An SMT-Based Approach to Motion Planning for Multiple Robots with Complex Constraints

Frank Imeson, Student Member, IEEE, Stephen L. Smith, Senior Member, IEEE

Abstract—In this paper we propose a new method for solving multi-robot motion planning problems with complex constraints. We focus on an important class of problems that require an allocation of spatially distributed tasks to robots, along with efficient paths for each robot to visit their task locations. We introduce a framework for solving these problems that naturally couples allocation with path planning. The allocation problem is encoded as a Boolean Satisfiability problem (SAT) and the path planning problem is encoded as a Traveling Salesman Problem (TSP). In addition, the 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 (SMT) 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 our algorithm significantly outperforms state-of-the-art mathematical programming solvers.

Index Terms—Path Planning for Multiple Mobile Robot Systems, Surveillance Systems, Autonomous Agents, Scheduling and Coordination

I. INTRODUCTION

Robots are increasingly being asked to perform complex missions that require the efficient coordination of multiple robots over large and complex physical spaces. In this paper, we focus on motion planning problems where a robot or group of robots are dispatched to complete a set of spatially distributed and interdependent tasks. Such problems arise in a variety of robot applications. In environmental monitoring and patrolling, robots with on-board sensing are tasked with selecting and visiting viewpoints to assess traffic conditions [1], crop health [2], security threats [3], [4], or for general data collection [5]. In exploration tasks, robots are used to observe and potentially collect samples from a previously unexplored environment [6], [7], [8] (an example sample collection task for two robots is shown in Figure 1). In flexible manufacturing and material transport, a group of robots must repeatedly visit a set of locations with a specified frequency to move and deliver materials within a factory [9], [10], [11].

In all of these applications, the common aspects are that tasks must be allocated to the robots and each robot must plan a route to accomplish its set of tasks (note, the problems of allocation and routing are typically coupled). Each problem has additional constraints such as battery life, visit frequencies, robot-task incompatibilities, and carrying capacities. The typical approach for solving these problems is to create a roadmap for each robot that captures the environment and its transition costs [12]. This separates the planner into two parts, a high-level planner, which is responsible for optimizing motion plans within the roadmap, and a low-level planner [13], [12], [14], which tracks the resulting path and avoids collisions. In this paper we focus on the high-level planner and introduce a common framework in which a broad class of motion planning problems (including their additional constraints) can be expressed and efficiently solved. Additionally, for this paper, we assume collision avoidance between multiple robots is handled by the low-level planner.

Research in high-level motion planning has followed two main approaches. The first is to design a custom algorithm for specific problems. These custom algorithms draw from a broad range of areas including approximation algorithms [15], [2], branch-and-bound techniques [3], and advanced heuristics [4]. The algorithms typically leverage some specific structure in the problem, and thus a drawback is that they are often not portable to related problems in which a subset of the constraints differ. The second approach is to express the problem in a standard form on which a powerful commercial-grade solver can be used. For example, a commonly used standard is integer linear programming (ILP), for which several advanced and user-friendly solvers exist, including CPLEX [16] and Gurobi [17]. However, a common difficulty in this approach is scalability, since there is no way to leverage problem specific structures such as allocation and routing aspects that are present in multi-robot planning problems. It is this observation that we seek to address in our proposed solution methodology.

In this paper we introduce a framework for expressing and solving high-level motion planning problems. The framework is designed to leverage the problem-specific structure of high-level planning problems, while maintaining the ability to express a wide variety of additional constraints such as interdependencies between tasks, incompatibilities between certain robots and tasks, battery life, and capacity constraints. To this end, we introduce a general problem called SAT-TSP...
and an algorithm for solving these problems, called cnTsp. The problem takes as input a set of weighted graphs, a Boolean formula, and a cost budget. The graphs are used to represent the motion roadmap for each robot. The Boolean formula is used to express the logical constraints on the robots’ motion, such as task dependencies, incompatibilities, and capacity constraints. The cost budget is a constraint on the robots’ total transition costs (distance or time). A SAT-Tsp solution is a set of paths for each robot that visits a subset of the locations in the environment, satisfy the constraints of the problem, and stay within the cost budget. To solve this problem, we propose an algorithm based on Satisfiability Modulo Theory (Smt). The algorithm leverages a state-of-the-art Boolean Satisfiability (Sat) solver [18] to satisfy the constraints of the problem, and a traveling salesman problem (Tsp) solver [19] to route the robots through their assigned tasks. The Smt approach enables these two solvers to coordinate their efforts, with the Sat solver proposing candidate allocations, and the Tsp solver evaluating their cost, helping to guide the search.

Contributions: The contributions of this paper are as follows. We formally introduce the SAT-Tsp problem and propose the novel solver cnTsp. We characterize the complexity of this problem, and show that the proposed algorithm is correct for instances in which the robotic roadmaps are Tsp-monotonic (a generalization of metric graphs). We compare cnTsp to 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 cnTsp often outperforms the ILP solver. Additionally, we have included a supplementary video, which demonstrates a solution for a multi-robot sample collection problem in a high-fidelity ROS [20] simulation.

The SAT-Tsp problem was first introduced in our preliminary works [21] and [22]. These papers proposed solving problems expressed in SAT-Tsp through 1) a reduction to Tsp, 2) a reduction to the generalized version of Tsp, and 3) a reduction to the constraint satisfiability problem. The work also proposed a preliminary version of our brute force solver as presented in Section IV-A. The contributions of this paper over the preliminary work is to propose a powerful solution methodology based on Smt that far outperforms the methods in this early work. We also prove that SAT-Tsp is NP-hard, even when the constraints and the routing aspect of the problem can be efficiently solved independently. Finally, we express three planning problems in SAT-Tsp and compare the performance of the proposed solver against a state-of-the-art commercial-grade ILP solver.

Organization: The rest of this paper is organized as follows. In the remainder of this section we review methods for solving high-level planning problems, and the use of Smt-based solvers. In Section II we cover the necessary background needed for this paper. In Section III we formally introduce the SAT-Tsp problem. In Sections IV and V we describe the cnTsp solver and our ILP formulation respectively used for the simulations. In Section VI we detail the three different path planning problems used for our simulations and describe how they are expressed in SAT-Tsp and ILP. In Section VII we detail the simulation setup for each problem and compare the solution quality of cnTsp to the ILP solver. In Section VIII we conclude the paper and review future work.

Related work: There are a number of methods for solving high-level path planning problems. One of the most common is to use Mixed Integer Linear Programming (MILP) or Integer Linear Programming (ILP) [23]. This framework allows the user to express their problem as a series of linear constraints along with an optimization objective. There are a number of powerful solvers that one can use to find optimal solutions such as CPLEX [16] and Gurobi [17]. In [24] the authors use MILP to plan optimal trajectories, and in [25], [26] ILP is used to solve a multi-robot charging problems. In [27] the authors give an ILP solution for collision-free multi-robot planning. In this paper we provide a direct comparison of our SAT-Tsp solver, cnTsp, to the well known ILP solver, Gurobi, on a similar set of robotic path planning problems and our results show that cnTsp often outperforms the ILP solver.

Another common approach for high-level path planning is to use Linear Temporal Logic (LTL) [28]. The LTL language allows a user to express a problem as a set of state transitions such as “if the robot is at location x then the next location it will visit is y.” Solvers developed in the model checking community can then be used to compute runs (expressed as an automaton) that satisfy the LTL formula [29], [30]. In [31], 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 [32], LTL is used to express multi-robot planning problems. A drawback of LTL is that the problem of determining if a LTL formula is satisfiable is in the complexity class PSpace-complete [33], while we show that deciding if a SAT-Tsp problem is satisfiable (the decision version of SAT-Tsp) is in the complexity class NSpace-complete (see Section IV). 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. However, many important path planning problems lie in NP, and thus our goal in this paper is to produce a solver that is tailored to problems in this class.

Other approaches for high-level planning include the STRIPS problem specification language [34] and its successor, PDDL [35]. Like LTL, the language allows a user to express a problem as a set of state transitions, although PDDL is even more expressive than LTL. There are a number of good solvers for these expressions such as FF [36] and LAMA [37]. These languages are capable of expressing any problem in EXPSpace [38], which is widely believed to be a strict superset of PSpace and thus NP. In this work we focus on problems in NP, and thus ILP (which lies in the same complexity class as SAT-Tsp) will serve as our point of comparison.

Our main solver, cnTsp 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 [39] and [40], 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 [41], [42] and [43], 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. 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.

The Constraint Satisfiability Problem (Csp) is a further generalization of SMT and thus SAT. Both SMT and Csp are capable of expressing and solving the types of problems proposed by this paper. In [44, Appendix B] we expressed SAT-Tsp instances in both SMT and Csp and then solved instances with state-of-the-art SAT and Csp solvers. Both methods performed poorly compared to an ILP solver, and thus we do not use these approaches as a comparison in this paper.

II. BACKGROUND

In this section we define SAT and Tsp, review SMT and the aspects of the SMT solver approach DPLL(T) used by cnTsp. As well, we introduce some notation for expressing Boolean formulae.

A. SAT, Graphs, and Tsp

Decision problems are problems that have a yes or no answer (satisfiable or unsatisfiable). In this paper we also study optimization problems that aim to minimize the cost of the solution and, for simplicity, we formally define these optimization problems as their decision problems versions.

The Boolean satisfiability problem SAT is expressed as a Boolean formula that contains literals and operators. A literal is either a Boolean variable (x) or its negation (¬x). The operators are conjunction (∧, and), disjunction (∨, or) and negation (¬, 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 and a formula F is in its canonical form if the formula is a conjunction of clauses, where each clause is a disjunction of literals. In this paper we allow for formulae to be expressed in non-canonical form.

Definition II.1 (SAT). Given a SAT Boolean formula F, determine if it is satisfiable.

A graph is expressed as a tuple, (V, E, w), where V is the set of vertices, E is the set of edges, and w is the weight function that assigns a cost in \( \mathbb{R}_{\geq 0} \) to each edge. Note, for this paper we assume all edges are directed (\( (v_i, v_j) \) is not the same as \( (v_j, v_i) ) \). An induced subgraph, which will be used in the upcoming sections, is defined as follows.

Definition II.2 (Induced Subgraph). An induced subgraph of a graph \( G = (V, E) \) for \( V' \subseteq V \) is a graph \( G' = (V', E') \) with \( E' = \{ (v_i, v_j) \in E \mid v_i, v_j \in V' \} \). We say that \( G' \) is induced by \( V' \).

A cycle in a graph that visits each vertex exactly once is called a Hamiltonian cycle, or a tour. The traveling salesman problem (Tsp) is typically defined as finding the shortest tour in a graph, and its decision version is as follows.

Definition II.3 (Tsp). Given a complete and weighted graph \( (V, E, w) \) and a cost budget \( c \), determine if there exists a Hamiltonian cycle with total cost \( c \) or less.

The optimization version of Tsp minimizes \( c \).

B. SMT and DPLL(T)

The SMT problem is an extension of SAT that additionally allows for the expression of multiple decidable first-order logic problems, such as arithmetic logic or quantifiable Boolean logic. We refer to each additional first-order formal t as a theory and the set of theories as T. Each theory \( t \in T \) is linked through the propositional logic formula F (the SAT formula) using a Boolean variable \( x_t \in X \), where the value of \( x_t \) must agree with the evaluation of t (\( x_t \) is true if and only if t evaluates to true). The power of this approach is that a SAT solver can be used to solve the SMT-aspect of the problem, while a set of specialized solvers can be leveraged for each class of theory (specializations of first-order logic).

Definition II.4 (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 \( X \subseteq Q \), and
3) 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).

The DPLL(T) algorithm for solving SMT instances is based on the DPLL algorithm for 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). The DPLL(T) algorithm extends DPLL to allow for theories by linking the predicates \( x_t \in X \) to the theories \( t \in T \). Informally the algorithm works as follows: As a partial solution for \( F \) is constructed by the DPLL algorithm, the theory solvers are called to confirm that the partial SAT solution is consistent with the theory instances. If the theory instances are consistent, then the DPLL algorithm continues, if not, then the theory solver that detected the conflict constructs a learnt clause \( f_{\text{conflict}} \) which captures the conflict over the 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 \( F \). A trimmed down algorithm for DPLL(T) is shown in Algorithm 2.

Note II.5. The above description of DPLL(T) only captures the mechanisms of DPLL(T) that are used by cnTsp, a full description of DPLL(T) can be found in [45].

III. PROBLEM STATEMENT

In this paper we consider path planning problems that require a set of robots to visit a set of locations in the environment to accomplish the problem goals (the robots complete a set of tasks at their locations). An optimal solution is one that allows the robots to accomplish the goals of the
problem, while minimizing their travel costs. We model these problems with \textit{Sat-Tsp} by capturing the transition costs of the robots in their environment as a set of discrete graphs (one for each robot) and the logic of the problem is encoded as a \textit{Sat} formula. The formula is used to dictate which locations are visited by which robots. In this way, we can have a location/task visited by multiple robots or just one robot. Additionally, the use of multiple graphs for multiple robots allows for heterogeneous robots (the transition costs can be different for each robot).

In the rest of this section, we introduce \textit{Sat-Tsp} and show how it is used to model a simple path planning example. Additionally, we classify \textit{Sat-Tsp}'s complexity and show that even when it is composed of easy to solve \textit{Sat} and \textit{Tsp} problems, the combination can still be hard.

A. \textit{Sat-Tsp} Definition

Before we define the decision version of \textit{Sat-Tsp} and its two optimization problems, we start with some notation. A \textit{Sat-Tsp} instance can take as input multiple graphs \( G_i \), each with a cost budget \( c_i \). To simplify the notation we sometimes absorb the cost budget \( c_i \) into the graph’s tuple \( G_i = (V_i, E_i, w_i, c_i) \) (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.

**Definition III.1 (\textit{Sat-Tsp}).** The \textit{Sat-Tsp} decision problem takes as input \((G_1, G_2, \ldots, G_n, F, C)\), where:

- \( G_i = (V_i, E_i, w_i, c_i) \) is a directed, weighted graph with edge weights \( w_i : E_i \rightarrow \mathbb{R}_{\geq 0} \) 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 \( \sum_{i=1}^n c'_i \leq C \),
- 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 \( v \in V'_1 \cup V'_2 \cup \ldots \cup V'_n \).

**Note III.2.** The above definition implies that there are two main types of variables, vertex variables and auxiliary variables. Vertex variables are used to indicate if a location is visited or not and auxiliary variables are used to construct the logic of the problem into a concise formula (all problems presented in this paper utilize auxiliary variables). Vertex variables share the same labels as the vertices to emphasize the link between the variable and vertex, where auxiliary variables take on arbitrarily labels such as \( x_i \).

In this paper we consider two optimization problems for \textit{Sat-Tsp}: 1) minimize the total cost budget (minimize \( C \)) and 2) minimize the maximum cost budget of any graph \( G_i \) (minimize \( \max c_i \)).

Furthermore, the instances studied in this paper can be expressed as metric-\textit{Sat-Tsp} instances.

**Definition III.3 (Metric-Sat-Tsp).** A metric-\textit{Sat-Tsp} instance is one where each input graph \( G = (V, E, w) \) is complete and the edges satisfy the triangle inequality: \( w(v_i, v_k) \leq w(v_i, v_j) + w(v_j, v_k) \) for all \( v_i, v_j, v_k \in V \) such that \( v_i \neq v_k \).

Note that robotic roadmaps are typically metric, as the time and/or distance to transition from a configuration \( A \) to another configuration \( C \) is less than or equal to that of transitioning from \( A \) to an intermediate point \( B \), followed by transitioning from \( B \) to \( C \). Additionally, if the roadmap is missing edges (not complete) then one can fill in the missing edges with shortest paths. For example if \( \langle v_i, v_j \rangle \notin E \) then we can add \( \langle v_i, v_j \rangle \) to \( E \) and set \( w(v_i, v_j) \) to the cost of the shortest path between \( v_i \) and \( v_j \) in the roadmap.

**Example III.4 (Modeling 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 \textit{Sat-Tsp} instance by first creating two graphs (i.e., roadmaps), \( G_1 = (V_1, E_1, w_1, c_1) \) and \( G_2 = (V_2, E_2, w_2, c_2) \) to capture the transition costs 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 time for the corresponding robot to travel from location \( i \) to location \( j \). If the distance from \( i \) to \( j \) is \( d_{ij} \), then the weight for robot 1 is \( w_1(i, j) = d_{ij}/2 \) and the weight for robot 2 is \( w_2(i, j) = d_{ij} \). Now the tuple \( \langle V_1, E_1, w_1, 600 \rangle \) captures the transition system for robot 1 and its battery budget, similarly tuple \( \langle V_2, E_2, w_2, 720 \rangle \) for robot 2.

Let \( R = \{1, 2\} \) be the set of robots. Then we construct the formula \( F \) using the set of variables \( X = \{v_{ir} | i \in L, r \in R\} \) to represent if vertex \( v_i \in G_r \) is in the solution or not (true or false). We start by adding the set of clauses \( (v_{i,1} \lor \neg v_{i,2}) \) 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 \( (\neg v_{i,1} \lor \neg v_{i,2}) \) 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 \( C \), to be any value \( \geq 1320 \), 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 \( C \).

**Remark III.5 (Modeling Path Planning Problems).** \textit{Sat-Tsp} as a modeling language, is most suited for expressing planning constraints as a \textit{Sat} formula and motion transitions as one or more \textit{Tsp}s. Theoretically, like \textit{Ilp}, \textit{Sat-Tsp} can efficiently model any decision problem in \textit{NP} but practically there are limitations. For example, it is arguably cumbersome to use \textit{Sat-Tsp} for counting constraints (e.g., visit five red locations). This is due to the simple structure that \textit{Sat} uses for expressing logic. However, this structure has led to the success of modern \textit{Sat} and \textit{Smt} solvers [18], [46], [45], which \textit{cnTsp} takes advantage of. Furthermore, we demonstrate in Section VI and VII that \textit{Sat-Tsp} can be successfully used for small counting constraints.
B. Complexity of Sat-Tsp

The decision version of Sat-Tsp is \(\text{NP}\)-complete. This follows from the fact that \(\text{Sat}\) (an \(\text{NP}\)-hard problem) reduces to Sat-Tsp and a Sat-Tsp solution can be verified in polynomial time.

We also classify the complexity of Sat-Tsp when it is composed of easy Sat and Tsp problems. Let \(\text{Sat}^* \subseteq \text{Sat}\) and \(\text{Tsp}^* \subseteq \text{Tsp}\) be the set of problem instances that are solvable in polynomial time. An example of a Tsp instance that is easy to solve (in \(\text{Tsp}^*\)) is an instance with a complete 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 [47]. Similarly, Sat solvers are quite efficient on practical Sat problems, such as instances in the Sat library [48]. So does this mean that if our Sat-Tsp problem is composed of easy instances in Sat* and Tsp*, then it is easy to solve?

**Theorem III.6.** Consider the subset of Sat-Tsp problems composed of instances from Sat* and Tsp*, then this subset of Sat-Tsp remains \(\text{NP}\)-complete.

**Proof.** We prove the above result by reducing a \(\text{NP}\)-complete problem to a Sat-Tsp problem composed of Sat* and Tsp* problems. Specifically, we do this for the Set-Cover problem. This problem takes in as input \((U, S, C)\), where \(U\) is a universe of finite elements (a set), \(S\) is a collection of sets, for which each set \(S_i \subseteq U\) contains a subset of the elements from \(U\) \((S_i \subseteq U)\), and \(C\) is a cost budget, then a solution is a subset \(S' \subseteq U\) that covers all of the elements in \(U\) with \(S'\) and \(|S'| \leq C\). The reduction maps the sets \(S_i \subseteq U\) to vertices \(v_i\) in the complete graph \(G = (V, E, w)\), 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 \((G, F, C)\) (\(C\) 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 \(\text{NP}\)-hard, then it must be the case that Sat-Tsp remains \(\text{NP}\)-complete despite the fact that the Sat and Tsp problems are in Sat* and Tsp* respectively.

Theorem III.6 proves that Sat-Tsp is \(\text{NP}\)-hard even when its sub-problems are easy. Thus, we cannot expect to construct a polynomial time solver for Sat-Tsp even when we have access to polynomial time solvers for its subproblems (unless \(\text{P} = \text{NP}\)). One could speculate that a similar consequence of the theorem would be that a good Sat-Tsp solver would require more sophistication than a naive combination of a Sat and Tsp solver, which would be to solve the Sat instance then solve the Tsp instance.

In the next section we start by exploring the Brute solver, which naively solves the Sat and Tsp instances separately. The result is that it explores every Sat solution (possibly an exponential number of solutions). The \(\text{cbTsp}\) solver improves upon the Brute solver by adding the ability to negate partial solutions and leverages the sophistication of the Sat solver's branch heuristics. The ability to negate partial solutions allows \(\text{cbTsp}\) to more effectively prune the search space and the branch heuristics prioritizes the assignment of variables that are involved in the most number of conflicts.

IV. 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. Then we provide a high-level description of \(\text{cbTsp}\) and the conditions under which it can be used.

A. 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 sequence of the Brute solver is as follows. First it uses a Sat solver to find a feasible set of included vertices \(V'\) (Lines 2 and 3). Next, it uses the 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 to prohibit it 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\((G, F, C)\)**

1. while SATISFIABLE\((F)\) do
2. \(M' \leftarrow \text{Solve}(F)\)
3. \(V' \leftarrow \{v \in V | v^* \in M'\}\)
4. \(G' \leftarrow \text{Subgraph}(G, V')\)
5. \(\langle p', c' \rangle \leftarrow \text{Tsp-Solve}(G', C)\)
6. if \(c' \leq C\) then
7. return \(\langle M', p', c' \rangle\)
8. else
9. \(f' \leftarrow (\bigwedge_{v \in V', v} v) \land (\bigwedge_{v \in V \setminus V', \neg v})\)
10. \(F \leftarrow F \land \neg f'\)
11. return \(\emptyset\)

If we were solving metric instances, a less naively 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 \( \text{ctTsp} \) approach works in this way, but it is also able to negate partial solutions. In fact the \( \text{ctTsp} \) solver’s parameters (found in Appendix B) 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 \( \text{ctTsp} \) is a sophisticated extension of the \( \text{Brute} \) approach.

**B. The \( \text{ctTsp} \) Solver**

The \( \text{ctTsp} \) solver builds on the \( \text{Brute} \) approach by using partial solutions to prune the search space. This allows for significant computation savings. We begin by casting the \( \text{SAT-Tsp} \) problem in the \( \text{SMT} \) framework. We use the \( \text{DPLL(T)} \) algorithm (Algorithm 2) to couple a \( \text{SAT} \) and \( \text{Tsp} \) solver. The \( \text{ctTsp} \) solver only works for \( \text{Tsp} \)-monotonic instances (Definition IV.2), which essentially means that the cost of partial solutions cannot be reduced by adding vertices. \( \text{Tsp} \)-monotonicity is a necessary property needed to prune partial solutions. Additionally, metric \( \text{SAT-Tsp} \) instances are \( \text{Tsp} \)-monotonic (Theorem IV.6).

**Definition IV.1 (Partial Solution).** Given a \( \text{SAT-Tsp} \) 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 the corresponding variables in \( M \) are true.

**Definition IV.2 (Tsp-Monotonicity).** A graph \( G = \langle V, E, w \rangle \) is \( \text{Tsp} \)-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 \( \text{Tsp} \) costs \( c_1 \leq c_2 \).

The \( \text{Tsp} \)-monotonic property allows for the negations of partial solutions that exceed the cost budget(s). Since 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 reoccuring (the negation details are given in Algorithm 3, Lines 6 and 9).

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

**Definition IV.3.** The \( \text{Tsp-Theory} \) takes as input the tuple \( \langle G_1, G_2, \ldots, G_n, C \rangle \) and \( M \), 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 \( \text{Smt} \) solver.
- Each \( G_i = \langle V_i, E_i, w_i, c_i \rangle \) is a \( \text{Tsp} \)-monotonic graph with 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 \( (x_{\text{tsp}} = 1) \) if and only if

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

In the \( \text{Smt} \) formulation, the \( x_{\text{tsp}} \) predicate is forced to be true, thus all solutions and partial solutions must not exceed the cost budget(s). If the \( \text{Tsp} \) theory finds that there is no solution, then a learnt clause is constructed (Lines 6 and 9 of Algorithm 3) and added back to the formula \( F \) (Line 6 of Algorithm 2). The \( \text{DPLL(T)} \) solver is subsequently tasked with solving the new formula (which includes the learnt clause).

At a high-level, \( \text{ctTsp} \) solves decision instances as follows:

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

The optimization version of \( \text{ctTsp} \) uses binary search to find a solution with the minimum cost budget (minimize \( C \) or minimize the maximum \( c_i \)) by solving a series of \( \text{SAT-Tsp} \) decision instances with different cost budgets. Details of the binary search algorithm is found in Appendix A.

**Algorithm 2: Overview of \( \text{DPLL(T)} \) on \( F \)**

**precondition: \( \text{Tsp-Theory} \).setup\( (G_1, \ldots, G_n, C) \)**

1. \( M \leftarrow \emptyset \)
2. \( \text{while } \exists x \in X \text{ s.t. } \{x^T, x^F\} \cap M = \emptyset \text{ do} \)
3. \( \text{Add a new variable assignment to } M \)
4. \( f_{\text{conflict}} \leftarrow \text{Tsp-Theory}(M) \)
5. \( \text{if } f_{\text{conflict}} \neq \emptyset \text{ then} \)
6. \( F \leftarrow F \land f_{\text{conflict}} \)
7. \( \text{Backtrack } M \text{ to some point that does not conflict with } F \)
8. \( \text{if } M \text{ solves } F \text{ then} \)
9. \( \text{return } M \)
10. \( \text{else} \)
11. \( \text{return } \emptyset \)

**Note IV.4.** Algorithm 2 is a simplified version of the \( \text{DPLL(T)} \) logic used in \( \text{ctTsp} \). The real algorithm is more sophisticated than what is depicted. For example, in addition to keeping track of \( M \), we also keep track of the solution paths and their costs, and we only call the \( \text{Tsp} \) theory if \( M \) contains a change to the set of included vertices.

**Example IV.5 (Illustration of the \( \text{ctTsp} \) Approach).** This example demonstrates the interaction between the \( \text{SAT} \) solver (based on \( \text{DPLL} \)) and the \( \text{Tsp} \) solver (the \( \text{Tsp} \) theory) in
Algorithm 3: Overview of Tsp-Theories (M)

1. for each graph $G_i$ do
2.   $V_i' = \{v_j \in V_i | v_j \in M \}$
3.   $G_i' \leftarrow \text{InducedGraph}(G_i, V_i')$
4.   $c_i' \leftarrow \text{TspSolve}(G_i', c_i)$
5.   if $c_i' > c_i$ then
6.      return $\text{¬}\left(\bigwedge_{v_j \in V_i} v_j\right)$
7.   if $\sum c_i' > C$ then
8.      $V' = V_1 \cup \ldots \cup V_n'$
9.      return $\text{¬}\left(\bigwedge_{v_j \in V} v_j\right)$
10. return $\emptyset$

$	ext{cnTsp}$ Suppose we have one input graph $G = (V, E, w, c)$ and suppose the solver has constructed a partial solution $M = \{x_{v_0}^0, v_1^1, v_2^1, x_{v_3}^1, v_4^1, v_5^1\}$. Then suppose that the $\text{S-A-T}$ solver extends the partial solution by assigning $v_0$ to be true. Once the assignment $v_0^1$ is added to $M$, the $\text{Tsp}$ theory solver is called to make a consistency check. The $\text{Tsp}$ theory uses $M$ to construct the $\text{Tsp}$ problem $G'$ for the set of included vertices $v_1, v_4, v_5$ and $v_0$. Note $x_{v_0}^0$ and $x_{v_3}^1$ are not vertex variables (they are auxiliary variables) and so they do not appear in this set. The $\text{Tsp}$ theory then calls the $\text{Tsp}$ solver with input $(G', c)$ and if a tour is found within the budget the theory returns true (consistent). The DPLL solver ($\text{S-A-T}$ solver) continues to build upon the partial solution. If no tour is found within the budget the check is inconsistent and the learnt clause $f_{\text{conflict}} = \text{¬}(v_1 \land v_4 \land v_5 \land v_6)$ is constructed to be added to $F$ so that the $\text{S-A-T}$ solver avoids this solution in the future. The $\text{S-A-T}$ solver then backtracks (revert some of the partial solution) to avoid the inconsistency and searches for a new solution that satisfies $F \land f_{\text{conflict}}$.

C. Correctness of $\text{cnTsp}$

In this section we show that metric instances are $\text{Tsp}$-monotonic (Definition IV.2) and prove that $\text{cnTsp}$ yields the correct solution for $\text{Tsp}$-monotonic instances.

Theorem IV.6. Metric graphs are $\text{Tsp}$-monotonic.

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 $\text{Tsp}$ 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 $(v_i, v_j)$ 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 $w'_i(k, j) = w_2(i, j) + w_2(j, k)$. 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 $\text{Tsp}$ cost of the new graph. Consequently, the $\text{Tsp}$ costs $c_1$ and $c_2$ for $V_1$ and $V_2$ satisfies $c_1 \leq c_2$. Therefore metric graphs are $\text{Tsp}$-monotonic.

Theorem IV.7. The $\text{cnTsp}$ approach is correct (i.e., sound and complete) on $\text{Tsp}$-monotonic decision instances.

Proof. We first prove that the $\text{Smt}$ formulations used by $\text{cnTsp}$ are sound and then we prove that the $\text{Smt}$ solver approach (DPLL(T) on $F$) used by $\text{cnTsp}$ is sound and complete.

The soundness of the $\text{Smt}$ formulation follows from the definition of $\text{S-A-T}$ (Section III) and Definition IV.2. Specifically, the $\text{Smt}$ formulation allows for the negation of partial solution vertex sets and supersets for induced subgraphs that exceed the $\text{Tsp}$ budget(s). This does not remove any solutions in the search space that could have $\text{Tsp}$ costs within the budget, since the graph(s) are $\text{Tsp}$-monotonic. Furthermore, a full $\text{Smt}$ solution consists of $\text{Tsp}$ tour(s) of the included vertices and a satisfying assignment of $F$, which is a solution to the $\text{S-A-T}$ instance. Therefore, the $\text{Smt}$ formulation is sound, since it has the same solution set as the $\text{S-A-T}$ formulation.

It follows that the solver approach for $\text{Smt}$ instances is sound and complete, since it is based on the sound and complete algorithm DPLL(T) [45]. Therefore the $\text{cnTsp}$ approach is correct on $\text{Tsp}$-monotonic instances.

D. Relaxing $\text{Tsp}$-Monotonicity

In this section we expand on the class of instances solvable by $\text{cnTsp}$ ($\text{Tsp}$-monotonic instances). Specifically, we describe a relaxation of $\text{Tsp}$-monotonicity that can be used in practice with $\text{cnTsp}$ when we have prior knowledge of a vertex set $A \subset V$ that must be included in the solution. The knowledge of this set allows us to relax the $\text{Tsp}$-monotonicity property to apply to sets $V_1 \supseteq A$. The $\text{cnTsp}$ 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 $F$ to be $F = F \land v_s \land v_g$. This example is not $\text{Tsp}$-monotonic. However, the addition to the formula is so simple that the $\text{cnTsp}$ 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 $\text{cnTsp}$ checks will satisfy the $\text{Tsp}$-monotonicity property, since the graph is otherwise metric (preprocessing happens before any calls to the $\text{Tsp}$ theory).

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

$$F \land \left(\bigcup_{v \in A} v\right)$$

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 $\text{cnTsp}$ (before the DPLL solver is called, which is not discussed in this paper). This happens for atomic clauses (clauses with one literal), e.g., given the clause $(v)$, the preprocessing phase would assign a true value to $v$ so that the clause is satisfied.

Definition IV.8 (A-Tsp-Monotonicity). Given a graph $G = (V, E, w)$ and a subset $A \subset V$, then $G$ is $A$-Tsp-monotonic if...
Fig. 2: A-Tsp-monotonic example, where the black locations represent the set $A$. Interconnecting edges to or from a graph $G_i$ represent a collection of edges that connect every vertex in the source to every vertex in the destination. The edge weights of the interconnecting edges satisfy the triangle inequality.

for any vertex subsets $V_1 \subseteq V_2 \subseteq V$ such that $A \subseteq V_1$, the induced subgraphs $G_1$ and $G_2$ have Tsp costs $c_1 \leq c_2$.

**Proposition IV.9.** The $\text{cSTsp}$ approach is correct on A-Tsp-monotonic decision instances if the decision to include the vertices in $A$ is done so in the preprocessing phase of the solver (before DPLL is called).

**Proof.** The proof follows Theorem IV.7’s proof — the instance behaves like a Tsp-monotonic instance once DPLL is called.

**Remark IV.10** (A Sufficient Construction for Achieving A-Tsp-Monotonicity). Given a graph $G = \langle V, E, w \rangle$ the problem is A-Tsp-monotonic for a set $A$ if the following is true for every pair of edges $\langle v_i, v_j \rangle, \langle v_j, v_k \rangle \in E$ such that $v_i, v_k \in V$ and $v_j \in V \setminus A$:

1) the edge $\langle v_i, v_k \rangle$ must also be in $E$,
2) and $w(v_i, v_k) \leq w(v_i, v_j) + w(v_j, v_k)$.

Notice that our motivating example with $G, v_s$, and $v_g$ satisfies the above construction for $A = \{v_s, v_g\}$. However, if we were to remove the edge $\langle v_s, v_g \rangle$, then the instance would not satisfy the above construction and it would not be A-Tsp-monotonic. This is due to the fact that the infeasible partial solution of $v_s$ and $v_g$ can be made feasible by adding any additional vertex to the solution.

Figure 2 illustrates a more complex example that was constructed using Remark IV.10. Here a set of metric subgraphs $G_1, G_2, \ldots, G_n$ are connected together with a ring graph to dictate the order that the metric graphs are to be traversed. The ring graph is constructed from vertices in $A$ (the black vertices) and the metric subgraphs are connect to $A$ in between adjacent vertices in the ring.

**V. An Integer Program Formulation for Sat-Tsp**

In this paper 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 (i.e., robotic roadmap). This section describes the standard method for expressing such problems as an integer linear program. 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 [49] and is a standard method used in the operations research community [49].

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 $e_{i,j}^k \in \{0, 1\}$ represent the inclusion or exclusion of the edge $\langle v_i, v_j \rangle \in E_k$ from the solution ($e_{i,j}^k = 1$ indicates inclusion). Similarly let $v_i^k$ 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 $v_i^k$ 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 $\sum_{k=1}^n \sum_{i=1}^{\vert V^k \vert} \sum_{j=1}^{\vert V^k \vert} w_k(v_i, v_j) e_{i,j}^k$. minimize

$$c_{\text{max}}$$

subject to

$$\sum_{k=1}^n \sum_{i=1}^{\vert V^k \vert} \sum_{j=1}^{\vert V^k \vert} w_k(v_i, v_j) e_{i,j}^k \leq C$$

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

$$v_i^k = 1,$$

$$\sum_{i=1}^{\vert V^k \vert} e_{i,j}^k = v_j^k, \text{ for each } v_j \in V_k$$

$$\sum_{j=1}^{\vert V^k \vert} e_{i,j}^k = v_i^k, \text{ for each } v_i \in V_k$$

$$\sum_{i \in P} e_{i,j}^k \leq \sum_{i \in P} v_i^k - v_i^l,$$

for each $P \subseteq \{2, \ldots, \vert V^k \vert\}$, and some $l \in P$

$$\sum_{i=1}^{\vert V^k \vert} \sum_{j=1}^{\vert V^k \vert} w_k(v_i, v_j) e_{i,j}^k \leq \min(c_k, c_{\text{max}})$$

Constraint (2) ensures the total solution cost is within the budget $C$. Constraint (3) forces the solution to include the home vertices. Constraints (4) and (5) restrict the incoming and outgoing degree for each location to be one only if the vertex is included. Constraint (6) is the sub-tour elimination constraint, where the sub-tour is represented by the set $P$, which can not include the home location. The sub-tour is eliminated by ensuring that the number of edges between vertices in $P$ is less than $\vert P \vert$ (a sub-tour would have $\vert P \vert$ edges). If one or more locations are not visited in $P$, i.e., $P$ is not a sub-tour, then the inequality is satisfied by the fact that each included vertex has at most one incoming and one outgoing edge (a strict subset of $P$ can have at most $\vert P \vert - 1$ edges). In practice the sub-tour elimination constraint is lazily implemented (if a constraint in (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 (7) ensures the individual tours are within the budgets $c_k$ and it
VI. 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 (also known as mutual exclusivity). We define this type of constraint here and use it throughout the rest of the section.

Definition VI.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:

\[
\bigwedge_{x,y \in X'|x \neq y} \neg(x \land y).
\]

A. 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 [50], [51]. Our patrolling problem is defined on a metric environment that contains \( m \) points of interest, \( n-1 \) “observation 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 \( v_i \) is complementary to \( v_j \) 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 (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 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 \((v_h)\) the robot’s home) and the set of clauses

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

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 V. Let the set of binary variables \( \hat{v}_{p,i} \in \{0,1\} \) 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.

\[
\text{for each } p \in \{1,2,\ldots,m\}:
\begin{align*}
\hat{v}_{p,i} &\leq v_i, \text{ for each } v_i \in V_p \quad (8) \\
\hat{v}_{p,i} &\leq \sum_{v_j \in V_{p,i}} v_j, \text{ for each } v_i \in V_p \quad (9) \\
\sum_{v_i \in V_p} \hat{v}_{p,i} &\geq 1 \quad (10)
\end{align*}
\]

Constraint (8) restricts the indicator \( \hat{v}_{p,i} \) from being true \((\hat{v}_{p,i} = 1)\) if the location \( v_i \) itself is not visited. Constraint (9) restricts the indicator from being true if none of the complementary locations of \( v_i \) are visited, and Constraint (10) ensures that at least one complementary pair is visited.

B. The Sample Collection Problem

The following problem is inspired by sample collection problems that arise for science rovers, as described in [6]. 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 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 (on the right). In this example there are no samples with mineral 2 and so a feasible solutions do not collect mineral 2. Additionally, we demonstrate a sample collection solution in our supplementary video. 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 that exist in the environment.

**SAT-TSp expression:** The problem is encoded as a SAT-TSp instance by constructing a graph \( G_r \) for each robot \( r \in R \) 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 with incompatible location/robot pairs to simplify the SAT-TSp expression for this paper.

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 and large sample) \( \bigwedge_{i,r} \neg v_{i,r} \), where \( X_L = \{v_{i,r}|v_i \in V_L, r \in R_S\} \cup \{v_{i,r}|v_i \in V_S, r \in R_L\} \). We encode the goal of collecting at least one of each mineral (if the mineral exists), with a disjunctive clause for each mineral

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

To restrict multiple robots from visiting the same location \( v_i \), we use at-most-one-in-a-set constraints (Definition VI.1) for the sets \( \{v_{i,r}|r \in R\} \) 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 \( r \in R_L \).

The tuple \( \langle G_1, G_2, \ldots, G_R, F, C \rangle \) now encodes the problem as a SAT-TSp instance, where \( G_r = (V_r, E_r, w_r, c_r) \) 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 V. 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 V). The following is the extension of the ILP formulation:

\[
\sum_{v_i \in V_t} v_{i,r} \geq 1, \text{ for each } t \in \{1, 2, \ldots, m\} \quad (8)
\]
\[
\sum_{r \in R} v_{i,r} \leq 1, \text{ for each } v_i \in V \quad (9)
\]
\[
\sum_{v_i \in V_M, r \in R} v_{i,r} \leq 1, \text{ for each } r \in R_S \quad (10)
\]
\[
\sum_{v_i \in V_L, r \in R} v_{i,r} \leq 1, \text{ for each } r \in R_L \quad (11)
\]
\[
v_{i,r} = 0, \text{ for each } v_i \in I \quad (12)
\]

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 (8) encodes the goal of retrieving at least one of each mineral type, Constraint (9) restricts multiple robots from retrieving the same sample, Constraint (10) restricts small robots from collecting more than one medium sample, Constraint (11) restricts large robots from collecting more than one large sample, and Constraint (12) negates the incompatible location/robot pairs.

**C. The Period Routing Problem**

The period routing problem [9], [11] requires a robot 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 allowable. For this paper, all period combinations that avoid back to back visits are allowed (the first and last period are considered back to back). This problem arises in collection [11] and delivery [9] 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

---

1See supplementary video attachment.
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 [52], which are efficiently translated to a Boolean formula [53] and added to $F$. A brief overview of how adder circuits are used within this context can be found in Appendix C. 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. Next, the outputs of the adder circuit are forced to the capture the binary encoding of how many of the inputs are true. The back to back period restriction is exhaustively 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 $F \leftarrow F \land \neg b_{i,1} \land b_{i,2} \land \neg b_{i,4} \land \ldots$.

We force the robot to visit the home locations with the clause $\land_{p \in P} v_{h,p}$, 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.

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

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

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 V. The following are the additional constraints in the ILP formulation,

$$\sum_{p \in P} v^p_i = f(v_i), \text{ for each } v_i \in V \quad (6)$$

$$v^p_i + v^{p+}_i \leq 1, \text{ for each } v_i \in V \text{ and } p \in P \quad (7)$$

where $p^+ = (p + 1) \mod m$. Constraint (6) encodes the goal of servicing each location the proper number of times and Constraint (7) restricts back to back visits.

## VII. Simulation Results

In this section we present simulation results that compare cbTSP to an ILP solver on the instances presented in Section IV. Additionally, we explore the scalability of SAT-TSP with respect to the number of robots. The problem instances in this section all have metric travel costs and are thus solvable by cbTSP (metric instances are TSP-monotonic).

### A. 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 [18]) to test partial solutions by making external callbacks to a TSP solver (a version of LKH [19] 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 best solution found within the time budget is returned. The solver parameters are used to configure the behavior of cbTSP’s SAT and TSP solvers as well as a few cbTSP specific behaviors. The details of these parameters and their values are given in Appendix B. The ILP solver, Gurobi [17], was accessed through Python and 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 was 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 solver’s results as they 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 overnight (or longer) to completion. In the latter case of running the solvers to completion, comparing solution quality would be meaningless since both approaches would find optimal solutions. Thus, using a fixed time budget allows us to compare the solver’s ability to find solutions quickly. Typically, both solvers consume their entire time budget and so we do not explicitly report solver times as they would all be 300 seconds. Instead we compare the average solution quality found over ten separate runs for each solver with tables and track the solution quality found over time in Figure 5. The time budget does not include the time taken to create the SAT-TSP and ILP instances (reduction time) as we do not wish to benchmark this process. However, the reduction times are polynomial with respect to the input size.

<table>
<thead>
<tr>
<th>No.</th>
<th>n</th>
<th>m</th>
<th>Best</th>
<th>Avg. Cost</th>
<th>Best</th>
<th>Avg. Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>40</td>
<td>5</td>
<td>2.268</td>
<td>2.292</td>
<td>2.260</td>
<td>2.260</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td>10</td>
<td>1.812</td>
<td>1.812</td>
<td>1.812</td>
<td>1.812</td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td>20</td>
<td>3.171</td>
<td>3.171</td>
<td>3.111</td>
<td>3.111</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td>7</td>
<td>1.825</td>
<td>1.925</td>
<td>1.801</td>
<td>1.801</td>
</tr>
<tr>
<td>5</td>
<td>60</td>
<td>15</td>
<td>2.615</td>
<td>2.845</td>
<td>2.430</td>
<td>2.430</td>
</tr>
<tr>
<td>6</td>
<td>60</td>
<td>30</td>
<td>3.252</td>
<td>3.346</td>
<td>3.176</td>
<td>3.176</td>
</tr>
<tr>
<td>7</td>
<td>80</td>
<td>10</td>
<td>2.020</td>
<td>2.227</td>
<td>1.903</td>
<td>1.903</td>
</tr>
<tr>
<td>8</td>
<td>80</td>
<td>20</td>
<td>2.948</td>
<td>4.681</td>
<td>2.553</td>
<td>2.602</td>
</tr>
<tr>
<td>9</td>
<td>80</td>
<td>40</td>
<td>4.640</td>
<td>6.920</td>
<td>3.244</td>
<td>3.585</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>12</td>
<td>3.259</td>
<td>3.265</td>
<td>1.937</td>
<td>1.939</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>25</td>
<td>2.825</td>
<td>4.324</td>
<td>2.431</td>
<td>2.473</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>50</td>
<td>7.385</td>
<td>8.273</td>
<td>3.803</td>
<td>4.104</td>
</tr>
</tbody>
</table>

TABLE I: 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 highlighted.
B. Patrolling Problem Setup and Results

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 x 1000 meter two dimensional square. A point of interest $p = (x_p, y_p)$ is observable from location $v = (x_v, y_v)$ if

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

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

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

An illustrative example of this problem is given in Figure 3.

We compared cbTSP to the ILP solver Gurobi, on a set of 12 instances as shown in Table I. As we can see cbTSP outperforms Gurobi on almost every instance and as the instances get more difficult (lower in the table) the performance gap increases as shown by the first and last instances in the table — Gurobi has an average cost of 1.01 times more than cbTSP on the first instance and an average cost of 2.01 on the last.

C. Sample Collection Setup and Results

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 x 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 2 m/s, while the large robots travel at half that speed (1 m/s).

Note VII.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 II. 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); then as the instances get more difficult (instances 4–7), cbTSP starts to outperform Gurobi; and for the most difficult instances (instances 8–12), 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 instances.

Physics-based simulations: We have also performed simulation experiments for sample collection in more complex environments (shown in Figure 1) 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 [20] to move within the physics simulator. An example simulation, visualized with RViz, is included as a supplementary video. 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.

D. Period Routing Setup and Results

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 x 1000 meter two dimensional square. There are six service periods and 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 4.

We have compared cbTSP to the ILP solver, Gurobi, on 12 period routing instances, for which the results can be found in Table III. The table shows that cbTSP outperforms Gurobi on the majority 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. The results in the table show that Gurobi starts out finding solutions that are close to the quality of cbTSP (1.08 times the cost) then degrade as the instances become more difficult to 1.69 times the cost found by cbTSP.

### E. Solution Quality as a Function of Runtime

Both cbTSP and Gurobi provide updates on the quality of the solutions found during run time. This is particularly useful for online robotic applications where the robot can decide to accept a solution that is good enough instead of waiting for a better solution.

Figure 5 compiles all of the solver data for all three libraries. From the data we can see that cbTSP is able to find more feasible solutions than Gurobi with less time. Additionally, we can see that the quality of these solutions quickly improve within the first 60 seconds, while the solutions found by Gurobi improve at a slower rate. One thing to note is that the solution quality data is an average of the solutions found by these approaches, so making a direct comparison of the two approaches can be misleading since the data for one solver may include an average of an instance that the other solver has not been able to solve yet. This accounts for the degradation of Gurobi’s solution quality at around 10 seconds. Here new solutions are added to the average that were not part of the average before and in the case that these new solutions have a larger normalized cost, the new results will degrade the overall average.

### F. Scaling with Number of Robots

In this section we explore scalability with the number of robots. We consider the patrolling problem as described in Section VI-A. To scale the problem we allow multiple robots to visit the same location and set the objective to minimize the maximum length robot tour. This is a natural alternative to minimizing the sum of tour lengths, and encourages all robots to be utilized in the solution (rather than having a single robot tour the observation locations). The formula is updated with the following set of clauses for each $v_i \in V$

$$v_i \iff (v_{i,1} \lor v_{i,2} \lor \ldots \lor v_{i,|R|})$$

To capture whether or not a location was visited by any robot.

The simulations range from 1 to 20 robots and the solver is allowed to run to completion. The environment is scaled as follows: the number of points of interest in the environment is allowed to run to completion. The environment is scaled as

$$n = 2212$$

and the number of observation locations is

$$V = 2212$$

with close to 20 robots. Note that the log scale in the plot highlights that the growth in runtime is not exponential even though the search space grows exponentially with $R$. Figure 6 plots the time needed to solve the instances as a function of the number of robots available. The runtimes range from a few seconds for instances with a single robot, to approximately 5,000 seconds for instances with close to 20 robots. The best results are highlighted. The boxes capture 50% of the trials, while the tails capture 100% of the trials. The curve shown for scale is $10|R|^2$.

### VIII. Conclusion and Discussion

We introduced the Sat-Tsp problem and the effective solver cbTSP, which enables us to express and solve discrete robotic
motion planning problems with complex constraints. We have demonstrated that cTsp often outperforms a commercial grade ILP solver, showing that cTsp is a good choice for the problems demonstrated in this paper. One can obtain the cTsp solver from https://github.com/fcimeson/cbTSP.

There are a number of directions we are pursuing for our future work. One direction is to implement some of the advanced techniques used by DPLL and DPLL(T), such as clause forgetting, propagations, and custom search heuristics. Although, we have not found memory blow up to be a problem, clause forgetting would help avoid this potential problem by forgetting conflict clauses that our method added to the formula but are rarely if ever part of a conflict. With propagations, we could implement custom SMax theories useful for planning such as location ordering constraints, which in turn would offload some of the logic from the SAT formula and put it into a more compact form that the custom SMax theory can solve. Currently cTsp uses search heuristics from the SAT solver to choose the order of variable assignments, which does not utilize knowledge of the Tsp problem. We believe we could use this knowledge to improve the search heuristic and thus improve the solver’s performance. Additionally, we plan to investigate the effectiveness of using SAT-Tsp for collision avoidance problems. Our approach would extend the theories useful for the new problem.

ACKNOWLEDGMENTS

This research is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

The cbTSP solver uses a modified version of binary search to find optimal solutions. Algorithm 4 shows this approach for optimization problems that aim to minimize the cost $C$. The $\text{bdiv}$ parameter is configured through the cbTSP interface (default setting is 10) and $\text{upper_bound}$ is an upper bound on the worst cost solution that can be set to $\sum_{i=1}^{n} \max_{(v_a,v_b)} \epsilon E_i w_i(v_a,v_b)|V_i|$. The same basic algorithm is used to find optimal solutions for problems that aim to minimize the maximum cost $c_i$ (subgraph path cost) by replacing $C$ with $c_1$ and for each $G_i = \{V_i, E_i, w_i, c_i\}$ replace $c_i$ with $c_j$ so that $c_j$ now constrains the maximum subgraph cost for each $G_i$.

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

1. $C^- \leftarrow 0$
2. $C^+ \leftarrow \text{upper_bound}$
3. $C^* \leftarrow \text{upper_bound}$
4. while $C^- < C^+$ do
   5. $C \leftarrow C^+ - \left\lfloor \frac{C^+ - C^-}{\text{bdiv}} \right\rfloor$
   6. $C' \leftarrow \text{cbTSP}(G_1, G_2, \ldots, G_n, F, C)$
   7. if $C' \geq 0$ then
      8. $C^- \leftarrow C' - 1$
   9. else
      10. $C^* \leftarrow C' + 1$
11. return $C^*$

### APPENDIX A

#### cbTSP: Binary Search

The cbTSP solver can take in a number of parameters to configure the TSP solver, the SAT solver and the cbTSP solver. This section details the cbTSP specific parameters as well as the default settings.

#### A. $\text{MiniSat}$

The configurable parameters of $\text{MiniSat}$ are the conflict budget and the propagation budget. Both parameters by default are unlimited. All other $\text{MiniSat}$ parameters are used as default and are non-configurable through the cbTSP interface.

#### B. LKH

The LKH parameters configured by cbTSP are:

- $\text{PROBLEM_FILE}$, $\text{TOUR_FILE}$, $\text{TIME_LIMIT}$, $\text{STOP_AT_MAX_COST}$, and $\text{MAX_COST}$. 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 that deviate from the LKH default settings are given in Table IV.

#### C. $\text{cbTSP}$

This section documents the configurable parameters for the cbTSP solver. Default values are given in parenthesis.

### APPENDIX B

#### cbTSP: Solver Parameters

The $\text{cbTSP}$ solver can take in a number of parameters to configure the TSP solver, the SAT solver and the cbTSP solver. This section details the cbTSP specific parameters as well as the default settings.

#### A. $\text{MiniSat}$

The configurable parameters of $\text{MiniSat}$ are the conflict budget and the propagation budget. Both parameters by default are unlimited. All other $\text{MiniSat}$ parameters are used as default and are non-configurable through the cbTSP interface.

#### B. LKH

The LKH parameters configured by cbTSP are:

- $\text{PROBLEM_FILE}$, $\text{TOUR_FILE}$, $\text{TIME_LIMIT}$, $\text{STOP_AT_MAX_COST}$, and $\text{MAX_COST}$. 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 that deviate from the LKH default settings are given in Table IV.

#### C. $\text{cbTSP}$

This section documents the configurable parameters for the cbTSP solver. Default values are given in parenthesis.

### APPENDIX B

#### cbTSP: Solver Parameters

The $\text{cbTSP}$ solver can take in a number of parameters to configure the TSP solver, the SAT solver and the cbTSP solver. This section details the cbTSP specific parameters as well as the default settings.

#### A. $\text{MiniSat}$

The configurable parameters of $\text{MiniSat}$ are the conflict budget and the propagation budget. Both parameters by default are unlimited. All other $\text{MiniSat}$ parameters are used as default and are non-configurable through the cbTSP interface.

#### B. LKH

The LKH parameters configured by cbTSP are:

- $\text{PROBLEM_FILE}$, $\text{TOUR_FILE}$, $\text{TIME_LIMIT}$, $\text{STOP_AT_MAX_COST}$, and $\text{MAX_COST}$. 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 that deviate from the LKH default settings are given in Table IV.

#### C. $\text{cbTSP}$

This section documents the configurable parameters for the cbTSP solver. Default values are given in parenthesis.

### APPENDIX B

#### cbTSP: Solver Parameters

The $\text{cbTSP}$ solver can take in a number of parameters to configure the TSP solver, the SAT solver and the cbTSP solver. This section details the cbTSP specific parameters as well as the default settings.

#### A. $\text{MiniSat}$

The configurable parameters of $\text{MiniSat}$ are the conflict budget and the propagation budget. Both parameters by default are unlimited. All other $\text{MiniSat}$ parameters are used as default and are non-configurable through the cbTSP interface.

#### B. LKH

The LKH parameters configured by cbTSP are:

- $\text{PROBLEM_FILE}$, $\text{TOUR_FILE}$, $\text{TIME_LIMIT}$, $\text{STOP_AT_MAX_COST}$, and $\text{MAX_COST}$. 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 that deviate from the LKH default settings are given in Table IV.

#### C. $\text{cbTSP}$

This section documents the configurable parameters for the cbTSP solver. Default values are given in parenthesis.

### APPENDIX B

#### cbTSP: Solver Parameters

The $\text{cbTSP}$ solver can take in a number of parameters to configure the TSP solver, the SAT solver and the cbTSP solver. This section details the cbTSP specific parameters as well as the default settings.

#### A. $\text{MiniSat}$

The configurable parameters of $\text{MiniSat}$ are the conflict budget and the propagation budget. Both parameters by default are unlimited. All other $\text{MiniSat}$ parameters are used as default and are non-configurable through the cbTSP interface.

#### B. LKH

The LKH parameters configured by cbTSP are:

- $\text{PROBLEM_FILE}$, $\text{TOUR_FILE}$, $\text{TIME_LIMIT}$, $\text{STOP_AT_MAX_COST}$, and $\text{MAX_COST}$. 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 that deviate from the LKH default settings are given in Table IV.

#### C. $\text{cbTSP}$

This section documents the configurable parameters for the cbTSP solver. Default values are given in parenthesis.
These carry over bits are used as input in the next level of adder circuit. XOR gate. AND gate.

2-Bit Adder circuits

in Table V. 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 VII. As we can see from the Table, a value of \texttt{cb\_interval}=1 has the best performance. One thing to note from the results, is how the non-naïve \texttt{Brute} approach (\texttt{cb\_interval}=999) fails 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.

\textbf{Conflicts (nConflicts=-1):} The \texttt{cnTsp} solver adds conflict clauses back to the sat formula to narrow down the search space, as described in Section IV. The number of conflict clauses \texttt{cnTsp} adds back to the formula can be exponential in size, thus the parameter \texttt{nConflicts=x} limits the number of additional clauses to \( x \) (1 for unlimited) and once that number is reached, it returns with the best known solution. One can check the output text of the solver to confirm whether or not the budget was reached.

\textbf{Query budget (max\_query\_time=-1):} The \texttt{cbTsp} solver searches for solutions with a series of \texttt{SxTsp} decision queries, where each search has a different cost budget set by either a binary or linear search algorithm. By default (-1) the queries are given an unlimited amount of time but if this parameter has a positive value of \( x \) then each query is terminated after \( x \) seconds and is assumed to be unsatisfiable.

\textbf{Search method (search\_method=binary):} This parameter is used to choose between binary and linear search.

\textbf{The binary search parameter (bdiv=10):} This parameter configures the division size of the binary search as shown in Algorithm 4. Due to the fact that most unsatisfiable instances are harder to prove than satisfiable instances, it is desirable to have this parameter larger than the nominal value of two. The default settings for the binary search parameter \texttt{bdiv} was chosen after comparing different values of \texttt{bdiv} (and comparing it to a linear search) on a set of small experiments documented in Table VI. 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 VII. As we can see from the Table, a value of \texttt{bdiv}=10 yields a good result.

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|c|c|}
\hline
\textbf{instance/cb\_interval} & \multicolumn{4}{c|}{\textbf{Avg. Solver Cost}} \\
\hline
& 1 & 2 & 10 & \textbf{BRUTE} \\
\hline
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 \\
\hline
\end{tabular}
\caption{Tuning experiments for different values of the \texttt{cbTsp} parameter, \texttt{cb\_interval} (\texttt{cbTsp} becomes \texttt{BRUTE} when \texttt{cb\_interval} > |\texttt{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 VII. The best results are highlighted.
}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|c|c|}
\hline
\textbf{instance/bdiv} & \multicolumn{4}{c|}{\textbf{Avg. Solver Cost}} \\
\hline
& 2 & 10 & 20 & \textbf{Linear} \\
\hline
patrolling06 & 3177 & 3177 & 3177 & 3177 \\
patrolling10 & 3653 & 3442 & 3496 & 3972 \\
sample04 & 1899 & 1883 & 1882 & 2291 \\
sample08 & 12993 & 1387 & 4071 & - \\
period11 & 2925 & 2804 & 2937 & 3578 \\
period12 & 2973 & 2669 & 2708 & 3148 \\
\hline
\end{tabular}
\caption{Tuning experiments for different values of the \texttt{cbTsp} parameter \texttt{bdiv} (the search is linear is when \texttt{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 VII. The best results are highlighted.
}
\end{table}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{adder_circuit.png}
\caption{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 all the carry out bits.}
\end{figure}

In this section, we provide an overview of how adder circuits are used to count the number of true variables in a set \( X' \). In general, we can input the negation of a variable (its negative literal) but for this document, we only count variable inputs if they are assigned true. Adder circuits are used in electronics to do rudimentary mathematical operations [52]. The example circuit shown in Figure 7 takes as input \( X' = \{x_1, x_2, \ldots, x_5\} \) and outputs \( b_1 \). This circuit is composed of four two-bit adder circuits and correctly constrains the binary one bit \( b_1 \) to the number of true input variables in \( X' \), there would be a similar circuit for the two-bit that takes in as input all of the carry out bits from the example. The complete binary circuit can be constructed using techniques in [52]. The circuit is translated to a Boolean formula (\texttt{SxTsp}) in polynomial time using methods from [53].

\section*{Appendix C

\textbf{Adder Circuits}

In this section, we provide an overview of how adder circuits are used to count the number of true variables in a set \( X' \). In general, we can input the negation of a variable (its negative literal) but for this document, we only count variable inputs if they are assigned true. Adder circuits are used in electronics to do rudimentary mathematical operations [52]. The example circuit shown in Figure 7 takes as input \( X' = \{x_1, x_2, \ldots, x_5\} \) and outputs \( b_1 \). This circuit is composed of four two-bit adder circuits and correctly constrains the binary one bit \( b_1 \) to the number of true input variables in \( X' \), there would be a similar circuit for the two-bit that takes in as input all of the carry out bits from the example. The complete binary circuit can be constructed using techniques in [52]. The circuit is translated to a Boolean formula (\texttt{SxTsp}) in polynomial time using methods from [53].

\begin{appendices}

\section{Proof of Theorem 1

\begin{proof}

\end{proof}

\end{appendices}