Advances in Electrical and Computer Engineering **Volume 14, Number 1, 2014**

Visibility-based Planners for Mobile Robots Capable to Handle Path Existence Queries in Temporal Logic

Mihai POMARLAN
Politehnica University of Timişoara, 300006, Romania
mihai.pomirlan@etc.upt.ro

***Abstract*—Over the past few years, sampling based planner algorithms have been applied to planning queries formulated in path existence temporal logic, a formal system that allows more complex specifications on a solution path and is useful for task planning for mobile robots or synthesizing controllers for dynamical systems. In this paper, we extend the visibility heuristic to planners capable to handle finite path existence temporal logic queries. Our interest is justified by the visibility heuristic's ability to construct small roadmaps that are fast to search. We find that the visibility heuristic must be amended so that it can reliably handle temporal logic queries and we propose a suitable modification of the heuristic. We then present a method to extract a solution path from a roadmap, if such a solution exists. Finally, we show how the planner can be used to generate looping paths by augmenting it with a gap reduction step.**

***Index Terms*—collision avoidance, computational efficiency, mobile robots, motion planning, reachability analysis, robot motion.**

I. INTRODUCTION

The motion planning problem for a robotic system (like a manipulator arm or a mobile robot) is typically formulated as the requirement to find a path between a start and a goal configuration, such that following the path does not result in the robot colliding with obstacles in its work environment. We'll thereafter refer to such problems as point-to-point planning queries, and the methods used to solve them as point-to-point planners. However, the tasks a robot may be asked to perform, especially if it aims for some degree of autonomy, are often more complex than simple point-to-point queries and involve sequences of actions, each with different constraints [1]. For example, a robot might need to carry a glass of water to a sink; while carrying the glass, it should keep it upright and move slowly enough to prevent spilling, however once it reaches a point above the sink it would be allowed to tip it over. As another example, a robotic vehicle may be required to visit several waypoints in sequence, using a path that allows a fallback to a safety region; inertia and non-holonomic constraints limit which directions the robot can take and how fast it can move and still have a quick retreat trajectory.

Temporal logic [2-4] has been researched in recent years as an elegant way to describe and reason about tasks more complicated than simple point-to-point queries for mobile robots [5-7]. A temporal logic is a formal system capable to describe (say what is) or specify (say what should be) the behavior of a time-varying system, and it uses the operators of propositional calculus (AND, OR, NOT) together with some temporal operators, depending on the particular system of temporal logic, to do so. Originally, temporal logic was proposed as a means for computer program verification; it has recently been applied to motion planning for mobile robots because it allows more robust handling of tasks [5], as well as eases integration of control considerations into planning [6-7]. The cited papers provide examples of planners capable of handling problems formulated as temporal logic statements, not just point-to-point queries.

Exact algorithms for motion planning are known to be computationally intractable [8], and as a result the state of the art in the field is represented by the so called sampling-based algorithms. These generate a usually random collection of points in the robot's state space, called samples, which they then attempt to link via simple trajectories generated by a local planner procedure that ignores obstacles. The samples and connections that do not collide with obstacles are kept in a graph called a roadmap, which is an approximate representation of the connectivity of the robot's state space. Sampling-based planners do not guarantee a plan will always be found if a feasible path exists in the state space; the path might not be in the roadmap. However, another guarantee, called probabilistic completeness, is usually offered. Probabilistic completeness means that, as the number of samples in the roadmap goes to infinity, the chance of finding a plan if one exists goes to one. This means that, almost surely, a solution will be eventually found if the sampling process is continued for long enough.

In this paper we propose and verify through simulation a motion planner capable to handle queries expressed as temporal logic statements, which can solve such queries even though it keeps a smaller roadmap than previous algorithms. Our interest in reducing the size of a planner's roadmap is two-fold. First, the smaller the roadmap, the faster it is to query and produce a plan. Second, should some further processing be needed on its contents (because, for example, the environment has changed), this processing is faster if the roadmap is small. Obviously, the roadmap cannot be too small or else it ceases to be useful, because it doesn't capture the connectivity of the configuration space.

Our starting point is the visibility heuristic [9-10] and related sparse planners [11-12], which keep a new sample

This work was partially supported by the strategic grant POSDRU 107/1.5/S/77265, inside POSDRU Romania 2007-2013 co-financed by the European Social Fund – Investing in People.

Digital Object Identifier 10.4316/AECE.2014.01009

only if it improves coverage (the new sample cannot be connected by non-colliding simple trajectories to any roadmap sample) or if it improves connectivity (the new sample can be connected to roadmap samples from different connected components); as a result, few random samples are kept and roadmaps constructed by visibility planners tend to be very small. The visibility heuristic was formulated for point-to-point planners and is probabilistically complete in that context. However, in general it will not be probabilistically complete for specifications in temporal logic. For example, suppose we want to reach the goal by a path that is either completely inside a region "p" or completely inside a region "q" of the free space. The visibility heuristic guarantees we can find a path to the goal through free space, but does not guarantee we'll find a path with the required property because it uses no information about the regions p and q. In this paper we present an adapted version of the visibility heuristic for path existence temporal logic queries.

The visibility heuristic as originally proposed in [9] and [10] and the visibility heuristic proposed here for temporal logic queries do not guarantee optimality of paths. Visibility heuristics can be "softened", ie. made to allow more samples in the roadmap if some path length improvement condition holds, which has been done for the classic visibility heuristic for point-to-point planning queries [13], but this is beyond the scope of this paper. We aim here to present a base visibility heuristic that can handle temporal logic queries. Augmenting the heuristic with asymptotic near-optimality considerations is left for future work.

Point-to-point planning algorithms use a shortest path graph search algorithm like Dijkstra or A* to find a solution to a simple, start to goal planning query in a roadmap. For a problem specification in temporal logic, shortest path algorithms are in general not applicable. Consider again the task of getting from a start region to a goal, via a path that stays inside some region "p" of the state space; there is no reason to expect this would be the shortest path. In general, previous planners for temporal logic specifications [5-6] have used model checkers like the ones described in [14], [15] to find a plan from the roadmap. We propose a simpler procedure here, tailored to the temporal logic fragment we study.

We limit discussion to a path-existence fragment of a temporal logic system known as LTL [2]. A statement expressible in this temporal logic requires a finite path with some given property to exist. We restrict to path existence under the assumption that if a solution exists, then the robot is free to choose it. Note, while path existence LTL can only specify finite trajectories, it is possible to instruct the planner to close a path into a loop, if one is required, for example in the case of a robot that needs to patrol through several regions. We show here how to use an adapted visibility planner for path existence LTL specifications together with a gap reduction step to generate loops.

In summary the contributions of this paper are as follows: a visibility heuristic adapted to path existence LTL, a way to estimate how well a robot's state space is covered, a procedure to extract a plan from a roadmap, and a procedure to generate looping paths.

The paper is organized as follows. Section II offers theoretical background on graph theory, sampling-based planners, and temporal logic. In section III we describe the visibility heuristic adapted for path existence LTL, indicate how coverage can be estimated, describe a procedure to extract plans from the roadmap, and show how the planner can be used to create paths containing loops. In section IV we compare our method to previously existing algorithms in the literature: PRM, the classical visibility method, and RRG [7], using both shortest path search (Dijsktra) and the path search we propose here, and show our method can drastically reduce the number of samples needed in the roadmap. Finally, section V presents conclusions and future work.

II. THEORETICAL BACKGROUND

A. Graphs and roadmaps for planning

A graph G is a mathematical structure consisting of two sets: $G = \{V, E \subseteq V \times V\}$ where V is referred to as the vertex set and E, the edge set, is a set of ordered pairs of vertices. Graphs have many applications in computer science and engineering but in this paper we focus on two of them, roadmaps for planning and graphs of strongly connected components.

The graph as defined above is a directed graph, in that if $(v_1, v_2) \in E$ it doesn't follow that $(v_2, v_1) \in E$. The significance of an edge is that one can "go" from one end to the other directly by "moving along" that edge. Unless an edge in the opposite direction exists, one cannot move backwards directly.

A path from one vertex v_1 to a vertex v_n is a sequence of vertices starting with v_1 and ending with v_n, and such that for any v_k, v_{k+1} consecutive vertices in the sequence, one finds that $(v_k, v_{k+1}) \in E$. Paths that begin and end at the same vertex are called cycles.

Some oriented graphs contain no cycles and are called acyclic (any unoriented graph with at least one edge contains cycles). On acyclic graphs one can define a partial order relation, the topological sort $<$, defined by requiring that, if an edge (v_1, v_2) exists, then it must be the case that $v_1 < v_2$. Algorithms for the topological sort of directed acyclic graphs (DAGs) exist [16], including versions that can efficiently redo the sorting as changes happen to the graph, without resorting to recomputing the sort from scratch [17].

A subgraph of a graph is a subset of its vertices $U \in V$, together with a subset of the edges that connect vertices from U.

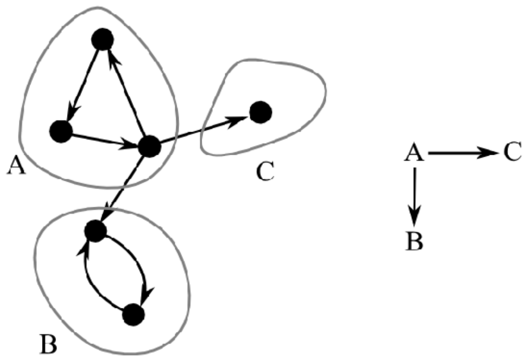

Figure 1. Directed graph (left) with maximal strongly connected components circled. Graph of SCCs of the original graph (right).

A strongly connected component *S* of a graph is a non-empty subset of the vertex set *V*, along with all edges from the graph that are between vertices in *S*, which has the following two properties: (connectivity) between any two vertices v_1, $v_2 \in S$, there exists cycles using only vertices from *S*; and (maximality) for any $v_1 \in S$ and any $v_2 \in V\text{-}S$, there is no cycle between them in G.

A graph may have several strongly connected components, and it may be the case that there are paths linking a vertex from one to a vertex from the other. In this case one says that the components are linked by some connection, therefore one can define a graph of the strongly connected components of some graph *G* (see Fig. 1). Denote this graph by SCC(*G*). Since each strongly connected component of *G* has at least a vertex, the size of SCC(*G*) is less than that of *G*. SCC(*G*) is also a directed graph, and one finds that, if two vertices from SCC(*G*) have paths between them in both directions, or in other words if a cycle exists, then the strongly connected components that they represent can in fact be merged into a single one. Updating SCC(*G*) as *G* is changed can be done by applying algorithms based on topological sort methods [17].

For motion planning, two kinds of graphs are important. One is referred to as "the roadmap" of the free space, and its vertices are allowed configurations (or subspaces of allowed configurations) while its edges correspond to non-obstacle-colliding trajectories between these configurations generated by some "local planning" procedure. The other kind of graph important for visibility based planners is the strongly connected component graph of the roadmap or its subgraphs.

B. Sample-based planners

Sampling planners construct a graph called a roadmap to represent the connectivity of the manipulator or mobile robot's free space (the part of configuration space without obstacle overlaps) by running a sequence of sample-and-connect steps [18]. Each step involves generating a random, usually uniformly distributed, sample in the free space of the robot, then attempting to connect it to samples already existing in a graph called the roadmap by some "simple" local planner procedure. Here, "simple" means the local planner is not mindful of obstacles when generating trajectories [19-20]. These simple trajectories are included in the roadmap only if they are free of obstacle collisions. Several variations of the above framework exist; several sampling [21-22], and connection heuristics [23-24] have been proposed. Once a roadmap for the configuration space is constructed, solving point-to-point planning problems reduces to a path search in the roadmap: the planner first attempts to connect the start and goal configurations to the roadmap via simple trajectories, then searches for a path between the two new vertices.

Note that there is a difference between a planning problem having solutions and the planning problem having solutions in a roadmap. A planning problem has solutions if there exist paths in free space between the start and goal; it may be the case that none of these paths are also in the roadmap however. Therefore sampling based planners do not guarantee that, if a planning problem has solutions, the planner will find one of them. However, they offer a weaker guarantee called probabilistic completeness: as the number of samples and connections stored in the roadmap increases to infinity, then if a planning problem has solutions in free space, the chance it also has a solution in the roadmap goes to one.

In practice, roadmaps can offer a good chance to find solutions even when a small number of samples is stored. In particular, the visibility heuristic of [9-10] is efficient at keeping roadmaps small as well as capable to solve planning queries. The heuristic works by filtering which samples generated in the sample-and-connect steps are stored in the roadmap; a new sample is kept only if it cannot be connected to other samples in the roadmap, therefore it has been placed in a region previously invisible to the roadmap and improves its coverage, or if it is a bridge between samples in different maximally connected components of the roadmap and improves connectivity. Therefore, the probability that a sample is accepted is related to the volume of free space that is either invisible to the roadmap or at the overlap of several connected components in the roadmap, which represents regions where the roadmap 'should' know of a path but doesn't yet. As the number of samples in the roadmap increases, this volume decreases to zero. One can use the rate at which samples are rejected to estimate the volume of freespace covered by the roadmap, and its overall chance to solve planning queries. As a consequence, the visibility heuristic is probabilistically complete for simple point-to-point queries [9], which here means that as the number of sample-and-connect steps used to build the roadmap increases to infinity, so does the chance to solve a planning query increase to one.

C. Temporal logic

Temporal logic is a catch-all term for formal languages designed to describe specifications about the behavior, in time, of dynamic systems. Several such logics exist, of varying expressive power. Computational Tree Logic, Linear Temporal Logic [2], and their superset CTL* [3] are examples of temporal logics, and so is the stronger μ-calculus [4]. Temporal logic first appeared for program verification [2], where it allowed one to reason about, and sometimes check whether certain properties hold, like a program eventually reaching some state, or avoiding dead-lock, or obtaining some useful property infinitely often. In this paper we will focus on a fragment of Linear Temporal Logic (LTL) that allows statements about path existence.

As with any formal language, LTL and its path existence fragment have an alphabet of symbols, syntactic rules for combining the symbols, and semantics to interpret them. The alphabet of symbols contains parenthesis, "true" and "false" symbols, the logical operators "$\neg$" (NOT), "$\wedge$" (AND), "$\vee$" (OR) and "U" (UNTIL), and atomic propositions which we will denote by small Latin letters. Atomic propositions are interpreted in a motion planning context as regions in space. For example, we might say region "p" is the region of configuration space where the robot touches a particular table. We will denote the set of all atomic propositions by Π. What this set contains may vary by application, depending on what regions of interest need to be defined in the robot's environment.

The syntax of path existence LTL is recursively defined as follows. An atomic proposition is a statement in path existence LTL, and so is its negation. The AND operator applied to two statements is a statement, and the same goes for OR and UNTIL. In Backus-Naur Form [25] this is described as:

$$\varphi ::= true \,|\, false \,|\, p \,|\, \neg p \,|\, \varphi \wedge \varphi \,|\, \varphi \vee \varphi \,|\, \varphi U \varphi \qquad (1)$$

where p is some element of Π. Since the syntactic structures above are defined recursively, one can use them to construct a syntactic tree for a statement, in which each node is a subformula (another statement in LTL). The root is the statement itself, and children of a node are the two subformulas appearing to the left and right of an operator. If a node is an atomic proposition it has no children, i.e. it is a leaf node.

Note that we only allow negation to appear in front of atomic propositions. As will become clear from the operator semantics, this limits the possible statements to that describing path existence, which is not uncommon in a planning context [7], for otherwise one could write statements that impose conditions on all possible paths from a state. However, if only one agent, the planner, is doing the choosing, then it only makes sense to talk about path existence since if a path exists, the planner may choose it and solve the problem.

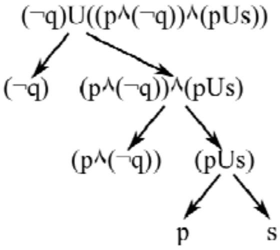

Figure 2. Syntactic tree of an LTL formula where each node represents a subformula. The tree is grown until the locally checkable subformulas are reached as leaves.

We will now describe how to interpret statements in path existence LTL as describing trajectories of a robot in its configuration space. We will say a statement φ holds at a point **x** in configuration space, and denote this by $\{\varphi\}@\mathbf{x}$, if there is a path starting at **x** over which the property described by the statement φ is true. The statement "true" holds at any point; the statement "false" holds nowhere.

An atomic propositions holds at points inside the region associated to it. For example, if p represents the surface of a table, for any point **x** on that table's surface we will have $\{p\}@\mathbf{x}$. Conversely, for any point not on that table's surface we will have $\{\neg p\}@\mathbf{x}$. The AND and OR operators have their usual meaning from propositional calculus; given two statements φ_1, φ_2 in LTL, then $\{\varphi_1 \wedge \varphi_2\}@\mathbf{x}$ if both $\{\varphi_1\}@\mathbf{x}$ and $\{\varphi_2\}@\mathbf{x}$, and $\{\varphi_1 \vee \varphi_2\}@\mathbf{x}$ if either $\{\varphi_1\}@\mathbf{x}$ or $\{\varphi_2\}@\mathbf{x}$. If a formula does not contain the UNTIL operator, then it is locally checkable: it can be determined whether it holds at a point in configuration space just by knowing the position of that point.

The interpretation of the UNTIL operator requires thinking about possible paths, not just positions in space. Let then φ_1 and φ_2 be statements in path existence LTL, and let $\boldsymbol{W}$ be the set of points in configuration space at which φ_2 holds. Then, $\{\varphi_1 U \varphi_2\}@\mathbf{x}$ if there exists a path starting at **x** and ending at some point in $\boldsymbol{W}$, such that at every location along the path (including therefore **x** itself) φ_1 is true.

As an example, consider the atomic proposition q as representing an obstacle region, and g as representing a goal region. Then, the basic planning query of reaching the goal while avoiding the obstacles is given by $(\neg q)U(g)$, and finding whether a starting point **x** can reach the goal becomes determining whether $\{(\neg q)U(g)\}@\mathbf{x}$. This statement is not locally checkable because it cannot be verified just by knowing the position of **x**; it also needs information about how the regions in configuration space are connected.

Except for locally checkable statements, a planner will not have access to exact representations of the regions where a statement is true. Instead, it must construct approximate representations by building a roadmap, as well as keep track of what statements hold at the vertices and connections in its roadmap. There exists a tangible benefit for restricting to path existence specifications we refer to as the "More isn't less" lemma, encountered for example in [7] (lemma III.1 in that text). Intuitively, the lemma states that as we improve our knowledge of the dynamics of a system and the environment it inhabits by adding more samples to a roadmap, then the set of path existence LTL statements that hold at a sample in the roadmap will not decrease; or, conversely, the set of roadmap samples at which a path existence LTL statement holds will never decrease as we add more samples to a roadmap. The lemma is useful in establishing why a planner which explores more of the configuration space will only become more able to solve planning queries, not less, at least as long as the queries are of the path existence type. We refer to [7] for its exact statement and a proof which easily translates to path existence LTL.

III. PLANNER ALGORITHM

A. Algorithm description

The visibility heuristic [9-10] accepts a new sample into a roadmap M if doing so will change the graph of connected components of the roadmap. If **x** is the new sample, and M⊕**x** is the roadmap after **x** and its possible connections is added to M, let us state the visibility heuristic symbolically by saying a sample **x** is accepted if SCC(M) and SCC(M⊕**x**) are different: either the number of connected components increases by one, because the new sample cannot be connected to other roadmap samples, or decreases by at least one, because the new sample allows cycles between vertices in different connected components, or a new one-way connection appears between two previously unconnected components. As stated in the Introduction section, this heuristic is not enough for handling queries in temporal logic because it does not use information about the regions represented by atomic propositions.

We therefore amend the visibility heuristic to consider not just the connected components of the roadmap as a whole, but also the connected components of subgraphs of the roadmap, where each subgraph is associated to a temporal logic statement. Let $[\varphi](M)$ be the subgraph of M that contains all vertices at which φ holds, and all edges for which φ holds at every point. Our heuristic then is: given a set of temporal logic statements $\mathcal{L}$ and a roadmap M, a new sample **x** is added to the roadmap if there exists $\varphi \in \mathcal{L}$ such that SCC($[\varphi]$(M)) and SCC($[\varphi]$(M$\oplus$**x**)) are different. We will next show how to construct $[\varphi]$(M).

Suppose we have an LTL statement φ for which we want to design a planner. The planner should construct a roadmap for the environment and then, when given a query point **x**, find a path that starts at **x** and satisfies the formula φ, if such a path exists. We first construct the syntactic tree of subformulas of φ; and in order to minimize the tree size, we allow locally checkable formulas to be leaves. Then, the set $\mathcal{L}$ contains the nodes in the syntactic tree. Belonging to a subgraph can be flagged by a binary value and does not require several copies of the roadmap to exist. We will denote by $\mathcal{L}_{\text{local}}$ the subset of $\mathcal{L}$ which contains locally checkable formulas.

We also need to define how to construct $[\upsilon]$(M) for some statement υ. In the case of vertices and locally checkable formulas, this is obvious as by definition locally checkable formulas are made of conjunctions and disjunctions of atomic propositions, which can be verified knowing only a position in configuration space. Verification of locally checkable formulas along an edge is analogous to collision checks along the edge in usual planners. If all along an edge, the same locally checkable formulas hold, then that edge is also a member of whatever locally checkable subgraphs its endpoints are members of. If instead one finds that along an edge there are regions where different locally checkable formulas hold, then one can cut it by either generating new vertices at the points of contact between regions, or inside each region. The resulting edges will then be bridge edges between locally checkable subgraphs.

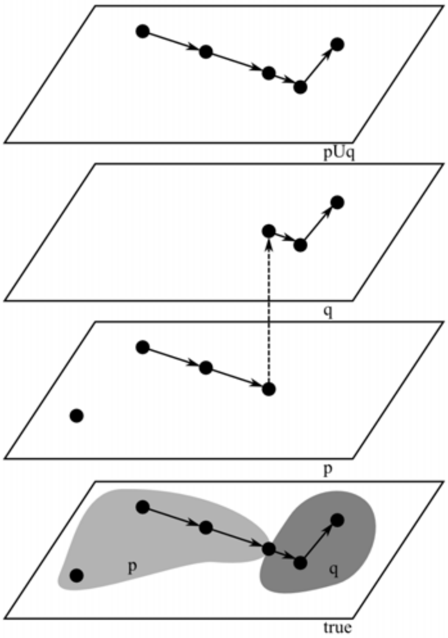

Figure 3. Roadmap and subgraphs associated to subformulas of a specification. The roadmap in its entirety corresponds to the "true" formula. Nodes are generated at the interface of the p and q regions, and the edge between them is "between" subgraphs.

Suppose then that υ_1 holds at a vertex, or respectively along an edge. Then that vertex, or respectively edge, is added to $[\upsilon_1 \vee \upsilon_2]$(M) formula (if it exists in $\mathcal{L}$).

Supposing that both υ_1 and υ_2 hold at a vertex, or along an edge, then that vertex (or edge) is added to $[\upsilon_1 \wedge \upsilon_2]$(M) (if it exists in $\mathcal{L}$).

If υ_2 holds at a vertex or all along an edge, then that vertex (or edge) is added to $[\upsilon_1 U \upsilon_2]$(M), if it exists in $\mathcal{L}$. Then, one can recursively add vertices from $[\upsilon_1]$(M) to $[\upsilon_1 U \upsilon_2]$(M), if they connect via an edge where υ_1 holds everywhere (except maybe in a region around the destination where υ_2 holds) to a vertex already in $[\upsilon_1 U \upsilon_2]$(M); the edges used for connection are also added to the $[\upsilon_1 U \upsilon_2]$(M).

An observation should be made here. The entire roadmap M is not the set of samples at which the top level specification φ holds. Rather, one would say that M is [true](M), the set of vertices and edges of M at which "true" holds.

Similar to the classic visibility heuristic [9], our proposed version allows an estimation of the coverage of the subspace corresponding to a statement υ. By definition of uniform sampling over a space, we have that the probability to place a sample in some region A of a space S is

$$p(A) = \frac{vol(A)}{vol(S)} \qquad (2)$$

where vol(A) is the volume of A (or, in general, some suitable measure function). However, we also have

$$p(A) = \frac{n(A)}{n(S)} \qquad (3)$$

where n(A) is the number of samples taken in region A.

Suppose then that, at some point during execution, we wish to estimate the volume not yet covered by the roadmap for some subformula υ. For the purpose of this estimation, we set n(υ) to 0 and increment it for each new sample that we generate in the υ subspace during sample-and-connect steps. If the heuristic decides the sample is worth keeping, then we also increment n(υ_{u}), the number of samples taken outside the current coverage of the roadmap. After several sample and connect steps, we may say that

$$\frac{vol(\upsilon_u)}{vol(\upsilon)} \approx \frac{n(\upsilon_u)}{n(\upsilon)} \qquad (4)$$

or in other words that the ratio of the uncovered volume to the total volume is similar to the ratio of samples accepted to the roadmap to the total number of samples taken.

While not something that can be verified to mathematical certainty, this estimation does have an intuitive justification. If the volume not yet covered by the roadmap is, for example, a quarter of the total volume of space, then one would expect that a quarter of a number of random uniformly distributed samples would be in the uncovered volume. The visibility heuristic would be able to determine whether a sample is inside the covered volume or not, because a sample that's inside the covered volume is connectable to samples already in the roadmap.

Such estimations of coverage are useful especially when several regions of space are considered, as is the case for LTL formulas. If one finds that the uncovered volume of some regions is large, then one can prioritize sampling to those regions, to speed up roadmap construction.

It will often be the case that sets of points where formulas hold are compact [26]. This is a property describing how well-behaved sets are, from a topological standpoint, and implies that a set can be covered by a finite collection of open sets. This is necessary so that we can have a finite roadmap covering the configuration space.

B. Changing the start configuration

In the previous section, we considered that the start configuration is known from the start and added to the roadmap. Should we desire to change it however, one simply adds the new start configuration to the roadmap, and thereafter proceeds with sample-and-connect steps, using the algorithm described before. If the new start is inside the subspace corresponding to the specification, it will eventually be added to the subgraph corresponding to that subspace of C_{free}.

C. Extracting a plan from the roadmap

Checking that a plan exists is made very easy by the process of roadmap construction. If the starting vertex is inside the subgraph corresponding to the specification, then a plan exists; otherwise, it does not.

Once a roadmap is constructed, the issue remains to extract a plan from it, if one exists, that meets a given specification. Tools for general LTL formulas exist, like NuSMV [14] and SPIN [15], which have also been used in a planning context [5]. They check a formula by providing a counter-example to its negation, if such a counter-example can be found, and if it can be, the counter-example is the sought after plan.

SPIN and NuSMV are capable of handling general LTL formulas, outside the subset of interest to this paper. Restricting to the subset of "existence of paths" formulas allows one to work with a formula directly, instead of requiring its negation, so we present a plan finding procedure specialized to this subset of LTL and which makes use of the auxiliary structures maintained by the planning algorithm.

The basic planning query, (true)Uq where q is some locally checkable statement, is typically done with a Dijkstra's shortest paths algorithm or A*, which provides a distance map over the vertices in the roadmap. Each vertex has associated with it the smallest cost required to reach it from some given starting vertex. Based on the distance map, the shortest path between the given starting vertex, and any other vertex in the graph, can be obtained. Note that the path search algorithms are applicable to a planning problem only after a roadmap has been constructed, for example by a visibility heuristic. Also, for more general queries in temporal logic, shortest path search algorithms may not be sufficient. If we require the path to pass through a certain region of the configuration space, it is not necessarily the shortest path between start and goal.

We can however construct a path that obeys some given temporal logic statement φ by concatenating segments produced by shortest path searches in subgraphs of the form $[\upsilon](M)$ where υ is some statement in $\mathcal{L}$, the set of statements known by the planner which includes the statement φ and all its subformulas. For example, to solve a query of the form pUq, the planner will run a shortest path search inside [p](M), where the destinations are samples that are also in $[p \wedge q](M)$. Of the samples in $[p \wedge q](M)$, those which are connected to the starting vertex by short paths that do not go through other samples in $[p \wedge q](M)$ are called a "front".

We can now define a function, FindFront, which takes as input a start configuration, and an LTL formula representing a specification on paths. It will return a list of front vertices and the paths to them from the starting vertex. We will now specify the behavior of FindFront in more detail.

If the starting vertex is not in the subgraph corresponding to the LTL formula, then the function returns one path, containing just the starting vertex, which is said to have infinite cost. We therefore have a quick test to check whether searching for a plan is fruitless because none exists.

If the LTL formula is locally checkable, and is obeyed at the starting vertex, then FindFront returns a path containing just the starting vertex, of cost *0*.

If the LTL formula is of the $\upsilon_1 U \upsilon_2$ type, then the function will restrict itself to $[\upsilon_1 U \upsilon_2](M)$. Assuming the starting vertex can be found in this subgraph (or else, an infinite cost path containing just the start vertex would have been returned), FindFront will run a Dijkstra algorithm and locate the υ_2 front vertices and the paths toward them. After that, for each front vertex **y**, a new instance of FindFront is called, with **y** as start vertex and υ_2 as the formula. The return value of the upper level of the recursion is then the set of paths obtained by concatenating, to the paths to each front vertex **y**, the paths obtained for that vertex by the lower level of the FindFront recursion.

If the LTL formula is of the $\upsilon_1 \wedge \upsilon_2$ type, where one of the formulas, say υ_1, is locally checkable, then FindFront first checks that the starting vertex is inside $[\upsilon_1 \wedge \upsilon_2](M)$. If it is (which implies that υ_1 also holds at it), then another instance of FindFront is called with the same starting vertex and υ_2 as the formula. The return value for the upper level of the recursion is then the return value from the lower level one.

If the LTL formula is of the $\upsilon_1 \wedge \upsilon_2$ type, where neither formula is locally checkable, then one should first use rewrite rules to bring it to a form in which the $\wedge$ operator always has at least one locally checkable formula as operand. Some useful rewriting rules are summarized in Table I; notice that the rewrite rules tend to shorten the formulas appearing as operands to the $\wedge$ operator, and therefore eventually we will only apply it to operands out of which at least one is locally checkable.

Table I: Some LTL formula rewrite rules. Formulas on the right simplify the operands of the conjunction operators.

LTL Formula	**Equivalent rewrite**
$(\varphi_1 \vee \varphi_2) \wedge \varphi_3$	$(\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3)$
$\varphi_1 \wedge (\varphi_2 U \varphi_1)$	φ_1
$\varphi_1 \vee (\varphi_2 U \varphi_1)$	$(\varphi_2 U \varphi_1)$
$\varphi_1 U (\varphi_1 U \varphi_2)$	$(\varphi_1 U \varphi_2)$
$(\varphi_1 U \varphi_2) \wedge (\varphi_3 U \varphi_4)$	$(\varphi_1 \wedge \varphi_3) U (\varphi_2 \wedge (\varphi_3 U \varphi_4)) \vee$ $(\varphi_1 \wedge \varphi_3) U (\varphi_4 \wedge (\varphi_1 U \varphi_2))$

Finally, if the LTL formula is of the $\upsilon_1 \vee \upsilon_2$ type, then two instances of FindFront are called, both with the same start vertex, but one with the υ_1 and the other with the υ_2 formula. The return value of the upper level recursion is the union of the return values of the lower level recursion.

Looking for a plan then requires that a FindFront be called, with the starting vertex and plan specification. From the resulting set of paths, one can pick the lowest cost one as the plan to follow when solving the planning specification.

The above is also given in pseudocode in Fig. 4. For ease of illustration, some simple auxiliary functions are invoked. LastInPath takes a path as an argument and returns the last vertex in the path. Count is a function that takes a path and an LTL formula υ as parameters, and returns how many vertices from the path are also flagged as belonging to $[\upsilon](M)$. MergePaths takes two paths as parameters and returns the path which results from appending the second to the first. Finally, DijkstraPaths takes a starting vertex q, a subgraph $[\upsilon_1 U \upsilon_2](M)$ and a formula υ as parameters, and returns the shortest paths that start at q, end on a vertex inside $[\upsilon](M)$, and use only vertices and edges in $[\upsilon_1 U \upsilon_2](M)$.

```
FindFront ($q$, $\varphi$):
if $q \notin [\varphi](M)$ then
  return $\{\{q; \infty\}\}$
else if "U" $\notin \varphi$ then
  return $\{\{q; 0\}\}$
else if $\varphi \cong \phi_1 \vee \phi_2$ then
  $A \leftarrow$ FindFront($q$, $\phi_1$)
  $B \leftarrow$ FindFront($q$, $\phi_2$)
  return $A \cup B$
else if $\varphi \cong \phi_1 \wedge \phi_2$ then
  if "U" $\notin \phi_1$ then
    return FindFront($q$, $\phi_2$)
  else if "U" $\notin \phi_2$ then
    return FindFront($q$, $\phi_1$)
else if $\varphi \cong \phi_1 U \phi_2$ then
  $FP \leftarrow$ DijkstraPaths($q$, $[\phi_1 U \phi_2](M)$, $\phi_2$)
  for $p \in FP$ do
    if $1 <$ Count($p$, $\phi_2$) then
      $FP \leftarrow FP - p$
  $A \leftarrow \emptyset$
  for $p \in FP$ do
    $S \leftarrow$ FindFront(LastInPath($p$), $\phi_2$)
    for $s \in S$ do
      $A \leftarrow A \cup$ {MergePaths($p$, $s$)}
  return $A$
```

Figure 4. The FindFront function.

IV. SIMULATION VERIFICATION

In this section we present two tests for the new planning method. The first compares it to previously existing algorithms like PRM [19], the classical visibility heuristic [9], and RRG [7] on a constrained path finding problem for a mobile robot; we present the problem and the results in subsections A and B. We then apply our method to a loop finding problem that was used in [7] to showcase RRG; we present the problem, customize our heuristic for loop creation, and present the results in the following subsections C, D, and E.

A. Constrained path: problem specification

The problem is to navigate a point robot between two configurations in a maze shown in Fig. 5 a). However, two regions are defined in this maze: a region 'p' shown in Fig. 5 b), and a region 'q' shown in Fig. 5 c).

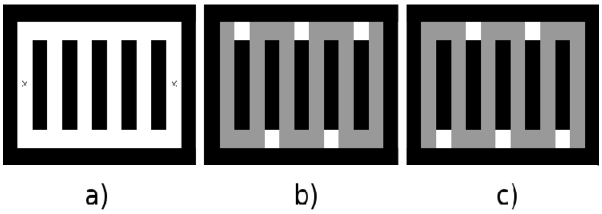

Figure 5. Problem environment for the constrained path problem (a); the 'p' region, marked in gray (b); the 'q' region, marked in gray (c).

The problem is to find a path between the two points marked with crosses in Fig. 5 a); however, the path is required to stay inside either the p or the q region until the destination is reached. Let 'd' be a region associated to the destination, then the LTL specification for the problem is

$$\varphi = (pUd) \vee (qUd) \tag{5}$$

B. Constrained path: simulation results

We compare our algorithm to RRG [7] (another algorithm capable to handle LTL specifications) as well as to classical planners like PRM [19] and the first visibility heuristic [9]. We run each algorithm for 1000 times until the roadmap produced by the algorithm contains a path in free space (the union of p and q regions) from source to destination, which is easy to check using a connected components structure using Tarjan's set union algorithm [16] which runs in 'almost constant' time (running time proportional to inverse Ackermann function of the number of vertices, and therefore grows very slowly).

Once an algorithm produces a roadmap where some path exists between source and destination, we run a Dijkstra shortest path algorithm to find the shortest path between source and destination, and we then check whether that path always stays inside the p, or always inside the q region; as expected, a shortest path search will not find a path that answers the posed problem. We also run path searches on the subformula subgraphs to see whether a planning

TABLE II: SOLUTION STATISTICS FOR 1000 RUNS ON THE CONSTRAINED PATH SEARCH PROBLEM.

Planner	**Avg. time (s)**	**Stdev time (s)**	**Avg. vertices**	**Stdev vertices**	**Successes (Dijkstra path search)**	**Successes (subgraph path search)**
PRM	0.001	0.0011	28.02	16.67	0	38
Visibility	0.0007	0.0005	9.82	2.1	0	0
RRG	0.0058	0.0038	77.61	24.82	0	1000
Our method	0.003	0.0015	18.99	2.74	0	1000

algorithm produced a roadmap that does contain a path which answers the posed problem, even if a simple path search was not able to find it.

We collect execution times, sample counts, and success rates for the planners and path search methods, which we summarize in Table II.

As can be seen from the table, both the classical visibility planner and our own method are able to connect the source and destination faster and with fewer vertices kept in the roadmap. However, the roadmap produced by the classical visibility planner doesn't contain a path that is either all inside the p region or all inside the q region, which strongly suggests the classical visibility heuristic would not be capable to handle general path-existence LTL specifications for mobile robots since it cannot handle even a fairly simple one as in this test case. The classical visibility heuristic aggressively prunes the samples accepted in a roadmap, and only consider the connectivity of the entire configuration space, not that of regions of it.

Searching for a path that answers the posed problem cannot be done with a simple shortest path search; the search needs to be informed of the subgraph structure induced by the problem specification, which is what we propose in this paper, or else the path search is not actually aware of the problem specification. If the path search does use the structure we propose, then it is always successful for roadmaps produced either by RRG or by our method. However, our method is twice as fast as RRG and produces roadmaps with about a quarter of the vertices needed by RRG.

A roadmap with fewer vertices that nonetheless manages to capture the connectivity of the environment is useful because it allows faster queries, as well as faster post-processing of the roadmap, should the environment change. Several schemes have been proposed in the literature to handle changing environments, for example vertex displacements that mimic a network of elastic bands pushed away by the moving obstacles [27] or vertex cost adjustments when obstacles are detected near roadmap vertices [28]. Having fewer vertices to consider for such processing is an advantage.

It should be noted that a path search on subgraphs will sometimes succeed for roadmaps produced by PRM, because PRM tends to use 'more samples' than necessary to connect two vertices and may, by chance, capture the connectivity of one of the regions p or q, including an entire path inside one of these regions. However, the rate of success is very low (less than 5% for our test case) which suggests that for LTL task specifications, one needs to use special planning algorithms like RRG or the method we proposed here.

C. *Loop construction: problem specification*

We apply the planner to the problem used for simulation verification of RRG in [7], which asks for a discrete time linear dynamic system in a two-dimensional configuration space to be steered towards a looping trajectory that passes through two specified regions while avoiding a third. The system is characterized by the following equations of state (a kinematic model, as no inertia is present):

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \mathbf{A} \cdot \begin{bmatrix} x_k \\ y_k \end{bmatrix} + \mathbf{B} \cdot \begin{bmatrix} u_k \\ v_k \end{bmatrix} \tag{6}$$

where

$$\mathbf{A} = \begin{bmatrix} 1.019 & -0.029 \\ 0.049 & 0.95 \end{bmatrix} \tag{7}$$

$$\mathbf{B} = \begin{bmatrix} 0.101 & -0.0015 \\ 0.0025 & 0.098 \end{bmatrix} \tag{8}$$

from which it is straightforward to define a local steering procedure between arbitrary positions. The local steering procedure will assume obstacles are not in the way, and the trajectories it produces must be checked for validity, as is typical for sampling based planners. Valid trajectories are kept in the roadmap. Note further that the system is fully reversible so we can use an unoriented graph for the roadmap.

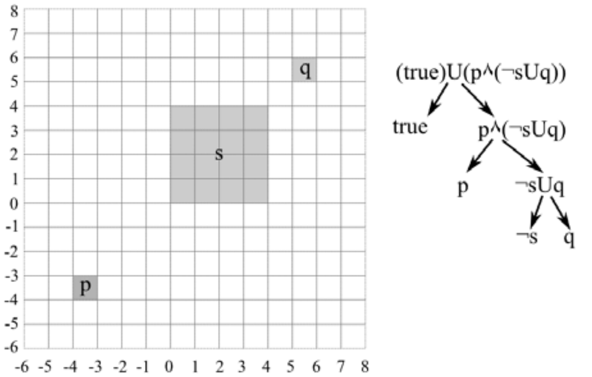

Figure 6. Left, the problem environment with special regions marked in gray. The specification and its syntactic tree are shown on the right.

The environment is shown in Fig. 6. The system starts at *(0, 0)*, on the edge of the *s* region. We require that it reach the *p* region, then the *q* region while avoiding *s*, then the *p* region again while avoiding *s*. We then formulate the LTL specification:

$$\varphi = (true)U(p \wedge ((\neg s)U(q))) \tag{9}$$

Note that, while the specification above produces a finite path, the problem in [7] requires a loop to be formed between *p* and *q*, which avoids *s*. To close the loop, notice that the system's reversibility allows the *s*-free path from *p* to *q* to be used in reverse. Were this not the case, then a different LTL specification would have required a path from *p* to *q*, then from *q* to *p*, and a gap reduction step would have been necessary to close the loop by linking the path endpoints in *p*.

We compare our planner's performance in terms of roadmap size and execution speed with that reported in [7] for the RRG algorithm. The reason we compare to RRG in this test is the fact that it, like our planner, is capable to handle queries in temporal logic. Point-to-point planners like PRM or RRT would not be able to solve such problems and are not fit for comparison. It should also be said that path search algorithms like A* or Dijsktra are not good comparisons either; a path search algorithm needs to have a graph to search, and the construction of that graph is the domain of the planner heuristic we propose here. Also, shortest path search algorithms on their own cannot find paths that obey temporal logic statements because such paths are not necessarily the shortest in the roadmap; such algorithms must be deployed in a FindFront procedure as described in section III.D to search among those paths that obey a temporal logic statement, as was shown in subsections IV-A and IV-B .

D. Loop construction: customized visibility heuristic

We will first make an inventory of the distinct formulas that we need to track for the given specification: $true$, p, q, $\neg s$, $(\neg s)U(q)$, $p\wedge((\neg s)U(q))$, and finally the specification itself, $(true)U(p\wedge((\neg s)U(q)))$. Each of these will have a subgraph in the roadmap to represent it; vertices and edges inside $[\upsilon](M)$ satisfy υ, meaning a path exists which starts at the vertex, or the point on the edge, and satisfies υ, where υ is some formula in the list given above.

We maintain connected components for each $[\upsilon](M)$ using Tarjan's set union algorithm [16]. If a new sample changes the graphs of strongly connected components or the connections between formula subgraphs, it is kept in the roadmap. Sampling for this problem will be uniform on the problem area.

E. Loop construction: simulation results

We run our tests on an Intel Pentium 4 processor with 3.6GHz clock frequency; we use a single thread for program execution (no parallelism). One thousand runs of the algorithm are performed, and statistics on final roadmap size and rejected sample counts are shown in Table III. As can be seen, the size of the roadmap is reliably small, as an average of nine samples is sufficient to find a suitable path. In comparison, RRG used more than 1000 samples for the same problem [7]. We compare with RRG because it, like our planner, is capable to handle queries in temporal logic, whereas point-to-point planners like PRM or the classical visibility are not.

One notices that the average number of rejected samples is around the same order of magnitude as the number of samples used by the RRG, suggesting that a uniformly sampling planner needs a few hundred attempts in order to pick some samples useful for a solution for the test problem considered here. The visibility based planner however can determine that most of the random samples taken do not improve the roadmap's ability to solve the problem, and instead only selects a much smaller set to keep for future use. In this test, the roadmap used by our method is about thirty-six times smaller than that used by RRG.

In terms of time spent, whereas RRG requires several seconds (3.5s on average) [7], our visibility based planner needs an average of 0.33s. Although roughly the same number of sample and connect steps are made, the fact that the roadmap is kept dramatically smaller makes each of these steps much less expensive as near-neighbor queries and other auxiliary steps for sample based planners will require less time.

TABLE III: STATISTICS FOR THE KEPT AND REJECTED SAMPLE COUNTS OVER 1000 RUNS.

Samples	Avg.	StdDev.	Max.	Min.
Kept	8.84	1.63	12	5
Rejected	368.6	257.43	1445	5

V. CONCLUSIONS AND FUTURE WORK

We have presented an extension of the visibility heuristic which is applicable to planning problems given as specifications in a path-existence subset of temporal logic, and defined a planning method to handle such specifications by using our proposed heuristic. These kinds of planning problems are relevant in task specifications for mobile robots and manipulators, as they allow a tighter integration between a symbolic level of task planning and the geometric level where motion planning actually occurs. Temporal logic capable motion planners can reason about geometric feasibility of subsequence tasks directly, and thus allow easier handling of branching and sequencing, and verification of candidate sequences of tasks.

It should be noted that while the kinds of specifications our method can natively handle are about existence of finite and open paths, it may also be useful in some cases where the existence of an infinite loop is sought. To form a loop, one would need to get a path from some starting point, to a destination, then back inside the region of the starting point, and then use some gap closing procedure to close the loop, if the system that planning is done for has non-reversible maneuvers. For a system with reversible maneuvers, producing loops is trivial.

The formulation of the planning problem used here assumed perfect actuation of the mobile robot. Probabilistic temporal logics exist which account for errors in motion [29-30], and it may be possible to extend the strategies presented here for planning specifications written in such formal systems. Other methods for obtaining sparse roadmaps besides visibility exist, in particular methods which soften the visibility heuristic and aim for some guarantee of partial optimality [11-13]. It may be fruitful to apply the subgraph constructions presented here to such methods, so that asymptotically near optimal, sparse roadmaps are made possible for general path existence temporal logics specifications, which would be useful in contexts like grasping, manipulation, and task planning. Both of the previous topics are left for future work.

ACKNOWLEDGMENT

I would like to thank my thesis supervisor, prof. dr. ing. Virgil Tiponuţ, for his help and guidance during the entirety of my doctoral studies at the Politehnica University of Timişoara.

REFERENCES

[1] I. A. Sucan, S. Chitta, “Motion planning with constraints using configuration space approximations”, in proceedings of the IEEE International Conference on Robotics Systems (IROS), 2012, doi: 10.1109/IROS.2012.6386092.

[2] A. Pnueli, “The temporal logic of programs”, in proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS), 1977, doi: 10.1109/SFCS.1977.32 .

[3] E. A. Emerson, J. Y. Halpern, “'sometimes' and 'not never' revisited: on branching versus linear time temporal logic”, journal of the ACM, vol. 33 nr. 1, pg. 151–178, 1986, doi: 10.1145/4904.4999.

[4] D. Kozen, “Results on the propositional μ-calculus”, in proceedings of the 9th Colloquium on Automata, Languages and Programming, pg. 348–359, 1982, doi: 10.1016/0304-3975(82)90125-6.

[5] G. E. Fainekos, H. Kress-gazit, G. J. Pappas, “Temporal logic motion planning for mobile robots”, in proceedings of the 2005 IEEE

International Conference on Robotics and Automation (ICRA), pg. 2020–2025, 2005, doi: 10.1109/ROBOT.2005.1570410.

[6] G. E. Fainekos, H. Kress-gazit, “Hybrid controllers for path planning: A temporal logic approach”, in proceedings of the IEEE Conference on Decision and Control (CDC), pg. 4885-4890, 2005, doi: 10.1109/CDC.2005.1582935 .

[7] S. Karaman, E. Frazzoli, “Sampling-based motion planning with deterministic μ-calculus specifications”, in proceedings of the IEEE Conference on Decision and Control (CDC), 2009, doi: 10.1109/CDC.2009.5400278 .

[8] John F. Canny, “The complexity of robot motion planning”, PhD thesis, MIT Press, pp. 1-20, 1987, ISBN: 9780262031363.

[9] C. Nissoux, “Visibilité et méthodes probabilistes pour la planification de mouvement en robotique” (PhD thesis), University Paul Sabatier Toulouse, 1999.

[10] T. Simeon, J.-P. Laumond, C. Nissoux, “Visibility-based probabilistic roadmaps”, in proceedings of the IEEE/RSJ International Conference on Robots and Systems (IROS), 1999, doi: 10.1109/IROS.1999.811662.

[11] J. D. Marble, K. Bekris, “Computing spanners of asymptotically optimal probabilistic roadmaps”, in proceedings of the IEEE/RSJ International Conference on Robots and Systems (IROS), 2011, doi: 10.1109/IROS.2011.6095070.

[12] J. D. Marble, K. Bekris, “Asymptotically near-optimal is good enough for motion planning”, in proceedings of the International Symposium of Robotics Research (ISRR), 2011, doi: 10.1109/IROS.2011.6095070 .

[13] D. Nieuwenhuisen and M. H. Overmars, “Useful cycles in probabilistic roadmap graphs”, in proceedings of IEEE International Conference on Robotics and Automation (ICRA), 2004, doi: 10.1109/ROBOT.2004.1307190 .

[14] A Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, “NuSMV 2: an OpenSource tool for symbolic model checking”, in proceedings of International Conference on Computer-Aided Verification (CAV), 2002, doi: 10.1007/3-540-45657-0_29.

[15] G.J. Holzmann, “The SPIN model checker primer and reference manual”, Addison-Wesley, pp. 167-190, 2004, ISBN: 978-0321773715.

[16] R. E. Tarjan, “Edge-disjoint spanning trees and depth-first search”, Acta Informatica, vol. 6 nr. 2, pg. 171–185, 1976, doi: 10.1007/BF00268499.

[17] B. Haeupler, T. Kavitha, R. Mathew, S. Sen, R. E. Tarjan, “Incremental cycle detection, topological ordering, and strong component maintenance”, ACM Transactions on Algorithms, vol. 8 nr. 1, 2012, doi: 10.1145/2071379.2071382.

[18] S. M. LaValle, “Planning Algorithms”, Cambridge University Press, pp. 185-247, 2006. Available online at: http://planning.cs.uiuc.edu/

[19] L. E. Kavraki, P. Svestka, J.-C. Latombe, M. H. Overmars, “Probabilistic roadmaps for path planning in high-dimensional configurations spaces”, IEEE Transactions on Robotics and Automation, vol. 12, nr. 4, 1996, doi: 10.1109/70.508439 .

[20] J. J. Kuffner, S. M. LaValle, “RRT-connect: an efficient approach to single-query path planning”, in proceedings of the IEEE International Conference on Robotics and Automation (ICRA), 2000, doi: 10.1109/ROBOT.2000.844730 .

[21] A. Yershova, L. Jaillet, T. Simeon, S. M. LaValle, “Dynamic domain RRTs: efficient exploration by controlling the sampling domain”, in proceedings of IEEE International Conference on Robotics and Automation (ICRA), 2005, doi: 10.1109/ROBOT.2005.1570709 .

[22] S. R. Lindemann, S. M. LaValle, “Steps toward derandomizing RRTs”, in proceedings of the 4th International Workshop on Robot Motion and Control (RoMoCon), 2004, doi: 10.1109/ROMOCO.2004.240739 .

[23] Sertac Karaman, Emilio Frazzoli, “Incremental sampling-based algorithms for optimal motion planning”, in proceedings of Robotics: Science and Systems (RSS), 2010. Available online at: http://www.roboticsproceedings.org/rss06/p34.pdf

[24] Sertac Karaman, Emilio Frazzoli, “Incremental sampling-based algorithms for motion planning”, 2010. Available online at: http://arxiv.org/abs/1005.0416

[25] D. Grune, C. J. H. Jacobs, “Parsing techniques- a practical guide”, Springer, pp. 5-60, 2008, ISBN: 978-0387202488.

[26] Andrew H. Wallace, “An Introduction to Algebraic Topology”, Dover Books on Mathematics, Dover Publications, pp. 50-54, 2007, ISBN: 978-0486457864.

[27] R. Gayle, A. Sud, M. C. Lin, D. Manocha, “Reactive deformation roadmaps: motion planning of multiple robots in dynamic environments”, in proceedings of the IEEE/RSJ International Conference on Robots and Systems (IROS), 2007, doi: 10.1109/IROS.2007.4399287.

[28] M. Pomarlan, I. A. Sucan, “Motion planning for manipulators in dynamically changing environments using real-time mapping of free space”, in proceedings of the 14th IEEE International Symposium on Computational Intelligence and Informatics (CINTI), 2013, doi: 10.1109/CINTI.2013.6705245.

[29] M. Lahijanian, J. Wasniewski, S. B. Andersson, C. Belta, “Motion planning and control from temporal logic specifications with probabilistic satisfaction guarantees”, in proceedings of IEEE International Conference on Robotics and Automation (ICRA), 2010, doi: 10.1109/ROBOT.2010.5509686.

[30] I. Cizelj, C. Belta, “Control of noisy differential-drive vehicles from time-bounded temporal logic specifications”, CoRR, abs/1209.1139, 2012, available online at http://arxiv.org/pdf/1209.1139v4, doi: 10.1109/ICRA.2013.6630847.