A Language For Robot Path Planning in Discrete Environments: The TSP with Boolean Satisfiability Constraints

Frank Imeson Stephen L. Smith

***Abstract*— In this paper we introduce a new language in which discrete path planning problems for mobile robots can be specified and solved. Given an environment represented as a graph and a Boolean variable for each vertex to represent its inclusion/exclusion on the path, we consider the problem of finding the shortest path (or tour) in the graph subject to a Boolean satisfiability (SAT) formula defined over the vertex variables. We call this problem SAT-TSP. We show the expressiveness of this language for specifying complex motion planning objectives in a discrete environment. We then present three solution techniques for this problem, including a novel reduction to the well known traveling salesman problem (TSP). We present extensive simulation results which compare the performance of the three solvers on standard benchmarks from TSP, SAT, and Generalized TSP (GTSP) literature.**

I. INTRODUCTION

A key problem in robotics is in providing a natural and expressive language in which a user can specify a desired task, and from which a planner can compute a robot motion plan. A common approach is to represent the environment as a finite transition system (i.e., a graph) and to specify a task in a formal language such as linear temporal logic (LTL). Linear temporal logic contains the usual Boolean operators—and, or, and not—along with temporal operators—next, always, eventually, and until. Tools from model checking can then be used to synthesize a robot motion plan that satisfies the task specification. Early work looked at finding a satisfying motion plan for a given task specification [1], [2], and recent work has looked at optimizing over the set of satisfying motion plans [3], [4].

The typical problem with these approaches is computational complexity. The problem of satisfying a general LTL formula is PSPACE-complete [5]. To combat this, researchers have proposed to look at fragments of the LTL problem, such as general reactivity [2] and the fragment introduced in [6], for which satisfaction and optimization can be performed more efficiently. This highlights the inherent trade-off between expressivity of the language, and complexity of computing the solution.

An interesting feature of temporal logic based motion planning is that it generates plans over an infinite horizon. This is a result of the temporal operators, "always" and "eventually", which specify logic over infinite time. Thus, any optimization objective must be defined over an infinite horizon. Common objective functions in LTL problems include a discounted cost, the average time between a repeating events, or the worst time between repeating events. Of course, in application, robot plans will be finite in duration.

Many finite path planning problems can be cast as optimization problems on graphs. Finding a shortest path between a start and end vertex can be solved in polynomial time using Dijkstra's algorithm. Finding a shortest path that visits all vertices in a set is known as the travelling salesman problem (TSP), which is **NP**-hard. However, if the graph is metric, then good approximation algorithms exist [7]. Problems that are easily expressed as TSP arise in surveillance and monitoring applications where a robot needs to visit the set of all vantage points [8]. A more general problem is the generalized TSP (GTSP), in which there are several sets of vertices, and the goal is to find the shortest path that visits at least one vertex in each set. The GTSP arises in several applications, including surveillance problems [9], and even certain instances of temporal logic motion planning [6]. One solution technique for the GTSP is to reduce it to the standard TSP using the Noon-Bean transformation [10]. In practice, this technique has been quite successful, and by leveraging the power of TSP solvers [11], large instances of the GTSP have been approximately solved with reasonable efficiency [9].

In this paper we seek to explore the middle-ground between these two extremes in path planning (i.e., simple graph path planning vs. LTL planning). To this end, we look at finding the shortest path in a graph subject to a set of Boolean constraints on the vertices that indicate their inclusion or exclusion from the path. We call this problem SAT-TSP. In this problem we can no longer express temporal (i.e., ordering) constraints, but the Boolean operators on their own are quite expressive—Boolean satisfiability (SAT) is an **NP**-complete problem, and thus can efficiently express any other problem in **NP**. As an example, our problem language can easily express SAT, TSP, and GTSP instances.

What is interesting, though, is the notion of expressivity. The decision versions of TSP, SAT, GTSP, and SAT-TSP are all in the complexity class **NP**-complete. Thus, they are all, in a theoretical sense, equally expressive: If we had a polynomial-time algorithm for one problem, then we would have a polynomial-time algorithm for all the other problems. However, practically, we believe that expressing complex path planning problems as SAT-TSP instances (i.e., in the SAT-TSP language) is more natural than attempting to encode them directly as TSP, GTSP, or SAT instances.

The contribution of this paper is follows. We introduce a new language SAT-TSP to allow a user to more "easily" express high-level path planning problems. We demonstrate the expressiveness of this language and we then provide

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

The authors are with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo ON, N2L 3G1 Canada (fcimeson@uwaterloo.ca; stephen.smith@uwaterloo.ca)

three methods for solving SAT-TSP instances: 1) an efficient and novel reduction to TSP, 2) a reduction to the constraint satisfaction problem (CSP), and 3) a mixed solver approach which leverages standard SAT and TSP solvers independently. We provide simulation results benchmarking these three approaches and give a coarse classification of the types of problem instances that best suit each solver approach.

The organization of this paper is as follows. In Section II we provide the necessary background on TSP, GTSP, and SAT. In Section III we formalize the SAT-TSP problem. In Section IV we provide the details on the three methods we use for solving SAT-TSP instances. In Section V we provide simulation results which shed some light on the regimes in which each solver excels. In Section VI we discuss some of the difficulties in solving SAT-TSP and other potential approaches. Finally, we conclude in Section VII.

II. BACKGROUND

In this paper we present a new language SAT-TSP which is based on the SAT and TSP languages, and we compare our generalization to the GTSP. To do so we must first provide the background on the languages SAT, TSP and GTSP and we also provide a means of comparing GTSP to TSP, which we do in the context of expressivity. We also provide background on the CSP language as we use it later in the paper.

A. Languages

The Boolean satisfiability problem SAT is expressed as a Boolean formula which contains literals and operators. A literal is either a Boolean variable (x_i) or its negation ($\neg x_i$). The operators are conjunction ($\wedge$, and), disjunction ($\vee$, or) and negation ($\neg$, not) which operate on the literals and other Boolean formulae. An assignment of the variables (true or false) will now result in the formula being satisfied or not (true or false). The conjunctive normal form (Cnf-SAT) is the standard form of SAT in which the formula F has the form $F = c_1 \wedge c_2 \wedge \cdots \wedge c_n$ and each clause $c_i = l_{i.1} \vee l_{i.2} \vee \cdots \vee l_{i.|c_i|}$ is a disjunction of literals. The problem language is defined as follows:

$\textbf{SAT} = \{\langle F \rangle : F \text{ is a satisfiable Boolean formula}\}$.

The constraint satisfiability problem CSP consists of a set of variables, a domain of values, and a set of constraints. Each constraint is a condition on a subset of the variables that must be satisfied, some examples are $\text{LESSTHAN}(x_1, x_2)$, $\text{EQUAL}(x_1, x_2)$ or $\text{ALLDIFFERENT}(x_1, x_2)$. The language is as follows:

$\textbf{CSP} = \{\langle X, D, C \rangle : X \text{ is a set of variables, } D \text{ is a domain of values and } C \text{ is a set of constraints, and there exists an evaluation } v : X \to D \text{ satisfying all constraints in } C\}$.

The travelling salesman problem TSP is traditionally posed as the following: given a list of cities and distances between each pair of cities, what is the shortest possible path that the salesman can take to visit each city exactly once and return to the first city? GTSP is the variation on the TSP where the salesman has to visit at least one city in each set of cities. A tour in a graph that visits each vertex exactly once is called a Hamiltonian tour. The languages can be written as follows:

$\textbf{TSP} = \{\langle G, c \rangle : G = \langle V, E, w \rangle \text{ is a complete graph with edge weights } w : E \to \mathbb{R}_{\geq 0} \text{ and } G \text{ contains a Hamiltonian tour with cost at most } c\}$.

$\textbf{GTSP} = \{\langle G, S, c \rangle : G = \langle V, E, w \rangle \text{ is a complete and weighted graph, } S = \{S_1, S_2, ..., S_m\} \text{ where } S_i \subseteq V \text{ for each } i \in \{1, \ldots, m\}, G \text{ contains a tour that visits at least one vertex in each set } S_i \text{ and has cost at most } c\}$.

The optimization versions of these problems find a solution that minimizes c.

B. Expressivity

To compare GTSP to TSP we use a concept known as expressivity.

Definition II.1 (Notions of expressivity)**.** The *theoretical expressivity* of a language is the breadth of problems that can be encoded in the language [12]. The *practical expressivity* of a language is an informal notion that captures the ease of which the problems can be encoded into the language.

In terms of theoretical expressivity, GTSP is no more is more expressive then TSP. However it can be argued that GTSP is practically more expressive then TSP. To understand why, note that there is a trivial reduction from TSP to GTSP: for each vertex $v_i \in V$ create a set $S_i = \{v_i\}$ and add it to the group of sets S, yielding a GTSP instance $\langle G, S \rangle$ that solves the TSP problem. On the other hand, the reduction of GTSP to TSP is a research area, with the Noon-Bean transformation [10] being one of the most widely used. In literature, the ease of which a problem is encoded is sometimes measured as the succinctness of the encoding [13]. But such encodings may be nontrivial to find. In the comparison between GTSP and TSP we could easily encode TSP instances as GTSP. However we do not have a method as easy for encoding GTSP instances as TSP instances and thus we argue that GTSP is practically more expressive than TSP. We make a similar argument in Section III as to how SAT-TSP is more expressive then GTSP in the practical sense.

III. PROBLEM STATEMENT

Our goals in this paper are two fold: we study TSP problems with additional constraints of inclusion/exclusion on the vertices in the TSP graph, and we provide the language SAT-TSP which can be used to easily express these instances.

To define SAT-TSP we consider a complete and weighted graph $G = (V, E, w)$, where $V = \{v_1, \ldots, v_n\}$. For each vertex $v_i \in V$ we associate a Boolean variable $x_i \in \{0, 1\}$. We write the corresponding set of Boolean variables as $X(V) = \{x_1, \ldots, x_n\}$. Each path in G induces exactly one assignment to the Boolean variables: $x_i = 0$ if and only if v_i is on the path. Then given a graph G, a SAT formula F defined over $X(V)$ and a possible set of additional Boolean variables (which can be used to define extra constraints), our task is to find the shortest cycle in G that satisfies F. The language is defined as follows:

SAT-TSP $= \{\langle G, F, c\rangle : G = \langle V, E, w\rangle$ is a complete and weighted graph, F is a Cnf-SAT formula defined over $X(V)$ along with a possible set of auxiliary variables, and G contains a cycle that satisfies F and has cost at most $c\}$.

Note, we have defined SAT-TSP as finding the cycle, or tour, but we could equivalently define the problem as finding the shortest path.

We claim that SAT-TSP is practically more expressive and thus more general than GTSP. To show this consider the following trivial reduction from GTSP to SAT-TSP: given a GTSP instance $\langle G, S, c\rangle$ we construct the formula $F = \prod_{j=1}^{|S|} \sum_{v_i \in S_j} x_i$, where each S_j is a set in S of the GTSP instance and now we have a SAT-TSP instance $\langle G, F, c\rangle$[1]. On the other hand, given an instance of SAT-TSP, there does not appear to be a straightforward reduction to GTSP.

To demonstrate the expressiveness of SAT-TSP, consider the following illustrative example.

Example III.1 (Expressivity of SAT-TSP)**.** Let us first consider a GTSP example where a robot has suffered severe damage to its collection unit and needs to retrieve a set of parts from different suppliers. The required parts are 1) a collection scoop, 2) a motor, and 3) a subframe. The robot can visit suppliers A, B, C for the scoop, suppliers U, V, W for the motor and X, Y, Z for the subframe. Each supplier has a physical location on a map as shown in Figure 1 and the robot must choose a tour with minimal travel distance to a set of suppliers that retrieves all three parts. This problem has a natural encoding as a GTSP instance where the sets are $S = \{\{A, B, C\}, \{U, V, W\}, \{X, Y, Z\}\}$ and the graph edge weights are given by travel times between the suppliers.

However, what if the problem contained incompatibilities in the choices? For example what if the scoop brackets are incompatible with certain subframes. Let us update our problem: supplier A has scoops with type 1 brackets, while B and C have scoops with type 2 brackets. Supplier X supplies a subframe that accepts type 1 brackets, supplier Y supplies a subframe that accepts type 2 brackets, and supplier Z supplies a subframe that accepts type 1 and 2 brackets. It is not obvious how to express this as a GTSP problem. However, we can easily encode it as SAT-TSP instance: let G be the graph with edge weights equal to the Euclidean distances between suppliers and the formula $F = (A \vee B \vee C) \wedge (U \vee V \vee W) \wedge (X \vee Y \vee Z) \wedge ((A \wedge (X \vee Z)) \vee (\neg A \wedge (Y \vee Z)))$ which we easily translated to cnf [14] for our language. We also have the freedom to introduce new (or auxiliary) variables. In this case we could have represented the choice of bracket type with a separate variable to aid in readability. □

While the above example is simple, it demonstrates that complex constraints consisting of dependencies and incompatibilities can be easily represented in the SAT-TSP language.

IV. APPROACH

In this section we present three approaches for solving SAT-TSP instances. The first approach is to reduce SAT-TSP instances to TSP instances, which can then be solved using an existing TSP solver. The second approach reduces SAT-TSP instances to CSP instances, on which we use an existing solver to find the optimal solution. The third approach uses a combination of solvers for SAT and TSP to solve the problem.

[1]Summation is used to represent a series of disjunctions $\sum a_i = a_1 \vee a_2 \vee \cdots \vee a_n$, while product is used to represent a series of conjunctions $\prod a_i = a_1 \wedge a_2 \wedge \cdots \wedge a_n$.

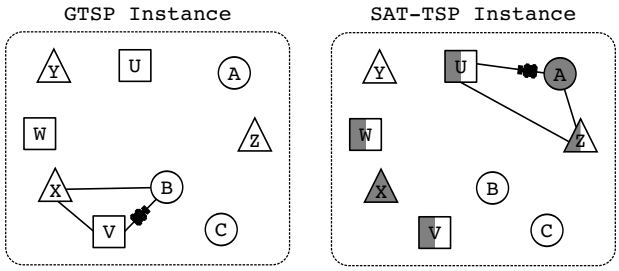

Fig. 1: Visual representation of our GTSP and SAT-TSP example. Shapes represent the different groups, circles for scoop suppliers, squares for motor suppliers and triangles for subframe suppliers. The colour in each shape represents the bracket compatibility, grey for type 1 and white for type 2. The shortest tour is shown for each instance.

In the rest of this section we outline the details of these three approaches, for which we use the set of symbols x_i to represent the boolean variables in the SAT formula F, c_j to represent the clauses in the formula, l_k to represent literals in a clause and v_i to represent vertices in the TSP graph G. In Section V we compare the performance of these approaches in simulation.

A. Approach 1: Reduction of SAT-TSP To TSP

For this approach we reduce the SAT-TSP instance to a series of TSP instances that each have a different starting vertex v_s. After solving each instance we return the optimal result. This procedure is shown in Algorithm 1.

Algorithm 1: TSP APPROACH(F, G)

```
c_min ← ∞
Φ_min ← ∅
for v_i ∈ V[G] do
    G' ← REDUCE2TSP(F, G, v_i)
    ⟨Φ, c⟩ ← SOLVETSP(G')
    if c < c_min and FEASIBLE(F, G, Φ, c) then
        c_min ← c
        Φ_min ← Φ
return ⟨Φ_min, c_min⟩
```

Algorithm 2: REDUCE2TSP(F, G, v_s)

```
H, W ← ∅
V[H] ← V[H] ∪ {v_s, v_{c_1}, v_{c_2}, ..., v_{c_m}}
for each v_i ∈ V[G]/v_s do
    ⟨H, W_i⟩ ← ADDWIDGET(F, G, H, v_i)
    W ← W ∪ W_i
H ← CONNECTWIDGETS(H, W, v_s)
return H
```

The reduction from SAT-TSP to TSP is shown in Algorithms 2, 3 and 4. It consists of the construction of "widgets" that allow for a TSP instance to represent inclusion/exclusion of a vertex. For conciseness this reduction does not include the widgets for auxiliary variables. However, the widgets for auxiliary variables are constructed in the same manner. The only difference is the connections between widgets, which in this case force a solution to traverse all auxiliary widgets at the end of the tour. This design is inspired by the reduction from SAT to the Hamiltonian Cycle Problem [15].

Algorithm 3: ADDWIDGET(F, G, H, k)

1 $W_k \leftarrow \{v_{k.1}, v_{k.2}\}$
2 **for** *each* $c_i \in F$ *and each* $l_j \in c_i$ **do**
3 **if** $l_j = x_k$ *or* $l_j = \neg x_k$ **then**
4 $n \leftarrow |W_k|$
5 $W_k \leftarrow W_k \cup \{v_{k.n+1}, v_{k.n+2}, v_{k.n+3}\}$
6 **if** $l_j = x_k$ **then**
7 $E[H] \leftarrow E[H] \cup \{\langle v_{k.n+1}, c_i\rangle, \langle c_i, v_{k.n+2}\rangle\}$
8 $w_H(v_{k.n+1}, c_i) \leftarrow 0$
9 $w_H(c_i, v_{k.n+2}) \leftarrow 0$
10 **else**
11 $E[H] \leftarrow E[H] \cup \{\langle v_{k.n+2}, c_i\rangle, \langle c_i, v_{k.n+1}\rangle\}$
12 $w_H(v_{k.n+2}, c_i) \leftarrow 0$
13 $w_H(c_i, v_{k.n+1}) \leftarrow 0$
14 $W_k \leftarrow W_k \cup \{v_{k.|W_k|+1}\}$
15 $V[H] \leftarrow V[H] \cup W_k$
16 $v_a \leftarrow v_{k.1}$
17 **for** *each* $v_b \in W_k / v_{k.1}$ **do**
18 $E[H] \leftarrow E[H] \cup \{\langle v_a, v_b\rangle, \langle v_b, v_a\rangle\}$
19 $w_H(v_a, v_b) \leftarrow 0$
20 $w_H(v_b, v_a) \leftarrow 0$
21 $v_a \leftarrow v_b$
22 **return** $\langle H, W_k\rangle$

A widget is shown in Figure 3. If the widget is traversed from left to right, then the vertex is included in the solution. If the widget is traversed from right to left, then the vertex is excluded. These widgets are also connected to clause vertices. When the path visits a clause vertex then the clause is satisfied as shown in Figure 3. The edge weights connecting widgets to widgets and vertices are shown in Figure 2.

Remark IV.1 (Creating a Complete Graph). The input to a TSP solver is a complete graph, and thus for any missing edges in our construction, we add edges with an infinite cost to the graph.

We now prove that Algorithm 2 is indeed a reduction. We begin with a definition and some useful results.

Definition IV.2 (Feasibility)**.** A feasible TSP solution is a tour that has non-infinite cost.

Lemma IV.3 (Reduction Results)**.** *For the reduction from SAT-TSP to TSP in Algorithms 2, 3, and 4, the following hold:*

(i) *Algorithm 2 runs in $O(|V|^2 + |V||L|)$ where $|V|$ is the number of vertices in G and $|L|$ is the number of literals in F.*
(ii) *A feasible TSP tour must traverse a chain from one end to the other before visiting another widget.*
(iii) *A feasible TSP tour must visit all included vertex widgets followed by all excluded vertex widgets.*
(iv) *A feasible TSP tour translates to a solution for the SAT-TSP instance.*
(v) *A SAT-TSP solution translates to a feasible tour for the TSP instance.*

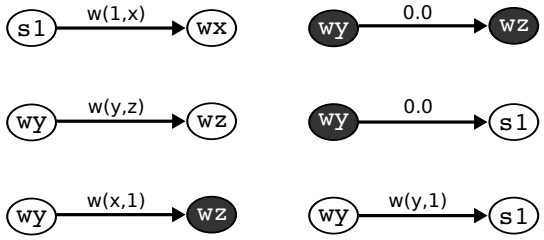

Fig. 2: This diagram is a legend of the different edge weights connecting widgets to widgets and widgets to the start vertex. A widget wy that is white in colour indicates inclusion of the vertex y and black colour represents exclusion. The weight function is $w(x, y)$ is with respect to the original graph G in the SAT-TSP instance.

Algorithm 4: CONNECTWIDGETS(H, W, v_s)

1 **for** *each* $W_i \in W$ **do**
2 $E[H] \leftarrow E[H] \cup \{\langle v_s, v_{i.1}\rangle, \langle v_{i.|W_i|}, v_s\rangle, \langle v_{i.1}, v_s\rangle\}$
3 $w_H(v_s, v_{i.1}) \leftarrow w_G(v_s, v_i)$
4 $w_H(v_{i.|W_i|}, v_s) \leftarrow w_G(v_i, v_s)$
5 $w_H(v_{i.1}, v_s) \leftarrow 0$
6 **for** *each* W_j / W_i **do**
7 $E[H] \leftarrow E[H] \cup$
 $\{\langle v_{i.|W_i|}, v_{j.1}\rangle, \langle v_{i.|W_i|}, v_{j.|W_j|}\rangle, \langle v_{i.1}, v_{j.|W_j|}\rangle\}$
8 $w_H(v_{i.|W_i|}, v_{j.1}) \leftarrow w_G(v_i, v_j)$
9 $w_H(v_{i.|W_i|}, v_{j.|W_j|}) \leftarrow w_G(v_i, v_s)$
10 $w_H(v_{i.1}, v_{j.|W_j|}) \leftarrow 0$
11 **return** H

Proof. We will establish each of the five results in turn.

Proof of (i): For lines 2-13 in ADDWIDGET it traverses the set of literals L, creating widgets of size $O(|L|)$, for lines 17-21 it connects all the widget's vertices together $O(|L|)$, and so the total running time of ADDWIDGET is $O(|L|)$. The CONNECTWIDGETS algorithm populates all edges in between widgets $O(|V|^2)$. The reduction in Algorithm 2 calls the ADDWIDGET Algorithm $O(|V|)$ times and the CONNECTWIDGETS once and so we have a running time of $O(|V|^2 + |V||L|)$.

Proof of (ii): Refer to Figure 3 to see that a widget only has two possible directions ($1 \to 9$ or $9 \to 1$) for the tour to visit all of the vertices on the chain. Each path starts and ends at opposite extremes of the chain and by considering all possible paths from the vertex before clause c_1 to the vertex after the clause $(\{2, 3, 4, 5\}, \{2, 3, c_1, 4, 5\}, \{5, 4, 3, 2\})$ we see that there does not exists a feasible tour that can visit

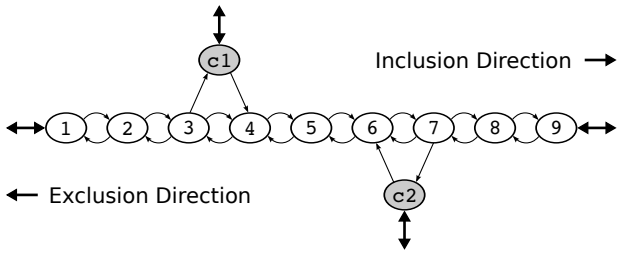

Fig. 3: Above we show an example of a widget for vertex v_i attached to two clause vertices c_1 and c_2. The clause c_1 contains literal x_i and the clause c_2 contains literals $\neg x_i$. A path that traverses the vertices from left to right ($1 \rightarrow 9$) indicates that v_i is included in the SAT-TSP tour.

another widget from a clause vertex and since there are no other connections to widgets, a path must traverse the entire widget before visiting the next widget.

Proof of (iii): feasible tours include the starting vertex v_s, there are no connections from v_s to any excluded vertices, and there are no connections from any excluded vertices to any included vertices. Thus all tours must start with v_s and visit all included vertices before the excluded vertices.

Proof of (iv): The traversal direction of the chain indicates the inclusion or exclusion of the vertex in the tour. Since the chain must be traversed from one end to the other, the only clause vertices that can be visited are those that are satisfied by the direction chosen (inclusion or exclusion) and since a feasible tour must visit all clause vertices, then the corresponding SAT instance must be satisfied.

Proof of (v): Construct a tour for the TSP instance to visit all included vertices followed by the excluded vertices (arbitrary order). For each clause vertex, find a pair of edges that the tour can deviate from to visit the clause vertex. This deviation must exist since there is at least one true literal in each clause.

□

Theorem IV.4. *(Reduction) Algorithm 2 is a reduction from SAT-TSP with starting vertex v_s to TSP with starting vertex v_s. Specifically the algorithm runs in polynomial time and there exists a solution to the SAT-TSP instance if and only if there is a solution to the TSP instance. Moreover, the TSP and SAT-TSP solutions have equivalent costs and thus have the same minimum and maximum solutions.*

Proof. The proof has two parts, the first part follows directly from Lemma IV.3. The second part follows from the fact that a feasible tour visits all included vertices followed by excluded vertices and the edge weights between included widgets $\langle w_i, w_j \rangle$ are the same as the edge weights between vertices $\langle v_i, v_j \rangle$ in the SAT-TSP instance. The edge weight of the last included vertex v_l to any other vertex (including v_s) is $w_G(v_l, v_s)$ which is also the same weight as the edge $\langle v_l, v_s \rangle$ in the SAT-TSP instance and thus the entire tour has the same cost. □

B. Approach 2: Reduction of SAT-TSP To CSP

For this approach we translate the SAT-TSP instance into CSP instances with a fixed tour length and a maximum cost. We solve these instances with an existing solver. Once we have exhausted the search for a tour that has cost less than our best solution, we return the optimal result. This procedure is shown in Algorithm 5.

The reduction of SAT-TSP to CSP, consists of reducing SAT to CSP, TSP to CSP and then constraining the tour to visit all vertices that the SAT formula has included in the solution. We used a similar reduction from SAT to CSP as shown in Walsh [16]. For the TSP reduction, we simply use a set of variables $\{t_1, t_2, ..., t_s\}$, with domain $\{1, 2, ..., |V[G]|\}$, and the constraints that all variables have different values. We also use a linear equation to constrain a variable `cost` to equal the tour cost (as computed from the edge weights). Finally, we constrain the tour cost to satisfy `cost` $< c_{\min}$.

Remark IV.5 (Binary Search). Algorithm 5 constructs CSP instances that restrict a solution to have cost less than $c_{\min}$. This is not an efficient approach as we may explore an exponential number of solutions. However we find in practice we are not exploring exponential number of solutions and binary search takes longer since it takes longer to return a negative result than a positive result. □

Algorithm 5: CSP APPROACH(F, G)

```
1  c_min ← ∞
2  Φ_min ← ∅
3  for s ∈ {1, 2, .., |V[G]|} do
4      while True do
5          ⟨X, D, C⟩ ← REDUCE2CSP(F, G, s, c_min)
6          ⟨Φ, c⟩ ← SOLVECSP(X, D, C)
7          if c < c_min then
8              c_min ← c
9              Φ_min ← Φ
10         else
11             break
12 return ⟨Φ_min, c_min⟩
```

C. Approach 3: MIXED Approach

The MIXED approach uses a SAT solver to exhaustively find all the solutions of the SAT instance. It then constructs a sub-graph G' from G for each solution of the SAT formula, by removing all vertices that the SAT solution indicates to exclude. For each sub-graph G' a TSP solver finds the shortest tour and the optimal result out of the set of solutions is returned. This procedure is outlined in Algorithm 6. The approach is inefficient since the SAT instance may have an exponential number of solutions. However, we have observed good performance on a subset of our testbed, and so we use this approach as a comparison for the other two approaches.

V. RESULTS

In this section we summarize the results of our simulations and give some insight into the strengths and weaknesses of our proposed approaches. The simulations were run on the cloud computing system `sharcnet.ca`. All the

Algorithm 6: MIXED APPROACH(F, G)

```
c_min ← ∞
Φ_min ← ∅
S ← FINDALLSOLNS(F)
for each S_i ∈ S do
    G' ← SUBGRAPH(G, S_i)
    ⟨Φ, c⟩ ← SOLVETSP(G')
    if c < c_min then
        c_min ← c
        Φ_min ← Φ
return ⟨Φ_min, c_min⟩
```

implementations are single threaded and programmed in python. The python programs are responsible for parsing the input, translating the instances, invoking external solvers (LKH [17] for TSP instances[1], minisat [18] for SAT instances and GeCode [19] for CSP instances), timing out the external solvers if necessary, interpreting the results and measuring the efficiency.

The instances that we tested on were obtained from three sources: the TSP instances were obtained from the TSPLIB [20], of which we used both symmetric and non-symmetric instances of up to 1300 vertices, the SAT instances were obtained from SATLIB [21], of which we used a subset of satisfiable instances with up to 260 variables and the GTSP instances were obtained from Karapetyan's GTSPLIB [22], which contained instances that have up to 1000 vertices. We combined the TSPLIB and the SATLIB instances in three different configurations: 1) EASY-SAT, which uses TSP instances as is from TSPLIB, constructs the SAT formula to be $F = \prod_{i \in |V[G]|} x_i$ (note, this SAT formula has only one solution: $x_i = 1$ for each i), 2) EASY-TSP, which uses SAT instances as is from SATLIB, constructs the complete graph of the TSP instance to have all the same edge weights $w(v_i, v_j) = 1$ (note, in this graph, every tour corresponding to a given SAT solution has the same cost), and 3) HARD-SAT, which uses the SAT instances as is from the SATLIB library, constructs a graph to be the induced subgraph of a TSP instance in TSPLIB, where all vertices with index labels larger then the number of variables in F are excluded.

In the rest of this section we compare the computation efficiency and soundness of the results (optimality) for our three approaches.

A. *Efficiency*

The running time of an algorithm is often considered an important metric for comparison. Here we compare the running time of all three approaches. Since the MIXED approach is not efficient for SAT instances that have exponential number of solutions, we do not run it on GTSPLIB instances.

Our first comparison seeks to study how well the TSP approach handles EASY-SAT instances. To do this, we compare the running time to the MIXED approach as shown in Figure 4. The TSP approach implements a Turing reduction that solves $|V[G]|$ number of instances. In the case of EASY-SAT, each instance is essentially equivalent. Thus, to remove this aspect we compare the normalized running time of solving one out of $|V[G]|$ instances. From Figure 4 we see that this approach often takes more than $10\times$ the amount of time as the MIXED approach (which amounts to a single call to a TSP solver). Most of the TSPLIB instances we have considered are metric. Unfortunately, even when the SAT instance is trivial, the reduction from SAT-TSP to TSP will not preserve the metric property. This, we believe, is the reason for the large increase in run-time.

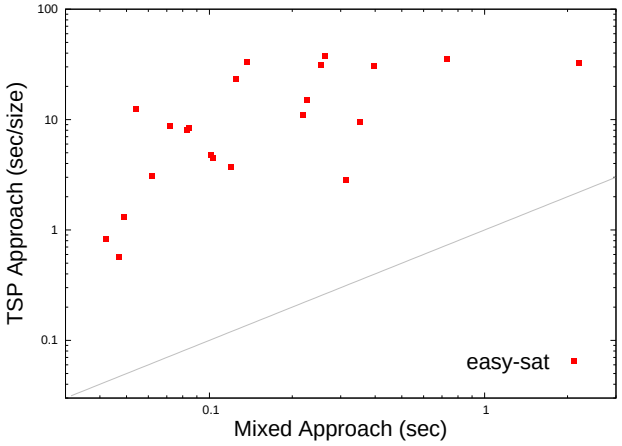

Fig. 4: Comparison of the time taken (including reduction) for the TSP approach with respect to the time taken for the MIXED approach on the set of EASY-SAT instances. Each data point shows one problem instance in the EASY-SAT class.

Figure 5 compares the CSP approach to the MIXED approach. The CSP approach is slower in most of our instances by many orders of magnitude when compared to the MIXED approach. However, the CSP approach is able to generate a solution for most input instances, which we cannot say about the TSP approach or the MIXED approach. For instance, the MIXED approach times-out before it is able to find an optimal solutions to GTSP instances and the TSP approach is not able to solve any EASY-TSP or HARD-SAT instances, thus those simulation results are absent in the figures.

We can also see that the CSP approach seems to be able to handle SAT instances from SATLIB in a comparable amount of time to the MIXED approach as seen with the EASY-TSP results, but for the EASY-SAT and HARD-SAT instances it does much worse, which is an indicator that the CSP approach is not handling the TSP instances very well.

In Figure 6 we compare the TSP approach to the CSP approach and as we can see there is no clear winner for EASY-SAT instances, but CSP does outperforms the TSP approach on GTSP instances. However if we were to normalize the TSP results again the TSP approach would do better on the EASY-SAT instances but no such normalization exists for GTSP instances.

[1]We also tried exact solvers but found that for the problem instances under consideration, LKH had significantly better performance.

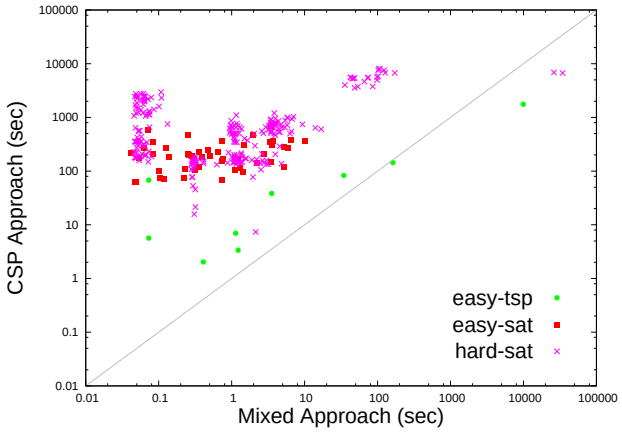

Fig. 5: Comparison of the time taken (including reduction) for the CSP approach with respect to the time taken for the MIXED approach on the set of input instances: EASY-SAT, EASY-TSP and SATLIB. Each data point corresponds to one problem instance.

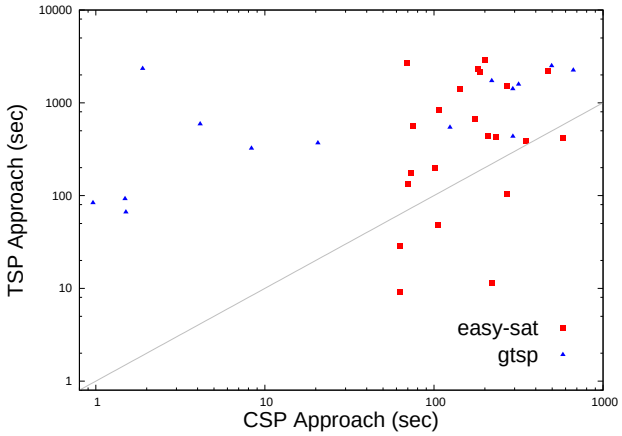

Fig. 6: Comparison of the time taken (including reduction) for the TSP approach with respect to the time taken for the CSP approach on the set of input instances: EASY-SAT and GTSP.

B. Optimality

Another metric for comparing algorithms is the soundness of the solution, in the case of optimization algorithms a measure of soundness is whether or not the algorithm returns the optimal (or best known) result.

For our simulations we have compared the output of all the approaches as well as included the best known solutions from the GTSPLIB for our analysis. The first thing we note is that when the TSP approach or the MIXED approach finds a solution it is almost always optimal and so we do not show any graphs for these results. However the CSP approach does not always obtain the optimal tour, which is due to the fact that we limit its total computation time. Note that we also limit the computation time for the TSP approach, but it usually returns an optimal value or none at all.

In Figure 7 we see that for EASY-TSP instances the CSP approach is always optimal. Similarly most of the solutions found for GTSPLIB instances were optimal but we can see that EASY-SAT and HARD-SAT instances deviated quite a bit from the best known solution and thus our original assessment of the TSP approach and the CSP approach being

Solver	EASY-SAT	EASY-TSP	HARD-SAT	GTSPLIB
TSP	34%	0%	0%	43%
CSP	4%	71%	17%	90%
MIXED	100%	92%	99%	0%

TABLE I: This table presents the percentage of instances that the respective approaches were able to obtain the best known solution.

comparable for EASY-SAT instances does not have the same meaning. In this case we would rather choose a solver with comparable time and better solutions (the TSP approach). We also observe this result in Table I, where CSP is only able to find 4% of the optimal solutions for EASY-SAT instances. Encouragingly we see that CSP is able to find 71% of the EASY-TSP optimal solutions and 90% of the GTSPLIB optimal solutions, but does not perform as well for HARD-SAT solutions. The TSP approach does not do well overall but as we can see from Table I it is able to find about 30-40% of the optimal solutions for EASY-SAT and GTSPLIB. Not surprisingly the MIXED approach does well on all but GTSPLIB instances. This motivates the question, might there exist an efficent mixed approach?

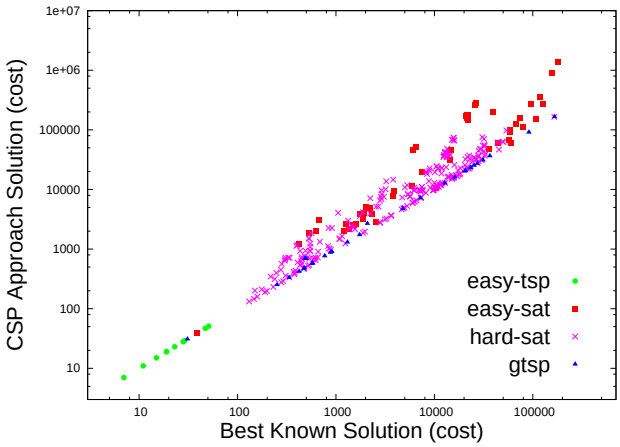

Fig. 7: Comparison of the optimal cost found with the CSP solver approach with the best know solution.

VI. DISCUSSION

Given a SAT solver and a TSP solver, there is some subset of instances SAT* $\subseteq$ SAT and TSP* $\subseteq$ TSP which can be solved efficiently[2].

We have observed that the majority of our instances from the SATLIB and TSPLIB library are in SAT* and TSP*. This implies that our reduction from SAT-TSP to TSP does not in general map into TSP* instances. However since the MIXED approach does not translate the TSP instance, we expect it to be handled efficiently, which is what we observed. Recall that we have stated that the MIXED approach is not efficient and so we wonder if there is an efficient implementation that avoids translating the instances and hence avoids the possibility of mapping an instance in TSP* to an instance in TSP $\setminus$ TSP*. The answer to this question is no, and to understand why

[2]For this argument, we adopt a crude classification of efficiency to be the set of instances that are on average solved in less then one second.

we present the following theorem, but first we need two definitions. A sub-instance of a SAT instance F is any SAT instance F' obtained by fixing an assignment to a sub-set of variables in F. A sub-instance of a TSP instance $\langle G, c\rangle$ is any instance $\langle G', c'\rangle$ where G' is a complete subgraph of G.

Theorem VI.1. *The decision version of SAT-TSP is **NP**-complete even when we have access to oracles for the specific SAT and TSP instances or any of their sub-instances.*

This result states that unless **P = NP**, we have no hope of finding an efficient algorithm that does not include a translation of either the SAT instance and or the TSP instance. We now prove this result:

Proof. To prove the above result let us reduce a known **NP**-complete problem to SAT-TSP, namely SET-COVER.

SET-COVER $= \{\langle S, k\rangle : S$ is a collection of subsets of elements from a finite universe U for which there exists a subset $S' \subset S$ which covers all the elements in U and $|S'| \leq k\}$.

The reduction maps the sets in SET-COVER to vertices in a complete graph G, in which all edges have a weight of 1. The inclusion/exclusion of a set S_i is indicated by a variable x_i, which is used to ensure that each element u_j is covered. This is accomplished with the formula $F = \prod \sum_{u_j \in S_i} v_i$. Now a minimal solution to the SAT-TSP instance $\langle G, F\rangle$ has a tour of length l which is also the minimum number of sets needed to cover all the elements U.

The TSP instance and sub-instances have trivial solutions (an arbitrarily ordering of the vertices). The SAT instance also has a trivial solution, which is to arbitrarily pick a variable to be true in each clause (there are no negative literals), for a sub-instance, remove all false literals from the formula and repeat the previously mentioned procedure. These instances are solved in linear time which is the same time that it would take an oracle to read off the solution. Thus in this case, it is as if we have oracles for SAT and TSP, and since the SET-COVER problem is **NP**-hard, it must be that SAT-TSP is **NP**-hard even when we have oracles for the SAT and TSP instances or any sub-instances. Since SAT-TSP is in **NP** then SAT-TSP is **NP**-complete. □

Despite this negative result, all is not lost. There may still exist an efficient algorithm that guarantees the result is within some constant factor of the optimal solution (approximation). Alternatively, there may exist an efficient algorithm to solve instances from SAT-TSP* = SAT* + TSP* by reducing to instances in TSP* and SAT*.

VII. CONCLUSION

To summarize, we have introduced a new problem language SAT-TSP which enables us to easily express discrete robotic path planning problems with complex constraints. We have provided three approaches to solve SAT-TSP problems which we have simulated to see how each approach performs on different types of instances.

We plan to continue our investigation of solving SAT-TSP instances by using reductions to TSP, SAT and CSP so that we can better understand what kind of millage we can get from existing solvers before we turn our attention to custom solvers. As we discussed in Section VI we are particularly interested to know if we can construct an efficient algorithm that reduces SAT-TSP to TSP and SAT instances that are solved efficiently or if we can find an approximation algorithm that does not require a translation of the sub-problems (SAT and TSP).

REFERENCES

[1] C. Belta, V. Isler, and G. J. Pappas, "Discrete abstractions for robot motion planning and control in polygonal environments," *Robotics, IEEE Transactions on*, vol. 21, no. 5, pp. 864–874, 2005.

[2] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Temporal-logic-based reactive mission and motion planning," *Robotics, IEEE Transactions on*, vol. 25, no. 6, pp. 1370–1381, 2009.

[3] S. L. Smith, J. Tůmová, C. Belta, and D. Rus, "Optimal path planning for surveillance with temporal-logic constraints," *The International Journal of Robotics Research*, vol. 30, no. 14, pp. 1695–1708, 2011.

[4] E. M. Wolff, U. Topcu, and R. M. Murray, "Optimal control with weighted average costs and temporal logic specifications," in *Robotics: Science and Systems*, 2012.

[5] A. P. Sistla and E. M. Clarke, "The complexity of propositional linear temporal logics," *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 733–749, 1985.

[6] E. M. Wolff, U. Topcu, and R. M. Murray, "Optimal control of non-deterministic systems for a computationally efcient fragment of temporal logic," in *IEEE Conference on Decision and Control*, 2013, to appear.

[7] B. Korte and J. Vygen, *Combinatorial Optimization: Theory and Algorithms*, 4th ed., ser. Algorithmics and Combinatorics. Springer, 2007, vol. 21.

[8] Y. Chevaleyre, "Theoretical analysis of the multi-agent patrolling problem," in *IEEE/WIC/ACM Int. Conf. Intelligent Agent Technology*, Beijing, China, Sep. 2004, pp. 302–308.

[9] K. J. Obermeyer, P. Oberlin, and S. Darbha, "Sampling-based path planning for a visual reconnaissance unmanned air vehicle," *Journal of Guidance, Control, and Dynamics*, vol. 35, no. 2, pp. 619–631, 2012.

[10] C. Noon and J. Bean, "An efficient transformation of the generalized traveling salesman problem," *Ann Arbor*, vol. 1001, pp. 48 109–2117, 1989.

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

[12] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov, "Complexity and expressive power of logic programming," *ACM Computing Surveys (CSUR)*, vol. 33, no. 3, pp. 374–425, 2001.

[13] G. Ausiello and L. Cabibbo, "Expressiveness and complexity of formal systems," *Functional Models of Cognition*, 1999.

[14] M. N. Velev, "Efficient translation of boolean formulas to cnf in formal verification of microprocessors," in *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*. IEEE Press, 2004, pp. 310–315.

[15] É. Tardos and J. Kleinberg, "Algorithm design," 2006.

[16] T. Walsh, "Sat v csp," in *Principles and Practice of Constraint Programming–CP 2000*. Springer, 2000, pp. 441–456.

[17] K. Helsgaun, "An effective implementation of the lin–kernighan traveling salesman heuristic," *European Journal of Operational Research*, vol. 126, no. 1, pp. 106–130, 2000.

[18] N. Sorensson and N. Een, "Minisat v1. 13-a sat solver with conflict-clause minimization," *SAT*, vol. 2005, p. 53, 2005.

[19] G. Team, "Gecode: Generic constraint development environment, 2006," 2008.

[20] G. Reinelt, "Tspliba traveling salesman problem library," *ORSA journal on computing*, vol. 3, no. 4, pp. 376–384, 1991.

[21] H. H. Hoos and T. Stützle, "Satlib–the satisfiability library," *Web site at: http://www. satlib. org*, 1998.

[22] G. Gutin and D. Karapetyan, "A memetic algorithm for the generalized traveling salesman problem," *Natural Computing*, vol. 9, no. 1, pp. 47–60, 2010.