus, CBTSP is able to solve more instances and find higher quality solutions than Gurobi.

**Physical simulations** We have also performed simulation experiments for sample collection in more complex environments where travel costs are shortest collision free paths. These simulations utilize the Clearpath Husky robot model within Gazebo. The robots use the ROS navigation stack [75] to move within the physical simulator. An example

| | | | Gurobi | | cbTSP | |
|---|---|---|---|---|---|---|
| No. | $n$ | $m$ | Best | Avg. Cost | Best | Avg. Cost |
| 1 | 10 | 10 | 3,500 | 3,500 | 3,500 | 3,500 |
| 2 | 20 | 10 | 3,355 | 3,356 | 3,355 | **3,355** |
| 3 | 20 | 20 | 8,563 | 8,563 | 8,563 | 8,563 |
| 4 | 40 | 10 | 1,876 | 1,893 | 1,876 | **1,880** |
| 5 | 40 | 20 | 5,251 | 5,429 | **5,117** | **5,397** |
| 6 | 40 | 40 | 9,117 | 9,682 | **8,812** | **9,235** |
| 7 | 60 | 10 | 1,613 | 1,730 | **1,591** | **1,603** |
| 8 | 60 | 30 | - | - | **11,809** | **13,077** |
| 9 | 80 | 10 | 1,382 | 1,580 | **1,299** | **1,332** |
| 10 | 80 | 40 | 10,941 | **11,009** | **10,453** | 12,073 |
| 11 | 100 | 10 | 1,883 | 2,863 | **1,363** | **1,419** |
| 12 | 100 | 50 | - | - | **14,668** | **14,949** |

Table 3.2: Experimental results for sample collection instances (300 second trials). On the left we indicate the instance number, the number of samples ($n$), and the maximum number of different minerals ($m$) in the environment. The best results are shown in bold. Results that are shown with a dash indicate that the solver was not able to solve the instance.

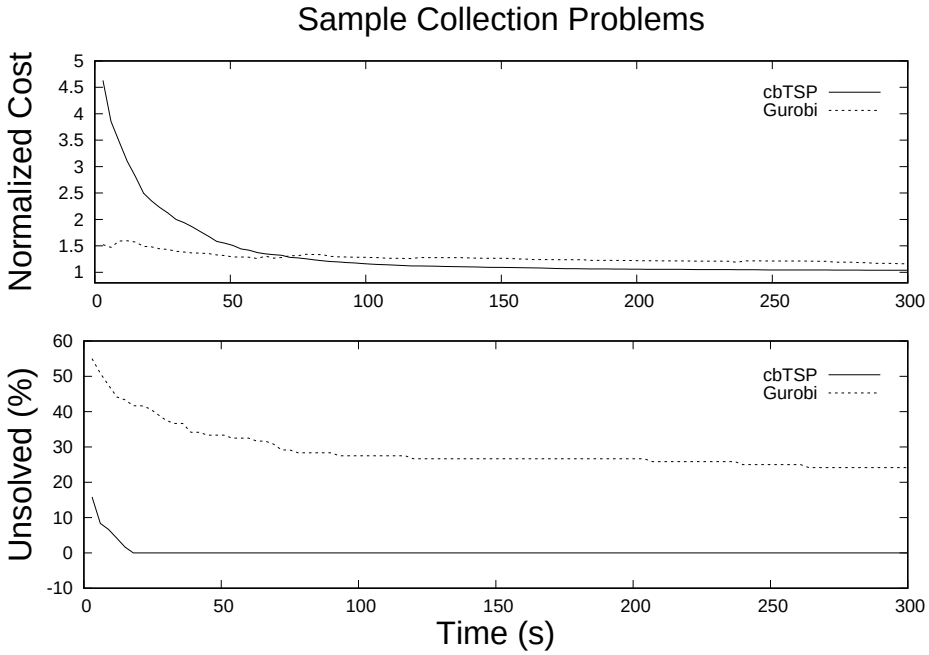



Figure 3.5: The top plot captures the average normalized solution quality obtained during the time budget for sample collection instances solved by CBTSP and Gurobi. The bottom plot captures the % of unsolved runs (10 runs per instance) over the time budget.

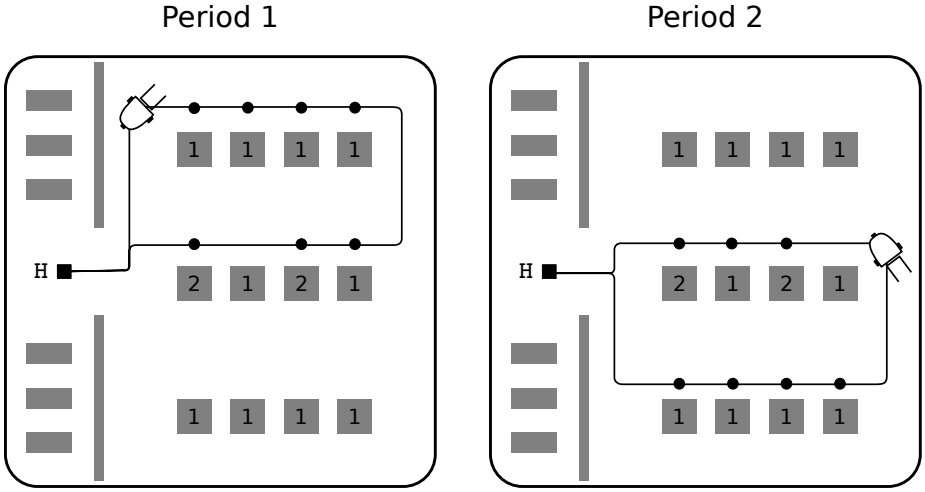



Figure 3.6: A simple period routing example for material transport within a factory. There are only two periods of service and a location can either require service for one or both periods (as indicated on the graph). The home location is labelled with an "H".

simulation, visualized with RViz is shown in Figure 1.1 and demoed in our video[3]. In the video there are two robots, each with different speed. The slower Husky emulates the large sample collecting robot and the faster Husky emulates the small robot. The collection task requires collecting seven different mineral types from the environment.

### 3.7.4 Period Routing

The period routing problem instances were generated as follows. All of the locations (the $n-1$ service locations and the robot's home) were uniformly randomly distributed in a $1000 \times 1000$ meter two dimensional square. There are six service periods; each service location may require either one, two, or three periods of service (uniformly randomly assigned) out of the six periods. An illustrative example is shown in Figure 3.6.

We have compared CBTSP to the ILP solver Gurobi, on 12 period routing instances, for which the results can be found in Table 3.3 and Figure 3.7. The table shows that CBTSP outperforms Gurobi on all of the instances and like the other two applications, as the instances become more difficult the performance gap between CBTSP and Gurobi grows. This can be seen as a trend between the first and last instance in the table. Gurobi is on

[3] `https://ece.uwaterloo.ca/~sl2smith/SAT-TSP/demo.mp4`

| | | Gurobi | | cbTSP | |
|---|---|---|---|---|---|
| No. | $n$ | Best | Avg. Cost | Best | Avg. Cost |
| 1 | 15 | 2,278 | 2,392 | **2,212** | **2,212** |
| 2 | 15 | 1,968 | 2,055 | **1,904** | **1,904** |
| 3 | 20 | 2,200 | 2,382 | **2,022** | **2,022** |
| 4 | 20 | 1,807 | 2,016 | **1,670** | **1,670** |
| 5 | 25 | 2,706 | 3,718 | **2,310** | **2,347** |
| 6 | 25 | 2,638 | 2,869 | **2,057** | **2,057** |
| 7 | 30 | 2,877 | 3,133 | **2,342** | **2,358** |
| 8 | 30 | 3,040 | 3,811 | **2,433** | **2,439** |
| 9 | 35 | 3,738 | 3,944 | **2,502** | **2,610** |
| 10 | 35 | 3,667 | 3,902 | **2,404** | **2,506** |
| 11 | 40 | 4,369 | 4,680 | **2,727** | **2,954** |
| 12 | 40 | 3,934 | 4,529 | **2,594** | **2,678** |

Table 3.3: Experimental results for period routing problem instances (300 second trials). The left two columns indicate the instance number and the number of locations in the environment. The best results are shown in bold.

par with CBTSP for the first instance and then for the last instance Gurobi has a solution quality that is 3.66 times worse than CBTSP's. From the figure, we see that CBTSP is able to find feasible solutions for all of the runs within the first 10 seconds, while in the same time frame Gurobi was only able to find feasible solutions for approximately 70% of the runs. Additionally, Gurobi struggled finding high quality solutions for these problems, while CBTSP was able to find reasonably good quality solutions within the first 60 seconds.

## 3.8 Summary

In summary, this chapter provided an alternative approach for solving high-level path planning problems, called SAT-TSP. We used the SAT-TSP problem as the modelling language for expressing high-level path planning problems. We provided the SAT-TSP solver CBTSP for finding path planning solutions and proved its correctness for TSP-monotonic instances. Additionally, we provided a relaxation of TSP-monotonicity that expanded the set of solvable problems by CBTSP. Through a series of experiments we demonstrated that CBTSP often outperforms a commercial grade ILP solver — especially on difficult instances. Thus showing that CBTSP is a strong candidate for solving discrete path planning problems. The CBTSP solver can be downloaded from `https://github.com/fcimeson/cbTSP`.

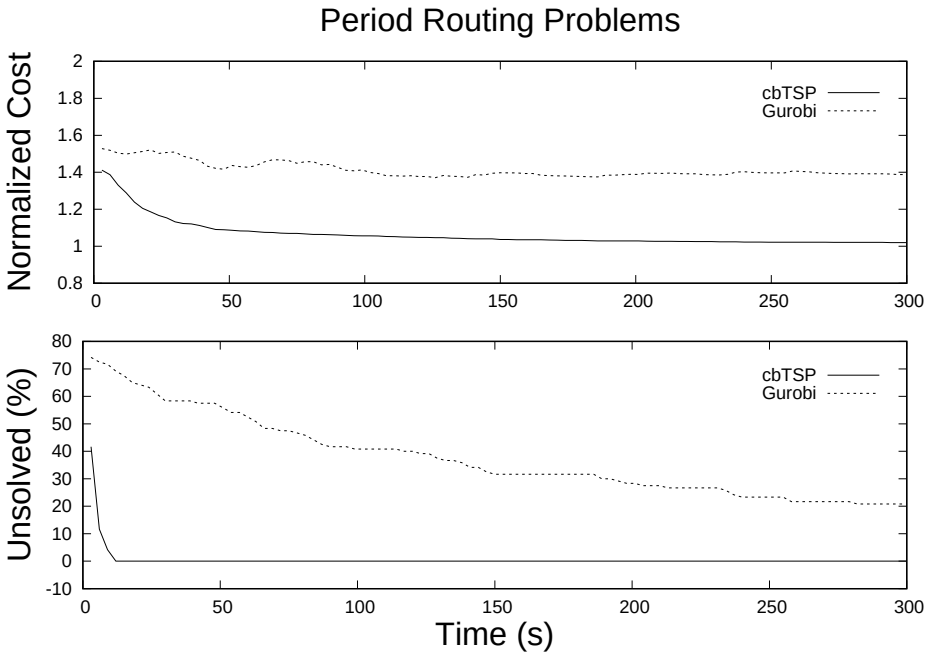



Figure 3.7: The top plot captures the average normalized solution quality obtained during the time budget for period routing instances solved by CBTSP and Gurobi. The bottom plot captures the % of unsolved runs (10 runs per instance) over the time budget.

# Chapter 4

# Pruning Solutions

This chapter introduces two approaches for pruning path planning solutions from the search space. These approaches are used in conjunction with existing path planning solvers to improve the solver's efficiency. The boost in efficiency comes at the expense of solution quality. However, we show in this chapter that solutions found using these approaches are within a constant factor of the optimal. We focus on the class of discrete path planning problems that do not contain ordering constraints.

The first approach, *coupled planning*, starts by partitioning the environment into a set of clusters/regions. Then it creates a new version of the problem, *the clustered version*, to restrict how feasible paths visit the set of clusters. This restriction prunes a significant portion of the solution search space.

The second approach, *hierarchical planning*, is an extension of the first approach that further prunes the solution search. This approach uses the clusters to decompose the problem into a hierarchy of small independent problems, where each problem represents a coarse approximation of the environment or sub-region. The solver starts by building a path on the coarsest level and then extends its solution to the next level by visiting the nested components of its coarsened path. This process repeats until a solution is found for the full problem.

These approaches achieve their solution quality guarantees (bounds) from how they partition the environment. Specifically, these approaches use a new clustering/partitioning method called Γ-Clustering that we developed specifically for path planning. A Γ-Clustering is a collection of clusters that have their locations close to each other and far from all other locations. See Figure 4.1 for a sample Γ-Clustering of an office environment (the 5th floor of the E5 building at the University of Waterloo). The parameter Γ is used

to specify the degree of separation that each Γ-Cluster must achieve (distance from other locations). Unlike other clustering methods, the Γ-Clustering method does not take in as input, an integer $k$ for finding the best $k$ set of clusters. Instead it takes in as input the parameter Γ and finds the Γ-Clusters within the environment that achieve the standards imposed by the Γ parameter. Thus some problems contain many Γ-Clusters while others contain little to none.

The contributions of this chapter are as follows. We introduce a new clustering method, Γ-Clustering, that is specifically designed for discrete path planning. We provide an efficient algorithm based on minimum spanning trees (MST) for computing the optimal clustering and prove that this clustering is unique. We introduce the coupled planning approach as a method for pruning the search space and prove that solutions found using this approach are within a constant factor of $\min(2, 1 + \frac{3}{2\Gamma})$ of the optimal. We then introduce hierarchical planning with Γ-Clusterings and prove that solutions found by this method are within a constant factor of $\min(2 + \frac{4}{\Gamma}, 1 + \frac{13}{2\Gamma})$ of the optimal. Our benchmark compares these two approaches to a standard non-clustered approach, where we use an ILP solver with a time budget of 900 seconds to find path planning solutions. The results show that the path quality obtained by these two methods are much closer to the optimal cost than the constant factor bounds. Most solutions obtained from our approaches were within 10% of the optimal, where the coupled approach was able to outperform the hierarchical approach on the easy instances. For the instances that finished within the allotted time budget, the coupled and hierarchical approaches showed a substantial time savings compared to the non-clustered approach. For the instances that consumed the time budget, the clustered approaches were able to maintain their performance longer than the non-clustered approach, where the hierarchical approach was able to maintain its performance longer than the coupled approach. Additionally, our benchmark compares the quality of Γ-Clusterings to clusterings found with five different methods. The results for these comparisons show that when planning with clusters, Γ-Clusterings produce higher quality solutions than other types of clusterings methods.

The rest of this chapter is organized as follows. Section 4.1 reviews the related work for these approaches. Section 4.2 provides the background needed for this chapter. Section 4.3 defines the set of path planning problems considered by this chapter. Section 4.4 formally defines what Γ-Clusters are and how to find them. Section 4.5 defines the coupled planning approach, then Section 4.6 introduces decoupled planning as a lead-in to hierarchical planning presented in Section 4.7. The benchmark is presented and discussed in Section 4.8. The chapter summary is given in Section 4.9.

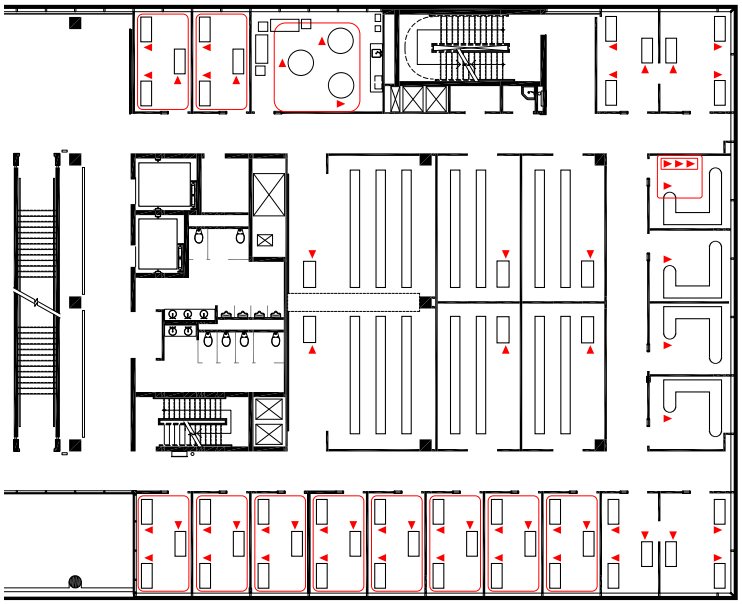


Figure 4.1: The results of Γ-Clustering used on an office environment. The red triangles represent locations of interest and the red boxes surround clusters of size two or greater.

## 4.1 Related Work

This section builds upon Section 1.1 to provide a more in depth review of the literature related to this chapter.

There are a number of pruning techniques for reducing the search space, such as branch and bound [51, 13]. Like our approach, these approaches can sometimes be used in tandem with each other to prune larger regions of the search space. In fact, the *hierarchical planning* approach presented in this chapter is a combination of the *coupled planning* approach with a hierarchical approach.

The general form of hierarchical planning decomposes the environment into regions and plans those regions as approximations instead of planning on the full resolution problem. This practice has been done under the guise of hierarchical highways [82], cell decomposition [104, 9], hierarchical planning with maps [105], and multi-resolution planning [43, 53]. Most of these approaches are used for point-to-point planning. However, there is a body of work that addresses solving vehicle routing problems, such as TSP for Euclidean environments [22, 31, 58, 5]. Our approach goes beyond Euclidean environments and the TSP, as it handles metric environments and works for a class of vehicle routing problems that

encompasses TSP.

Multilevel refinement is a framework similar to hierarchical planning and has been used to solve vehicle routing problems. At its most basic level, the framework recursively coarsens a problem into a hierarchy of approximations, where each coarsening creates a smaller problem than the last. Typically, a solution is found for the coarsest level of the problem and projected back onto its predecessor. Here, the solution can be refined to improve its quality before projecting it back onto its predecessor. The authors of [71] used multilevel refinement to more effectively guide a variable neighbourhood search for period routing problems. In [79], the authors used multilevel refinement to solve the capacitative vehicle routing problem by fixing cheap edges in the path to concentrate on minimizing large edges. In [97], they developed a travelling salesman problem (TSP) solver using multilevel and the Lin-Kernighan heuristic [54] to improve upon a state-of-the-art TSP solver, LKH [32]. A key difference in our approach is that we identify a hierarchy of sub-problems that can be solved independently and provide solution quality guarantees.

The typical motivation for using clustering is to find structure within data and/or compress the data [42]. One of clustering's most popular uses is data mining. Applications range from hand writing analysis [42], to understanding retail customer behaviour [29], and for mining data to guide unsupervised machine learning [83]. These applications often have a large amount of high denominational data. Using clusters to compress the data makes the data more manageable, as such it is often desirable to control the size and number of the clusters that are imposed onto the data [42, 45, 46]. These methods force structure onto the problem which is unlike how Γ-Clustering structures are found within the data.

In these applications, data is clustered into disjoint sets based on similarity which is measured with a distance metric, typically the $l^2$ norm. Using the $l^2$ norm as the distance metric has the advantage of being able to leverage geometry to find relationships in the data. A number of clustering algorithms take advantage of distance metrics such as k-means [42], CURE [28], and BIRCH [103]. Γ-Clustering on the other hand, works for graphs. Graphs are able to explicitly capture distances between points instead of representing the data in $\mathbb{R}^n$ space where geometric relationships can be used. This allows Γ-Clusters to be used on environments that are and are not explicitly represented in $\mathbb{R}^n$.

The task of finding an optimal partitioning (a clustering) is often in the complexity class **NP**-hard [42, 83]. This is unlike the task of finding an optimal Γ-Clustering which is in the complexity class **P**. In this chapter, we provide an algorithm for finding the unique and optimal Γ-Clustering.

Community structures are a type of clustering that work with graphs or multigraphs (typically referred to as a network) [83]. Here, the goodness of a clustering is often measured

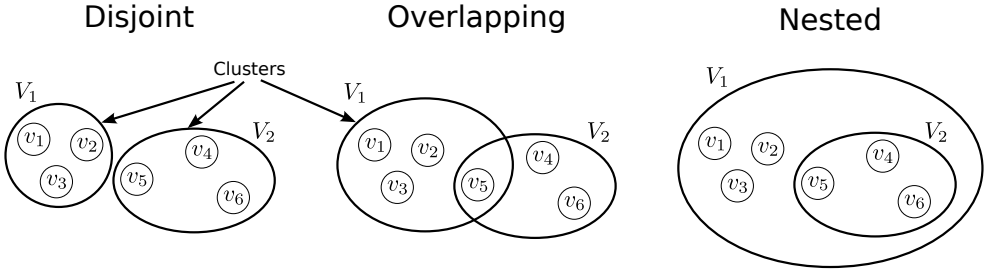



Figure 4.2: This illustration shows three examples of how two clusters can overlap or not overlap with each other.

with a common metric called modularity [62]. Thus many clustering algorithms work towards the same goal. However, since the clustering problem is an **NP**-hard task [83], the algorithms often produce a range of different solutions in their attempt to find good quality solutions. In this chapter we use these algorithms to find desirable clusters for path planning and compare the quality of these clusters to Γ-Clusters for path planning.

## 4.2 Background

This section reviews the concepts needed for the chapter.

### 4.2.1 Clusters

A *cluster* $V_i$, is a subset of the graph's vertices. Given two clusters, $V_i$ and $V_j$ we say that $V_i$ and $V_j$ are *disjoint* if $V_i \cap V_j = \emptyset$; $V_i$ and $V_j$ *overlap* if $V_i \cap V_j \neq \emptyset$, $V_i \nsubseteq V_j$, and $V_j \nsubseteq V_i$; and $V_j$ is *nested* in $V_i$ if $V_j \subseteq V_i$. See Figure 4.2 for a visual of these classifications. A *clustering* is a set of clusters, denoted by $C = \{V_1, \ldots, V_m\}$. A clustering $C$ is said to be nested if there exists some $V_i \subset V_j$, where $V_i, V_j \in C$.

A nested clustering $C$ can be visualized as a forest, where the roots of the forest are clusters that are not nested within any other cluster. The children of a cluster $V_i$, represent the clusters nested within $V_i$ that are not nested within any other cluster $V_j \subset V_i$. See Figure 4.3 for a clustering visualized as a forest. In this example the clustering

$$C = \{\{2,3,4,5\}, \{3,4\}, \{6,7\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}\}$$

has roots $V_1$, $\{1\}$, and $V_2$. The children of $V_1$ is $\{2\}, V_3$ and $\{5\}$. The children of $V_2$ is $\{6\}$ and $\{7\}$.

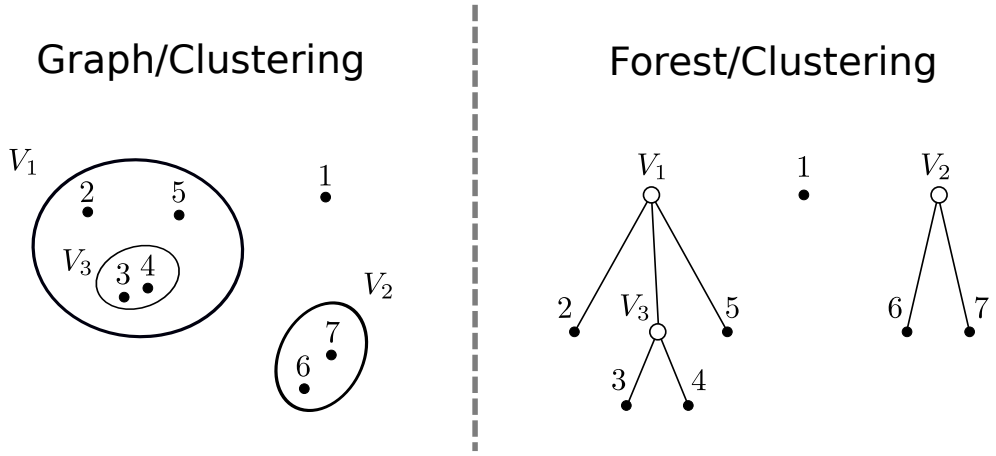



Figure 4.3: On the left we show an example of clustering in its graph environment with the edges omitted and on the right we show the same clustering depicted as a forest.

### 4.2.2 Search Space

The solution search space of a path planning problem is the set of all feasible paths. For example, a TSP problem with a directed graph of size $n$ has $(n-1)!$ different feasible tours. Thus the search space size of the problem is $(n-1)!$. Adding constraints to the problem can only serve to reduce the size of the search space. Suppose, we were to constrain the tour to visit location 2 directly after location 1. Then the search space size of the problem would be $(n-2)!$, since we have essentially shrunken the problem by removing the choice of what follows location 1.

**Computational Complexity** The difficulty of finding solutions, let alone optimal solutions, is dependent on the problem and solver. For some problems finding feasible solutions is an **NP**-hard task. In this chapter, we analyze the reduction in *computational complexity* from the view point of reducing the size of the search space. In this way we are attempting to capture the difficulty of finding an optimal ordering of a path while ignoring the difficulty of finding feasible solutions.

### 4.2.3 Multigraphs

This chapter requires a definition of multigraphs for the clustering methods that we compare against (we do not use multigraphs for our clustering problem or the path planning

problem).

Multigraphs/networks are graphs that allow for multiple edges between its vertices. For example, the multigraph $G$ may have five edges that connect vertex $v_i$ to vertex $v_j$. In this chapter we only consider unweighted multigraphs.

## 4.3 Path Planning Problem Statement

In this section, we present the class of vehicle routing problems considered by this chapter (Problem 4.3.2) and defined the clustered version of this problem as a means to reduce the problem's search space (Problem 4.3.3).

First we introduce some terminology. Let $\mathcal{P}$ represent the set of all possible paths in $G$. Then abstractly, a path planning constraint defines a subset $\mathcal{P}_1 \subseteq \mathcal{P}$ of feasible paths. Given a path planning problem $P = \langle G, \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_m, C \rangle$ where $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_m$ are the set of problem constraints, then the set of feasible paths is $\mathcal{P}_1 \cap \mathcal{P}_2 \cap \cdots \cap \mathcal{P}_m$.

**Definition 4.3.1** (Order-Free Constraints)**.** A constraint $\mathcal{P}_1$ is *order-free* if, given any $p \in \mathcal{P}_1$, then all paths obtained by permuting the vertices of $p$ are also in $\mathcal{P}_1$.

**Problem 4.3.2** (The Non-Clustered Path Planning Problem)**.** Given a complete weighted graph $G = \langle V, E, w \rangle$ and a set of order-free constraints $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_m$, find the minimum length feasible path or tour.

Many discrete vehicle routing problems can be expressed in the above framework so long as they do not restrict the ordering of vertex visits (i.e., no constraints of the form "visit $A$ before $B$"). Some examples include single and multi-robot travelling salesman problems, point-to-point planning, and patrolling problems. As a specific example the generalized travelling salesman problem (GTSP) is a problem where the robot is required to visit exactly one location in each non-overlapping set of locations (the location sets are given as input). This is naturally expressed in the above framework by having one constraint for each set (i.e., for each set $V_i$ we would construct the set of all paths that visit the set of $V_i$ exactly once). This could be expressed as $\mathcal{P}_{V_i} = \{p | V[p] \setminus V_i = V \setminus V_i \text{ and } |V[p] \cap V_i| = 1\}$, where $p$ is a non-repeating path of vertices in $V$ and $V[p]$ is the set of vertices visited by $p$).

We use clusters to constrain the feasible paths. This in turn reduces the search space size of the problem.

**Problem 4.3.3** (The Clustered Path Planning Problem)**.** Given a discrete path planning problem $P = \langle G, \mathcal{P} \rangle$ and a clustering $C$, the *clustered version of* $P$ is $\bar{P} = \langle G, \mathcal{P}, C \rangle$, which has the additional constraint that the path cannot visit any cluster $V_i \in C$ more than once.

We say a path $p$ visits a cluster $V_i$ *consecutively* only if the path visits $V_i$ once. Otherwise, there exists a continuous path segment $p_{V_i} \subseteq p$ that visits vertices in $V_i$ such that $|V_i \cap V[p]| = |p_{V_i}|$ and $V[p_{V_i}] \subset V_i$.

## 4.4 Γ-Clustering

In this section, we define what Γ-Clusterings and Γ-Clusters are and provide an efficient algorithm to find a Γ-Clustering that maximizes the search space reduction.

### 4.4.1 Definitions

We use Γ-Clusters to reduce the search space of path planning problems by restricting feasible paths to visit the Γ-Clusters, at most once. This restriction allows us to handle non-TSP problems since this restriction in no way imposes which vertices must be visited. Below we define what a Γ-Cluster is and what a Γ-Clustering is, then we formally define the restrictions that we impose on the path planning problem to reduce its search space size.

**Definition 4.4.1** (Γ-Cluster)**.** Given a graph $G = \langle V, E, w \rangle$, an input parameter $\Gamma > 1$, and a cluster $V_i \subset V$, we say $V_i$ is a Γ-Cluster if and only if the minimum inter-set edge weight, $\alpha_i$ is at least a factor of $\Gamma$ larger than the maximum intra-set edge weight, $\beta_i$.

Formally, $\alpha_i$ and $\beta_i$ satisfy the following.

$$\begin{aligned} \alpha_i &= \min \{w(v_a, v_b) | v_a \in V_i, v_b \in V \setminus V_i\} \\ \beta_i &= \max \{w(v_a, v_b) | v_a, v_b \in V_i,\ v_a \neq v_b\} \\ \alpha_i &\geq \Gamma \beta_i. \end{aligned}$$

**Definition 4.4.2** (Γ-Clustering)**.** Given a graph $G = \langle V, E, w \rangle$ and an input parameter $\Gamma > 1$. Then a clustering $C = \{V_1, V_2, \ldots, V_m\}$ is said to be a Γ-Clustering if and only if $V$ is covered by $V_1 \cup V_2 \cup \ldots \cup V_m$, non-overlapping (nested is permitted) and each cluster $V_i \in C$ is a Γ-Cluster.

*Remark* 4.4.3 (Overlap). Note that in Definition 4.4.2 overlapping clusters are not permitted. This is necessary for the problem in Definition 4.3.3 to be well defined. In addition, we will see in the following section that Γ-Clusters cannot overlap.

Given the above definitions, we are interested in finding the Γ-Clustering $C^*$ that minimizes the search space size for our path planning problem.

**Problem 4.4.4** (The Clustering Problem)**.** Given a path planning problem $P = \langle G, \mathcal{P} \rangle$ and a parameter $\Gamma > 1$, find a $\Gamma$-Clustering $C^*$ of $G$ such that the search space size of the clustered problem $\bar{P}$ is minimized.

**Note 4.4.5.** $\Gamma$-Clusters are found within the environment, not imposed onto the environment. As such some problems have many $\Gamma$-Clusters while others, little to none. Additionally, a larger choice of $\Gamma$ will tend to result in a $\Gamma$-Clustering with fewer $\Gamma$-Clusters and for some sufficiently large choice of $\Gamma$ there will be no non-trivial (of size two or more) $\Gamma$-Clusters to be found within the environment.

The methods presented in this chapter rely on finding $\Gamma$-Clusters, as such problems that contain $\Gamma$-Clusters will benefit from these approaches. Furthermore, it has been our experience that most path planning environments contain $\Gamma$-Clusters (see Section 4.8).

### 4.4.2 Finding $\Gamma$-Clusters

Based on Problem 4.3.3 and Problem 4.4.4, the task of finding a maximal clustering comes down to finding a clustering $C^*$ that contains as many clusters as possible. Each cluster adds more constraints to the problem and can only serve to reduce the problem's search space size — at worst, it stays the same size. In this section we show that there exists a unique maximal set $C^*$ and that all other $\Gamma$-Clusterings are a subset of $C^*$. We provide an efficient algorithm based on MSTs to calculate $C^*$. Below is a series of properties that we leverage in the algorithm.

#### Overlap

A special property of $\Gamma$-Clustering is that there are no overlapping clusters. Specifically, there are no two clusters $V_i$ and $V_j$ that have a non-zero intersection unless one cluster is nested within the other.

**Lemma 4.4.6.** Given a graph $G$ and $\Gamma$-Clusters $V_i$ and $V_j$ with $\Gamma > 1$, then $V_i$ and $V_j$ do not overlap.

*Proof.* We prove the above result by contradiction. Before we begin, recall that edges cut by the cluster $V_i$ have edge weights $\geq \alpha_i$ and edges within the cluster have edge weights $\leq \beta_i$. Let us assume that $V_i$ and $V_j$ overlap but are not nested. Without loss of generality let $\beta_j \leq \beta_i$. Then there exists an edge $\langle v_a, v_b \rangle$ with weight $w(v_a, v_b) \leq \beta_j$ for $v_a \in V_i \cap V_j$ and $v_b \in V_j \setminus (V_i \cap V_j)$. Since this edge does not exist in the cluster $V_i$ and it is cut by $V_i$, then it must be the case that $w(v_a, v_b) \geq \alpha_i$. However, this edge does exist in the

cluster $V_j$ and so $\alpha_i \leq \beta_j$, since $w(v_a, v_b) \leq \beta_j$. This result highlights the contradiction: $\frac{\alpha_i}{\beta_i} \leq \frac{\beta_j}{\beta_i} \leq \frac{\beta_i}{\beta_i} = 1$. □

**Uniqueness**

The non-overlapping property for Γ-Clusterings implies that there exists a unique maximal Γ-Clustering $C^*$ (more clusters equates to more reductions in the search space size). This result follows from the simple property that if a Γ-Cluster $V_i$ exists and is not in our clustering $C^*$, then we can add it to $C^*$ to obtain additional search space reductions.

**Proposition 4.4.7.** Given graph $G$ and a parameter $\Gamma > 1$, the problem of finding a Γ-Clustering $C^*$ that maximizes the search space reduction has a unique solution $C^*$. Furthermore $C^*$ contains all clusters $V_i$ with separation $\alpha_i \geq \Gamma\beta_i$.

*Proof.* We use contradiction to prove that $C^*$ is unique. Suppose there exists two different Γ-Clusterings $C_1$ and $C_2$ that maximize the search space reductions for a $\Gamma > 1$. Therefore, there is a cluster $V_1$ that is in $C_1$ but not in $C_2$ (or vice versa). This implies that either $V_1$ is somehow incompatible with $C_2$, which we know is not the case due to Lemma 4.4.6 (i.e., clusters do not overlap unless one is a subset of the other) or this cluster can be added to $C_2$. This is a contradiction that proves the first result. The second result directly follows since adding a cluster can only further reduce the search space size of the problem. □

**Corollary 4.4.8.** Given a graph $G = \langle V, E, w \rangle$ and two optimal Γ-Clusterings $C_i^*$ and $C_j^*$ such that $\Gamma_j > \Gamma_i > 1$, then $C_j^* \subseteq C_i^*$.

*Proof.* We prove this result by contradiction. Suppose we have a cluster $V_1 \in C_j^*$ that is not in $C_i^*$. By the definition $\alpha_1 \geq \Gamma_j\beta_1$, where $\alpha_1$ is the minimum inter-set edge weight for $V_1$ and $\beta_1$ is the maximum intra-set edge weight of $V_1$. Since $\Gamma_j > \Gamma_i$ then $\alpha_1 \geq \Gamma_i\beta_1$, which implies that $V_1$ could be in $C_i^*$. This contradicts Proposition 4.4.7, which states that all clusters that could be in the optimal clustering are in the optimal clustering. Therefore, if $\Gamma_j > \Gamma_i > 1$, then $C_j^* \subseteq C_i^*$. □

**Corollary 4.4.9.** There exists a minimum $\Gamma^* > 1$ and its optimal Γ-Clustering $C^*$ that acts as a superset for all other Γ-Clusterings.

*Proof.* We prove this result by contradiction. Suppose, we have $\Gamma^*$ and $C^*$. Then assume, there exists a $\Gamma > \Gamma^*$ with a Γ-Clustering $C$ that is a superset of $C^*$. Corollary 4.4.8 states that $C \subseteq C^*$, which highlights the contradiction. Additionally, every graph has a finite number of clusters. Therefore, there must exist a minimum value of $\Gamma^* > 1$ that finds the Γ-Clustering $C^*$ that is a superset of all other Γ-Clusters. □

### An MST Approach For Finding Γ-Clusters

Given a graph $G$ and an input $\Gamma > 1$, Algorithm 5 computes the optimal Γ-Clustering for $G$ (i.e., the Γ-Clustering with maximum number of clusters). Informally, the algorithm deletes edges in the graph from largest to smallest (Lines 6-7) to look for Γ-Clusters. It uses a minimum spanning tree (MST) to keep track of when the graph becomes disconnected and tests the disconnected components to see if they are Γ-Clusters (Lines 9-11). Regardless, disconnected components of size two or more are added back to the queue (Lines 13-14) to find Γ-Clusters within those components. If the disconnected component was a Γ-Cluster then the algorithm is searching for nested Γ-Clusters.

**Algorithm 5:** Γ-CLUSTERING$(G, \Gamma)$

**Input:**
$G$: a graph with vertices $V$, edges $E$, and weights $w$.
Γ: a real valued constant greater than 1.
**Output:**
$C$: a maximal Γ-Clustering.

```
1  C ← ∅
2  M ← {MST(G)}
3  while |M| > 0 do
4      m ← M.pop()
5      α ← largest edge cost in m
6      M' ← disconnected trees after removing edge(s) of cost α from m
7      for m' ∈ M' do
8          G' ← graph induced by V[m']
9          β ← max edge cost of G'
10         if α ≥ Γβ then
11             C ← C ∪ {V[m']}
12         if |m'| > 1 then
13             M ← M ∪ m'
14 return C
```

**Theorem 4.4.10.** Given a $G = \langle V, E, w\rangle$ and a $\Gamma > 1$, Algorithm 5 finds the optimal Γ-Clustering $C^*$ for $G$ in $O(|V|^3)$ time.

*Proof.* First we show that Algorithm 5 runs in polynomial time. Let $n = |V|$. Minimum spanning trees can be found in $O(n^2)$ time [74] (Line 3). The rest of the algorithm modifies

the MST found in Line 3, which originally has $n$ edges. Thus the while loop for the rest of the algorithm runs at most $n$ times (we can't remove more than $n$ edges). Finding the largest edge(s) and removing them in Line 6 and 7 takes $O(n)$ time. Creating the induced subgraph and finding the maximum edge cost in Lines 9 and 10 takes $O(n^2)$ time. Testing if the subgraph is a clique (Line 11) takes $O(n^2)$ time. Thus Lines 1-3 run in $O(n^2)$ time and Lines 4-14 run in $O(n \cdot n^2)$, meaning the entire algorithm runs in $O(n^3)$ time or $O(|V|^3)$.

Next, we show that Algorithm 5 finds the optimal Γ-Clustering. To do so, we show that each $V[m']$ found in Line 11 is a Γ-Cluster as defined by Definition 4.4.1. Then we show that the algorithm does not omit any candidate clusters.

Let us start by understanding how MSTs are used to find Γ-Clusters. A Γ-Cluster with separation $\Gamma > 1$ is a subgraph of $G$ that is connected with intra-edge weights less than the inter-edge weights. Let $\alpha$ be the minimum edge weight connected to the subgraph (minimum inter-edge weight) and $\beta$ be the maximum edge weight within the subgraph (maximum intra-edge weight). Thus one method of searching for Γ-Clusters is to delete all edges of weight $\geq \alpha$ in the graph and test the disconnected components (subgraphs). We can use MSTs to efficiently keep track of these deleted edges. By definition, an MST is a tree that connects vertices within the graph with minimum edge weight. When we remove edges of size $\geq \alpha$ to search for disconnected subgraphs, the graph is disconnected if and only if the MST is disconnected (the cut that disconnects the graph also disconnects the MST). Furthermore, the algorithm incrementally removes the edge(s) in the MST from largest to smallest so that we can emulate removing edges in the graph of cost $\geq \alpha$. Thus if the induced subgraph $G'$ created in Line 9 has a maximum intra-edge weight of $\beta$, then it is a Γ-Cluster if $\alpha \geq \Gamma\beta$. For these reasons it is added to the clustering $C$ in Line 12.

Now, we show that the Algorithm does not omit any candidate clusters. We have already argued that only disconnected MST trees need to be considered for Γ-Clusters. What is left to show is that the algorithm tests every possible value of $\alpha$ for disconnecting the MST tree. This is true since it considers every edge in the MST. Thus every candidate cluster of size two or more is considered.

Therefore, this algorithm finds all of the candidate Γ-Clusters and Proposition 4.4.7 implies that this clustering is the unique optimal Γ-Clustering for the given $\Gamma > 1$. □

## 4.5 Coupled Planning

In this section we solve the clustered problem $\bar{P}$ instead of the non-clustered problem $P$ (Problem 4.3.3 and Problem 4.3.2 respectively). We show that the clustered problem can

have a search space size that is exponentially smaller than the non-clustered problem. We also show that the cost $\bar{c}^*$ of optimal solutions of $\bar{P}$ are within a constant factor of $c^*$, where $c^*$ is the cost of optimal solutions for $P$.

### 4.5.1 Search Space Reduction

The task of measuring the reduction in search space size from Problem $P$ to $\bar{P}$ for a clustering $C$, depends on the set of feasible paths for $P$ and $C$. For our analysis, we study the reduction of search space size for the path planning problem TSP. This problem's set of feasible paths are all tours that visit each vertex exactly once. Thus the size of the search space for this problem is

$$N_0 = (|V| - 1)!.$$

The size of the clustered problem's search space (Problem 4.3.3) is

$$N_1 = (|\textsc{Roots}(C)| - 1)! \prod_{V_i \in C} |\textsc{Children}(V_i)|!,$$

where the function $\textsc{Roots}(C)$ returns the root clusters of $C$ and the function Children$(V_i)$ returns the children of $V_i$ (see Section 4.2.1 for the definition of roots and children).

**Note 4.5.1.** The functions ROOTS and CHILDREN, return a set of sets instead of a hybrid of sets and vertices. We do this to simplify the algorithms within this chapter.

To simplify the analysis, we study clusterings that contain $m$ equally sized clusters of size $x = |V|/m$, where $x \in \mathbb{Z}$. Now we have

$$N_1 = (m-1)! \prod_{i=1}^{m} x!.$$

**Lemma 4.5.2.** Given $|V| = mx$, for $m, x \in \mathbb{Z}_+$ the following holds $|V|! \geq m!^x x!^m$.

*Proof.*

$$\begin{aligned}
|V|! &= \prod_{i=1}^{m}\prod_{j=0}^{x-1}(ix-j) \\
&= \prod_{i=1}^{m}\prod_{j=0}^{x-1}(i)\left(x-\frac{j}{i}\right) \\
&\geq \prod_{i=1}^{m}\prod_{j=0}^{x-1}(i)\,(x-j) \\
&= \left(\prod_{i=1}^{m} i^x\right)\left(\prod_{j=0}^{x-1}(x-j)\right)^m \\
&= (m!)^x(x!)^m.
\end{aligned}$$

□

We show through a series of manipulations the ratio $r = N_0/N_1$ is exponential in size, which in turn shows that $N_0$ is exponentially larger than $N_1$.

$$\begin{aligned}
r &= \frac{N_0}{N_1} \\
&= \frac{(|V|-1)!}{(m-1)!\prod_{i=1}^{m} x!} \\
&\geq \frac{(m!)^x(x!)^m}{|V|(m-1)!(x!)^m} && \text{Lemma 4.5.2} \\
&\geq \frac{(m!)^{x-1}}{|V|} \\
&= \frac{(m!)^{x-1}}{mx} \\
&= \frac{(m-1)!(m!)^{x-2}}{x} \\
&= (m-1)!(m!)^{x-3} && \text{for } x \geq 3 \text{ and } m \geq 2 \\
&\geq (m-1)!^{x-2}.
\end{aligned}$$

To gauge the magnitude of $r$, consider a TSP instance of size 100 and a clustering that divides the vertices into four equal parts. Here $r \approx 2.69 \times 10^{54}$, which means the clustered

problem has search space that is more than $10^{54}$ smaller than the non-clustered problem's search space. However $N_1$ is still extremely large, $N1 \approx 3.47 \times 10^{101}$.

### 4.5.2 Solution Quality Bounds

In this section, we show that when $P = \langle G, \mathcal{P} \rangle$ is metric (the graph $G$ is metric) the solutions to the clustered problem $\bar{P} = \langle G, \mathcal{P}, C \rangle$ have cost $c$ bound by

$$\bar{c}^* \leq \min\left(2, \left(1 + \frac{3}{2\Gamma}\right)\right) c^*,$$

where $c^*$ is the optimal cost of $P$. We start by proving the first half of the result ($\bar{c}^* \leq 2c^*$). We do this with a series of theoretical results that use minimum spanning trees (MST) to construct feasible solutions.

**Lemma 4.5.3.** Given a metric graph $G$ and a $\Gamma$-Clustering $C$ with $\Gamma > 1$, then every MST will have exactly one inter-set edge for each cluster $V_i \in C$.

*Proof.* We prove the above result by contradiction. Suppose the MST has at least two inter-set edges connected to $V_i$. Thus, there are at least two sets of vertices in $V_i$ that are not connected to each other using intra-set edges. We can lower the cost of the MST by removing one of these inter-set edges of weight $\geq \alpha_i$ and replace it with an intra-set edge of weight $\leq \beta_i < \alpha_i$. This highlights the contradiction. Therefore, every MST will have exactly one inter-set edge for each $\Gamma$-Cluster $V_i \in C$. □

**Lemma 4.5.4** (Two-Factor Approximation)**.** Consider a metric discrete path planning problem $P$ with an optimal solution path $p^*$ and cost $c^*$. Then given a $\Gamma$-Clustering $C = \{V_1, V_2, \ldots, V_m\}$ with $\Gamma > 1$, the optimal solution $\bar{p}^*$ for the clustered problem $\bar{P}$ over the same set of vertices $V[p^*]$ is a solution to $P$ with cost $\bar{c}^* \leq 2c^*$.

*Proof.* To prove the above result, we use the MST approach described below and in [49] to construct a path $\bar{p}$ over the set of vertices in $V[p^*]$. This approach yields a solution $\bar{p}$ for $\bar{P}$ that has the desired cost bound, $\bar{c} \leq 2c^*$ [49]. The MST procedure is described below.

1. Find a minimum spanning tree for the vertices $V[p^*]$.
2. Duplicate each edge in the tree to create a Eulerian graph.
3. Find an Eulerian tour of the Eulerian graph.

4. Convert the tour to a TSP: if a vertex $v_i$ is visited more than once after the first visit, create a shortcut: from the vertex before $v_i$ to the vertex after $v_i$ (i.e., create a tour that visits the vertices in the order they first appeared in the tour).

What remains to prove is that the above tour is a feasible solution for $\bar{P}$. First we note that the above approach yields a single tour of all the vertices in $V[p^*]$ (i.e., there are no disconnected tours). Note that Lemma 4.5.3 states that every MST uses exactly one inter-set edge for each cluster $V_i \in C$. Thus when the edges are duplicated and a Eulerian tour is found, there are only two inter-set edges used for each $V_i \in C$. Furthermore shortcutting the path does not change the number of inter-set edges used by the tour, thus the final solution $\bar{p}$ only has one incoming and one outgoing edge for each cluster $V_i \in C$. As a result it is a clustered solution for $\bar{P}$ that satisfies the bound since the MST approach also yields a solution with cost $\bar{c} \leq 2c^*$. □

Next, we prove the second half of the bound $\bar{c}^* \leq \left(1 + \frac{3}{2\Gamma}\right) c^*$ by using Algorithm 6 to construct $\bar{p}$ (a feasible solution for $\bar{P}$). Additionally, we use a modified graph (Definition 4.5.8) to show that the cost of the solution satisfies our desired bound.

**Lemma 4.5.5** (Correctness of Algorithm 6)**.** Given a feasible path $p$ for $P$ and a cluster $V_i \in C$ that is not visited consecutively, then $p' \leftarrow \textsc{Deform}(p, V_i)$ visits $V_i$ consecutively and any subsequent deformed paths also visits $V_i$ consecutively.

*Proof.* By construction, $V_i$ is visited consecutively in $p' \leftarrow \textsc{Deform}(p, V_i)$. The remaining claim that $V_i$ continues to be visited consecutively is proven by showing that the set of intra-set edges for $V_i$ remains unchanged.

We prove this result by contradiction. Suppose, there exists some deformed path $p'''$ that came some time after $p'$, such that the edge $\langle v_a, v_b \rangle \in p'$ for some $v_a, v_b \in V_i$ but not in $p''$ ($\langle v_a, v_b \rangle \notin p''$). Furthermore, let $p'''$ be the first deformed path that does not have this edge and let $p''' \leftarrow \textsc{Deform}(p'', V_j)$ be the path before the deformation.

This means that Algorithm 6 inserts one or more vertices in between $v_a$ and $v_b$. This cannot happen in Lines 5-7 since the path for the $1^{st}$ vertex to the $k^{th}$ is unchanged. Also this cannot happen in Lines 8-9 since this part of the algorithm is only connecting vertices within the cluster $V_j$ together. Finally, this cannot happen in Lines 11-13, since the path is not changing the order of the appearance of $v_a$ and $v_b$ (no insertions, just deletions).

Thus $\langle v_a, v_b \rangle$ must be in $p'''$, which highlights the contradiction. Therefore, all subsequent paths must also visit $V_i$ consecutively, since all intra-set edges remain intact. □

**Algorithm 6:** DEFORM$(p, V_i)$

**Input:**
$p$: a path represented as an array.
$V_i$: a cluster.
**Output:**
$\bar{p}$: a deformed path that visits $V_i$ at most once.

```
1  if p has two or less inter-set edges for V_i then
2      return p
3  k ← 1
4  p̄ ← [ ]                                          /* empty array */
5  while p[k] ∉ V_i do
6      p̄.append(p[k])
7      k ← k + 1
8  for l ∈ [k, k + 1, . . . , |p|] do
9      if p[l] ∈ V_i then
10         p̄.append(p[l])
11 for l ∈ [k, k + 1, . . . , |p|] do
12     if p[l] ∉ V_i then
13         p̄.append(p[l])
14 return p̄
```

*Remark* 4.5.6 (Uniqueness). When Algorithm 6 is applied to $p^*$ iteratively for each $V_i \in C$ to generate the solution $\bar{p}$, the solution is unique despite the order that DEFORM$(p, V_i)$ was called. Furthermore the order of the clusters is determined by their first appearance in $p$.

This is how Algorithm 6 reorders the vertices within the tour. Specifically, the order of vertices within each cluster $V_i$ is preserved as DEFORM$(p, V_i)$ is called, as well as the ordering of the remaining vertices.

**Lemma 4.5.7** (Deformation cost)**.** Consider a feasible path $p$ for $P$ that has $2(n+1) \geq 4$ inter-set edges for $\Gamma$-Cluster $V_i$ such that $\Gamma > 1$ and $n \in \mathbb{Z}_{\geq 0}$. Then the cost to deform $p$ into $\bar{p} \leftarrow$ DEFORM$(p, V_i)$ is $\bar{c} - c \leq (2n+1)\beta_i$.

*Proof.* We analyze the cost to deform $p$ into $\bar{p}$ for each type of deformation the algorithm uses.: There are three types of deformations: 1) shortcuts are created within the cluster; 2) shortcuts are created outside of the cluster; and 3) the cluster adopts a new outgoing edge. These deformations are illustrated in Figure 4.4. A classification of the edges in the figure are as follows: 1) edges $\langle v_3, v_4 \rangle$ and $\langle v_6, v_7 \rangle$ are shortcuts within the cluster; 2) edges $\langle v_{10}, v_{11} \rangle$ and $\langle v_{12}, v_{13} \rangle$ are shortcuts outside of the cluster; and 3) edge $\langle v_8, v_9 \rangle$ is the new outgoing edge for the cluster.

We start by examining the incurred cost to shortcut paths within the cluster. Consider a path segment $\langle v_a, v_b, v_c, \ldots, v_x, v_y, v_z \rangle$ of $p$ such that $v_a$ is directly connected to $v_z$ in $\bar{p}$ with the edge $\langle v_a, v_z \rangle$ and $v_a, v_z \in V_i$. The incurred cost of each of these edges is $\leq \beta_i$ due to the fact that the cost of any intra-set edge has weight $\leq \beta_i$. There are $n$ such shortcuts of this nature incurred from DEFORM$(p, V_i)$ ($n$ captures the number of extra visits to the cluster) and so the total incurred cost for this type of shortcut is $\leq n\beta_i$.

Next, we examine the incurred cost to shortcut paths outside of the cluster. Consider a path segment $\langle v_a, v_b, v_c, \ldots, v_x, v_y, v_z \rangle$ of $p$ such that $v_a$ is directly connected to $v_z$ in $\bar{p}$ with the edge $\langle v_a, v_z \rangle$, $v_a, v_z \notin V_i$ and $v_b, v_c, \ldots, v_y \in V_i$. The incurred cost for each of these shortcuts is again $\leq \beta_i$. This is due to the metric property of $G$. The cost of the direct path from $v_a$ to $v_z$ is less than or equal to any path from $v_a$ to $v_z$, specifically $c(v_a, v_z) \leq c(v_a, v_b) + c(v_b, v_y) + c(v_y, v_z) \leq c(v_a, v_b) + \beta_i + c(v_y, v_z)$. Thus the incurred cost, $\Delta$, of this shortcut is bound by the difference between the cost of the new edges in $\bar{p}$ and the removed edges in $p^*$, namely:

$$\begin{aligned}
\Delta &= c(v_a, v_z) - c(v_a, v_b) - c(v_y, v_z) \\
&\leq c(v_a, v_b) + \beta_i + c(v_y, v_z) - c(v_a, v_b) - c(v_y, v_z) \\
&\leq \beta_i.
\end{aligned}$$

There are $n$ such shortcuts of this nature incurred by $\textsc{Deform}(p, V_i)$. Thus, the total incurred cost for this type of shortcut is also $\leq n\beta_i$.

Lastly we examine the incurred cost of the new outgoing edge. Consider the path $\langle v_a, v_b, v_c, \ldots, v_x, v_y, v_z \rangle$ of $p$ such that $\langle v_b, v_c \rangle$ is the first outgoing edge of $V_i$ and $\langle v_x, v_y \rangle$ is the last outgoing edge of $V_i$, thus $\langle v_x, v_c \rangle$ is the new outgoing edge. Then due to the metric property we know that $c(v_x, v_c) \leq c(v_x, v_b) + c(v_b, v_c) \leq \beta_i + c(v_b, v_c)$. The incurred cost, $\Delta$, of this deformation is the difference between the cost of the new edge $\langle v_x, v_c \rangle$ and the removed edge $\langle v_b, v_c \rangle$ (this edge has not been considered in any previous incurred cost calculation):

$$\begin{aligned}\Delta &= c(v_x, v_c) - c(v_b, v_c) \\ &\leq \beta_i + c(v_b, v_c) - c(v_b, v_c) \\ &\leq \beta_i.\end{aligned}$$

This accounts for all the incurred costs, and so the total cost to deform $p$ into $\bar{p}$ via $\textsc{Deform}(p, V_i)$ is $\bar{c} - c \leq (2n+1)\beta_i$. □

We introduce the modified graph $\hat{G}$ in the following definition to aid with our ongoing proof of the bound.

**Definition 4.5.8** (The modified graph $\hat{G}$)**.** Given a graph $G$ and a $\Gamma$-Clustering $C$, *the modified graph* $\hat{G}$ is a copy of $G$ with the following modifications: if $\langle v_a, v_b \rangle$ is an inter-set edge with $v_a \in V_i$ and $v_b \in V_j$, then $\hat{w}(v_a, v_b) = w(v_a, v_b) + \frac{3}{2}\max(\beta_i, \beta_j)$; otherwise $\hat{w}(v_a, v_b) = w(v_a, v_b)$, where $\beta_i$ and $\beta_j$ are defined as in Definition 4.4.1.

**Lemma 4.5.9** (Deformation cost in $\hat{G}$)**.** Consider a feasible path $p$ for $P$, a $\Gamma$-Cluster $V_i$ with $\Gamma > 1$, and $\bar{p} \leftarrow \textsc{Deform}(p, V_i)$, then the cost of $p$ and $\bar{p}$ in $\hat{G}$ satisfies $\hat{c}(\bar{p}) \leq \hat{c}(p)$, where $\hat{c}(\bar{p})$ and $\hat{c}(p)$ are the cost of $\bar{p}$ and $p$ in $\hat{G}$ respectively. In other words the cost to deform $p$ into $\bar{p}$ is $\leq 0$ for $\hat{G}$.

*Proof.* In this proof we analyze the cost to deform $p$ into $\bar{p}$ in $\hat{G}$, which is a result of single call $\bar{p} \leftarrow \textsc{Deform}(p, V_i)$ (Algorithm 6).

From Lemma 4.5.7, we see that the cost to deform $p$ into $\bar{p}$ with respect to $G$ is $\bar{c} - c \leq (2n+1)\beta_i$, where there are $2(n+1) \geq 4$ inter-set edges for $V_i$ in $p$. The cost of $p$ in $\hat{G}$ is

$$\hat{c}(p) \geq c + 2(n+1)\left(\frac{3}{2}\right)\beta_i = c + 3(n+1)\beta_i$$

and the cost of $\bar{p}$ in $\hat{G}$ is

$$\hat{c}(\bar{p}) \leq \bar{c} + 2\left(\frac{3}{2}\beta_i\right) \leq c + (2n+1)\beta_i + 3\beta_i.$$

Thus

$$\begin{aligned}\hat{c}(\bar{p}) - \hat{c}(p) &\leq (2n+1)\beta_i + 3\beta_i - 3(n+1)\beta_i \\ &= 2n\beta_i + \beta_i + 3\beta_i - 3n\beta_i - 3\beta_i \\ &= \beta_i - n\beta_i\end{aligned}$$

and since $n \geq 1$ (otherwise we did not need to deform the path), then we have $\hat{c}(\bar{p}) \leq \hat{c}(p)$. □

**Lemma 4.5.10** (Approximation Factor)**.** Given a metric discrete path planning problem $P$ with optimal solution cost $c^*$, $\Gamma > 1$, and a $\Gamma$-Clustering $C = \{V_1, V_2, \ldots, V_m\}$, then the optimal solution $\bar{p}^*$ to the clustered problem $\bar{P}$ over the same set of vertices is a solution to $P$ with cost $\bar{c}^* \leq (1 + \frac{3}{2\Gamma})c^*$.

*Proof.* To prove the above result we work with the modified graph $\hat{G}$ (Definition 4.5.8) and use Lemma 4.5.9 to imply there exists a clustered solution in $\hat{G}$ that has the same cost or lower than $\hat{c}(p^*)$, where $\hat{c}(p^*)$ is the cost of $p^*$ in $\hat{G}$. Then we relate $\bar{c}^*$ to $c^*$.

First, we show that there exists a clustered solution $\bar{p}$ that satisfies the following:

$$\hat{c}(\bar{p}) \leq \hat{c}(p^*),$$

where $\hat{c}(\bar{p})$ and $\hat{c}(p^*)$ are the cost of $\bar{p}$ and $p^*$ in $\hat{G}$ respectively. To find a solution $\bar{p}$ that satisfies the above we use Algorithm 6 to deform $p^*$ into $\bar{p}$. The deform algorithm is called for each $V_i \in C$, in any order, to form a solution for $\bar{P}$ (see Lemma 4.5.5). For each call $p' \leftarrow \textsc{Deform}(p_i, V_i)$ the incurred cost in $\hat{G}$ is $\hat{c}(p_i) - \hat{c}(p') \leq 0$ (see Lemma 4.5.9). Therefore, after the series of calls, we have a clustered solution $\bar{p}$ satisfying

$$\hat{c}(\bar{p}) \leq \hat{c}(p^*).$$

Next, we relate $c^*$ to $\hat{c}(p^*)$ by observing the following for an inter-set edge $\langle v_a, v_b \rangle \in p^*$:

$$\begin{aligned}\hat{w}(v_a, v_b) &= w(v_a, v_b) + \frac{3}{2}\max(\beta_i, \beta_j) \\ &\leq w(v_a, v_b) + \frac{3}{2\Gamma}\max(\alpha_i, \alpha_j) \\ &\leq \left(1 + \frac{3}{2\Gamma}\right) w(v_a, v_b).\end{aligned}$$

The above inequality is true for each edge $\langle v_a, v_b \rangle \in p^*$, inter-set edge or not. Therefore the path cost satisfies

$$\hat{c}(p^*) \leq \left(1 + \frac{3}{2\Gamma}\right) c^*.$$

Due to the construction of $\hat{G}$ and how $\hat{c}(\bar{p}) \leq \hat{c}(p^*)$ we deduce that

$$\bar{c}^* \leq \hat{c}(p^*),$$

since

$$\bar{c}^* \leq c(\bar{p}) \leq \hat{c}(\bar{p}) \leq \hat{c}(p^*),$$

where $c(\bar{p})$ is the cost of $\bar{p}$ in $G$. Therefore

$$\bar{c}^* \leq \left(1 + \frac{3}{2\Gamma}\right) c^*.$$

□

**Theorem 4.5.11** (Approximation Factor)**.** Given a metric discrete path planning problem $P$, an optimal solution $p^*$ with cost $c^*$, and a $\Gamma$-Clustering $C = \{V_1, V_2, \ldots, V_m\}$ with $\Gamma > 1$, then the optimal solution $\bar{p}^*$ to the clustered problem $\bar{P}$ over the same set of vertices is a solution to $P$ with cost $\bar{c}^* \leq \min\left(2, 1 + \frac{3}{2\Gamma}\right) c^*$.

*Proof.* The proof directly follows from Lemma 4.5.4 and Lemma 4.5.10. □

*Remark* 4.5.12 (Tightness of Bound). To better characterize the tightness of the bound provided by Theorem 4.5.11, we provide a lower bound for the solution quality of solving $\bar{P}$ instead of $P$ (Problem 4.3.3 and Problem 4.3.2 respectively). Additionally, Figure 4.6 visualizes the gap between the lower and upper bound for increasing values of $\Gamma$.

The lower bound problem is illustrated in Figure 4.5. In this example, the graph is scalable (add three vertices at a time). The clustered and non-clustered solutions for this example have a cost relation of

$$\lim_{|V| \to \infty} \bar{c}^* = \left(1 + \frac{2}{2\Gamma + 1}\right) c^*.$$

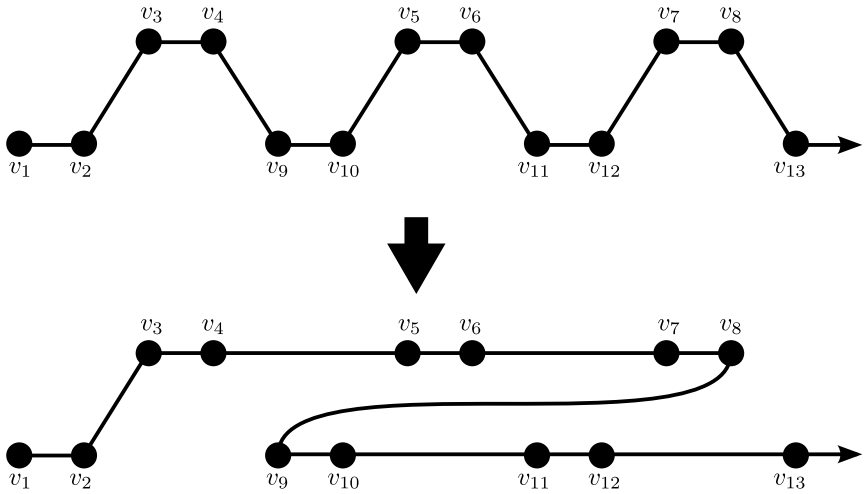



Figure 4.4: Path deformation example.

The bound for this graph is obtained as follows. Let $n = \frac{|V|}{3} - 1$ for $\frac{|V|}{3} \in \mathbb{Z}_{>0}$. Then the optimal non-clustered solution cost is:

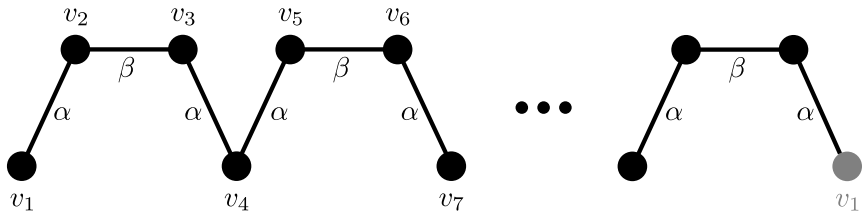
$$\begin{aligned} p^* &= \langle v_1, v_2, v_3, v_4, v_6, v_5, \ldots \rangle \\ c^* &= \alpha + \beta + \alpha + n\,(2\alpha + \beta) \\ &= (n+1)\,(2\alpha + \beta) \\ &= (n+1)\left(2 + \frac{1}{\Gamma}\right)\alpha \end{aligned}$$

(verified with a solver for $P$). The optimal clustered solution is:

$$\begin{aligned} \bar{p}^* &= \langle v_1, v_2, v_3, v_5, v_6, \ldots, v_4, v_7, \ldots \rangle \\ \bar{c}^* &= \alpha + \beta + 2n\beta + \alpha + n(2\alpha + \beta) \\ &= 2\alpha + 3\beta - 2\beta + n(2\alpha + 3\beta) \\ &= (n+1)(2\alpha + 3\beta) - 2\beta \\ &= \left((n+1)(2 + \frac{3}{\Gamma}) - \frac{2}{\Gamma}\right)\alpha \end{aligned}$$

(verified with a solver for $\bar{P}$).

Optimal Solution:




Optimal Clustered Solution:

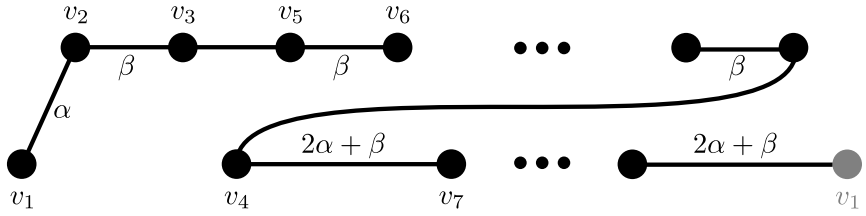



Figure 4.5: Metric example instance ($\alpha > \beta$). Vertices in the top row ($v_2, v_3, v_5, v_6, \ldots$) are in the cluster $V_i$. Edge weights connecting vertices within $V_i$ are $\beta$. Edge weights connecting vertices not in $V_i$ are $2\alpha + \beta$. Edge weights connecting vertices not in $V_i$ to vertices in $V_i$ are $\alpha + \beta$, unless shown differently in diagram.

As the instance grows ($|V| \to \infty$ and $n \to \infty$), we have the following:

$$\begin{aligned}
\lim_{n\to\infty} \bar{c}^* &= \frac{(n+1)\left(2+\frac{3}{\Gamma}\right) - \frac{2}{\Gamma}}{(n+1)\left(2+\frac{1}{\Gamma}\right)} c^* \\
&= \left(\frac{2+\frac{3}{\Gamma}}{2+\frac{1}{\Gamma}}\right) c^* \\
&= \left(1 + \frac{2}{2\Gamma+1}\right) c^*,
\end{aligned}$$

which confirms the stated bound.

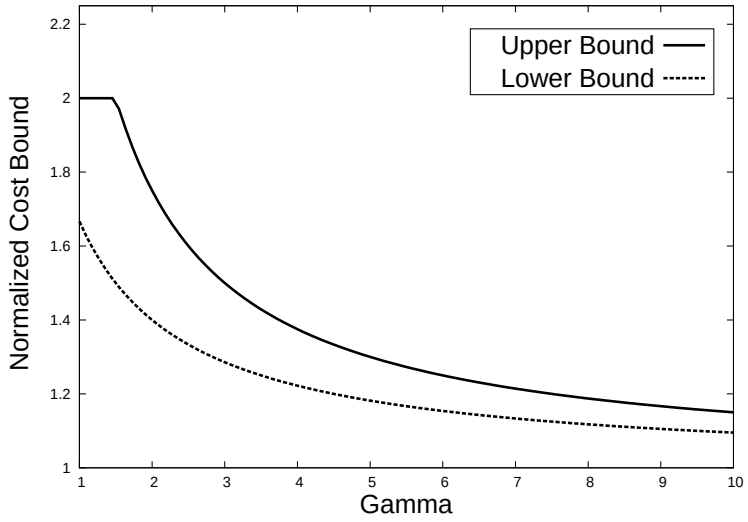



Figure 4.6: A plot showing the tightness of the approximation's upper bound. The gap between the two curves shows where the tightest upper bound can lie.

## 4.6 Decoupled Planning

We introduce decoupled planning as an intermediate step towards hierarchical planning. Decoupled planning, is a technique that significantly reduces the space of feasible solutions that are searched. This approach achieves its reduction in computational complexity, by how it finds solutions in the space (i.e., the search space of the problem is unchanged but a large portion is ignored by the approach). Decoupled planning solves the clustered path planning problem $\bar{P} = \langle G, \mathcal{P}, C \rangle$ by decomposing the problem into a series of smaller independent problems, one problem for each cluster $V_i \in C$. A path segment $p_{V_i}$ for cluster $V_i$ is found by planning the sequence of its children without consideration for the rest of the graph (independently planned). Its children are compressed into super vertices, where the incoming and outgoing edges take on the worst case edge weights. Algorithm 7 solves this problem for a given set of included vertices $V'$.

To construct a full solution, we also require a tour of the root clusters. The clusters visited in this tour again depend on the included set of vertices $V'$. However, if we were to naïvely plan a tour of the included root clusters we may end up with a solution path, instead of a tour. This would occur in the event that only one root cluster $V_i$ is visited. To compensate for this corner case, the planner would need to find a tour of $V_i$'s offspring at a level where there is more than one cluster to visit. Algorithm 9 solves this problem by constructing a clustering $C'$ of the visited clusters. Once all of the segments are planned,

**Algorithm 7:** DECOUPLED PATH$(G, C, V', V_i)$

**Input:**

$G$: a graph with vertices $V$, edges $E$, and weights $w$.

$C$: a clustering.

$V'$: a set of included vertices.

$V_i$: a cluster such that $V_i \in C$.

**Output:**

$p_{V_i}$: a locally optimal path of the non-nested components of $V_i$ that contain vertices in $V'$.

1 $C' \leftarrow \{V_j \in \text{CHILDREN}(V_i) | V_j \cap V' \neq \emptyset\}$

2 $\tilde{G}' \leftarrow \text{COARSEN}(G, C')$

3 $p_{V_i} \leftarrow \text{SHORTESTPATH}(\tilde{G}')$

4 Return $p_{V_i}$

**Algorithm 8:** COARSEN$(G, C)$

**Input:**

$G$: a graph with vertices $V$, edges $E$, and weights $w$.

$C$: a clustering such that $V_i \cap V_j = \emptyset \ \forall \ V_i, V_j \in C$.

**Output:**

$\tilde{G}$: a coarsened graph with vertices $C$.

1 Create a graph $\tilde{G}$ of size $|C|$ with vertices $V_i \in C$

2 Let $\tilde{w}(V_i, V_j) = \max\{w(v_a, v_b) | v_a \in V_i, v_b \in V_j\}$

3 Return $\tilde{G}$

then a full solution is constructed by recursively replacing the super vertices $V_i$ in the tour with their corresponding path segments $p_{V_i}$.

Decoupled planning can also be used to find solutions for $\bar{P}$ without the knowledge of $V'$. This is accomplished by re-coupling the smaller problems back together with some additional logic to ensure that a solution tour is found instead of a path. Then the re-coupled problem will find $V'$, a tour of the root clusters, and a set of path segments $p_{V_i}$ for each visited cluster $V_i$. The re-coupled problem still has a much smaller search space than $\bar{P}$ since for every feasible set $V'$ the problem is finding a tour in a decoupled manner.

**Algorithm 9:** DECOUPLED TOUR$(G, C, V')$

**Input:**
$G$: a graph with vertices $V$, edges $E$, and weights $w$.
$C$: a clustering.
$V'$: a set of vertices to be visited such that $V' \subseteq V$.
**Output:**
$p_C$: an optimal tour of the non-nested components of $C$ that contain vertices in $V'$.

```
C' ← {V_i ∈ C | V_i ∩ V' ≠ ∅}
while |ROOTS(C')| = 1 do
    Replace C' with the root cluster V_i of C'
G̃' ← COARSEN(G, ROOTS(C'))
p_C ← SHORTESTTOUR(G̃')
Return p_C
```

### 4.6.1 Search Space Reduction

In this section we show how decoupled planning can exponentially reduce the space of searched solutions. We measure this space by counting the number of feasible paths searchable by the algorithm. As with coupled planning, measuring the reduction in search space size depends on the set of feasible paths $\mathcal{P}$ for Problem 4.3.2 and the clustering $C$. For our analysis we again study the TSP problem $P = \langle G, \mathcal{P} \rangle$, where $\mathcal{P}$ is the set of all tours that visit ever vertex in $G$ exactly once. As before the size of the non-clustered problem is

$$N_0 = (|V| - 1)!$$

and the size of the clustered problem's search space (Problem 4.3.3) is

$$N_1 = (|\textsc{Roots}(C)| - 1)! \prod_{V_i \in C} |\textsc{Children}(V_i)|!.$$

The size of the space searched by the decoupled approach is

$$N_2 = (|\textsc{Roots}(C)| - 1)! + \sum_{V_i \in C} |\textsc{Children}(V_i)|!.$$

To simplify the analysis, we study clusterings $C$ with $m$ clusters at the top level of the hierarchy and each one of those clusters contains $m$ clusters. This continues for total of $m$ levels (each cluster has $m$ children) until each cluster in the last level contains $m$ vertices. The search space size of this clustered problem $\bar{P}$ is

$$N_1 = (m-1)! \prod_{i=1}^{m-1} \prod_{j=1}^{m^i} (m!) = (m-1)! \prod_{i=1}^{m-1} (m!)^{m^i}$$

and when solved with the decoupled approach it is

$$N_2 = (m-1)! + \sum_{i=1}^{m-1} \sum_{j=1}^{m^i} m! = (m-1)! + m! \sum_{i=1}^{m-1} m^i.$$

To show that the search space size $N_1$ for the coupled approach is exponentially larger than the space searched by the decoupled approach $N_2$, we manipulate the ratio $r = N_1/N_2$ to show that it is exponential in size.

$$
\begin{aligned}
r &= \frac{N_1}{N_2} \\
&= \frac{(m-1)!\prod_{i=1}^{m-1}(m!)^{m^i}}{(m-1)! + m!\sum_{i=1}^{m-1} m^i} \\
&\geq \frac{(m-1)!\prod_{i=1}^{m-1}(m!)^{m^i}}{m!\sum_{i=0}^{m-1} m^i} \\
&= \frac{\prod_{i=1}^{m-1}(m!)^{m^i}}{m\sum_{i=0}^{m-1} m^i} \\
&= \frac{\prod_{i=1}^{m-1} m^{m^i}\prod_{i=1}^{m-1}(m-1)!^{m^i}}{m\left(\frac{m^m-1}{m-1}\right)} \\
&= \frac{m^{\sum_{i=1}^{m-1} m^i}\prod_{i=1}^{m-1}(m-1)!^{m^i}}{m\left(\frac{m^m-1}{m-1}\right)} \\
&= \frac{m^{m\left(\frac{m^m-1}{m-1}\right)}\prod_{i=1}^{m-1}(m-1)!^{m^i}}{m\left(\frac{m^m-1}{m-1}\right)} \\
&\geq \frac{m^{m\left(\frac{m^m-1}{m-1}\right)}\prod_{i=1}^{m-1}(m-1)!^{m^i}}{m^{m+1}} \\
&= m^{m\left(\frac{m^m-1}{m-1}\right)-(m+1)}\prod_{i=1}^{m-1}(m-1)!^{m^i} \\
&\geq \prod_{i=1}^{m-1}(m-1)!^{m^i} \qquad \text{for } m \geq 2.
\end{aligned}
$$

To get a better understanding of this size, we now consider the concrete example of when $m = 3$. In this case there would be 27 vertices and 12 clusters (9 clusters with 3 vertices and 3 clusters with 3 nested clusters). The ratio $r$ would be approximately $5.9 \times 10^7$, meaning the decoupled method's search space is approximately $5.9 \times 10^7$ smaller. If we were to instead compare $N_2$ to $N_0$ then $r' = N_0/N_1 \approx 5.45 \times 10^2 4$.

Thus the size of the space searched by the decoupled approach is exponentially smaller than the search space of the coupled problem $\bar{P}$ and $P$.

### 4.6.2 Solution Quality Bounds

In this section we provide a solution quality bound for decoupled planning. We start by introducing a modified version of the problem environment that will yield the same set of solutions found by the decoupled approach. Then we step through a series of theoretical results that are used to prove the main result, which is that solutions found with decoupled planning have a cost $c$ bound by $c \leq 2\left(1 + \frac{2}{\Gamma}\right) c^*$, where $c^*$ is the cost of the optimal solution for the non-clustered path planning problem.

**Definition 4.6.1** (The modified graph $\tilde{G}$)**.** Given a graph $G = \langle V, E, w\rangle$ and a $\Gamma$-Clustering $C$, we define the *modified graph* $\tilde{G} = \langle V, E, \tilde{w}\rangle$ as a copy of $G$ with the following edge weight assignments. For each $\langle v_a, v_b\rangle \in E$, let $V_i$ be the largest cluster containing $v_a$ but not $v_b$ and $V_j$ be the largest cluster containing $v_b$ but not $v_a$, then $\tilde{w}(v_a, v_b) = \max\{w(v_x, v_y) | v_x \in V_i, v_y \in V_j\}$.

**Proposition 4.6.2.** Given a metric clustered path planning problem $\bar{P} = \langle G, \mathcal{P}, C\rangle$ and its modified clustered problem $\tilde{P} = \langle \tilde{G}, \mathcal{P}, C\rangle$, where $C$ is a $\Gamma$-Clustering with $\Gamma > 1$ and $\tilde{G}$ is the modified graph as defined in Definition 4.6.1, then the solution $p$ obtained by the decoupled approach is an optimal solution for $\tilde{P}$ and likewise, an optimal solution $\tilde{p}^*$ for $\tilde{P}$ could be obtained by the decoupled approach.

*Proof.* We prove the above result by showing that for every feasible set of included vertices $V'$, the optimal solution $\tilde{p}^*$ could be found using the decoupled approach (with the same set of included vertices). Let

$$C' = \{V_i \cap V' | V_i \in C \text{ and } V_i \cap V' \neq \emptyset\}$$

be the set of visited clusters. Let $C'_0 = \textsc{Roots}(C')$. Let the notation $\tilde{p}^*_{C''}$ represent the subset of inter-set edges between the clusters in $C''$,

$$\left\{\langle v_a, v_b\rangle \in \tilde{p} | v_a \in V''_i, v_b \in V''_j, V''_i, V''_j \in C''\right\},$$

where $C''$ has no nested clusters.

The first property we show is that $\tilde{p}^*_{C'_0}$ is a shortest tour of the root clusters of $C'$ and that $\tilde{p}^*_{C'_0}$ could be equally obtained from calling Algorithm 9 with input $(G, C, V')$. This result follows from the construction of $\tilde{G}$. The costs of the edges in the tour, depend on the sequence of the clusters in $C'_0$, not the choice of vertices that $\tilde{p}^*$ transitions to and from the root clusters. For example, if we are transitioning from $V'_i$ to $V'_j$ such that $V'_i, V'_j \in C'_0$ then each edge $\langle v_a, v_b\rangle$ with $v_a \in V'_i$ and $v_b \in V'_j$ has the same cost. Thus $\tilde{p}^*_{C'_0}$ is a shortest tour of the root clusters $C'_0$ since the cost of the tour is independent of the choice of vertices

transitioned within the clusters. The cost of the edges transitioning from $V_i'$ to $V_j'$ in $\tilde{G}$ also have equivalent cost to the coarsened graph $\tilde{G}'$ created by Algorithm 9. Thus $p_C$ from Algorithm 9 is a solution for $\tilde{p}^*_{C_0'}$ and vice versa since both paths are solutions for the same problem (find a minimum cost tour of the root clusters of $C'$).

For the second property, we show that for each $V_i' \in C'$, $\tilde{p}^*_{V_i'}$ is a shortest path of $V_i'$'s children with free choice of start and goal. Alternatively, $\tilde{p}^*_{V_i'}$ could be obtained from calling Algorithm 7 with input $(G, C, V', V_i)$, where $V_i \in C$ is used to construct $V_i' = V_i \cap V'$. This result again follows from the construction of $\tilde{G}$ using the same argument as before. The choice of first and last vertices visited in $V_i'$ is independent from the rest of the tour and the ordering of $V_i'$'s children depend only on the sequence of the children (not the vertices within). Thus $\tilde{p}^*_{V_i'}$ must also be a shortest path. Additionally in Algorithm 7, the transition costs between the children of $V_i'$ are equivalent to $\tilde{G}'$. Thus $p_{V_i}$ from Algorithm 7 is a solution for $\tilde{p}^*_{V_i'}$ and vice versa since both paths are solutions for the same problem.

Therefore, to obtain $\tilde{p}^*$ one can either solve $\tilde{P}$ optimally or solve $\bar{P}$ with the decoupled approach. □

**Proposition 4.6.3** (Inter-set Edge Weight Bound)**.** Given a metric graph $G$, its $\Gamma$-Clustering $C$, and any $V_i, V_j \in C$ such that $V_i \cap V_j = \emptyset$, then for any $v_a, v_x \in V_i$, and any $v_b, v_y \in V_j$, the following holds: $w(v_a, v_b) \leq (1 + \frac{2}{\Gamma}) w(v_x, v_y)$.

*Proof.* The above is proven using the triangle inequality and the relations from Definition 4.4.2. Specifically, we use the fact that the direct path from $v_a$ to $v_b$ is no longer than the path from $v_a$ to $v_x$ to $v_y$ to $v_b$. Then we make a series of inequality replacements using the relations from Definition 4.4.2.

$$\begin{aligned} w(v_a, v_b) &\leq w(v_a, v_x) + w(v_x, v_y) + w(v_y, v_b) \\ &\leq \beta_i + w(v_x, v_y) + \beta_j \\ &= \frac{\alpha_i}{\Gamma} + w(v_x, v_y) + \frac{\alpha_j}{\Gamma} \\ &\leq \frac{w(v_x, v_y)}{\Gamma} + w(v_x, v_y) + \frac{w(v_x, v_y)}{\Gamma} \\ &= \left(1 + \frac{2}{\Gamma}\right) w(v_x, v_y). \end{aligned}$$

□

**Lemma 4.6.4.** Consider a clustered TSP problem $\bar{P} = \langle G, \mathcal{P}, C \rangle$ with optimal solution $\bar{p}^*$ of cost $\bar{c}^*$ and its modified clustered problem $\tilde{P} = \langle \tilde{G}, \mathcal{P}, C \rangle$ with optimal solution $\tilde{p}^*$ of cost $\tilde{c}^*$, then $\tilde{c}^* \leq \left(1 + \frac{2}{\Gamma}\right) \bar{c}^*$.

*Proof.* To prove the above result we analyze the cost of $\bar{p}^*$ in $\tilde{G}$. Let $\langle v_a, v_b \rangle$ be an edge in $\bar{p}^*$ and let $V_i$ be the largest $\Gamma$-Cluster containing $v_a$ but not $v_b$ and let $V_j$ be the largest $\Gamma$-Cluster containing $v_b$ but not $v_a$ (as defined in Definition 4.6.1). We use the result from Proposition 4.6.3 to bound the weight of $\langle v_a, v_b \rangle$ in $\tilde{G}$ with $\tilde{w}(v_a, v_b) \leq (1 + \frac{2}{\Gamma}) w(v_a, v_a)$ and since every edge weight of $\bar{p}^*$ is bound by a factor of $(1 + \frac{2}{\Gamma})$ in $\tilde{G}$, then the cost of the entire tour $\bar{p}^*$ must also be bound by the same factor, $\tilde{c}(\bar{p}^*) \leq (1 + \frac{2}{\Gamma}) \bar{c}^*$. Therefore, $\tilde{c}^* \leq (1 + \frac{2}{\Gamma}) \bar{c}^*$, since $\tilde{c}^*$ represents the cost of the optimal solution of $\tilde{P}$. □

**Theorem 4.6.5.** Given a metric planning problem $P = \langle G, \mathcal{P} \rangle$ and a $\Gamma$-Clustering $C$ with $\Gamma > 1$, then the decoupled approach will find a path $p$ with cost $c$ that is bound by $c \leq \min(2 + \frac{4}{\Gamma}, 1 + \frac{13}{2\Gamma}) c^*$, where $c^*$ is the cost of the optimal solution.

*Proof.* The proof follows from Proposition 4.6.2 (the decoupled approach solves the modified problem $\tilde{P}$), Lemma 4.6.4 (bounds the modified problem $\tilde{P}$ with the clustered problem $\bar{P}$), and Lemma 4.5.11 (bounds the clustered problem $\bar{P}$ with the non-clustered problem $P$).

$$
\begin{aligned}
c &= \tilde{c} \\
&\leq \left(1 + \frac{2}{\Gamma}\right) \bar{c}^* \\
&\leq \left(1 + \frac{2}{\Gamma}\right) \min\left(2, 1 + \frac{3}{2\Gamma}\right) c^* \\
&= \min\left(2 + \frac{4}{\Gamma}, 1 + \frac{7}{2\Gamma} + \frac{3}{\Gamma^2}\right) c^* \\
&\leq \min\left(2 + \frac{4}{\Gamma}, 1 + \frac{13}{2\Gamma}\right) c^*.
\end{aligned}
$$

Therefore, $c \leq \min(2 + \frac{4}{\Gamma}, 1 + \frac{13}{2\Gamma}) c^*$. □

## 4.7 Hierarchical Planning

In this section we present our hierarchical method for solving high-level path planning problems as an improvement to decoupled planning. This approach achieves similar search

space reductions and solution quality bound.

The organization of this section is as follows: first we present a hierarchical planning algorithm for solving TSP problems; then we show that it obtains the same reduction in search space; followed by a proof of the solution quality bound; and finally we present an extension for solving non-TSP problems that achieve similar computational savings and solution quality.

### 4.7.1 A Hierarchical Method for TSP Problems

The solver approach for TSP problems (Algorithm 10) uses the clustering $C$ to decompose the problem into a set of semi-independent sub-problems. There are two external solvers that are used in conjunction with this method: SHORTESTTOUR and SHORTESTPATH. The SHORTESTTOUR method is a TSP solver. The SHORTESTPATH$(V_i, V_j, V_k)$ is a Hamiltonian path solver that finds paths for $V_j$'s children that start with $V_i$ and end with $V_k$. The solution path is returned without $V_i$ or $V_j$.

The first step in the process is to find a $\Gamma$-Clustering $C$ of $G$ (Section 4.4.2). The next step is to approximate the environment by compressing clusters on the level being solved into super vertices. The goal is to plan on the coarse levels and then extend the solution by expanding the visited super vertices. The visualization in Figure 4.7 shows how a cluster in level $n$ is represented as a super vertex in the coarsened problem. The details of the coarsening process is given in Algorithm 8.

The first tour $p_0$, is a sequence of super vertices that connect the roots clusters. The algorithm plans this path in Lines 2-4 on the graph $\tilde{G}_0$, which approximates the transition costs between the root clusters.

The next step is to expand each super vertex $V_j$ visited by $p_0$ to create $p_1$. This is done cluster by cluster. The algorithm creates the approximation $\tilde{G}_{V_j}$ of the environment for each $V_j$. The approximation includes the nested clusters within $V_j$, as well as the path's start and goal. This is visualized in level 1 of the figure and Lines 5-20 of the algorithm.

In this way, the algorithm refines the accuracy of its edges at each level. For example, suppose the tour $p_0$ transitioned from $V_i$ to $V_j$ with an edge weight of

$$\max \{w(v_a, v_b) | v_a \in V_i, v_b \in V_j\},$$

then in level 1, $p_1$ replaces that edge with a more accurate one. The new edge still transitions from $V_i$ to $V_j$, however, the edge weight now represents edges between $V_x$ and $V_y$,

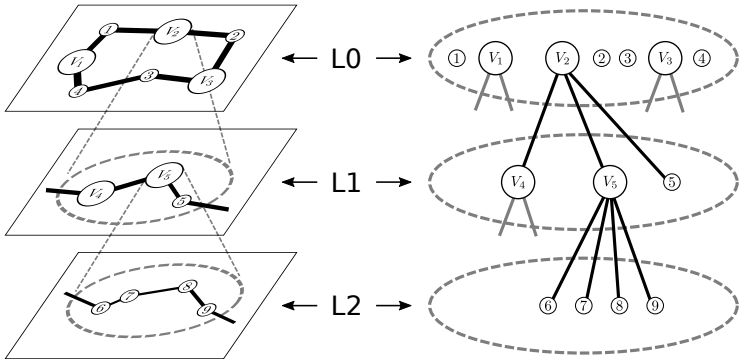



Figure 4.7: Illustration of how the hierarchical approach progresses. On the left, the progression of how the path is built. On the right, the levels of nesting in the clustering are illustrated as a forest (i.e., clusters $V_4$ and $V_5$, as well as vertex 5 are nested in cluster $V_2$).

where $V_x \subset V_i$ and $V_y \subset V_j$. As such, the new weight is

$$\max \{w(v_a, v_b) | v_a \in V_x \subset V_i, v_b \in V_y \subset V_j\}.$$

Thus each time the path is refined, the edge weight between $V_i$ and $V_j$ of the path, monotonically deceases until it reaches its final value obtained by $p_n$.

The extension process is repeated (Line 19) until a tour is found with the same resolution as $G$.

### Search Space Reduction

The search space reduction obtained by the hierarchical method is identical to the reduction obtained by the decoupled method. The only difference is when the hierarchical method finds a path visiting cluster $V_i$ (Line 18 of Algorithm 10). Here it has two additional vertices, the start and the goal. However, since these are fixed, the search space size is the same. For example, when planning a path for cluster $V_i$ of size $n = |V_i|$, the decoupled approach has a search space size of $n!$ ($n$ choices for the starting vertex, $n-1$ choices for the next vertex and so on). The search space size for the hierarchical approach is also $n!$ because we do not have a choice of start or goal vertices but we do have $n!$ for the remaining vertices.

**Algorithm 10:** HIERARCHICAL-TSP$(G, \Gamma)$

**Input:**
$G$: a graph with vertices $V$, edges $E$, and weights $w$.
$\Gamma$: a real valued constant greater than 1.

**Output:**
$p_n$: a Hamiltonian tour $G$.

```
/* Find initial coarse solution                                    */
```
$C \leftarrow$ FIND $\Gamma$-CLUSTERS$(G, \Gamma)$
$C_0 \leftarrow$ ROOTS$(C)$
$\tilde{G}_0 \leftarrow$ COARSEN$(G, C_0)$
$p_0 \leftarrow$ SHORTESTTOUR$(\tilde{G}_0)$
```
/* Extend p_0 to create p_1                                        */
```
$V_i \leftarrow p_0[-1]$
$p_1 \leftarrow []$ `/* empty array */`
**for** $x \in \{0, 1, \ldots, |p_0| - 1\}$ **do**
  $V_j \leftarrow p_0[x]$
  **if** $V_j = p_0[-1]$ **then**
    $V_k \leftarrow p_1[0]$
  **else**
    $V_k \leftarrow p_0[x + 1]$
  **if** $|V_j| = 1$ **then**
    Append $V_j$ to $p_1$
  **else**
    $C_{V_j} \leftarrow V_i \cup$ CHILDREN$(V_j) \cup V_k$
    $\tilde{G}_{V_j} \leftarrow$ COARSEN$(G, C_{V_j})$
    $p_{V_j} \leftarrow$ SHORTESTPATH$(V_i, \tilde{G}_{V_j}, V_k)$
    Append $p_{V_j}$ to $p_1$
  $V_i \leftarrow p_1[-1]$
Continue to extend $p_1, p_2, \ldots, p_n$ until we have an array of unit clusters (vertices)
**return** $p_n$

**Solution Quality**

Now we show that the hierarchical method for TSP problems, obtain solutions of cost

$$c \leq \min(2 + \frac{4}{\Gamma}, 1 + \frac{13}{2\Gamma})c^*,$$

where $c^*$ is the cost of a solution obtained from the decoupled approach.

**Lemma 4.7.1.** Given a metric TSP problem $P = \langle G, \mathcal{P} \rangle$, with an optimal solution of cost $c^*$, and a $\Gamma$-Clustering $C$ then the solution $p$ found by Algorithm 10 has a cost bound by $c \leq \tilde{c}$, where $\tilde{c}$ is the cost of a solution obtained by the decoupled approach.

*Proof.* We prove the above by comparing $p$ to the optimal solution $\tilde{p}^*$ for the clustered problem $\tilde{P} = \langle \tilde{G}, \mathcal{P}, C \rangle$, where $\tilde{G}$ is the modified graph defined in Definition 4.6.1 (Proposition 4.6.2 states that the decoupled approach and the modified problem $\tilde{P} = \langle \tilde{G}, \mathcal{P}, C \rangle$ have an equivalent set of optimal solutions).

To compare $p$ to $\tilde{p}^*$ we analyze the cost of $p$ as it is being constructed in Algorithm 10 (proof by induction). Let the function CLUSTERS operate on a partial solution path $p_x$ to return the set of super vertices/clusters being visited. Let the notation $\tilde{p}^*_{C''}$ represent the subset of inter-set edges between the clusters in $C''$,

$$\left\{ \langle v_a, v_b \rangle \in \tilde{p} | v_a \in V_i'', v_b \in V_j'', V_i'', V_j'' \in C'' \right\},$$

where $C''$ has no nested clusters.

We start with our initial solution for level 0 of the cluster hierarchy (the root clusters). Here, $p_0$ of the algorithm is an optimal path that visits the vertices in $G_0$ (the clusters in $C_0$). The cost of an edge $\langle V_i, V_j \rangle$ in $G_0$ has equal cost of any edge in $\tilde{G}$ that transitions from $V_i$ to $V_j$. Thus, the path $\tilde{p}^*_{C_0}$ built from edges in $\tilde{p}^*$ that connect the clusters in $C_0$ has the same total cost as $p_0$, since $\tilde{p}^*$ is an optimal tour of $C_0$ (Proposition 4.6.2).

In level 1 of the algorithm, the path $p_1$ is constructed to visit all the expanded super vertices of the clusters visited in $p_0$ (assign $p_1 = p_0$ to start). We incrementally expand the super vertices inherited by $p_0$. This view point is equivalent to how Algorithm 10 builds $p_1$.

The induction step is as follows. We start with a path $p_1$ that has equal or lower cost than the path $\tilde{p}^*_{\text{CLUSTERS}(p_1)}$. Then we expand the next super vertex in $p_1$ to show that this new path $p_1'$ has equal or lower cost than its counterpart, $\tilde{p}^*_{\text{CLUSTERS}(p_1')}$.

Given a super vertex $V_j$ in $p_1$, which is preceded by $V_i$ and followed by $V_k$. The algorithm constructs $\tilde{G}_{V_j}$ and finds the path $p_{V_j}$ that visits the $V_j$'s children (Line 18). The first thing

to note is that the incoming edges and outgoing edges for $V_j$ in $\tilde{G}_{V_j}$ are more accurate than they were before the expansion. They were $w(V_i, V_j) = \max\{w(v_a, v_b) | v_a \in V_i, v_b \in V_j\}$ and $w(V_j, V_k) = \max\{w(v_a, v_b) | v_a \in V_j, v_b \in V_k\}$ and now in $\tilde{G}_{V_j}$ they have lesser or equal cost. For example, $w(V_i, V_x) = \max\{w(v_a, v_b) | v_a \in V_i, v_b \in V_x \subset V_j\}$. If we were to use $\tilde{p}^*_{C_{V_j}}$ as a candidate path for $p_{V_j}$ then $p_1$ after the expansion would have lesser or equal cost to its counterpart. This is due to the cost of the path visiting $V_j$ being equal in both cases. However, the incoming edges and outgoing edges have lesser or equal cost and the rest of the path has lesser or equal cost. Thus, Line 18 in the algorithm would choose $p_{V_j}$ in such a way to create $p_1$ with equal or lower cost than the candidate path. Therefore, if we start with a path $p_1$ of equal or lower cost than its counter part $\tilde{p}^*_{\text{CLUSTERS}(p_1)}$ and expand it as in Algorithm 10, the expanded path $p'_1$ will have equal or lower cost than its counterpart.

This is the case when we start with $p_1 = p_0$. We repeat this step until all of the original super vertices of $p_0$ are expanded and then apply the same induction logic for each subsequent level until we get a path $p_n$ that can no longer be expanded with a solution cost less than or equal to $\tilde{p}^*$. Therefore, $c \leq \tilde{c}$. □

**Theorem 4.7.2** (Solution quality)**.** Given a metric TSP problem $P = \langle G, \mathcal{P} \rangle$, an optimal solution of cost $c^*$, and a $\Gamma$-Clustering $C$, then the solution $p$ found by Algorithm 10 has a cost bound by

$$c \leq \min(2 + \frac{4}{\Gamma}, 1 + \frac{13}{2\Gamma})c^*.$$

*Proof.* The proof follows from Lemma 4.7.1 (the hierarchical approach finds solutions of equal or lower cost than the decoupled approach) and Theorem 4.6.5 (the decoupled approach finds solutions of cost $c \leq \min(2 + \frac{4}{\Gamma}, 1 + \frac{13}{2\Gamma})c^*$). □

### 4.7.2 A Hierarchical Method for Non-TSP Problems

The hierarchical approach builds a path by planning cluster $V_i \in C$ semi-independently of the rest of the graph (it is semi-independent since it also considers the start and goal external to the cluster). This poses a challenge for solving non-TSP problems since the set of vertices included in the solution path is not known ahead of time. We provide Algorithm 11 to solve this problem by extending Algorithm 10.

The extension (Algorithm 11) combines the decoupled approach with the hierarchical approach. The algorithm first determines the set of included vertices by solving the problem using a decoupled approach (Line 2), then it finds a near optimal solution path using the hierarchical approach for TSP problems (Line 4).

**Algorithm 11:** HIERARCHICAL$(G, C, \mathcal{P})$

**Input:**

$G$: a graph with vertices $V$, edges $E$, and weights $w$.

$C$: a clustering.

$\mathcal{P}$: a set of constraints.

**Output:**

$p$: a tour of some or all of $V$ that satisfies $\mathcal{P}$.

1 Construct the modified graph $\tilde{G}$

2 Find the shortest path $\tilde{p}^*$ on $\tilde{G}$ that satisfies $\mathcal{P}$

3 Let $G'$ be induced by $G$ with vertices $V[\tilde{p}^*]$

4 $p \leftarrow$ HIERARCHICAL-TSP$(G')$

5 Return $p$

### Search space size

Algorithm 11 searches the same space as the decoupled approach, at most twice. The first time is in Line 2 when it finds the set of included vertices $V[\tilde{p}^*]$ and then searches it again in Line 4 when it uses the hierarchical method for TSP problems (this search space may be smaller, since the set of included vertices may be less than $V$). Thus the hierarchical method for non-TSP problems achieves similar search space reductions.

### Solution Quality

Now we show that the hierarchical extension for non-TSP problems achieves the same solution quality bounds.

**Theorem 4.7.3** (Solution quality)**.** Given a metric non-TSP problem $P = \langle G, \mathcal{P} \rangle$, an optimal solution of cost $c^*$, and a $\Gamma$-Clustering $C$, then the solution $p$ found by Algorithm 10 has a cost bound by

$$c \leq \min(2 + \frac{4}{\Gamma}, 1 + \frac{13}{2\Gamma})c^*.$$

*Proof.* The hierarchical method for non-TSP problems also achieves the same solution quality bounds as the decoupled approach. This is easily proven using Theorem 4.6.5 and Lemma 4.7.1. In Line 2 of the algorithm we use the decoupled approach to find a solution of cost $c$ bound by $c \leq \min(2+\frac{4}{\Gamma}, 1+\frac{13}{2\Gamma})c^*$, where $c^*$ is the optimal solution cost. In Line 4

we improve the solution quality, thus preserves the cost bound. Therefore, Algorithm 11 produces solutions with cost $c$ bound by

$$c \leq \min(2 + \frac{4}{\Gamma}, 1 + \frac{13}{2\Gamma})c^*.$$

□

## 4.8 Experiments

This section presents the results of our experiments that demonstrate the effectiveness of Γ-Clustering for solving discrete path planning problems. We benchmark the coupled planning approach and the hierarchical planning approach against a non-clustered approach. All three methods use an ILP solver to find minimum length paths/tours. The benchmark consists of three different vehicle routing problems, TSP, sample collection, and period routing problems. The TSP instances are taken from TSPLIB [77], which provide a variety of symmetric and non-symmetric environments. The sample collection and the period routing problems are taken from Section 3.6. They demonstrate the effectiveness of the proposed approaches on non-TSP problems. Additionally, we compare the quality of Γ-Clusters to clusters obtained by five other clustering approaches. In the rest of this section, we detail the generation of the sample collection and period routing problems followed by an analysis of the results.

### 4.8.1 Problem Library

**Multi-Agent Sample Collection** The setup of the sample collection problem is found in Section 3.6.2 and the specifics are as follows. The robot's environment is an obstacle free Euclidean environment of size $1000 \times 1000$ meters. The instances contain 20 to 100 locations which are uniformly randomly distributed in the environment (easier instances have less locations). One location is randomly chosen to be the home location and the remaining locations randomly contain either a small, medium, or large sample. The distribution of the samples are 3:1 for small to large and 2:1 for small to medium. The collection of samples in the environment contain 10-40 different minerals (easier problems have less minerals). Each small sample has a mineral that is chosen uniformly randomly, similarly medium samples have two minerals and large samples have three minerals chosen uniformly randomly (medium and large samples may choose the same mineral more than once). The small robots have batteries that allow them to travel 3000 meters and the large robots

have batteries that allow them to travel 1500 meters. In Table 4.1 the naming convention `#loc_#mnr` captures the number of locations and minerals of the problem.

**Period Routing** The setup of the period routing problem is found in Section 3.6.3, with the exception that back to back visits are allowed for the first and last periods. The specifics of the problem instances are as follows. The environment is an obstacle free Euclidean environment of size $1000 \times 1000$ meters. The instances contain 25 to 35 locations which are uniformly randomly distributed in the environment (easier instances have less locations). One location is randomly chosen to be the home and the remaining locations are customer locations. There are five service periods for which each customer may require either one, two, or three periods of service (uniformly and randomly assigned). In Table 4.1 the naming convention `#loc_a` captures the number of locations of the problem.

### 4.8.2 Setup and Execution

The Γ-Clustering algorithm was implemented in Python and run on an Intel Core i7-6700, 3.40GHz with 16GB of RAM. All Γ-Clustering experiments were conducted with a $\Gamma = 1.000001$, for which Theorem 4.5.11 implies that the coupled planning approach finds solutions within $\min\left(2, 1 + \frac{3}{2\Gamma}\right)$-factor of the optimal. The hierarchical approach finds solutions within $\min(2 + \frac{4}{\Gamma}, 1 + \frac{13}{2\Gamma})$-factor of the optimal. However, the experiments will show that the observed gap in performance is considerably closer to optimal than the bound. The ILP expression of the problem and clustered problem were solved on the same computer using Gurobi, also accessed through python.

The following is an ILP expression for TSP problems. We use this as a starting point

for the expressions used in the coupled, hierarchical, and non-clustered methods.

minimize

$$\sum_{a=1}^{|V|}\sum_{b=1}^{|V|} e_{a,b}\ w(v_a, v_b) \tag{4.1}$$

subject to

$$\sum_{b=1}^{|V|} e_{a,b} = 1, \quad \text{for each } a \in \{1, 2, \ldots, |V|\} \tag{4.2}$$

$$\sum_{a=1}^{|V|} e_{a,b} = 1, \quad \text{for each } b \in \{1, 2, \ldots, |V|\} \tag{4.3}$$

$$\sum_{\forall v_a \in V_i, v_b \notin V_i} e_{a,b} = 1, \quad \text{for each } i \in \{1, 2, \ldots, m\} \tag{4.4}$$

$$\sum_{\forall v_a \notin V_i, v_b \in V_i} e_{a,b} = 1, \quad \text{for each } i \in \{1, 2, \ldots, m\} \tag{4.5}$$

$$\sum_{e_{a,b} \in E'} e_{a,b} \leq |E'| - 1, \quad \text{for each subtour } E' \tag{4.6}$$

The formulation was adapted from a TSP formulation found in [27], where the Boolean variables $e_{a,b}$ represent the inclusion/exclusion of the edge $\langle v_a, v_b \rangle$ from the solution. Constraints (4.2) and (4.3) restrict the incoming and outgoing degree of each vertex to be exactly one (the vertex is visited exactly once). Similarly Constraints (4.4) and (4.5) restrict the incoming and outgoing degree of each cluster to be exactly one (these constraints are present only in the clustered version of the problem). Constraint (4.6) is the subtour elimination constraint, which is lazily added to the formulation as conflicts occur to avoid expressing an exponential number of constraints.

We use the above expression to find one tour or replicate it to find a series of independent tours. We can also use the above to find shortest paths. This allows us to solve TSP problems and the sub-problems SHORTESTPATH and SHORTESTTOUR used by the hierarchical approach.

To adapt the above for non-TSP problems we change the equality of (4.2)-(4.5) to inequalities ($\leq 1$) and add the following for each vertex $v_a \in V$

$$Nv_a - \sum_{b=1}^{|V|}(e_{a,b} + e_{b,a}) \geq 0,$$

where $N$ is some number bigger than $|E|$.

We solve the shortest tour problem in Algorithm 11, Line 2 with the decoupled approach by constructing a set of decoupled non-TSP problems and solving them as one instance. Recall from Section 4.6 that some additional logic is required to ensure that the decoupled approach generates a solution tour instead of a path. This logic is required for the sample collection and period routing problems. The construction of both of these problems contain a home location that must be visited. We use the home location to create a hybrid coupled/decoupled problem, where the clusters that are not within the root cluster containing the home vertex are solved with the decoupled approach and the clusters within the root cluster that contain the home vertex are solved with the coupled approach. Clusters that are solved with the decoupled approach that appear in the coupled problem, appear as a super vertices.

The problem's constraints $\mathcal{P}$, also need to be encoded into the formulation. For example, the sample collection problem requires additional constraints to ensure each mineral in the environment is collected. Concretely, if location $v_a$ and $v_b$ were the only locations to contain mineral $x$ then we would add the following to the formulation

$$\sum_{r=1}^{|R|} v_{a,r} + v_{b,r} \geq 1,$$

where $v_{a,r}$ is the variable for robot $r$'s vertex location $v_a$ and $R$ is the set of robots. Similarly, to restrict back to back service for the period routing problem at location $v_a$, we add the following constraint

$$v_{a,p} + v_{b,p+1} \leq 1, \text{ for each } p \in \{1, 2, \ldots, |P| - 1\},$$

where $v_{a,p}$ is the variable for the robot to visit location $v_a$ on day $p$ and $P$ is the set of days.

### 4.8.3 Path planning with Γ-Clusters

In this section we benchmark the coupled approach and the hierarchical approach against a non-clustered approach. The benchmark includes TSP, sample collection, and period routing problems. Each problem is solved three times and the average error and run times are reported (run time includes time to find clusters as well as planing time). In the case of the multi-agent sample collection problem and the periodic monitoring problem, the error is measured from the best known solution (optimals are not known). Each problem

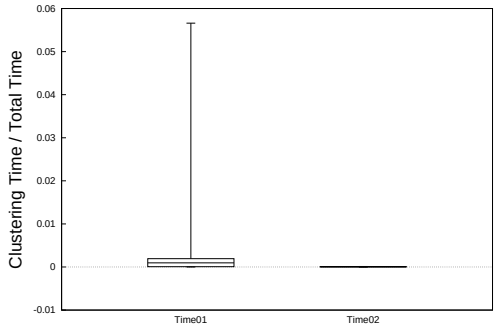



Figure 4.8: Box plot of the clustering time ratio with respect to the Γ-Clustering approach. The data is categorized by instances that did not time out (Time01) and instances that did time out (Time02).

is given a maximum time budget of 900 seconds to find the lowest cost solution tour. If the time budget is reached before termination, the best solution up to that point is returned. The TSP instances shown in Table 4.2 are instances that have Γ-Clusters and are difficult enough so that the non-clustered approach took more than 30 seconds to solve but was also able to find solutions within 50% of optimal.

The first fact to note is that all three libraries have Γ-Clusters. Specifically, 63 out of 70 instances in the TSPLIB have Γ-Clusters and every sample collection and period routing problem contained Γ-Clusters. This shows that Γ-Clusters naturally occur in path planning problems since instances from the TSPLIB, along with the other two sets of problems represent a diverse collection of path planning environments. Overall there were more than 3700 non-trivial Γ-Clusters (clusters of size two or more) in TSPLIB and 242 non-trivial Γ-Clusters in the remaining instances. Additionally, the time spent finding Γ-Clusters was insignificant compared to the total time spent finding path planning solutions. Specifically, most instances spent less than 1% of their time finding clusters (all spent less than 6%). Although, the data is not explicitly presented in this chapter we noticed a trend, that as the instances became more difficult, the ratio of time spent finding Γ-Clusters to the time spent finding solutions, shrank.

The results in Table 4.2 show how the two approaches improved the solver efficiency. Specifically, the non-clustered approach used 90% of their time budget compared to 73% and 41% for the coupled and hierarchical approaches, respectively. A few noteworthy examples are gr96, bier127, and ch130. In all three examples the non-clustered approach took more than 20 times longer than the other two approaches. We see a similar improvement in efficiency for the coupled and hierarchical approaches. For example, the coupled approach took more than 20 times longer for the instances kroB150, kroA200, and gr202.

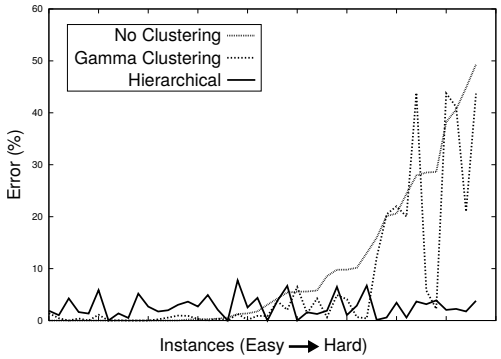



Figure 4.9: A plot of the average error for each solver method. Instances are sorted from least to most difficult.

The results show these approaches also achieve good solution quality. Specifically, all solutions obtained by these approaches were within 50% of optimal and most were within 5%. As a comparison, the non-clustered approach was able to solve 24 out of 44 instances to within 5% of optimal while the coupled and hierarchical approaches were able to respectively solve 32/44 and 42/44 instances to within 5% of optimal. Additionally, there is a trend in the data, that as instances become more difficult to solve, the performance of the solvers degrade. However, we see from the data that the coupled approach degrades slower than the non-clustered approach and the hierarchical approach degrades at an even slower rate than the other two approaches. This trend is shown more clearly in Figure 4.9, which plots the percent error verses the difficulty of the instance. As we can see from the figure, the hierarchical approach on average incurs a random error centred around 5% and that error does not degrade as the instances become more difficult. Thus, as instances become more difficult to solve, we employ the coupled approach or the hierarchical approach to find better cost solutions.

*Remark* 4.8.1 (Using ILP solvers). It is worth emphasizing that although we use an ILP solver to solve path planning problems, we are not recommending using ILP solvers in general. For example, there are some very good TSP solvers that would greatly outperform a ILP solver for TSP instances. We are simply using ILP solvers to demonstrate how Γ-Clustering can be used to improve solver efficiency for path planning problems. Many discrete path planning problems are solved with ILP solvers and as such, we hope our results provide insight for how effective the proposed approaches are for path planning problems.

| | | % Error / Time ($s$) | | |
|---|---|---|---|---|
| **TspLib** | $\|C\|$ | No Clustering | Γ-Clustering | Hierarchical |
| pr76 | 27 | **0.00** / 40 | 1.40 / 11 | 1.88 / 1 |
| st70 | 23 | **0.00** / 45 | 0.44 / 13 | 1.04 / 1 |
| lin105 | 42 | 0.00 / 225 | 0.00 / 8 | 4.25 / 15 |
| kroE100 | 43 | **0.00** / 414 | 0.39 / 20 | 1.63 / 1 |
| kroC100 | 46 | 0.00 / 445 | 0.00 / 12 | 1.35 / 1 |
| kroA100 | 44 | **0.00** / 602 | 1.11 / 23 | 5.86 / 3 |
| gr96 | 32 | **0.00** / 845 | 0.05 / 38 | 1.37 / 3 |
| gr137 | 44 | **0.00** / 900 | 0.00 / 154 | 2.68 / 36 |
| bier127 | 37 | **0.00** / 900 | 0.23 / 24 | 1.74 / 3 |
| kroD100 | 42 | **0.00** / 900 | 0.57 / 471 | 1.98 / 2 |
| kroB100 | 42 | **0.01** / 900 | 0.96 / 900 | 3.02 / 12 |
| ch130 | 59 | **0.20** / 900 | 0.90 / 29 | 3.63 / 2 |
| ch150 | 53 | **0.22** / 900 | 0.31 / 900 | 2.70 / 198 |
| kroB150 | 65 | 0.35 / 900 | **0.35** / 395 | 2.08 / 1 |
| kroA150 | 58 | 1.37 / 900 | **0.15** / 492 | 2.52 / 31 |
| rat195 | 57 | 1.75 / 900 | **0.89** / 900 | 4.36 / 46 |
| kroA200 | 82 | 5.59 / 900 | **1.38** / 900 | 1.59 / 15 |
| pr124 | 18 | 5.78 / 900 | 4.24 / 900 | **1.26** / 900 |
| gr202 | 74 | 8.56 / 900 | **0.64** / 703 | 1.90 / 18 |
| pr136 | 48 | 9.78 / 900 | **4.88** / 900 | 6.48 / 900 |
| kroB200 | 80 | 10.16 / 900 | **0.67** / 900 | 2.86 / 900 |
| pr107 | 6 | 13.01 / 900 | **0.40** / 900 | 6.71 / 900 |
| a280 | 11 | 20.11 / 900 | 20.46 / 900 | **0.58** / 900 |
| pr144 | 40 | 24.40 / 900 | 20.03 / 900 | **0.57** / 900 |
| tsp225 | 66 | 28.50 / 900 | 5.76 / 900 | **3.14** / 450 |
| gr229 | 73 | 28.63 / 900 | **2.24** / 900 | 3.88 / 900 |
| pr152 | 44 | 40.64 / 900 | 41.15 / 900 | **2.24** / 900 |
| gil262 | 98 | 44.81 / 900 | 21.00 / 900 | **1.75** / 65 |
| **Multi-Agent Sample Collection** | | | | |
| 20loc_20mnr | 9 | 0.00 / 785 | 0.00 / 596 | 0.01 / 15 |
| 20loc_10mnr | 10 | 0.00 / 900 | 0.00 / 900 | 0.51 / 7 |
| 40loc_10mnr | 16 | 0.27 / 900 | **0.00** / 900 | 4.90 / 193 |
| 60loc_10mnr | 22 | 0.57 / 900 | 0.00 / 900 | 0.00 / 100 |
| 40loc_20mnr | 17 | 3.04 / 900 | 0.87 / 900 | **0.00** / 810 |
| 40loc_40mnr | 13 | 5.38 / 900 | **2.04** / 900 | 6.67 / 810 |
| 80loc_10mnr | 31 | 9.79 / 900 | 4.12 / 900 | **1.05** / 810 |
| 100loc_10mnr | 38 | 27.93 / 900 | 43.86 / 900 | **3.66** / 810 |
| **Period Routing** | | | | |
| 20loc_a | 6 | 0.00 / 900 | 0.00 / 900 | 5.18 / 810 |
| 20loc_b | 9 | **1.24** / 900 | 1.28 / 900 | 7.73 / 1 |
| 40loc_a | 15 | 4.23 / 900 | **3.75** / 900 | 3.79 / 343 |
| 30loc_a | 13 | 5.55 / 900 | 6.46 / 900 | **0.00** / 420 |
| 40loc_b | 14 | 15.94 / 900 | 12.12 / 900 | **0.13** / 810 |
| 30loc_b | 9 | 20.60 / 900 | 22.01 / 900 | **3.43** / 810 |
| 50loc_a | 21 | 38.20 / 900 | 43.75 / 900 | **2.05** / 810 |
| 50loc_b | 23 | 49.32 / 900 | 43.71 / 900 | **3.80** / 810 |

Table 4.1: Experimental results for the TSP, sample collection, and the periodic routing problems. We report the average % error and solver time for each instance as well as the number of clusters $|C|$. The solver method with the best average error is shown in bold. Results are sorted from least to most difficult for the non-clustering method.

### 4.8.4 Other Clustering Methods

In this section we compare the quality of Γ-Clusters to the quality of clusters obtained by other methods.

The ideal test of a clustering's quality for path planning is a measure of how much the optimal path(s) degrade in quality when the clustered problem is solved (Problem 4.3.3 solved with the coupled approach) instead of the non-clustered problem (Problem 4.3.2). Solving these problems (finding an optimal) may require large amounts of time and so we opted to make this comparison on TSP problem instances since they are easier to solve than non-TSP problems. Additionally, the TSP problem instances we solve cover a diverse set of environments, thus giving us a good basis for comparison.

We compare the Γ-Clustering method to a set of community structure methods from the iGraph library [16]. Specifically, we tested against a fast greedy approach [14], Newman's leading eigenvector method [61], a multilevel algorithm [8], spinglass [76], and walktrap [72].

The objective of community structures is to find clusters that are intimately connected in a sparse network (a multigraph). The connections (edges) usually represent a relationship or common trait between the nodes, such as a friendship in a social network. Each connection is represented with an edge; more edges mean more shared traits.

To utilize community structures for path planning (find clusters of locations that are in close proximity) we transform the problem graph into a network by "inverting" the weighted graph into a unweighted multigraph (network). Specifically, we translate an edge $v_{i,j}$ with weight $w(i,j)$ in the problem graph to $\max_{j,k} w(j,k) - w(i,j)$ edges in the network. The nodes in the new network, are highly connected, if they were in close proximity in the original graph. Thus the obtained community structures in the network resemble the type of clusters we study in this chapter.

The goodness of a community structure is often measured with a metric called modularity [62] (not necessarily the same metric for each algorithm), meaning that many community methods work towards the same goal. However, since the task of finding the optimal clustering is **NP**-hard, the different algorithms often produce different results.

Some of the methods in the iGraph library provide metadata that gives insight into the optimal number of clusters and how the clusters were built. We used this information and the threshold modularity parameter to produce clusterings of a similar size to those found by the Γ-Clustering approach.

The results of the experiments are given in Table 4.2. They show that the quality of the Γ-Clusterings often outperform the quality of the clusterings found by other methods.

| | Fastgreedy | | Newman's | | Multilevel | | Spinglass | | Walktrap | | Γ-Clustering | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\|C\|$ | % Error | $\|C\|$ | % Error | $\|C\|$ | % Error | $\|C\|$ | % Error | $\|C\|$ | % Error | $\|C\|$ | % Error |
| pr76 | 32 | 7.86 | 27 | 11.69 | 29 | 7.41 | 38 | 8.18 | 43 | 6.16 | 27 | **1.40** |
| st70 | 32 | 8.44 | 27 | 5.48 | 23 | 4.00 | 30 | 3.70 | 32 | 6.07 | 23 | **0.44** |
| lin105 | 52 | 9.31 | 43 | 19.15 | 52 | 16.13 | 60 | 16.67 | 39 | 12.46 | 42 | **0.00** |
| kroE100 | 35 | 8.68 | 38 | 10.15 | 45 | 9.30 | 47 | 8.46 | 52 | 13.44 | 43 | **0.39** |
| kroC100 | 40 | 3.68 | 42 | 13.93 | 56 | 2.82 | 51 | 3.87 | 51 | 13.15 | 46 | **0.00** |
| kroA100 | 48 | 15.38 | 44 | 7.81 | 39 | 8.35 | 37 | 4.92 | 41 | 8.81 | 44 | **1.11** |
| gr96 | 25 | 6.06 | 27 | 6.85 | 25 | 6.06 | 27 | 6.47 | 14 | 2.96 | 32 | **0.05** |
| bier127 | 0 | 0.00 | 0 | 0.00 | 0 | 0.00 | 0 | 0.00 | 0 | 0.00 | 37 | 0.23 |
| kroD100 | 48 | 12.56 | 47 | 17.79 | 45 | 8.14 | 53 | 10.90 | 41 | 7.29 | 42 | **0.57** |
| gr137 | 30 | 9.71 | 63 | 19.10 | 51 | 17.15 | 52 | 13.51 | 47 | 15.63 | 44 | **0.00** |
| kroB100 | 43 | 10.03 | 38 | 11.07 | 40 | 9.25 | 43 | 8.64 | 46 | 9.66 | 42 | **0.92** |
| ch130 | 58 | 4.44 | 56 | 12.14 | 59 | 4.12 | 64 | 8.25 | 63 | 9.44 | 59 | **0.90** |
| ch150 | 75 | 10.55 | 68 | 17.57 | 58 | 11.03 | 72 | 12.62 | 51 | 9.24 | 53 | **0.31** |
| kroB150 | 67 | 6.89 | 72 | 9.57 | 60 | 3.57 | 67 | 3.11 | 75 | 15.76 | 65 | **0.35** |
| kroA150 | 62 | 11.17 | 70 | 11.10 | 57 | 8.74 | 63 | 9.95 | 78 | 10.36 | 58 | **0.15** |
| rat195 | 93 | 18.55 | 105 | 19.54 | 91 | 16.23 | 95 | 19.50 | 116 | 16.01 | 57 | **0.65** |
| kroA200 | 99 | 10.81 | 97 | 12.29 | 86 | 6.66 | 90 | 9.37 | 87 | 8.48 | 82 | **0.73** |
| pr124 | 70 | 10.36 | 52 | 11.64 | 68 | 9.44 | 55 | 8.28 | 63 | 11.99 | 18 | **0.08** |
| gr202 | 0 | 0.00 | 0 | 0.00 | 0 | 0.00 | 0 | 0.00 | 0 | 0.00 | 74 | 0.57 |
| pr136 | 62 | 1.41 | 62 | 1.41 | 62 | 1.41 | 57 | 2.78 | 58 | 1.41 | 48 | 4.88 |
| kroB200 | 95 | 12.62 | 103 | 17.45 | 81 | 11.04 | 84 | 13.08 | 95 | 11.96 | 80 | **0.52** |
| pr107 | 60 | 0.62 | 68 | 0.84 | 60 | 0.62 | 66 | 3.00 | 34 | 0.00 | 6 | 0.00 |
| a280 | 94 | 13.42 | 133 | 14.35 | 112 | 11.63 | 104 | 15.86 | 113 | 19.00 | 11 | **0.12** |
| pr144 | 58 | **2.89** | 66 | 5.25 | 58 | 6.85 | 82 | 6.79 | 72 | 8.91 | 40 | 3.34 |
| tsp225 | 99 | 9.24 | 108 | 11.41 | 90 | 9.35 | 94 | 11.26 | 85 | 15.02 | 66 | **1.05** |
| gr229 | 37 | 2.71 | 78 | 12.21 | 36 | 3.53 | 67 | 5.70 | 51 | 15.03 | 73 | **0.88** |
| pr152 | 54 | 1.02 | 74 | 7.39 | 53 | 1.02 | 66 | 3.08 | 47 | 7.06 | 44 | 3.76 |
| gil262 | 126 | 12.24 | 109 | 13.75 | 101 | 11.23 | 111 | 11.02 | 130 | 10.98 | 98 | **0.46** |

Table 4.2: Results comparing the quality of the clusterings on TSP instances for the different clustering methods. The results with the lowest percent error are shown in bold.

Specifically, the Γ-Clustering method produced better quality clustered problems for 22 out of 28 TSP instances and the average error is less than 1% compared to an average error of more than 7% for each non-Γ-Clustering approach.

## 4.9 Summary

This chapter presented two methods for pruning high-level path planning solutions. Both methods lead to an improvement of solver efficiency and performance. The two approaches, *coupled planning* and *hierarchical planning* relied on the the new clustering method Γ-Clustering (also presented in this chapter). The chapter provided an efficient algorithm for finding the optimal Γ-Clustering and proved that both planning approaches find solutions within a constant factor of the optimal.

We validated the two planning approaches on a benchmark of three different path planning problems. The benchmark compares these approaches to a non-clustered approach. The comparisons were done with an ILP solver and a fixed time budget (the proposed approaches are used in tandem with the ILP solver to improve its efficiency). The results show that both approaches improve solver efficiency, where the hierarchical approach

achieves more time savings than the coupled approach but poorer solution quality for instances solved within the time budget. Both approaches found solutions much closer to optimal than their quality bounds — most solutions were within 10% of optimal. For the more difficult instances the results showed that these approaches maintained their performance longer than the non-clustered approach (the hierarchical approach maintained its performance longer than the coupled approach). Thus these approaches can be used to find higher quality solutions than the non-clustered approach for more difficult instances. Additionally, this chapter compared the quality of Γ-Clusterings to clusterings found by five other methods. The results showed that Γ-Clusterings provided the highest quality solutions for use in path planning.

# Chapter 5

# Conclusions

This thesis concentrated on solving high-level path planning problems. We started out learning how to solve high-level path planning problems with an ILP approach and then learned about an alternative approach called SAT-TSP in Chapter 3. This approach used the problem language SAT-TSP to model high-level path planning problems. The problem itself is a combination of two well known problems, SAT and TSP. The structure of this problem allowed us to leverage the SMT framework to combine state-of-the-art SAT and TSP solvers together to create the CBTSP solver. We used this solver to find solutions for a variety of path planning problems and the results showed that the CBTSP solver often outperformed a commercial ILP solver. Thus making CBTSP a good candidate for solving high-level path planning problems. The CBTSP solver is available at `https://github.com/fcimeson/cbTSP`.

In Chapter 4, we explored two approaches for pruning high-level path planning solutions and thus improving solver efficiency. The first approach, *coupled planning*, achieves its performance gains by trading off solution quality. The trade however, is not a drastic. We proved in this chapter that solutions found by this approach are within a constant factor of $\min\left(2, 1 + \frac{3}{2\Gamma}\right)$ of the optimal. The second approach, *hierarchical planning*, provides a more aggressive method to achieve even more performance gains while still finding solutions within a constant factor of $\min(2 + \frac{4}{\Gamma}, 1 + \frac{13}{2\Gamma})$ of the optimal. We tested these approaches with an ILP solver on a series of path planning problems. The results showed that, not only are these approaches more efficient than a non-clustered approach, they can also be used to find higher quality solutions when solver time is limited.

Both of these approaches rely on a new clustering method introduced in this chapter called Γ-Clustering. We provided an efficient method for find the optimal Γ-Clustering.

We compared the quality of Γ-Clusterings to clusterings found by other methods, to show that Γ-Clusterings are more useful for path planning problems.

## 5.1 Future Work

### 5.1.1 SAT-TSP

Below we provide a list of the areas of future work for related to SAT-TSP.

**A Modelling Language** We do not expect that every user will be willing to model their problem as a SAT-TSP expression. As such we are in the process of developing a modelling language for SAT-TSP. This language will allow the user to express/model their problem in a more user-friendly format. The format will preserve the important structures our software needs so that it can translate the model to SAT-TSP. Additionally, the use of a modelling language will allow the user to separate the problem definition from the problem instance. For example, one would be able to express the TSP problem as a model and then separately express the problem instance as a transition environment (a graph). This benefits the user in two ways: readability and repeatability. The problem model is more readable because the large environmental data is not cluttering the problem definition and the problem model can be reused to prevent user error in re-specifying the problem, making it more repeatable.

**Search Heuristics** We are working on improving the CBTSP solver, specifically, CBTSP's search heuristics. Currently CBTSP uses the DPLL search heuristic for choosing the order of variable assignments. We believe incorporating knowledge of the TSP problem into the heuristics would improve the solvers performance.

**Other SMT Theories** Another area of development, is to expand upon the CBTSP solver by incorporating more SMT theories (currently CBTSP only uses one theory, the TSP theory). In this way we could natively handle more path planning constraints such as ordering constraints.

**SMT Propagation** We can further improve the CBTSP solver by incorporating SMT propagation. Currently the only SMT technique CBTSP uses clause negations. If we were

to incorporate into the SMT theories some knowledge of the problem structure, such as set cardinality constraints and/or ordering constraints. We could use those theories to propagate variable assignments to the SAT solver instead of encoding those constraints as SAT formulae. Briefly consider the "exactly one-in-a-set" cardinality constraint for the set $S$ as an example. In this case, if the SAT solver chose one of the literals in $S$ to be true, then the theory would propagate the knowledge that all other literals in the set must be false. This example reduces the size of the SAT formula by $|S|^2$ which we expect would improve the solver's efficiency.

**Collisions** We plan to investigate the effectiveness of using SAT-TSP for collision avoidance problems. Our approach would extend the environment graph to incorporate discretized time steps. In this way, we can use the location/time vertices to prohibit multiple robots from occupying the same location during the same discrete time step as in [102].

**Parallelization** We are looking into upgrading CBTSP to utilize multiple threads. To accomplish this we would implement a multi-threaded SAT solver, which in turn would make threaded calls to a TSP solver.

### 5.1.2 Γ-Clustering

The areas of future work for using Γ-Clusterings to improve solver efficiency are as follows.

**Improve Solution Quality Bounds** We are currently working towards improving the tightness of the solution quality bounds presented in Theorems 4.5.11, 4.7.2, and 4.7.3. Additionally, we are trying to find problem instances that better characterize the tightness of these bounds.

**Parallelization** We plan on developing new algorithms that use parallel processes to find high quality solution paths for discrete path planning problems. The decoupled and hierarchical planning approach is setup to work with parallelization (decoupled more than hierarchical). To parallelize the hierarchical approach we can change how the algorithm expands the super vertices in the coarsened tour. Specifically, we could use parallel processes to expand every other super vertex in the tour. Then, after those processes are done we would again use parallel processes to expand the remaining super vertices.

**Dynamic Planning** We are currently working on techniques based on Γ-Clusterings for dynamic planning. Here Γ-Clusters are used to decompose an existing solution path into regions that may be preserved and regions that need to be replanned. Then the hierarchical approach is used to fill in the missing regions of the path and find solutions that are within a constant factor of the optimal.

**Applications** Finally, we plan to search for more applications that may benefit from the Γ-Clustering approaches presented in this thesis. Specifically, we are looking to find common solver approaches that can be improved by the proposed approaches.

# References


[1] IBM ILOG CPLEX Optimizer, 2010.

[2] Tobias Achterberg. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[3] David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. Concorde TSP solver. Accessed: 2015-03-05.

[4] David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[5] Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM (JACM)*, 45(5):753–782, 1998.

[6] Luca Bertazzi and M Grazia Speranza. Inventory routing problems with multiple customers. *EURO Journal on Transportation and Logistics*, 2(3):255–275, 2013.

[7] Graeme Best, Jan Faigl, and Robert Fitch. Multi-robot path planning for budgeted active perception with self-organising maps. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3164–3171, 2016.

[8] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[9] Stanislav Bochkarev and Stephen L Smith. On minimizing turns in robot coverage path planning. In *IEEE International Conference on Automation Science and Engineering (CASE)*, pages 1237–1242, 2016.

[10] Howie Choset. Coverage for robotics–a survey of recent results. *Annals of Mathematics and Artificial Intelligence*, 31(1-4):113–126, 2001.

[11] Nicos Christofides and John E Beasley. The period routing problem. *Networks*, 14(2):237–256, 1984.

[12] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002.

[13] Jens Clausen. Branch and bound algorithms-principles and examples. *Department of Computer Science, University of Copenhagen*, pages 1–30, 1999.

[14] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical Review E*, 70(6):066111, 2004.

[15] Leandro C Coelho and Gilbert Laporte. The exact solution of several classes of inventory-routing problems. *Computers & Operations Research*, 40(2):558–565, 2013.

[16] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal: Complex Systems*, page 1695, 2006.

[17] Arun Das, Michael Diu, Neil Mathew, Christian Scharfenberger, James Servos, Andy Wong, John S Zelek, David A Clausi, and Steven L Waslander. Mapping, planning, and sample detection strategies for autonomous exploration. *Journal of Field Robotics*, 31(1):75–106, 2014.

[18] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[19] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[20] Markus Eich, Ronny Hartanto, Sebastian Kasperski, Sankaranarayanan Natarajan, and Johannes Wollenberg. Towards coordinated multirobot missions for lunar sample collection in an unknown environment. *Journal of Field Robotics*, 31(1):35–74, 2014.

[21] Kutluhan Erol, Dana S Nau, and Venkatramana S Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial intelligence*, 76(1):75–88, 1995.

[22] Abdulah Fajar, Nur Azman Abu, and Nanna Suryana Herman. Clustering strategy to Euclidean TSP Hamilton path role in tour construction. In *IEEE International Conference on Computer Modeling and Simulation*, volume 3, pages 508–512, 2010.

[23] Dave Ferguson and Anthony Stentz. Using interpolation to improve path planning: The field D* algorithm. *Journal of Field Robotics*, 23(2):79–101, 2006.

[24] Richard E Fikes and Nils J Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[25] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[26] Marc J Gallant, Alex Ellery, and Joshua A Marshall. Rover-based autonomous science by probabilistic identification and evaluation. *Journal of Intelligent & Robotic Systems*, 72(3-4):591–613, 2013.

[27] Luis Gouveia and Jose Manuel Pires. The asymmetric travelling salesman problem and a reformulation of the miller–tucker–zemlin constraints. *European Journal of Operational Research*, 112(1):134–146, 1999.

[28] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. CURE: an efficient clustering algorithm for large databases. In *ACM SIGMOD Record*, volume 27, pages 73–84, 1998.

[29] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. ROCK: A robust clustering algorithm for categorical attributes. In *International Conference on Data Engineering*, pages 512–521, 1999.

[30] Gregory Gutin and Daniel Karapetyan. A memetic algorithm for the generalized traveling salesman problem. *Natural Computing*, 9(1):47–60, 2010.

[31] Yll Haxhimusa, Walter G Kropatsch, Zygmunt Pizlo, and Adrian Ion. Approximative graph pyramid solution of the E-TSP. *Image and Vision Computing*, 27(7):887–896, 2009.

[32] Keld Helsgaun. An effective implementation of the Lin–Kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[33] Keld Helsgaun. Solving the equality generalized traveling salesman problem using the Lin–Kernighan–Helsgaun Algorithm. *Mathematical Programming Computation*, 7:1–19, 2014.

[34] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, pages 253–302, 2001.

[35] Geoffrey A Hollinger and Gaurav S Sukhatme. Sampling-based robotic information gathering algorithms. *The International Journal of Robotics Research*, 33(9):1271–1287, 2014.

[36] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley, 2004.

[37] Holger H Hoos and Thomas Stützle. SATLIB–the satisfiability library, 1998.

[38] William NN Hung, Xiaoyu Song, Jindong Tan, Xiaojuan Li, Jie Zhang, Rui Wang, and Peng Gao. Motion planning with satisfiability modulo theories. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 113–118, 2014.

[39] Frank Imeson and Stephen L Smith. A language for robot path planning in discrete environments: The TSP with Boolean satisfiability constraints. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 5772–5777, 2014.

[40] Frank Imeson and Stephen L Smith. Multi-robot task planning and sequencing using the SAT-TSP language. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 5397–5402, 2015.

[41] Paul Jackson and Davisniel Sheridan. Clause form conversions for Boolean circuits. In *Theory and Applications of Satisfiability Testing*, pages 183–198. Springer, 2005.

[42] Anil K Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651–666, 2010.

[43] Subbarao Kambhampati and Larry Davis. Multiresolution path planning for mobile robots. *IEEE Journal on Robotics and Automation*, 2(3):135–145, 1986.

[44] Nitin Kamra and Nora Ayanian. A mixed integer programming model for timed deliveries in multirobot systems. In *IEEE International Conference on Automation Science and Engineering (CASE)*, pages 612–617, 2015.

[45] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[46] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[47] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley, 2006.

[48] Marius Kloetzer and Calin Belta. Automatic deployment of distributed teams of robots from temporal logic motion specifications. *IEEE Transactions on Robotics*, 26(1):48–61, 2010.

[49] Bernhard Korte, Jens Vygen, B Korte, and J Vygen. *Combinatorial Optimization*, volume 2. Springer, 2012.

[50] James Kuffner, Koichi Nishiwaki, Satoshi Kagami, Masayuki Inaba, and Hirochika Inoue. Motion planning for humanoid robots. *Robotics Research*, pages 365–374, 2005.

[51] Ailsa H Land and Alison G Doig. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, pages 497–520, 1960.

[52] Steven Michael LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[53] Maxim Likhachev and Dave Ferguson. Planning long dynamically feasible maneuvers for autonomous vehicles. *The International Journal of Robotics Research*, 28(8):933–945, 2009.

[54] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21:498–516, 1973.

[55] M. Morris Mano and Michael D. Ciletti. *Digital Design*. Prentice-Hall, Inc., 4 edition, 2006.

[56] Neil Mathew, Stephen L Smith, and Steven Lake Waslander. Multirobot rendezvous planning for recharging in persistent tasks. *IEEE Transactions on Robotics*, 31(1):128–142, 2015.

[57] Clair E Miller, Albert W Tucker, and Richard A Zemlin. Integer programming formulation of traveling salesman problems. *Journal of the ACM (JACM)*, 7(4):326–329, 1960.

[58] Joseph SB Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems. *SIAM Journal on Computing*, 28(4):1298–1309, 1999.

[59] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[60] Srinivas Nedunuri, Sailesh Prabhu, Mark Moll, Swarat Chaudhuri, and Lydia E Kavraki. SMT-based synthesis of integrated task and motion plans from plan outlines. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 655–662, 2014.

[61] Mark EJ Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical Review E*, 74(3):036104, 2006.

[62] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2):026113, 2004.

[63] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[64] Karl J Obermeyer, Paul Oberlin, and Swaroop Darbha. Sampling-based roadmap methods for a visual reconnaissance UAV. In *AIAA Conference on Guidance, Navigation and Control*, 2010.

[65] Karl J Obermeyer, Paul Oberlin, and Swaroop Darbha. Sampling-based path planning for a visual reconnaissance unmanned air vehicle. *Journal of Guidance, Control, and Dynamics*, 35(2):619–631, 2012.

[66] Gurobi Optimization et al. Gurobi optimizer reference manual, 2012.

[67] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, 1998.

[68] Junghee Park, Sisir Karumanchi, and Karl Iagnemma. Homotopy-based divide-and-conquer strategy for optimal trajectory planning via mixed-integer programming. *IEEE Transactions on Robotics*, 31(5):1101–1115, 2015.

[69] Gábor Pataki. The bad and the good-and-ugly: formulations for the traveling salesman problem, 2000.

[70] Gábor Pataki. Teaching integer programming formulations using the traveling salesman problem. *SIAM review*, 45(1):116–123, 2003.

[71] Sandro Pirkwieser and Günther R Raidl. Multilevel variable neighborhood search for periodic routing problems. In *EvoCOP*, pages 226–238. Springer, 2010.

[72] Pascal Pons and Matthieu Latapy. Computing communities in large networks using random walks. In *ISCIS*, volume 3733, pages 284–293, 2005.

[73] David Portugal and Rui P Rocha. Cooperative multi-robot patrol with bayesian learning. *Autonomous Robots*, 40(5):929–953, 2016.

[74] Robert Clay Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[75] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, volume 3.2, page 5, 2009.

[76] Jörg Reichardt and Stefan Bornholdt. Statistical mechanics of community detection. *Physical Review E*, 74(1):016110, 2006.

[77] Gerhard Reinelt. TSPLIB–a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[78] Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39(1):127–177, 2010.

[79] Demane Rodney, Alan Soper, and Chris Walshaw. The application of multilevel refinement to the vehicle routing problem. In *IEEE Symposium on Computational Intelligence in Scheduling (SCIS)*, pages 212–219, 2007.

[80] Indranil Saha, Rattanachai Ramaithitima, Vijay Kumar, George J Pappas, and Sanjit A Seshia. Automated composition of motion primitives for multi-robot systems from safe LTL specifications. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1525–1532, 2014.

[81] Mitul Saha, Tim Roughgarden, Jean-Claude Latombe, and Gildardo Sánchez-Ante. Planning tours of robotic arms among partitioned goals. *The International Journal of Robotics Research*, 25(3):207–223, 2006.

[82] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *European Symposium on Algorithms*, pages 568–579. Springer, 2005.

[83] Satu Elisa Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, 2007.

[84] Peter Schüller, Volkan Patoglu, and Esra Erdem. A systematic analysis of levels of integration between low-level reasoning and task planning. In *Workshop on Combining Task and Motion Planning (IEEE International Conference on Robotics and Automation)*, 2013.

[85] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. Modeling and programming with Gecode, 2010.

[86] Yasser Shoukry, Pierluigi Nuzzo, Indranil Saha, Alberto L Sangiovanni-Vincentelli, Sanjit A Seshia, George J Pappas, and Paulo Tabuada. Scalable lazy SMT-based motion planning. In *55th Conference on Decision and Control (CDC)*, pages 6683–6688. IEEE, 2016.

[87] A Prasad Sistla and Edmund M Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[88] Stephen L Smith and Frank Imeson. GLNS: An effective large neighborhood search heuristic for the generalized traveling salesman problem. *Computers & Operations Research*, 2017.

[89] Stephen L Smith, Jana Tůmová, Calin Belta, and Daniela Rus. Optimal path planning for surveillance with temporal-logic constraints. *The International Journal of Robotics Research*, 30(14):1695–1708, 2011.

[90] Niklas Sorensson and Niklas Een. MiniSat v1.13 – a SAT solver with conflict-clause minimization. *SAT*, 2005:53, 2005.

[91] Ioan Alexandru Sucan, Mark Moll, and Lydia E Kavraki. The open motion planning library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, 2012.

[92] Yuichi Tazaki and Takumi Suzuki. Constraint-based prioritized trajectory planning for multibody systems. *IEEE Transactions on Robotics*, 30(5):1227–1234, 2014.

[93] Pratap Tokekar, Joshua Vander Hook, David Mulla, and Volkan Isler. Sensor planning for a symbiotic UAV and UGV system for precision agriculture. *IEEE Transactions on Robotics*, 32(6):1498–1511, 2016.

[94] Paolo Toth and Daniele Vigo. *Vehicle routing: problems, methods, and applications*. SIAM, 2014.

[95] Mauro Vallati, Lukáš Chrpa, Marek Grzes, Thomas L McCluskey, Mark Roberts, and Scott Sanner. The 2014 international planning competition: Progress and trends. *AI Magazine*, 36(3):90–98, 2015.

[96] Toby Walsh. SAT vs CSP. In *Principles and Practice of Constraint Programming*, pages 441–456. Springer, 2000.

[97] Chris Walshaw. A multilevel approach to the travelling salesman problem. *Operations Research*, 50(5):862–877, 2002.

[98] Richard Wang, Manuela Veloso, and Srinivasan Seshan. Active sensing data collection with autonomous mobile robots. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2583–2588, 2016.

[99] Christopher Xie, Jur van den Berg, Sachin Patil, and Pieter Abbeel. Toward asymptotically optimal motion planning for kinodynamic systems using a two-point boundary value problem solver. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4187–4194, 2015.

[100] Baozhen Yao, Ping Hu, Mingheng Zhang, and Shuang Wang. Artificial bee colony algorithm with scanning strategy for the periodic vehicle routing problem. *Simulation*, 89(6):762–770, 2013.

[101] Jingjin Yu, Sertac Karaman, and Daniela Rus. Persistent monitoring of events with stochastic arrivals at multiple stations. *IEEE Transactions on Robotics*, 31(3):521–535, 2015.

[102] Jingjin Yu and Steven M LaValle. Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics. *IEEE Transactions on Robotics*, 32(5):1163–1177, 2016.

[103] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: an efficient data clustering method for very large databases. In *ACM Sigmod Record*, volume 25, pages 103–114, 1996.

[104] DJ Zhu and J-C Latombe. New heuristic algorithms for efficient hierarchical path planning. *IEEE Transactions on Robotics and Automation*, 7(1):9–20, 1991.

[105] Zoran Zivkovic, Bram Bakker, and Ben Krose. Hierarchical map building and planning based on graph partitioning. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 803–809, 2006.

# Appendices

# Appendix A

# CBTSP Solver Parameters

The CBTSP solver uses a set of input parameters to configure the solver (configure the SAT solver, the TSP solver, and CBTSP's search). This section details the parameters as well as, their default settings.

**MINISAT** The configurable parameters of MINISAT are the conflict budget and the propagation budget. Both parameters by default are unlimited. All other MINISAT parameters are used as default and are non-configurable through the CBTSP interface.

**LKH** The LKH parameters configured by CBTSP are: `PROBLEM_FILE`, `TOUR_FILE`, `TIME_LIMIT`, `STOP_AT_MAX_COST`, and `MAX_COST`. The last two parameters are customizations we added to allow for LKH to solve decision problems. All other LKH parameters are configurable. The default settings used by CBTSP that differ from the LKH's default settings are given in Table A.1.

| Parameter | Value | Parameter | Value |
|---|---|---|---|
| PRECISION | 10 | PATCHING_A | 2 |
| MOVE_TYPE | 5 | PATCHING_C | 3 |
| RUNS | 1 | | |

Table A.1: Default LKH Parameters

| | Avg. Solver Cost | | | |
|---|---|---|---|---|
| `instance/cb_interval` | 1 | 2 | 10 | BRUTE |
| patrolling06 | **3176** | 3182 | 3631 | 3631 |
| patrolling10 | **3442** | 3908 | 4018 | 4120 |
| sample04 | **1883** | 2089 | 4207 | 5324 |
| sample08 | **12395** | 13380 | 16157 | - |
| period11 | **2804** | 2896 | 1387 | 4071 |
| period12 | **2669** | 2697 | 3247 | 4078 |

Table A.2: Tuning experiments for different values of the CBTSP parameter, `cb_interval` (CBTSP becomes BRUTE when `cb_interval` $> |V|$, we chose a value of 999). Each test was run four times and the average cost is reported. The instance name captures the problem type and the instance number matches up with Section 3.7. The best results are highlighted.

**CBTSP** The following parameters pertain to how CBTSP searches for solutions. Default values are given in parenthesis.

**The callback interval (`cb_interval=1`)** This parameter configures the TSP callback interval. A setting of $x$ indicates that the TSP theory consistency checked is performed when the following is satisfied: $|V| \mod x = 0$. A value of $x > |V|$, behaves, as a non-naïve BRUTE solver (described in Section 3.4.1). The default settings for the parameter `cb_interval` was chosen after comparing different values of `cb_interval` on a set of small experiments documented in Table A.2. Here we take two instances from each problem application (patrolling, sample collection, and period routing), six in total and compare the quality of the solver's results with the same setup as in Section 3.7. As we can see from the Table, a value of `cb_interval=1` has the best performance. One thing to note from the results, is how the non-naïve BRUTE approach (`cb_interval=999`) fails to performs on instance sample08. We believe this is due to the battery constraint of the sample collection problem. Here the solver finds a solution and then negates it if it exceeds the battery budget. This is unlike the other two problems, which find feasible solutions and then negate them from reoccurring. Thus, without the guide of the TSP theory, the solver seems to struggle finding feasible solutions for the sample collection problem.

**Conflicts (`nConflicts=-1`)** The CBTSP solver adds conflict clauses back to the SAT formula to narrow down the search space. The number of conflict clauses CBTSP adds

back to the formula can be exponential in size, thus the parameter `nConflicts=`$x$ limits the number of additional clauses to a maximum of $x$ (-1 for unlimited). Once the maximum is reached the best known solution is returned. One can monitor the output text of the solver to confirm whether or not the time budget has been reached.

**Query budget (`max_query_time=-1`)** The CBTSP solver searches for solutions with a series of SAT-TSP decision queries (different cost budgets set by the binary or linear search algorithm). By default (-1) the queries are given an unlimited amount of time but if this parameter has the non-zero value $x$, then each query is terminated after $x$ seconds and is assumed to be unsatisfiable.

**Search method (`search_method=binary`)** This parameter is used to choose between binary and linear search.

**The binary search parameter (`bdiv=10`)** This parameter configures the division size of the binary search as shown in Algorithm 4. Most unsatisfiable instances are harder to prove than satisfiable instances. Thus it is desirable to have this parameter larger than two. The default settings for the binary search parameter `bdiv` was chosen after comparing different values of `bdiv` (and comparing it to a linear search) on a set of small experiments documented in Table A.3. Here we take two instances from each problem application (patrolling, sample collection, and period routing), six in total and compare the quality of the solver's results using the same solver setup as in Section 3.7. We can see from the Table a value of `bdiv=10` yields a good result.

| | Avg. Solver Cost | | | |
|---|---|---|---|---|
| `name/bdiv` | 2 | 10 | 20 | Linear |
| patrolling06 | 3177 | **3176** | 3177 | 3177 |
| patrolling10 | 3653 | **3442** | 3496 | 3972 |
| sample04 | 1899 | 1883 | **1882** | 2291 |
| sample08 | 12993 | 12395 | **12051** | 14731 |
| period11 | 2925 | **2804** | 2937 | 3578 |
| period12 | 2973 | **2669** | 2708 | 3148 |

Table A.3: Tuning experiments for different values of the CBTSP parameter `bdiv` (the search is linear is when `bdiv`=999999). Each test was run four times and the average cost is reported. The instance name captures the problem type and the instance number matches up with Section 3.7.

# Appendix B

# Additional SAT-TSP Approaches

In this appendix, we provide five additional SAT-TSP solvers, all of which reduce SAT-TSP to other problems. These approaches were designed to solve SAT-TSP problems with one input graph. Multiple robot problems that are solved in this appendix are done so using one large graph to represent the robot's environments. We benchmark these approaches against each other as well as, the CBTSP solver.

The rest of this appendix is organized as follows: Section B.1, provides some needed background for the constraint satisfaction problem; Section B.2 introduces a set of algorithms needed by the solvers to search for optimal solutions; Section B.3 provides a reduction from the Hamiltonian cycle problem to SAT that is used by two of the solvers; Section B.4 provides the five additional SAT-TSP solvers based on reductions to: SAT, TSP, GTSP, CSP, and SMT; Section B.5 details the benchmark; and Section B.6 reviews the results of the benchmark.

## B.1 The constraint satisfaction problem (CSP)

The constraint satisfaction problem (CSP) allows for the expression of non-Boolean constraints. It can be used to express high-level path planning problems and there are a number of good solvers such as Gecode [85]. In [92], the authors use CSP for multi-robot path planning. It is our experience that CSP solvers can be useful for finding feasible solutions.

The CSP problem takes as input a set of variables, a domain of values, and a set of constraints. Each constraint must be satisfied by finding an assignment of the variables

it constrains within the allowable domain of values. Some examples of CSP constraints are LESSTHAN$(x_1, x_2)$, EQUAL$(x_1, x_2)$ and ALLDIFFERENT$(x_1, x_2)$. The language is as follows:

**CSP** $= \{\langle X, D, C\rangle : X$ is a set of variables, $D$ is a domain of values, and $C$ is a set of constraints, and there exists a mapping $v : X \to D$ such that all constraints in $C$ are satisfied$\}$.

## B.2 Search Algorithms

The reduction techniques presented in this appendix consist of reducing one SAT-TSP instance to multiple (non-SAT-TSP) instances. The problem instances for these approaches are either a decision problem that tests if a solution exists within a cost budget or an optimization problem that encodes a temporary objective. Algorithm 12, 14, and 16 are used to find optimal solutions and Algorithm 13, 15, and 17 are used to solve decision problems.

These algorithms call functions SOLVE, REDUCE, and TRANSLATE pertain to algorithms that are specific to each solver approach. For example, if we are solving a SAT-TSP instance with a SAT solver, then SOLVE is the SAT solver, REDUCE reduces the SAT-TSP instance to a SAT formula, and TRANSLATE translates the SAT solution back to a SAT-TSP solution. The REDUCE and TRANSLATE algorithms must run in polynomial time to qualify as reductions.

Algorithms 12, 13, 14 and 15 run in $O(|V|)$ time and Algorithm 16 runs in $O(\log c_{max})$ time, where $c_{max} =$ MAXCOST$(G)$ is used to calculate an upper bound in polynomial time. The details of the REDUCE algorithms are given in Section B.4 for each reduction approach.

In the following algorithms we use the symbol $\Phi$ for the solution and $\Sigma$ for the reduction. Additionally, Algorithms 12 and 13 use $v_s$ to indicate the start vertex (a vertex that is included in the solution) and Algorithms 14 and 15 use $l$ to indicate the length of the solution.

**Algorithm 12:** OPTIMIZE$(G, F)$ for TSP and GTSP Approaches

```
1 Φ* ← ∅
2 for v_s ∈ V do
3 |   Σ ← REDUCE(G, F, v_s)
4 |   Φ ← SOLVE(Σ)
5 |   if Φ is better than Φ* then
6 |   |   Φ* ← Φ
7 return TRANSLATESOLN(Φ*)
```

**Algorithm 13:** DECIDE$(G, F, C)$ for TSP and GTSP Approaches

```
1 for v_s ∈ V do
2 |   Σ ← REDUCE(G, F, v_s)
3 |   Φ ← SOLVE(Σ, C)
4 |   if Φ ≠ ∅ then
5 |   |   return TRANSLATESOLN(Φ)
6 return ∅
```

**Algorithm 14:** OPTIMIZE$(G, F)$ for CSP Approach

```
1 Φ* ← ∅
2 for l ∈ {1, 2, ..., |V|} do
3 |   Σ ← REDUCE(G, F, l)
4 |   Φ ← SOLVE(Σ)
5 |   if Φ is better than Φ* then
6 |   |   Φ* ← Φ
7 return TRANSLATESOLN(Φ*)
```

**Algorithm 15:** DECIDE$(G, F, C)$ for CSP Approach

```
for l ∈ {1, 2, ..., |V|} do
    Σ ← REDUCE(G, F, l)
    Φ ← SOLVE(Σ, C)
    if Φ ≠ ∅ then
        return TRANSLATESOLN(Φ)
return ∅
```

**Algorithm 16:** OPTIMIZE$(G, F)$ for SAT and SMT Approaches

```
/* Binary Search                                                    */
Φ* ← ∅
⟨c_min, c_max⟩ ← ⟨0, MAXCOST(G)⟩
while c_max − c_min > 0 do
    c_t ← ⌊c_min + (c_max − c_min)/2⌋
    Σ ← REDUCE(G, F, c_t)
    Φ ← SOLVE(Σ)
    if Φ ≠ ∅ then
        Φ* ← Φ
        ⟨c_min, c_max⟩ ← ⟨c_min, c_t − 1⟩
    else
        ⟨c_min, c_max⟩ ← ⟨c_t + 1, c_max⟩
return TRANSLATESOLN(Φ*)
```

**Algorithm 17:** DECIDE$(G, F, C)$ for SAT and SMT Approaches

```
Σ ← REDUCE(G, F)
Φ ← SOLVE(Σ, C)
return TRANSLATESOLN(Φ)
```

## B.3 Reduction of the Hamiltonian Cycle Problem to SAT

In this section we reduce the Hamiltonian cycle problem (HCP) aspect of SAT-TSP to SAT (used in Sections B.4.1 and B.4.5 for the SAT and SMT reductions). Specifically, we translate the SAT-TSP instance $\langle G, F, c = \infty \rangle$ to SAT. There are existing translations from HCP to SAT, however, this translation requires a Hamiltonian cycle of the included vertices $V'$ (not necessarily the entire graph).

Our translation is inspired by the IP formulation of the Hamiltonian Cycle Problem (HCP) [47]. The first part of the translation creates a formula $\hat{F}$ that encodes the Hamiltonian cycle aspect of the SAT-TSP problem. We start with a set of constraints that capture the Hamiltonian cycle problem for the included vertices $V'$:

L1.1 If $\langle v_i, v_j \rangle \in E$ is in the solution, then $v_i$ and $v_j$ are in the solution.

L1.2 For each $v_i \in V$ there is at most one outgoing edge in the solution.

L1.3 For $v_j$ there is at most one incoming edge in the solution.

L1.4 If $v_i$ and $v_j$ are in the solution, then $v_j$ is reachable from $v_i$. To accomplish this we use the additional set of indicator variables $\{\hat{x}_{v_1,0}, \hat{x}_{v_2,0}, \ldots, \hat{x}_{v_{|V|},0}, \hat{x}_{v_1,1}, \ldots, \hat{x}_{v_{|V|},|V|}\}$.

- (a) Exactly one vertex in the set $\{\hat{x}_{v_1,0}, \hat{x}_{v_2,0}, \ldots, \hat{x}_{v_{|V|},0}\}$ is true.
- (b) Vertex $v_i$ is in the solution ($\hat{x}_{v_i} = 1$) if and only if at least one variable in the set $\{\hat{x}_{v_i,1}, \hat{x}_{v_i,2}, \ldots, \hat{x}_{v_i,|V|}\}$ is true.
- (c) Variable $\hat{x}_{v_j,k+1} = 1$ if and only if there exists $\hat{x}_{v_i,k} = 1$ and $\langle v_i, v_j \rangle$ is in the solution.

Now we encode these constraints into the SAT formula $\hat{F}$ using the set of vertices and edges as Boolean variables. As well, we use the set of variables introduced in constraint L2.4c. The following translates the above constraints into the SAT $\hat{F}$.

L2.1 This is encoded with an implies constraint:

$$\bigwedge_{\langle v_i, v_j \rangle \in E} \langle v_i, v_j \rangle \implies (v_i \wedge v_j).$$

L2.2 This is encoded with an at most one in a set constraint:

$$\bigwedge_{v_i \in V} \left( \bigwedge_{\langle v_i, v_x \rangle, \langle v_i, v_y \rangle \in E | x \neq y} \neg(\langle v_i, v_x \rangle \wedge \langle v_i, v_y \rangle) \right).$$

L2.3 This is also encoded with an at most one in a set constraint:

$$\bigwedge_{v_j \in V} \left( \bigwedge_{\langle v_x, v_j \rangle, \langle v_y, v_j \rangle \in E | x \neq y} \neg(\langle v_x, v_j \rangle \wedge \langle v_y, v_j \rangle) \right).$$

L2.4 (a) This is encoded with an exactly one in a set constraint:

$$\bigvee_{v_i \in V} \left( \hat{x}_{v_i,0} \left( \bigwedge_{v_j \in V \setminus v_i} \neg \hat{x}_{v_j,0} \right) \right).$$

(b) This is encoded with an if and only if constraint, nested with an at least one in a set constraint:

$$\bigwedge_{v_i \in V} \left( \hat{x}_{v_i} \iff \left( \bigvee_{j=0}^{|V|} \hat{x}_{v_i,j} \right) \right).$$

(c) This is also encoded with an if and only if constraint nested with an at least one in a set constraint:

$$\bigwedge_{v_j \in V, k=1}^{|V|} \left( \hat{x}_{v_j,k} \iff \left( \bigvee_{v_i \in V \setminus v_j | e_a = \langle v_i, v_j \rangle \in E} (\hat{x}_{v_i,k-1} \wedge \hat{x}_{e_a}) \right) \right).$$

Now the above formula $\hat{F}$ can be combined with $F \leftarrow F \wedge \hat{F}$ to finish the translation of $\langle G, F, c = \infty \rangle$ to SAT.

**Lemma B.3.1** (Reduction Results)**.** For the translation from the HCP to SAT the following holds:

(i) The formula $\hat{F}$ has $O(|L|+|V|^3)$ literals and is constructed in polynomial time, where $|L|$ is the number of literals in $F$.

(ii) A HCP tour of the vertices $V'$ translates to a solution for $\hat{F}$.

(iii) A satisfying assignment of $\hat{F}$ translates to a HCP solution over the vertices $V'$.

*Proof.* We will establish each of the three results in turn.

Proof of (i): The original formula $F$ contributes $|L|$ literals to $\hat{F}$, clauses L2.1, L2.4a, L2.4b, and L2.4c contribute $O(|V|^2)$ literals to $\hat{F}$ and clauses L2.2 and L2.3 contribute $O(|V|^3)$ literals to $\hat{F}$. Therefore, the formula $\hat{F}$ has $O(|L| + |V|^3)$ literals and since the formula $\hat{F}$ is directly constructed, it can be done in polynomial time.

Proof of (ii): Given a SAT-TSP solution $\langle M, p' \rangle$, where $p'$ is a tour of the included vertices $V'$ and $M$ is a satisfying assignment for $F$. We construct a solution to $\hat{F}$ by assigning the variables $\hat{x}_{v_i,k}$ true for each $v_i \in V'$. We also assign $\hat{x}_{e_a}$ for each edge $e_a \in p'$. The remaining variables either take their corresponding assignment from $M$ or they receive a false assignment if they have no corresponding variable ($\hat{x}_{v_i,k}$ and $\hat{x}_{e_a}$). This construction satisfies each of the constraints L1.1-L1.4c, since it is based off a tour. Since the original formula $F$ is also satisfied (the variables common to both $F$ and $\hat{F}$ are given the same assignment), $\hat{F}$ is satisfied ($\hat{F}$ is a combination of $F$ and the constraints L1.1-L1.4c).

Proof of (iii): Given a satisfying assignment $\hat{M}$ for $\hat{F}$, we construct the SAT-TSP solution $\langle M, p' \rangle$ as follows: for each variable assignment of $X$ in $\hat{M}$, we duplicate the assignment in $M$ (i.e., $M = \{x_i \in \hat{M} | x_i \in X\} \cup \{\neg x_i \in \hat{M} | x_i \in X\}$). We construct the graph cycle $p' = v_i, v_j, \ldots, v_k$ on the included vertices by including $v_i$ in $p'$ as the $k^{th}$ vertex in $p'$ if and only if $x_{v_i,k} = 1$ for $k \in [1, |V|]$ (there will be at most one true assignment in the set $\{x_{v_i,k} | v_i \in V\}$). To prove this is a solution, we need to prove that $M$ satisfies $F$ and that $p'$ is indeed a tour of the included vertices.

First, $M$ is clearly a satisfying assignment for $F$ since $\hat{F}$ includes $F$ in its formulation ($\hat{F} = F \wedge \ldots$). Second we show that the constraints L1.1-L1.4c do in fact encode the Hamiltonian tour constraints of the induced subgraph (i.e., every included vertex is visited exactly once and every edge $\langle v_i, v_j \rangle \in p'$ exists in $E$. To prove this result, we first provide insight of the construction and then we provide the proof.

As we can see from the translation of the $\hat{F}$ solution, variables $x_{v_i,k}$, pertain to the $k^{th}$ sequence that vertex $v_i$ is visited in the tour ($k = 0$ represents the start/finish vertex), for which constraint L1.4a states that there must be exactly one choice for this start. Constraint L1.4c pertains to which vertex can be visited next in the tour (only if the appropriate edge is included in the solution). Due to constraint L1.2, only one such vertex can be visited next in the sequence. Since each included vertex must be reachable even from itself (constraint L1.4b), it follows that each vertex must have an incoming edge, due to constraint L1.4c. The trick is that the starting vertex is in the tour since it has

an outgoing edge (constraint L1.1). However, it is not yet reachable, thus it needs to be visited at some level $k$ which is possible since it does not yet have an incoming edge associated with it. Since it appears in level $k = 0$ then it has an outgoing edge, thus it must be visited last, otherwise constraint L1.2 would be violated. Therefore, $p'$ is a tour that visits all included vertices without repetition and thus a satisfying assignment for $\hat{F}$ that translates to a solution for the SAT-HCP instance $\langle G, F \rangle$. □

**Corollary B.3.2.** The translation presented in this section is a reduction from $\langle G, F, c = \infty \rangle$ to SAT. Specifically, the translation is done in polynomial time and the translated problem has a solution if and only if the original problem has a solution.

## B.4 Solver Approaches

This section presents solver approaches for SAT-TSP that are based on reducing to SAT-TSP to: SAT, TSP, GTSP, CSP, and SMT.

### B.4.1 Reduction to SAT

The reduction of SAT-TSP to SAT consist of translating the SAT-TSP instance $\langle G, F, C \rangle$ into a SAT instance $\langle \hat{F} \rangle$, where the cost budget $C$. Algorithm 16 is used with this reduction to find optimal solutions and Algorithm 17 is used to solve the decision version. Now we present the details of REDUCE (we omit the details of TRANSLATEPROB).

The first part of the reduction reduces the Hamiltonian cycle problem to SAT (see Section B.3 for details). Then the cost budget is encoded into the SAT formula using adder circuits (see Section 3.2.2). The circuit contains a level for each significant digit (ones, twos, fours, ...). The adder circuit adds a binary representation of each edge weight if and only if the edge is included in the solution. For example, suppose the edge $e_1$ has an edge weight of 5 (101 in binary — a one bit, a four bit but no two bit), then the adder circuit responsible for adding up the first and third significant digits would have $e_1$ in their input. An example of an adder circuit for all the bits for the first significant level (ones) is shown in Figure B.1. The complete binary circuit can be constructed using techniques in [55].

The maximum solution cost of the SAT-TSP instance is bounded by $|V|w_{\max}$, where $w_{\max} = \max_i w(e_i)$. Thus the maximum solution requires $\log |V|w_{\max}$ bits to encode. Each output bit in the solution cost is generated by a circuit that adds the input edge bits and thus there are at most $|E|$ bits in each level of the circuit. A circuit adding up $|E|$

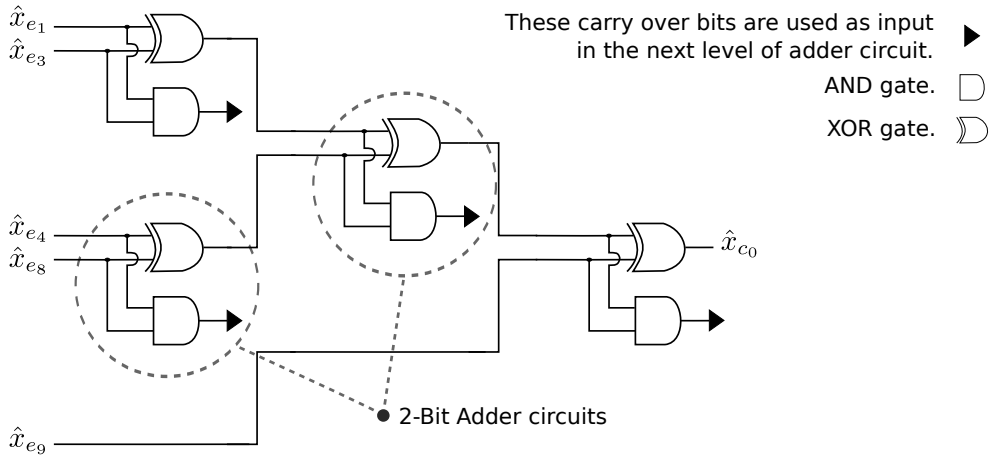



Figure B.1: An adder circuit summing up $x_{c_0}$, the one-bits of the solution cost. In this instance the edges $\{e_1, e_3, e_4, e_8, e_9\}$ have odd edge weights and all other edges have even weights.

bits would require at most $|E|$ 2-bit adder circuits (an XOR and an AND gate shown in Figure B.1). Therefore, the complete adder circuit is of size $O(|E| \log |V| w_{\max})$ and using methods from [41] this circuit is efficiently translated to a Boolean formula.

The cost budget constraints are constructed by forcing an assignment on a subset of the solution cost bits (variables) $\{\hat{x}_{\hat{c}_0}, \hat{x}_{\hat{c}_1}, \ldots, \hat{x}_{\hat{c}_n}\}$, where the total solution cost is $c = \sum_{i=0}^{n} 2^i \hat{x}_{\hat{c}_i}$. For example, if we wish to find a solution of cost seven or less, then all variables representing powers of three and higher are forced to be false. All cost constraints that use this approach are of size $O(\log(|V| w_{\max}))$.

### B.4.2 Reduction to TSP

The reduction of SAT-TSP to TSP consist of translating the SAT-TSP instance $\langle G, F, C \rangle$ to a series of TSP instances $G_{v_s}$, where the vertex $v_s$ is assumed to be in the solution. Algorithm 12 is used to find optimal solutions and Algorithm 13 is used to solve decision problems. Now we present the details of REDUCE (we omit the details of TRANSLATEPROB).

This reduction is inspired by the translation of SAT to the Hamiltonian Cycle Problem [47]. The first step in expressing $F$ as a graph $\hat{G}$ with the assumption that $v_s$ is in the solution, is to create a chain of vertices for each variable $x_i \in X$, for which we refer to as widget $\hat{\Omega}_{x_i}$. The direction that the TSP solution traverses the widget (chain) indicates the assignment of the SAT variable — traverse the widget $\hat{\Omega}_{x_i}$ from left to right then the

variable $x_i = 1$ (true), traverse the widget from right to left then $x_i = 0$ (false). Once the solution tour starts traversing the widget it cannot change directions and it must finish visiting the widget before moving onto the next widget. Figure B.2 shows an example of a widget and its connections.

The TSP graph $\hat{G}_{v_s}$ construction also includes a vertex $\hat{v}_{c_i}$ for each clause $c_i \in C$ in the formula $F$. The widget construction allows us to connect the clause vertices so the tour can only visit the clause vertex only if the clause would be satisfied by the variable assignment for the widget. For example, suppose clause $c_1 = (x_1 \vee x_2 \vee \neg x_3)$, then the clause vertex $\hat{v}_{c_1}$ is only connected to widgets $\hat{\Omega}_{x_1}, \hat{\Omega}_{x_2}$ and $\hat{\Omega}_{x_3}$ and it can only be visited if the tour traverses $\hat{\Omega}_{x_1}$ or $\hat{\Omega}_{x_2}$ from left to right, or if the tour traverses $\hat{\Omega}_{x_3}$ from right to left.

The the widgets are connected to each other to ensure feasible tours visit all the included vertices first (any widget $\hat{\Omega}_{x_{v_i}}$ that is traversed left to right that corresponds with $x_{v_i}$ for $v_i \in V$), then the non-included vertices, and then the auxiliary variable widgets. In this way, as the solution visits widgets that represent included vertices in $\hat{G}_{v_s}$, it does so with the same costs as if it were visiting vertices in $G$. This is accomplished by restricting the set of incoming and outgoing edges to the widgets. Figure B.3 shows how the widgets are connected to each other. With this construction a TSP solution of cost $c$ translates to a SAT-TSP solution of cost $c$.

The details of this approach and the proof of its correctness are given in our preliminary work [39]. For the sake of conciseness and since this approach is not among the most successful, we refer the reader to our earlier work for more details [39].

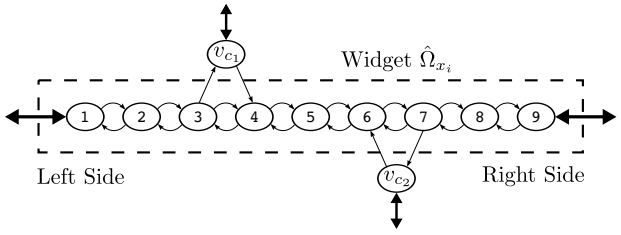



Figure B.2: An example of the widget $\hat{\Omega}_{x_i}$. In this instance the only clauses in $F$ that contain the variable $x_i$ are clauses $c_1$ and $c_2$. The clause $c_1$ contains the literal $x_1$ and $c_2$ contains the literal $\neg x_i$. A TSP solution that traverses the widget from left to right $(1 \to 9)$ indicates that $x_i = 1$ in the SAT-TSP solution and a solution that traverses the widget from right to left $(9 \to 1)$ indicates that $x_i = 0$.

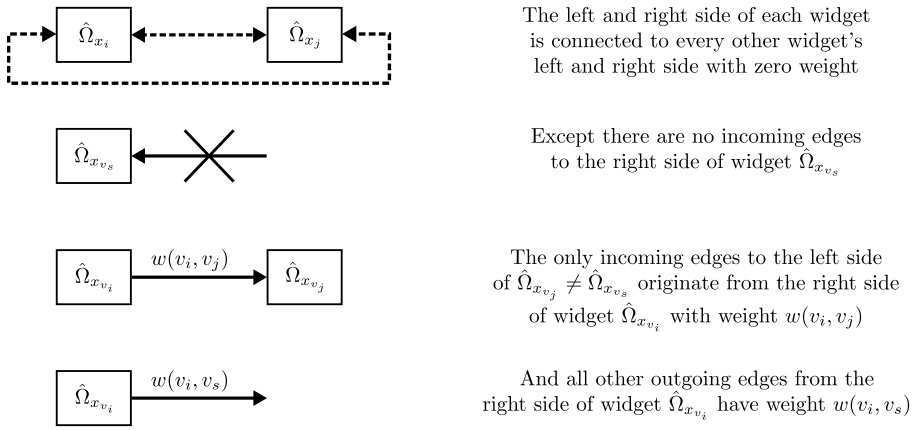



Figure B.3: The connections between widgets in the TSP graph. Dotted edges have zero weight. An edge going into or out of the left of the widget indicates a connection to left most vertex in the widget chain. Likewise, an edge going into or out of the right side indicates a connection to the right most vertex in the chain.

### B.4.3 Reduction to GTSP

The reduction of SAT-TSP to GTSP consist of translating the SAT-TSP instance $\langle G, F, C \rangle$ to a series of GTSP instances that assume a specific vertex $v_s$, is included in the solution. Algorithm 12 is used to find optimal solutions and Algorithm 13 is used to solve decision problems. Now we present the details of REDUCE (we omit the details of TRANSLATEPROB).

The reduction consist of creating a graph that contains vertices to represent the literals in $F$, sets to represent assignment of variables (true or false), and sets to represent clause satisfaction (at least one literal in each clause must be true). In this construction visiting a vertex literal translates to an assignment of the variable (true or false) and visiting a set that is associated with a clause, translates to a satisfying the clause.

The sets in the GTSP instance are used to restrict the assignments of the variable $x_i \in F$ — each variable can only be assigned true or false — not both. This is accomplished by creating a vertex $\hat{v}_{x_i}$ to represent the true assignment of $x_i$ and a vertex $\hat{v}_{\neg x_i}$ to represent the false assignment of $x_i$. We classify these vertices as root vertices. Creating the set $\{\hat{v}_{x_i}, \hat{v}_{\neg x_i}\}$ for each variable in the formula ensures that the GTSP solution can only visit either the true vertex $\hat{v}_{x_i}$ or the false vertex $\hat{v}_{\neg x_i}$. The sets are also used to ensure that each clause $c_i \in F$ is satisfied. This is accomplished by creating a set for each clause. However, a clause $c_i$ may be satisfied by multiple literals where a GTSP solution can only

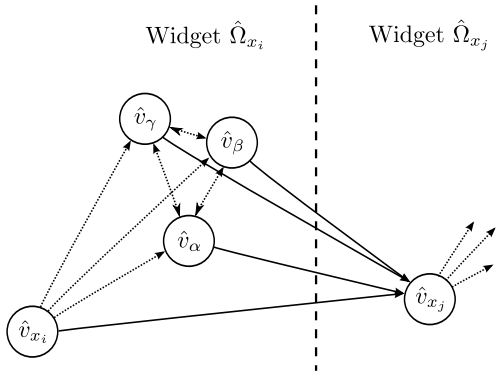



Figure B.4: An illustration of the connections between vertices in a widget and between widgets. In this example the literal $x_i$ appears in clauses $c_1, c_2$ and $c_3$, the vertices $\hat{v}_\alpha, \hat{v}_\beta$ and $\hat{v}_\gamma$ are short forms for vertices $\hat{v}_{x_i,c_1}, \hat{v}_{x_i,c_2}$ and $\hat{v}_{x_i,c_3}$ respectively. The vertices connected with dotted edges have zero edge weight and the solid edges all have the same weight.

visit one element in each set. Therefore, instead of using the root vertices $\hat{v}_{l_i}$ in the clause sets, we create a vertex $\hat{v}_{l_i,c_j}$ for each literal $l_i \in c_j$ in the clause. Then these vertices are used for the sets. For example, suppose $c_1 = x_1 \vee \neg x_2 \vee x_3$, then the clause set for $c_1$ is $\{\hat{v}_{x_1,c_1}, \hat{v}_{\neg x_2,c_1}, \hat{v}_{x_3,c_1}\}$. With this construction, a GTSP solution ensures that at least one literal in each clause will be visited.

In the GTSP graph we connect the vertex literals $\hat{v}_{l_i}$ to the clause literals $\hat{v}_{l_i,c_j}$ in such a way that the clause literals can only be visited if and only if $\hat{v}_{l_i}$ is visited. This is accomplished by constructing a widget $\hat{\Omega}_{l_i}$ for each set of vertices that pertain to a literal $l_i$. See Figure B.4 for an example. The following list enumerates the edge connections for the widgets.

1. $\langle \hat{v}_{l_i}, \hat{v}_{l_i,a} \rangle \in \hat{E}$ with weight 0 for $\hat{v}_{l_i}, \hat{v}_{l_i,a} \in \hat{V}$.
2. $\langle \hat{v}_{l_i,a}, \hat{v}_{l_i,b} \rangle \in \hat{E}$ with weight 0 for $\hat{v}_{l_i,a} \in \hat{V},\ \hat{v_{l_i,b}} \in \hat{V} \setminus \hat{v}_{l_i,a}$.
3. $\langle \hat{v}_{l_i,a}, \hat{v}_{l_j} \rangle \in \hat{E}$ with weight $\hat{w}(\hat{v}_{l_i}, \hat{v}_{l_j})$ for $\hat{v}_{l_i,a} \in \hat{V},\ \hat{v}_{l_j} \in \hat{V} \setminus \hat{v}_{l_i}$ if and only if $\langle \hat{v}_{l_i}, \hat{v}_{l_j} \rangle \in \hat{E}$, where $\hat{w}(\hat{v}_{l_i}, \hat{v}_{l_j})$ is to be defined.

The widgets are connected to each other so that the GTSP solution transition costs of included vertices have the same costs as the SAT-TSP instance. This is accomplished by constructing the graph to ensure that all included vertices are visited first. Let us create two sets of vertices $\hat{V}_\alpha = \{\hat{v}_{x_{v_i}} \in \hat{V} | v_i \in V\}$ and $\hat{V}_\beta = \{\hat{v}_{x_i}, \hat{v}_{\neg x_i} \in \hat{V} | \hat{v}_{x_i} \notin \hat{V}_\alpha\}$, to help

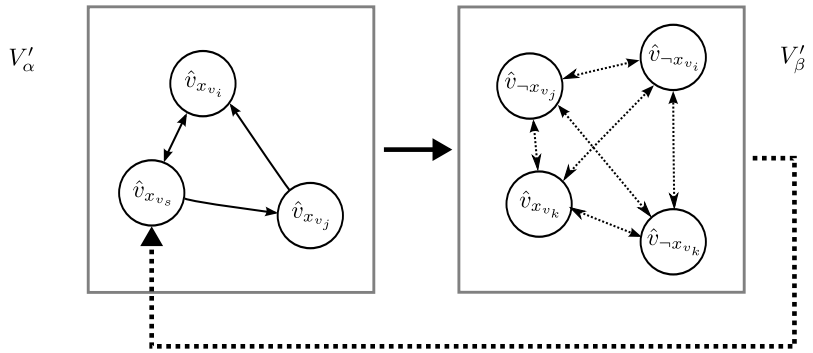



Figure B.5: The connections and edge weights between root vertices. If a connection is not shown, then it does not exist in the GTSP graph. The large arrow from $V'_\alpha$ to $V'_\beta$ indicates the connections between the two sets (unidirectional and weighted). The large dotted arrow from $V'_\beta$ to $\hat{v}_{x_{v_s}}$ indicates the connections between the two sets (unidirectional with zero edge weights).

us describe how the root vertices are connected. The vertices in $\hat{V}_\alpha$ are connected as $V$ is in $G$ and the vertices $\hat{V}_\beta$ are fully connected to each other with zero weight edges. Each vertex $\hat{v}_{x_{v_i}} \in \hat{V}_\alpha$ is connected to each vertex $\hat{v}_{x_j} \in \hat{V}_\beta$ with zero edge weight and each vertex $\hat{v}_{x_j} \in \hat{V}_\beta$ is connected to $\hat{v}_{x_{v_s}} \in \hat{V}_\alpha$ to close the cycle. See Figure B.5 for an example. The following list enumerates the remaining edges in the graph.

1. $\langle \hat{v}_{x_{v_i}}, \hat{v}_{x_{v_j}} \rangle \in \hat{E}$ with weight $w(v_i, v_j)$ for $\hat{v}_{x_{v_i}}, \hat{v}_{x_{v_j}} \in \hat{V}_\alpha$ if and only if $\langle v_i, v_j \rangle \in E$
2. $\langle \hat{v}_{l_i}, \hat{v}_{l_j} \rangle \in \hat{E}$ with weight 0 for $\hat{v}_{l_i} \in \hat{V}_\beta,\ \hat{v}_{l_j} \in \hat{V}_\beta \setminus \hat{v}_{l_i}$
3. $\langle \hat{v}_{x_{v_i}}, \hat{v}_{l_j} \rangle \in \hat{E}$ with weight $w(v_i, v_s)$ for $\hat{v}_{x_{v_i}} \in \hat{V}_\alpha,\ \hat{v}_{l_j} \in \hat{V}_\beta$
4. $\langle \hat{v}_{l_i}, \hat{v}_{x_{v_s}} \rangle \in \hat{E}$ with weight zero for $\hat{v}_{l_i} \in \hat{V}_\beta,\ \hat{v}_{x_{v_s}} \in \hat{V}_\alpha$

The construction ensures that a vertex literal $\hat{v}_{l_i,a}$ can only be visited if the vertex $\hat{v}_{l_i}$ is first visited and the costs are equivalent to the SAT-TSP instance. The proof of this approach's correctness is found in [40].

*Remark* B.4.1 (Special case of sets of cardinality one). In the special case that the set of literal vertices $\{\hat{v}_{l_i,a} \in \hat{V}\}$ for some $l_i$ has cardinally one, then the root vertex can be replaced with the vertex literal. Proper bookkeeping will be needed to reflect this change. □

### B.4.4 Reduction to CSP

The reduction of SAT-TSP to CSP consist of translating the SAT-TSP instance $\langle G, F, C\rangle$ to a series of CSP instances that assume that the set of included vertices $V'$ is of a specific size. Algorithm 14 is used to find optimal solutions and Algorithm 15 is used to solve decision problems. Now we present the details of REDUCE (we omit the details of TRANSLATEPROB).

The reduction consist of expressing the graph as a weight matrix in the CSP format, translating the formula $F$ into a CSP formula [96], and reducing a fixed length Hamiltonian cycle problem to CSP. The expression of the fixed length Hamiltonian cycle problem is as follows: first create a fixed length vector to represent the *tour*, next each variable in the tour vector is constrained to take on values $\{i|v_i \in V[G]\}$, and finally each element in the *tour* vector is constrained to be *all different* to ensure that no duplicate entries show up in the cycle. Additionally, there is a set of simple constraints used to link the tour vector with the weight matrix such that the solution cost can be calculated and constrained (vertex $v_i$ is included in the tour if and only if variable $x_i = 1$).

### B.4.5 Reduction to SMT

The reduction of SAT-TSP to SMT consist of translating the SAT-TSP instance $\langle G, F, C\rangle$ into an SMT instance with a cost budget $C$. Algorithm 16 is used with this reduction to find optimal solutions and Algorithm 17 is used to solve the decision version. Now we present the details of REDUCE (we omit the details of TRANSLATEPROB).

Like the SAT reduction, this reduction reduces the HCP to SAT (Section B.3). The cost is encoded and constrained using an SMT arithmetic theory. This saves $O(\log c_{\max})$ work compared to the SAT approach. The formulas used in calculating and constraining the costs are as follows: an edge $e_i$ contributes $w(e_i)$ to the solution cost $c'$ if and only if $\hat{x}_{\hat{e}_i}$ is assigned true in the formula. The binary search algorithm chooses a budget cost $c$ and the constraint $c' \leq c$ is encoded into the SMT instance. If the SMT solver finds a solution then the edge variables are translated into a SAT-TSP tour and the remaining variable assignments are used to construct a satisfying assignment for $F$.

## B.5 Benchmark Problems

In this section we detail the construction of the problem instances in the benchmark (these problems are different than the problems found in the main body of thesis).

The problems in libraries SetLib, CountLib, OrderedLib, and MultiRobotLib use the following environments.

**The Metric Library Environment** The robot is able to move within the two dimensional space $x \in [0, 1]$ and $y \in [0, 1]$ and a set of locations are uniformly randomly placed within the square. There is one location chosen at random to represent the robot's home. The rest of the locations contain items for the robot to retrieve. Each location has one item with a random shape and colour. The items can take one of eight colours and one of thee shapes. The graph is constructed to capture the euclidean distance between the locations.

**The Non-Metric Library Environment** The robot's environment graph is constructed with uniform random edge weights in the domain $[0, 1]$. One vertex is randomly chosen for the robot's home. The rest are item locations. Each location has one item with a random shape and colour. The items can take one of eight colours and one of thee shapes.

*Remark* B.5.1 (Vertex labels). In the rest of this section it will be convenient for us to talk about the set of vertices that have a specific shape or colour. We let the set $V_{\text{label}}$ represent the set of vertices with that label (shape, colour, or other). □

### B.5.1 SatLib

To test how well each approach performs on solving highly constrained SAT instances, we translated SAT instances from the SatLib library [37] into SAT-TSP instances. The construction of the SAT-TSP instances is as follows: create a fully connected undirected graph $G$ with $|V|$ vertices of zero edge weight and force each vertex to be included in the solution, $F \leftarrow F \wedge (x_{v_1} \wedge x_{v_2} \wedge \ldots)$. Then the SAT-TSP instance $\langle G, F, c = 0 \rangle$ is equivalent to the SAT instance $F$. The optimization version of the SAT-TSP translation is also equivalent to the SAT instance since every solution for the SAT-TSP instance has a cost of zero (every solution is optimal). There are 41 instances in this library ranging in size from 20 to 600 variables in the formula $F$.

### B.5.2 TspLib

To test how well each approach performs on simple patrolling problems, we translated TSP instances from the TspLib [77] into SAT-TSP instances. The construction of the SAT-TSP

instances is as follows: create a formula $F = (x_{v_1} \wedge x_{v_2} \wedge \ldots \wedge x_{v_{|V|}})$. Notice that there is only one solution to the formula $F$ (include every vertex in the solution), thus the SAT-TSP instance $\langle G, F \rangle$ is equivalent to the TSP instance $\langle G \rangle$. There are 66 instances in this library ranging in size from 16 to 1379 vertices.

### B.5.3 HardLib

We also created highly-constrained planning problems with complex environments by combining instances from the SATLIB and TSPLIB libraries. The SAT-TSP instances $\langle G, F \rangle$ uses the formula $F$ from the SAT instance and the graph $G$ from the TSP instance for combinations that satisfy $|X| \geq |V|$. There are 348 instances in this library.

### B.5.4 SetLib

We created the SETLIB library to test how well each solver performs on nested set problems. The problem instances are randomly generated as described in the metric and non-metric *library environments*. The problem is constrained to have one item of each shape, all the same colour in the solution. There are 200 problem instances in this library ranging in size from 10 to 100 vertices.

### B.5.5 GTspLib

To test how well each approach performs on one-in-a-set routing problems or similar problems, we translated GTSP instances from the GTSPLIB library [30] to SAT-TSP instances $\langle G, F \rangle$ where,

$$F = \bigwedge_{j=1}^{|S|} \left( \bigvee_{v_i \in S_j} x_{v_i} \right).$$

There are 83 instances in this library ranging in size from 17 to 1084 vertices in the graph.

### B.5.6 GTspLib$^+$

We created the GTSPLIB$^+$ library to test how well each approach performs on GTSP instances with additional SAT constraints. The problem instances are randomly generated as described in the metric and non-metric *library environments*. The GTSP sets are of

random size with random non-overlapping membership. There are 0 to 200 additional constraints of the form: negation, $\neg(v_i \wedge v_j)$ or implication, $(v_i \implies v_j)$. There are 262 instances in this library ranging in size from 10 to 500 vertices.

### B.5.7 CountLib

The robot is required to retrieve exactly one cube, at most three spheres, and at least five cylinders from its environment. The problem instances are randomly generated as described in the metric and non-metric *library environments*. These problems are constrained to retrieve an exact but random number of items for each shape (uniformly chosen from the set of possibilities). There are 200 instances in this library ranging in size from 10 to 100 vertices.

The problems are translated to SAT-TSP using adder circuits.

### B.5.8 OrderedLib

These problem instances are pickup and delivery problems. The environments are generated as in the metric and non-metric *library environments*, however, the item assignments are different. Specifically, the environment locations are a collection of pickup $v_i$ and drop off locations $v_j$ (chosen at random) — $v_i$ is paired with $v_j$. The robot must visit all the locations. For every pickup and delivery pair $v_i, v_j$ the tour must visit $v_i$ before $v_j$. There are 200 instances in this library ranging in size from 10 to 100 vertices.

These problems are translated to SAT-TSP instances by using the structure in Remark 3.4.10. Specifically, the graph is constructed with a series of widgets. When separated from the rest of the graph, each widget is simply an induced subgraph of the robot's environment $G$. The first widget $\hat{\Omega}^0$, contains only the vertex $v_s$ (the home) and the remaining widgets $\hat{\Omega}^1, \hat{\Omega}^2, \ldots$, contain the vertices $V \setminus v_s$. The widgets are connected with directed edges so that the path may visit them only in increasing order according of their index. Figure B.6, Figure B.7, and the following list details how the widgets are connected. First we require some notation: let the vertex label $\hat{v}_i^j$ represent the vertex $v_i$ in widget $\hat{\Omega}^j$.

1. $\langle \hat{v}_i^\lambda, \hat{v}_j^\gamma \rangle \in \hat{E}$ with weight $w(v_i, v_j)$ for $\hat{v}_i^\lambda, \hat{v}_j^\gamma \in \hat{V}$, $\gamma \geq \lambda$ if and only if $\langle v_i, v_j \rangle \in E$
2. $\langle \hat{v}_i^\lambda, \hat{v}_s^0 \rangle \in \hat{E}$ with weight $w(\hat{v}_i^\lambda, \hat{v}_s^0) = 0$ for all $\hat{v}_i^\lambda \in \hat{V} \setminus \hat{v}_s^0$

The entire graph has $O(|V|^2)$ vertices and $O(|V|^2|E|)$ edges.

If the SAT-TSP solution visits one of the copies of vertex $v_i$ in one of the widgets, then the solution visits $v_i$. To ensure that this happens at most once, we use an at-most-one-in-a-set constraint. To ensure the vertex ordering constraints are satisfied, we negate every pair of vertices that violate the ordering. For example, suppose vertex $v_i$ must precede vertex $v_j$ in the solution. That means for every pair $\hat{v}_i^\lambda$ and $\hat{v}_j^\sigma$ such that $\lambda \geq \sigma$ we add the negation clause $\neg(\hat{x}_{\hat{v}_i^\lambda} \wedge \hat{x}_{\hat{v}_j^\sigma})$ to the formula. There are $O(|V|^2)$ such negation clauses for each ordering constraint.

In the case that we have precedence constraints for a set of vertices $A = \{v_{a_1}, v_{a_2}, \ldots, v_{a_{|A|}}\}$ and a set $B = \{v_{b_1}, v_{b_2}, \ldots, v_{b_{|B|}}\}$, such that every vertex $v_a \in A$ must precede every vertex $v_b \in B$, then we use indicator variables to reduce the number of negation clauses from $O(|A||B||V|^2)$ to $O(|V|^2)$. For example, the set of indicator variables $\hat{x}_A^\lambda \in \{\hat{x}_A^0, \hat{x}_A^1, \ldots, \hat{x}_A^{|V|}\}$ and $\hat{x}_B^\lambda \in \{\hat{x}_B^0, \hat{x}_B^1, \ldots, \hat{x}_B^{|V|}\}$ are used to represent a vertex in set $A$ or $B$ while being visited in widget $\lambda$ respectively (logic below).

$$(\hat{x}_{\hat{v}_{a_1}^1} \vee \hat{x}_{\hat{v}_{a_2}^1} \vee \ldots \vee \hat{x}_{\hat{v}_{a_{|A|}}^1}) \implies \hat{x}_A^1$$

Then we negate the indicators instead of the vertex pairs to produce $O(|V|^2)$ negation clauses instead of $O(|V|^2)$ for each unique pair $v_i, v_j$ such that $v_i \in A$ and $v_j \in B$.

A solution to the above expression is a solution to the ordered problem since all subgraphs derived from solutions cannot violate the ordering constraints. The solution also has the equivalent cost since the corresponding transitions in $\hat{G}$ are equivalent to transitions in $G$.

### B.5.9 MultiRobotLib

A team of robots is tasked with working together to collect at least one item of each colour and at least one item of each shape from the environment. The problem instances are randomly generated as described in the metric and non-metric *library environments*. Each problem instance has a robot team of random size ranging from two to ten robots. There are 200 instances in this library ranging in size from 10 to 100 vertices.

These problems are translated to SAT-TSP instances by using the structure in Remark 3.4.10. Specifically, we construct a graph to contain a widget for each robot. The widgets $\hat{G}^r$ are the induced subgraph of the robot's environment with vertices $V \setminus v_s$, where $v_s$ is the robot's home location. The graph additionally contains the vertices $\hat{v}^r$ for each robot to represent its home. The edges are constructed to force the tour to visit each

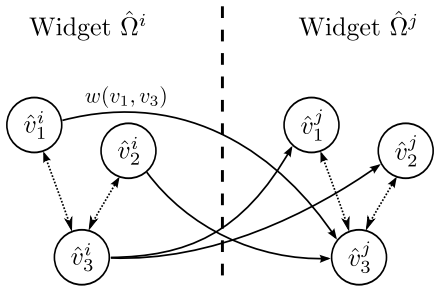



Figure B.6: The connection of widget $\hat{\Omega}^i$ to $\hat{\Omega}^j$ for the reduction of the ordering constraints. Note that only connections from $\hat{\Omega}^i$ to $\hat{\Omega}^j$ exist and not the other way around. Each widget is a copy of the original graph $G$, of which the edges are represented with dotted lines. The edges connecting vertices from one widget to another shown with solid lines are only present if the connection exists in $G$. The edge between vertex $\hat{v}_1^i$ and $\hat{v}_3^j$ highlights how the cost mimics the edge weights in $G$.

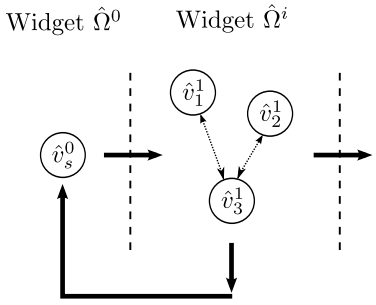



Figure B.7: The connections between widgets for the reduction of the ordering constraints. Widgets are connected with directed edges in sequential order. All widgets are connected back to the first widget to allow the solution tour to close.

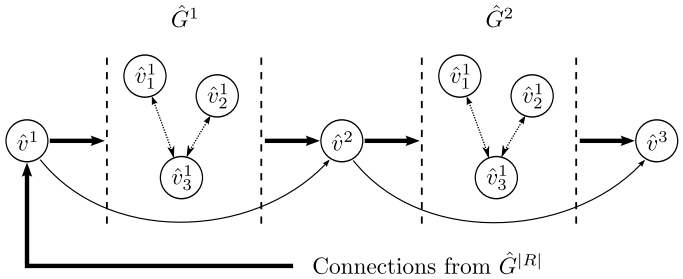



Figure B.8: The connections between widgets for the reduction of the multi-robot problem. Widgets are connected with directed edges in sequential order. All widgets are connected back to the first widget to allow the solution tour to close.

robot's home. The path is only able to visit vertex $\hat{v}_i^r$ for robot $r$ only after vertex $\hat{v}^r$ has been visited and before vertex $\hat{v}^{r+1}$ is visited. The connections are detailed in the list below and Figure B.8.

1. $\langle \hat{v}_i^\lambda, \hat{v}_j^\lambda \rangle \in \hat{E}$ with weight $w(v_i, v_j)$ if and only if $\langle v_i, v_j \rangle \in E^\lambda$.
2. $\langle \hat{v}^\lambda, \hat{v}_i^\lambda \rangle \in \hat{E}$ with weight $w(v_s, v_j)$ if and only if $\langle v_s, v_j \rangle \in E^\lambda$.
3. $\langle \hat{v}_i^\lambda, \hat{v}^{\lambda+1} \rangle \in \hat{E}$ with weight zero.
4. $\langle \hat{v}^\lambda, \hat{v}^{\lambda+1} \rangle \in \hat{E}$ with weight zero.

Since we cannot have multiple robots collect the same item, we use an at-most-one-a-set constraint to restrict multiple visits to the same location.

A solution to the above expression is a solution to the multi-robot problem. The solution also has the equivalent cost because the corresponding transitions in $\hat{G}^r$ are equivalent to transitions of robot $r$ in its environment graph.

## B.6 Benchmark Results

The results presented in this section we document the solver's run times and solution qualities. We omit the reduction times (time to translate SAT-TSP instances into other problems). The solver time reflects the time to find the optimal solution. In some instances the solver's search is terminated prematurely to comply with the time budget. The solvers

are given time budgets in 60 second intervals (60, 120, 180, ..., 900). We report only the solver's best solution with the best time. The simulations were executed on the cluster computing resource SHARCNET[1], which we used more than five years of computing time.

To evaluate each solver approach we use point plots and bar graphs. Both of these plots/graphs use a categorization scheme of the solution qualities which is found in Tables B.1 and B.2 respectively. We also compare the performance and/or quality to metadata for some of the approaches. We present this data only when the comparison reveals a correlation between performance and/or quality with respect to the metadata. Such metadata may be the total number of feasible solutions for the SAT-TSP instance or the number of additional constraints that have been added to the GTSPLIB$^+$ instance (Section B.5.6).

| **Solution Category:** Description | |
|---|---|
| **A** : | Solutions that achieve the best known cost. |
| **B** : | Solutions that achieve a solution cost within two times the best known cost, but not the best cost. |
| **C** : | Solutions that have cost worse than two times the best known cost. |

Table B.1: Descriptions of the mutually exclusive solution categories used in Figures B.9 to B.13.

### B.6.1 The Unsuccessful Approaches (SAT, TSP and SMT)

Three of the proposed approaches: SAT, TSP and SMT were unsuccessful compared to the other approaches (see Figure B.14). We developed the SAT and TSP approaches to explore how well SAT solvers and TSP solvers could respectively handle the addition of TSP and SAT problems. It turns out that both approaches performed poorly. Furthermore, the TSP approach performed poorly on TSP problems encoded as SAT-TSP problems. This forces us to conclude that our TSP reductions introduce additional complexity into the problem, which was needed to handle the SAT formula. Thus, it is unlikely that a pure TSP approach would be a good candidate for SAT-TSP problems.

We developed the SMT approach to allow for partial tour negations as in the CBTSP approach. However, it suffers a $O(|V|^3)$ increase in size due the translation of the Hamiltonian cycle problem (the SAT approach also suffers from this size increase). We suspect that this size increase is a contributing factor to its inefficiency.

[1] `https://www.sharcnet.ca`

### B.6.2 GTSP Approach

The GTSP approach reduces a SAT-TSP instance to $|V|$ TSP instances, where each instance has a different assumption of which $v_s \in V$ is in the solution. It uses the GTSP solver, GLNS [88], to find optimal tours. The best result out of the $|V|$ solutions is used. The solver divides the time budget evenly among each GTSP instance and it takes the time budget as input to terminate itself.

**Strengths** We developed the GTSP approach to help solve GTSP like problems as several of the proposed approaches perform poorly on these types of problems. As we had hoped, this approach does perform well on GTSP and GTSP-like instances (TSPLIB, GTSPLIB and GTSPLIB$^+$). Figures B.15b, B.16a and B.16b compile the results for this approach.

Figure B.14 shows that this approach is able to find a number of the best known solutions and Figure B.9a shows that if it does find the optimal solution then it is often the best performer.

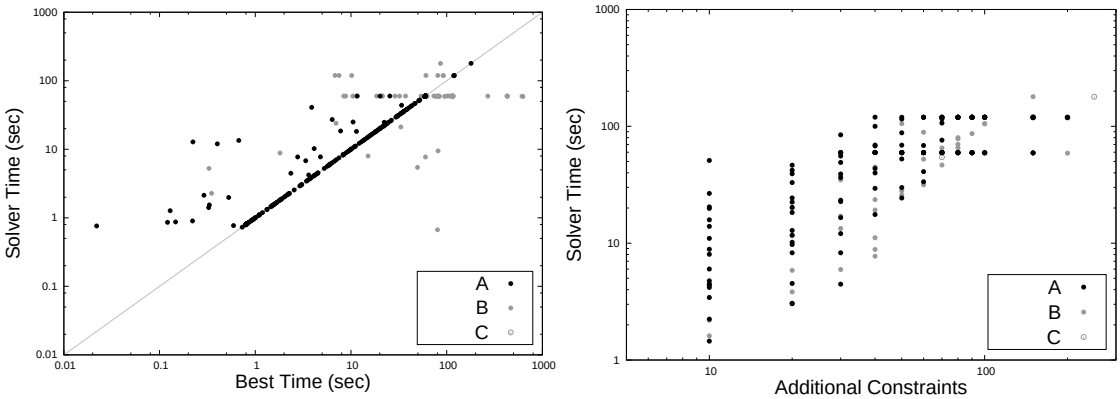
**Weaknesses** The GTSP approach was not designed to handle highly constrained problems such as SAT or counting problems. Thus this approach struggles with such problems, which is shown in Figures B.15a, B.15c, B.17a, B.17b, and B.18b.

Figure B.9b) shows that this approach degrades as the GTSP-like instances become less GTSP like.

### B.6.3 CSP Approach

The CSP approach reduces a SAT-TSP instance to $|V|$ CSP instances each with a different assumption of the solution tour size. It uses the CSP solver Gecode [85] to find the optimal solution of each translation and the best result is used. The CSP approach divides the time budget evenly among each CSP instance and the CSP solver takes the time budget as input to terminate itself.

**Strengths** We developed the CSP approach because it is relatively easy to express logic and optimization problems. Additionally, CSP solvers are relatively good at finding solutions, which we can see from Figure B.14.




(a) Time vs. Best

(b) GTspLib$^+$: Time vs. Constraints

Figure B.9: Performance results for the **GTSP** approach. The results are broken down into three categories described in Table B.1. **(a)** Compares the solver performance to the best performance achieved over all approaches. **(b)** Compares the solver performance to the number of additional constraints in GTspLib$^+$.

**Weaknesses** Despite the fact that this approach is able to find solutions, it often finds non-optimal solutions. A number of the non-optimal solutions are more than two times optimal as shown in Figure B.14. This makes the CSP approach difficult to trust (if we get a solution, we cannot trust that the solution is near optimal).

Unfortunately, this approach also has quite poor performance — up to 1000 times slower than the best approach — as shown in Figure B.11. Overall, this approach is not able to find optimal solutions within a reasonable time budget.

### B.6.4 BRUTE Approach

The BRUTE approach excels at finding feasible solutions but not necessarily optimal solutions. This approach uses the SAT solver, minisat, to find a feasible set of included vertices and then it uses the TSP solver, LKH, to find the optimal tour of these vertices. It then adds the negation of the solution to the formula preventing the solution from reoccurring, and then it starts the process over again. The solver is given multiple time budgets and the best result is used. The timeout behaviour is responsible for the grouping of solution times around the 60 second intervals as shown in Figure B.12. Even though this approach is inefficient, it is often able to exhaustively search the space and find the best solution in the best time or the approach randomly found the best solution before it terminated.

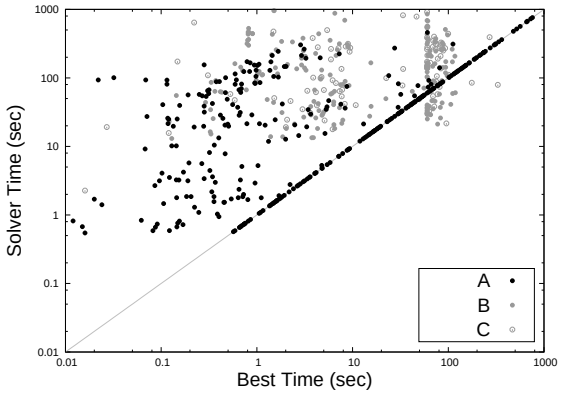



Figure B.10: CSP (Gecode)

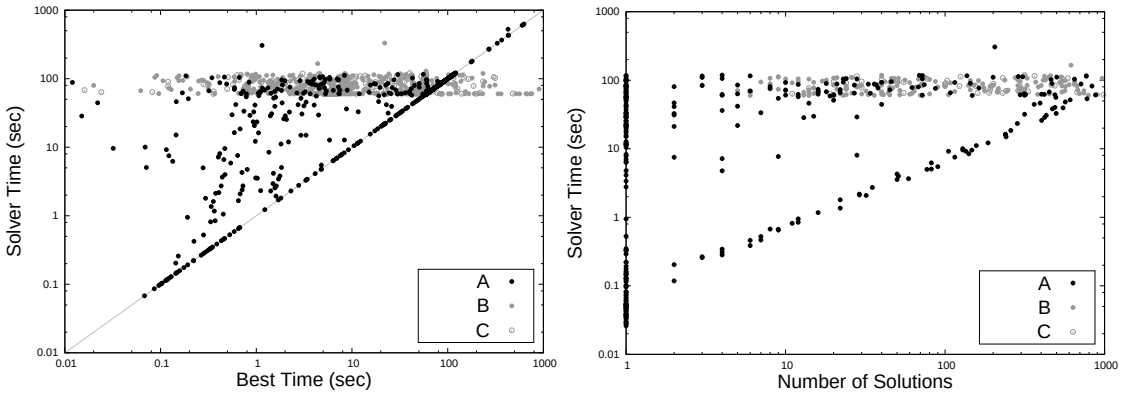
Figure B.11: Performance results of the **CSP** approach on the simulation library. The results are broken down into three categories described in Table B.1.

**Strengths** The BRUTE approach provides a straightforward method to leverage the best performing SAT and TSP solvers. As shown in Figure B.14 the approach is frequently able to find the best known solution and it often outperforms the other approaches. A good property of this solver is that if the solver quits before the time budget is exceeded and the TSP solver is an exact solver such as Concorde [3], then it has found the optimal solution. Our implementation uses LKH which sometimes returns sub-optimal solutions (LKH is more efficient than Concorde and has very good performance). We see this behaviour in Figures B.12a by observing that all the results below 60 seconds are the best known solutions (category $A$).

Figure B.15b verifies that this approach is the best choice for solving instances in the TSPLIB. This is not surprising, since the SAT formula in the SAT-TSP instance only has one solution to search. For such instances, the brute approach quickly solves the SAT formula and then uses the TSP solver, LKH, to solve the TSP problem.

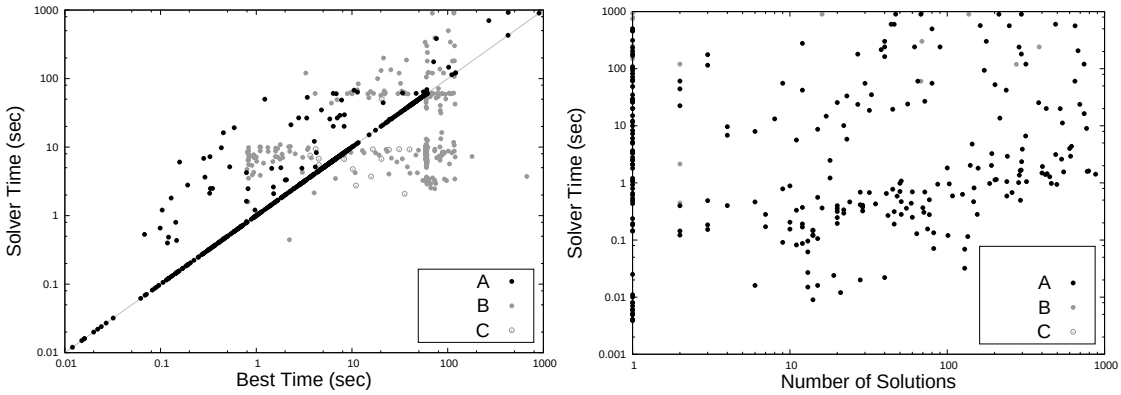
**Weaknesses** This approach searches all of the SAT solutions and only after it has explored each solution does it provide the optimal result. If the search is prematurely terminated (timed out), the solution quality can vary wildly. As such the solver's run time is heavily dependent on how many solutions it searches. Figure B.12b verifies this correlation.

This approach struggles GTSPLIB, SETLIB and COUNTLIB instances since they have a large number of feasible solutions to search (see Figures B.16a-B.17b).




(a) Time vs. Best (b) Time vs. Solutions

Figure B.12: Performance results of the **BRUTE** approach on the full library. Results are divided into categories described in Table B.1. **(a)** Compares the solver time to the best performance achieved over all approaches for SAT-TSP instances. **(b)** Compares the solver time to the number of SAT-TSP solutions of the instance.




(a) Time vs. Best (b) Time vs. Solutions

Figure B.13: Performance results of the **CBTSP** approach on the simulation library. The results are broken down into three categories described in Table B.1. **(a)** Compares the solver time to the best performance achieved over all approaches. **(b)** Compares the solver time to the number of SAT-TSP solutions of the instance.

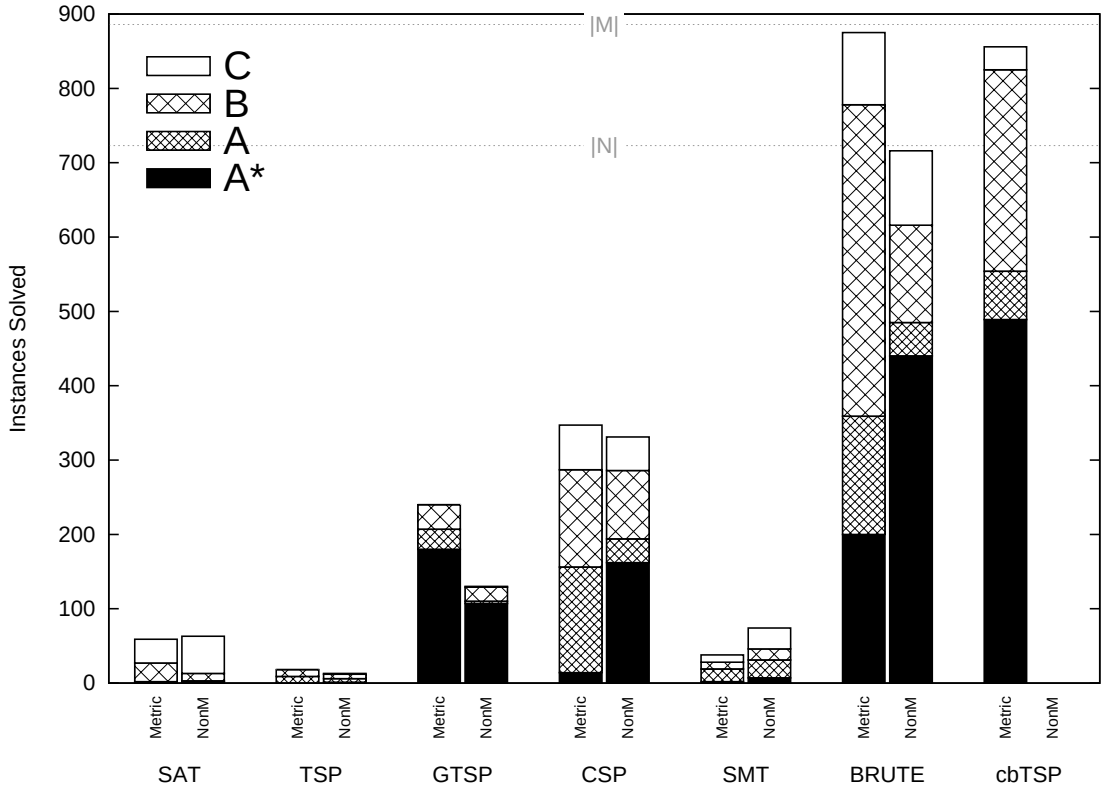
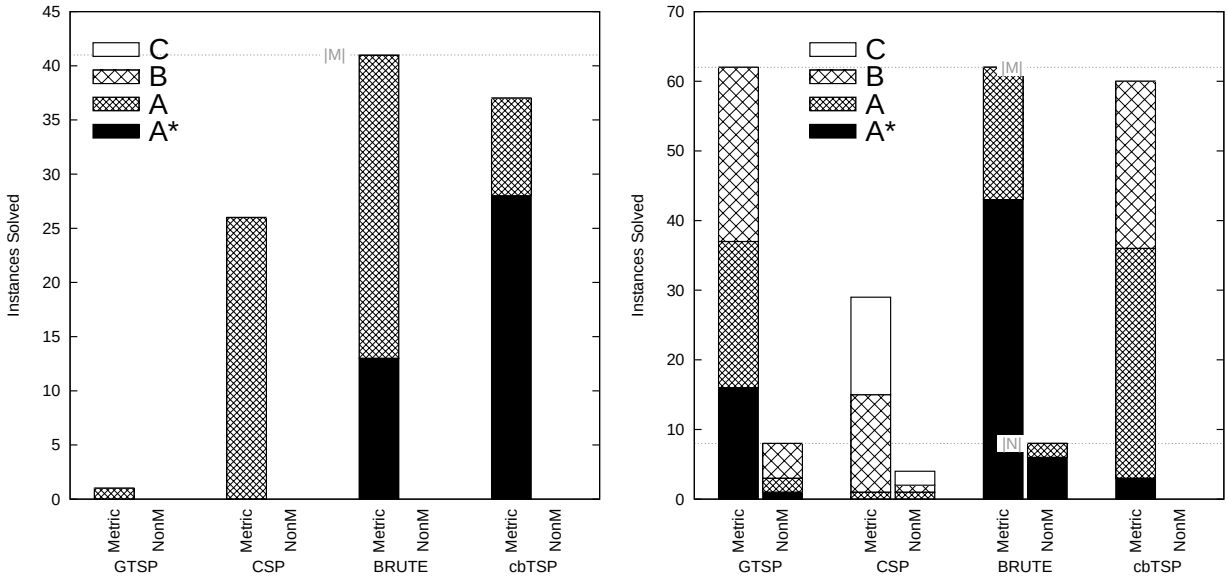
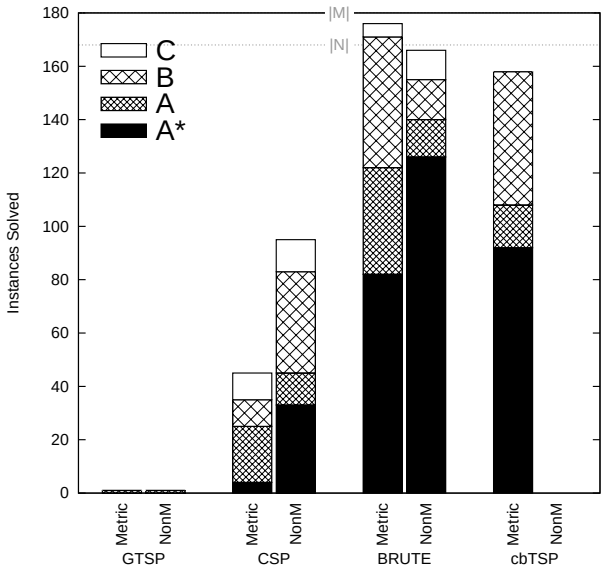



Figure B.14: The number of instances solved by each solver approach. The results are broken down into four categories described in Table B.2. The number of metric and non-metric instances are indicated on the graph with the dotted line and the $|M|$ and $|N|$ symbols respectively.

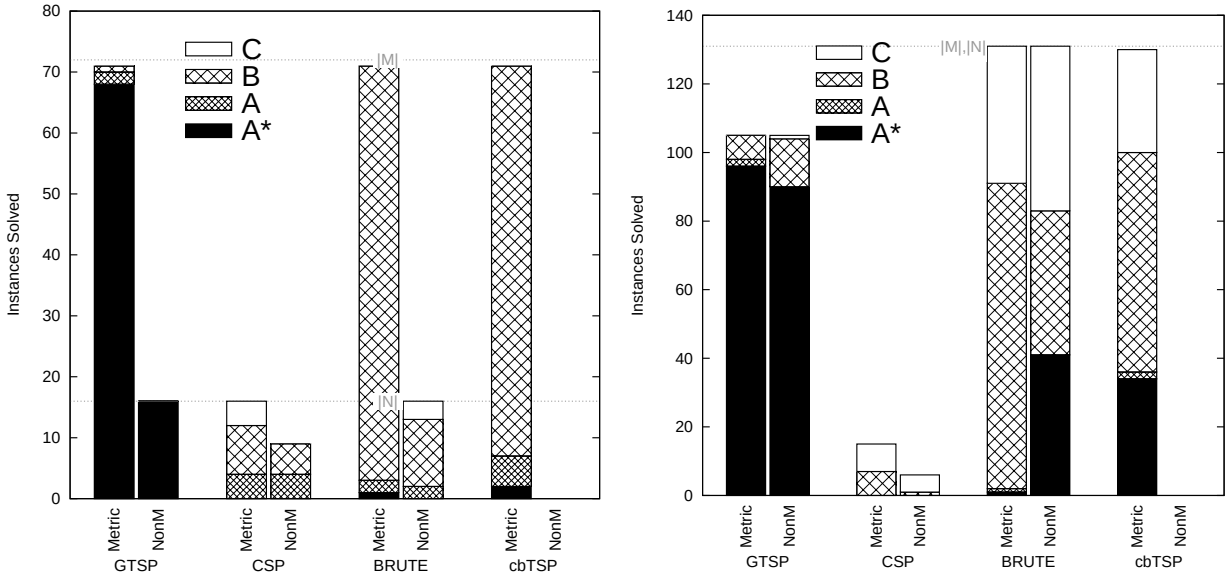



(a) SatLib (b) TspLib



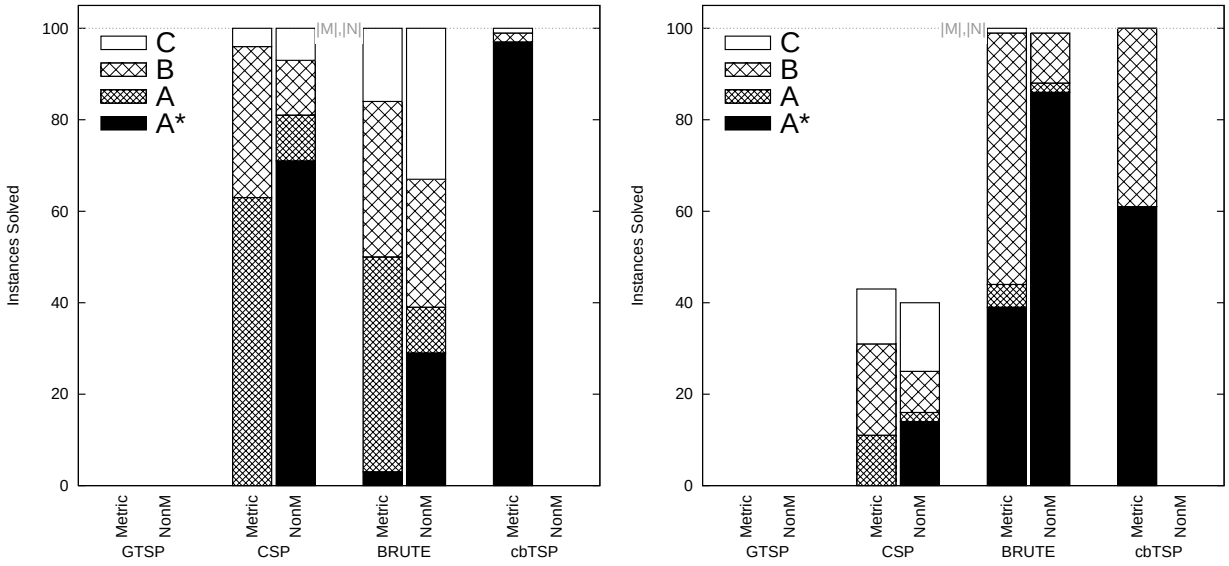

(c) HardLib

Figure B.15: The number of instances solved in their respective library by each successful solver approach. The results are broken down into four categories described in Table B.2. The number of metric and non-metric instances are indicated on each graph with the dotted line and the $|M|$ and $|N|$ symbols respectively (there are no non-metric instances in SatLib).



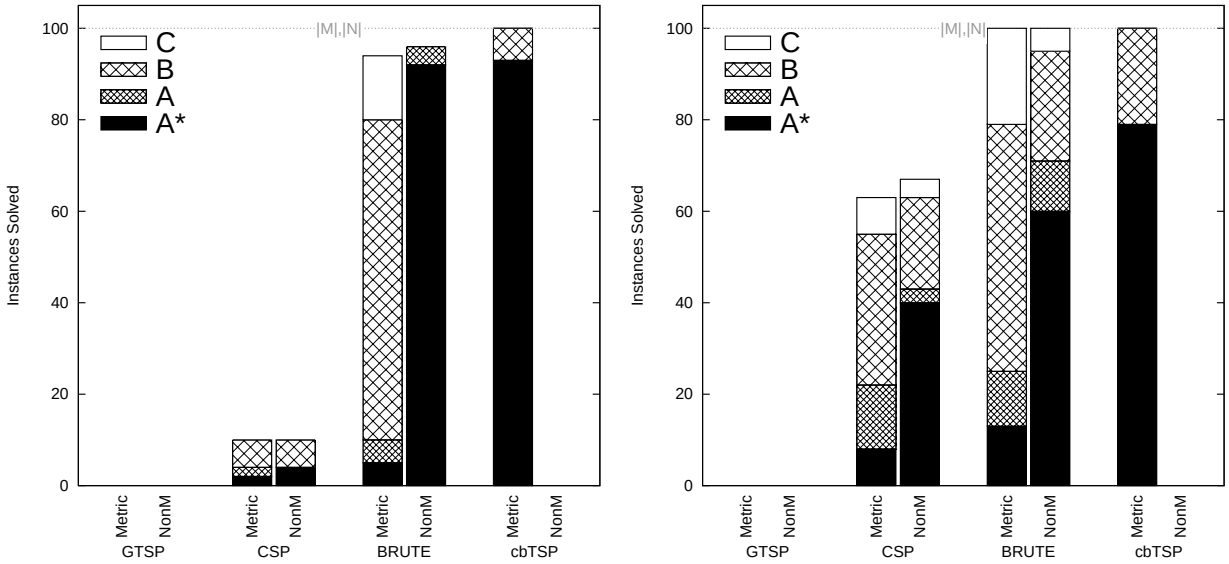

(a) GTspLib (b) GTspLib$^+$

Figure B.16: The number of instances solved in their respective library by each successful solver approach. The results are broken down into four categories described in Table B.2. The number of metric and non-metric instances are indicated on each graph with the dotted line and the $|M|$ and $|N|$ symbols respectively.




(a) SetLib (b) CountLib

Figure B.17: The number of instances solved in their respective library by each successful solver approach. The results are broken down into four categories described in Table B.2. The number of metric and non-metric instances are indicated on each graph with the dotted line and the $|M|$ and $|N|$ symbols respectively.




(a) OrderedLib (b) MultiRobotLib

Figure B.18: The number of instances solved in their respective library by each successful solver approach. The results are broken down into four categories described in Table B.2. The number of metric and non-metric instances are indicated on each graph with the dotted line and the $|M|$ and $|N|$ symbols respectively.

| **Solution Category:** | Description |
|---|---|
| $\mathbf{A}^*$ : | Solutions that achieve the best known cost and time. |
| **A** : | Solutions that achieve the best known cost but not time. |
| **B** : | Solutions that achieve a solution cost within two times the best known cost but not the best cost. |
| **C** : | Solutions that have cost worse than two times the best known cost. |

Table B.2: Descriptions of the mutually exclusive solution categories used in Figures B.14 to B.18b.