The 2010 IEEE/RSJ International Conference on
Intelligent Robots and Systems
October 18-22, 2010, Taipei, Taiwan

Using a string to map the world

Hui Wang, Michael Jenkin and Patrick Dymond

Abstract— **Literature and folklore is rife with a range of oracles that have been used by explorers to explore unknown environments. But how effective are these various oracles? This paper considers the power of string and string-like oracles to map an unknown embedded topological environment. We demonstrate that for undirected graphs, even very short strings can be used to explore an unknown environment but that significant performance improvements can be found when longer strings are available.**

I. Introduction

Robotic exploration and mapping is commonly referred to as *SLAM* or *Simultaneous Localization and Mapping* [8], [9], [5], and is considered to be a fundamental problem in robotics. While SLAM approaches can be classfied in various ways, one partitioning of the approaches is into those that use a topological representation and those which use a metric one. While metric representations capture the metric properties (e.g., Cartesian coordinates) of the environment, topological (graph-like) representations describe the *connectivity* of different places. Within a topological formalism, the goal of a SLAM algorithm is to construct a graph-like map representation that is isomorphic to the underlying world being explored.

The problem that lies at the core of SLAM algorithms is answering the question 'have I been here before?' This is also known as the 'loop closing' problem. (See [5] for different loop-closing approaches for metric representations.) One common approach to solving this problem in a graph-like world is to resort to the use of an additional device (an oracle) that can help the robot answer the question. Here 'oracle' refers to an aid or device that the robot can use to disambiguate locations during exploration. Different forms of oracles or aids have been employed in folk tales and literature, including pebbles, breadcrumbs, strings, and the like. (See [10] for a recent survey of the literature.) With an appropriate aid, the SLAM problem can be solved deterministically for embedded graph-like worlds. But how powerful must an oracle be in order to solve the SLAM problem and are there efficiencies to be found in using more powerful oracles? Motivated by work such as [7] that considers theoretical limits to particular classes of robots, this paper examines the power of various forms of string-like oracles in exploring topological environments[1]. We demonstrate that a very short string is sufficient to solve the SLAM problem for embedded topological worlds, and that sufficiently long strings can be used to solve the SLAM problem in time proportional to the size of the environment being mapped.

The authors are with the Department of Computer Science and Engineering, York University, 4700 Keele Street, Toronto, Ontario, Canada {huiwang,jenkin,dymond}@cse.yorku.ca

[1]For an early solution to partial mapping using a string-based aid see [1].

II. World and robot model

Environments in topological (graph-like) maps are typically represented as a set of significant places (vertices) connected via arcs (edges). Here we review a world model that has been adopted by a number of research efforts.

The world model Following [4] and [2] the world is modeled as an embedding of an undirected graph $G = (V, E)$ with a set of vertices $V(G) = \{v_1, ..., v_n\}$ and a set of edges $E(G) = \{(v_i, v_j)\}$. (In the following we denote the number of edges and vertices in G by $m = |E(G)|$ and $n = |V(G)|$ respectively.) The labels (if any) on vertices and edges of G are invisible to the robots, so that vertices and edges are not uniquely distinguishable to the robot. (In the literature this is referred to as an *unlabeled* or *anonymous* graph [6].) The graph is embedded within some space in order to permit relative directions to be defined on the edges incident upon a vertex. More formally, the definition of an edge within a graph is extended to allow for the explicit specification of the order of edges incident upon each vertex of the graph embedding. An edge $e = (v_i, v_j)$ incident upon vertices v_i and v_j is assigned two labels, one for each of v_i and v_j, representing the ordering of e with respect to the consistent local enumeration of edges at v_i and v_j, respectively. Note that as an unlabeled graph, the absolute edge ordering defined by the embedding is not accessible to the robot.

Robot motion and perception It is assumed that a robot can move from one vertex to another by traversing an edge. The robot can identify when it arrives at a vertex. The sensory information that the robot acquires at a vertex consists of *edge-related* perception and *string-related* perception. With *edge-related* perception, a robot can, by following the pre-defined ordering convention, determine the relative ordering of edges incident on the current vertex v_i in a consistent manner (e.g., by clockwise enumeration for a planar embedding). The robot can identify the edge through which it entered a vertex and assign a label to each edge in the vertex representing the current local edge ordering. Note that this local edge ordering is not, in general, equal to the absolute ordering specified by the embedding (which is not accessible to the robot), but rather is a permutation of it.

String-related operation and perception The robot is equipped with a string. The robot can manipulate the string in various ways. For example, it can 'tie' one end of the string at a vertex and play that string out as it moves through the graph, and perceive the string when encountering the string

at a given vertex. The complete set of the robot's potential operations and perception on the string are discussed later.

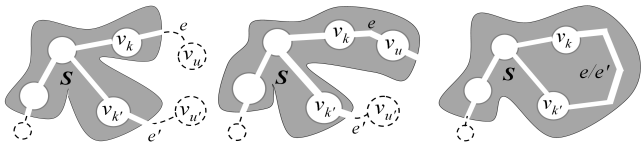
Memory The robot remembers all sensory information it has acquired and all of its actions. By "memorizing" a motion sequence and taking advantage of the local edge ordering, the robot can retrace any previously performed motion within the graph. Assume that the amount of local memory available within a robot is sufficient to store such information.

III. Mapping with strings

Can a robot explore and map an arbitrary anonymous graph-like world without any aid (oracle)? As shown in [4], given the minimal world model as described above, a robot lacking any additional aid (e.g., breadcrumbs, pebbles, strings, and the like) is *not* able to map its environments deterministically. Consider (embedded) single cycles graphs of any size $n \geq 3$. In each of the graphs all vertices have the same degree. Moreover, the degree information of vertices connected to any vertex are all the same. All of the vertices thus appear identical to the robot even if the degree information of arbitrarily large neighborhoods are taken into consideration. Thus even if the robot were to explore single cycles of size 3 and 4, it would not be able to tell them apart. Actually the robot would not tell apart any sized single cycles – in exploring both the graphs, the robot always observes a non-terminating sequence of '2-door rooms'.

Assuming that we have some aid that is sufficiently powerful, how expensive is it to explore an unknown environment deterministically? We begin by observing that the cost of physically moving a robot is likely to be several orders of magnitude more expensive (in terms of time, power expended, etc.) than is the cost associated with the computational effort. Thus in the following we consider physical steps moved in exploring the unknown environment (i.e., number of edge traversals) as the cost of the exploration algorithm. Clearly the robot must traverse every edge in the environment in the process of exploring (otherwise it would not know where all the edges go), and thus a trivial lower bound of the exploration cost is $O(m)$.

Assume the embedded graph representation described in Section II. Given the underlying graph G, the oracle-based algorithms developed in this paper proceed by building incrementally a known map out of an explored subgraph S of G. As new vertices are encountered, they are added to S and their incident edges are added to U which is the set of unexplored edges that lead to unknown places (and thus must be explored). Initially $S = \{v_0\}$ where v_0 corresponds to the initial location of the robot. Incident edges at v_0 are the initial elements of U. One step of the algorithms consist of selecting and removing an unexplored edge $e = (v_k, v_u)$ from U, (the robot) traversing to the known end v_k and then following e to the unknown end v_u (Fig. 1(a)). Upon arrival at v_u, the robot needs to answer 'have I been here before?' Specifically, it must answer: 1) Does v_u correspond to some known vertex in S ('where am I entering')? 2) If v_u corresponds to a known vertex $v_{k'}$, then which incident edge on $v_{k'}$ does e correspond to ('by which edge did I enter')? In the following discussions, inferring (1) and (2) are referred to as 'place validation' and 'back-link validation' respectively. Using different string oracles the algorithms discussed below conduct validations in different ways. If the validations show that unknown location v_u is not in S (i.e., no loop is formed), then both v_u and e can be added to S, augmenting S by one edge and one vertex as shown in Fig. 1(b) (non-loop augmentation). Other (unexplored) edges incident on v_u are added to U. If v_u is shown to correspond to the known vertex $v_{k'}$ (place validation) and e corresponds to the incident edge e' at $v_{k'}$ (back-link validation), then a loop is formed. In this case S is augmented by the edge $e/e' = (v_k, v_{k'})$ as shown in Fig. 1(c) (loop augmentation). Exploration terminates when the unexplored edge set U is empty.



(a) Robot explores e (b) No loop is formed (c) A loop is formed

Fig. 1. Basic exploration step. The robot must determine if $e = (v_k, v_u)$ corresponds to a loop or if v_u is an unvisited vertex.

The key question is 'how to do place and back-link validations deterministically using a string'? A string can be a powerful oracle when exploring a graph-like world in that there are many ways of manipulating it, resulting in validation aids of different powers. We examine different string classes and for each identify their relative power in doing validations, i.e., the required exploration cost.

A. Mapping with a very short string

We first present two algorithms for mapping deterministically a graph-like world with the shortest possible string. In its simplest form the string is unmarked (the surface of the string provides no specific information), and is only long enough to be tied at a particular location (vertex) and then laid out in some direction. We call such a short string a $l = \epsilon$ string where l denotes the length of the string.

A-1) Mapping by fixing a $l = \epsilon$ string

Probably the simplest way to manipulate such a short string is to tie it at the starting location, and never pick it up again. Suppose that the string is tied at vertex v_0. Now consider a path from a particular location (vertex or edge) within the underlying world to v_0. Given the embedding of the world, the path can be represented as the sequence of edge orderings at each vertex along the path, and it is true that there always exist *distinct* shortest paths (edge ordering sequences) from different locations of the world to v_0. That is, different locations cannot have the same shortest path to v_0. These paths can be exploited in the validation processes.

The robot can explore and map its environments using a fixed string in a manner similar to the fixed pebble exploration algorithm developed in [10]. Initially the robot enumerates incident edges at v_0 and sets edge labels (local

on S) based on the enumerated edge ordering. The robot then ties one end of the string at v_0, laying out the string toward one of the doors (edges) and remembers the (local) label of the door. The string-related sensory information that the robot acquires at a vertex includes whether the string is present at the vertex and the direction of the string if the string is present. The key idea is that whenever the robot re-enters v_0, by enumerating the doors and identifying the one that is pointed by the string's free end, the robot is able to infer the labels of all the incident edges at v_0. That is, the string not only identifies the unique vertex in which it is tied but it also provides the unique edge ordering at that vertex.

Algorithm 1: Mapping by fixing a $l = \epsilon$ string

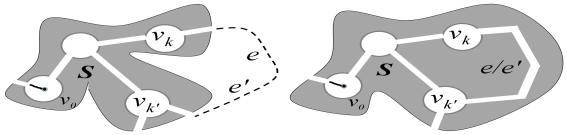
Input: the starting location v_0 in G
Output: a map representation S isomorphic to world G

```
the robot ties the string at v_0, laying out in a direction;
S ← v_0; U ← edges in v_0;          // initial S & U
while U is not empty do
    remove an unexplored edge e = (v_k, v_u) from U;
    the robot traverses S to v_k and follows e to v_u;
    for each edge (hypothesis) e' = (v'_k, v'_u) in U do
        compute the shortest path {v'_k, ..., v_0};
        the robot traverses path {v'_k, ..., v_0};
        based on the sensory info during traversal do
            case (1) or (2)
                the robot retraces to v_u;
                reject the hypothesis and continue;
            case (3)
                confirm the hypothesis and exit loop;
    if a hypothesis is confirmed then
        do 'loop augmentation' on S;
    else // all hypotheses are rejected
        do 'non-loop augmentation' on S and U;
return S;
```

The algorithm is outlined in Algorithm 1. Each step involves traversing an unexplored edge e to the unknown end v_u. Validations are conducted by disambiguating edge e against (other) unexplored edges in U. Each (other) unexplored edge $e' = (v_{k'}, v_{u'})$ incident on a known vertex $v_{k'}$ in S is considered a potential loop-closing hypothesis. That is, it is hypothesized that $e = (v_k, v_u)$ and $e' = (v_{k'}, v_{u'})$ correspond to the same edge (thus the robot entered $v_{k'}$ from v_k via e'), as shown in Fig. 2(a). Each hypothesis (edge) is validated by exploiting its (distinct) path to the string-marked v_0. For each hypothesis the *shortest* path $v_{k'}, ..., v_0$ (on S) is computed. The path is represented as a sequence of (relative) edge orderings at each vertex along the path, including the ordering at $v_{k'}$ (relative to the known ordering of e' at $v_{k'}$), and the ordering at v_0 (relative to the known ordering of the string-pointed edge at v_0). The robot then validates the hypothesis by attempting to traverse this path. The key fact is that if the hypothesis holds, the path would start from $v_{k'}$ and lead the robot to the string-marked vertex v_0 via the expected entry edge. The robot obtains sensory information during path execution, and distinguishes three possibilities:

1) The string is encountered at some point along the execution of the path prior to completion.
2) Upon completion of path execution, the string is not present at the vertex, or it is present but the entry edge does not match the expected entry edge in v_0.
3) Upon completion of path execution, the string is present at the vertex and the entry edge matches the expected entery edge in v_0.

As $v_{k'}, ..., v_0$ is the shortest possible path sequence, the robot should not have encountered the string prior to v_0, thus the hypothesis can be rejected in case (1). The hypothesis is also rejected in case (2). In this case the robot did not arrive at v_0 or did not arrive from the correct entry edge. Once a hypothesis is rejected, the robot retraces its steps by the reverse edge sequence (to v_u), and tries the next hypothesis of e (if any). In case (3) the hypothesis is confirmed and no other hypotheses of e are tried. The validation process for e terminates either when a hypothesis is confirmed, or, all the hypotheses have been tested (rejected). If all the hypotheses for e are rejected, the unknown location v_u is not in S (no loop is formed). Thus v_u and e are added to S, augmenting S by one vertex and one edge (non-loop augmentation). If a hypothesis e' is confirmed, then v_u corresponds to the known vertex $v_{k'}$ (place validated) and e corresponds to the incident edge e' at $v_{k'}$ (back-link validated). A loop is formed. Thus S is augmented by edge $e/e' = (v_k, v_{k'})$ as in Fig. 2(b) (loop augmentation). The algorithm terminates when U is empty, with S being isomorphic to underlying world G ([10]). The algorithmic cost (edge-traversals) of mapping a graph-like world G is $O(m^2n) \leq O(n^5)$.



(a) A hypothesis (b) Hypothesis is confirmed

Fig. 2. Mapping with a fixed $l = \epsilon$ string. S is augmented in (b).

A-2) Mapping by carrying a $l = \epsilon$ string

The high $O(m^2n)$ exploration cost of the above algorithm can be reduced by increasing the power of the string being used. One example is that the robot is not limited to only tying the string once, but rather can also pick up and carry the string, and tie the string again at different vertices as desired. Assume that the robot can put down and tie the string it carries at any vertex and that it can untie and pick up the string if the string is tied at the same vertex as the robot. Assume the same string-related perception as above.

With a movable string, after the robot traverses an unexplored edge e to the unknown end v_u, validations are carried out by tying the string at v_u and laying it out toward the entry edge (pointing toward v_k), and then searching S for the tied

string. If the string is not found at one of the vertices of S, then v_u is not in S (i.e., no loop is formed). Thus both v_u and e can be added to S (non-loop augmentation). If the string is found tied at some vertex $v_{k'}$ of S then v_u (where the string was tied) corresponds to the known vertex $v_{k'}$ (where the string was found). Moreover, the edge at $v_{k'}$ that the string points to is edge e' (that corresponds to edge e), whose label can be inferred immediately based on the relative ordering between e' and the current entry edge. The robot can thus add the edge $e/e' = (v_k, v_{k'})$ to S, augmenting S by one edge (loop augmentation). This is outlined in Algorithm 2.

Algorithm 2: Mapping by carrying a $l = \epsilon$ string

```
S ← v_0; U ← edges in v_0;
while U is not empty do
    remove an unexplored edge e = (v_k, v_u) from U;
    the robot traverses S to v_k and follows e to v_u; ties
    the string at v_u laying toward e;
    the robot traverses S searching for the string;
    if the string is found then
        do 'loop augumentation' on S;
    else
        do 'non-loop augmentation' on S and U;
    the robot (goes to v_u and) picks up the string;
return S;
```

This movable string algorithm is a simplified version of the $O(mn)$ undirectional movable marker algorithm of Dudek et al. [4]. An undirectional marker can mark a particular vertex but cannot provide edge ordering information. Thus when v'_k is identified the robot needs to do extra traversals for 'back-link validation'. Since the string is 'directional', here the extra traversals are avoided but the cost of the movable $l = \epsilon$ string algorithm is still $O(mn) \leq O(n^3)$.

B. Mapping with a longer string

Clearly a string of various (longer) lengths can be used as a short string as described above, incurring $O(m^2n)$ or $O(mn)$ cost, but are there efficiencies to be found in using longer strings? We examine below the power of strings of various lengths relative to the size of the world being explored. We start with the (simplest) case where the string is much longer than the size (total number of edges) of the graph-like world, i.e., $l \gg |E(G)|$ (assuming a unit edge length).

B-1) Mapping with a $l \gg |E(G)|$ string

If the robot is assured that the string is sufficiently long (e.g., $l \gg |E(G)|$), then there are several ways to alleviate the validation efforts in each step. One example is to allow the robot to tie knots at each unknown place v_u it is visiting. Assume that the robot ties the string at the initial vertex v_0 and plays the string out as it explores, and that during exploration the robot ties a distinct knot at each newly visited (blank) vertex v_u and 'remembers' the knot, and that when entering a string-marked (i.e., visited) v_u later the robot can sense the unique knot associated with the vertex. Then such a string answers not only 'whether v_u has been visited before', but also 'exactly which vertex does v_u refer to', as all the visited vertices are marked with their 'own' knots. Thus 'place validation' for v_u is never needed. Without exploiting other information, when entering a string-marked (visited) vertex v_u/$v_{k'}$ via e the robot may still need to conduct 'back-link validation' to infer the label of entry edge e' at $v_{k'}$ that e corresponds to. Probably the simplest solution is for the robot to traverse one of the *explored* edges e'' at $v_{k'}$, which is a stringed edge at $v_{k'}$. The knot at the other end of e'' (which must be there) 'tells' the robot which vertex it is visiting and thus making the label of e'' at $v_{k'}$ inferable. Then based on the relative ordering between e'' and e', the label of e' can be inferred. The exploration cost of the approach is $O(m)$.

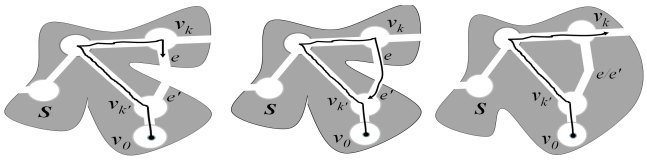
The extra traversal for back-link validation at each string-marked place can be avoided in various ways. For example, when entering a blank (unvisited) place for the first time, the robot ties the distinct knot near the edge (door) by which it enters the place, or, ties an extra knot on the string along the entry edge. Such a string defines a global ordering on each visited vertex thus marking all the visited vertices and edges. Both place and back-link validation are avoided. The above approach is thus simplified to search algorithms such as Depth-first search (DFS) and GREEDY, which have $O(m)$ cost. As shown in [3], DFS is an optimal algorithm for mapping an unknown undirected graph-like world.

B-2) Mapping with a $l = c(G)$ string

The problem becomes more challenging if the robot is not assured that the string is sufficiently long. We first consider the case where the robot is assured that once the string is tied, it can play the string out and traverse to any other vertex without running out of string, provided that the string never forms a complete loop (cycle) during traversals. We model the problem as mapping with a string of $l = c(G)$, where $c(G)$ denotes the *circumference* of the graph-like world G (i.e., the length of a longest simple cycle in the world).

With such a string, the robot can tie one end of the string at the initial vertex v_0, and play the string out as it explores, until it enters a string-marked place (i.e., the string forms a loop), or, until it 'saturates' the current vertex, i.e., the current vertex has no more unexplored edges. In these cases the robot retraces its steps along the string by picking up and rewinding the string, until it comes back a vertex (on the string) that has unexplored edge(s), and explores there. By enforcing that the robot rewinds the string when entering a string-marked vertex, the string never forms a loop and thus the robot never runs out of string during exploration. The robot also rewinds the string after 'saturating' the current vertex. Although the string might be rewound during exploration thus leaving some visited vertices 'blank', such vertices have all been saturated and thus are 'inaccessible' to the robot (in each exploration step the robot always follows an unexplored edge out of S). Thus if the robot enters a place that contains no string, that place must have *not* been visited.

The algorithm is outlined in Algorithm 3. Each step involves traversing an unexplored edge e to the unknown



(a) The robot explores e (b) v_u/v'_k has string (c) Augments S by e/e'

Fig. 3. Mapping with a $l = c(G)$ string. The robot rewinds the string after augmenting S (b)–(c).

end v_u (and laying the string along e). At v_u the robot leaves the string (free end) there (Fig. 3(a)–(b)). There are two possibilities at v_u: 1) v_u only contains the new string, 2) v_u (and up to two of its edges) already contain the string (Fig. 3(b)). In case (1), v_u must have not been visited before (as justified above). Thus both e and v_u can be added to S (non-loop augmentation) without any further validation. If v_u becomes saturated now, the robot rewinds the string along e back to v_k (if v_k becomes saturated now, the robot rewinds further). Otherwise the robot continues on an unexplored edge at v_u. In case (2), both place and back-link validation are needed, as the string at v_u might not reveal which known (visited) vertex $v_{k'}$ is, and which entry edge e' is. The newly laid string (along e) is exploited. Validations are conducted by the robot (reversely) visiting vertices along the string looking for the one that has one more stringed edge than it should have (based on S). (Or looking for the one that contains the string's free end.) Once the vertex $v_{k'}$ (corresponding to v_u) is identified, the edge e' that corresponds to e is also identified, which is the *unexplored* edge in $v_{k'}$ (shown on S) that now has the string laid along it (also the edge by which the string's free end comes in). This 'new stringed edge' $e/e' = (v_k, v_{k'})$ is added to S (Fig. 3(c)) (loop augmentation). The robot then rewinds the string along e/e' back to v_k, trying to explore there (Fig. 3(b)–(c)). If v_k becomes saturated now, the robot rewinds further.

Algorithm 3: Mapping with a $l = c(G)$ string

```
1  robot ties the string at v_0; S ← v_0; U ← edges in v_0;
2  while U is not empty do
3      remove a closest edge e = (v_k, v_u) from U;
4      the robot traverses S to v_k and follows e to v_u;
       unwinds the string along e to v_u;
5      if v_u only contains the new string then
6          do 'non-loop augmentation' on S and U;
7          if v_u becomes saturated now then
8              robot rewinds string to an unsaturated node;
9      else // v_u contains the string already
10         robot searchs back along the string looking for
           the vertex that has one more stringed edge;
11         do 'loop augmentation' on S;
12         robot rewinds string to an unsaturated node;
13 return S;
```

While the approach has $O(mn)$ exploration cost as the robot may exhaust all vertices currently on the string (bounded by n) for validating a single edge, it is expected to produce a reduced cost over the short movable $l = \epsilon$ string algorithms due to the reduced need for validation.

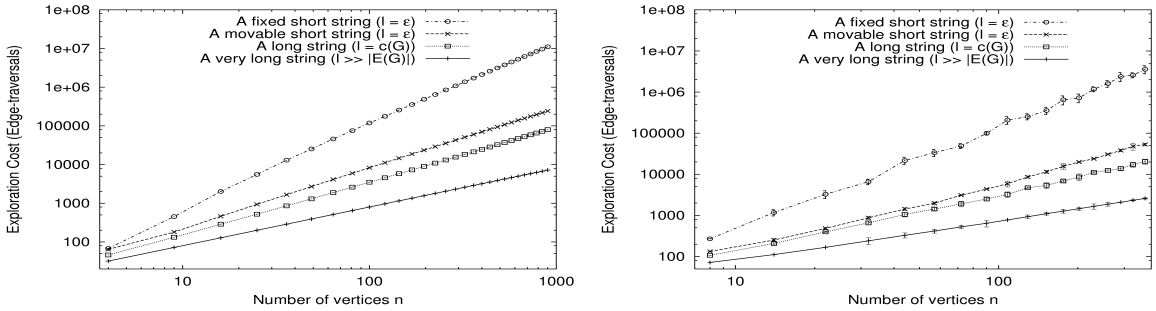
B-3) Exploring with a $l < c(G)$ string

Now consider the more general case where the robot may run out of string during exploration, even if the string never forms a cycle. This may happen when string length $l < c(G)$. This may also occur when the robot does not know the length of the string (relative to the environment size).

The proposed algorithm extends the above $l = c(G)$ algorithm. Initially the robot ties the string at the starting vertex v_0, and explores all the vertices that are within distance l from v_0. When string length l is reached, i.e., the robot runs out of string, the robot rewinds the string from the current vertex to try previously visited vertices on the string, even if the current vertex has *not* yet been saturated. When no exploration is possible without running out of the string, the robot unties the string from v_0 and chooses one of the known vertices v'_0 which is not yet saturated, tying the string there and starting exploration again. The robot then explores all vertices within a distance l from v'_0. The process repeats until the environment is fully explored. As above, each step involves exploring an unexplored edge e to its unknown end v_u. If v_u contains the string, as in the above algorithm the robot searches back along the string looking for the known vertex having one more stringed edges (or containing the string's free end), and then does 'loop augmentation'. If v_u does not contain the string, then unlike the above algorithm, the robot visits known vertices (which are not on the string) looking for the one having a stringed edge (or containing the string's free end). This visit is needed as the string is rewound whenever l is reached, leaving some visited places not yet saturated and potentially accessible.

Algorithm 4: Mapping with a $l < c(G)$ string

```
1  robot ties the string at v_0; S ← v_0; U ← edges in v_0;
2  while U is not empty do
3      remove a closest edge e = (v_k, v_u) from U;
4      the robot traverses e to v_u; unwinds the string to v_u;
5      if v_u only contains the new string then
6          robot searches non-string vertices on S looking
           for the vertex having a stringed edge;
7          do 'non-loop augmentation' on S and U;
8      else // v_u contains the string already
9          robot searches back along the string looking for
           the vertex having one more stringed edge;
10         do 'loop augmentation' on S;
11     if string length l is reached then
12         robot rewinds string to an unsaturated vertex;
13     if all vertices within distance l are explored then
14         robot unties string and reties it at a new node;
15 return S;
```



(a) Homogeneous lattices.

(b) Lattices with 20% missing vertices.

Fig. 4. Performance of mapping on lattices of varying sizes using different strings (log scale). Results for (b) are averaged over 30 graphs. Each graph has randomly generated holes (deleted vertices). Error bars in (b) show standard errors.

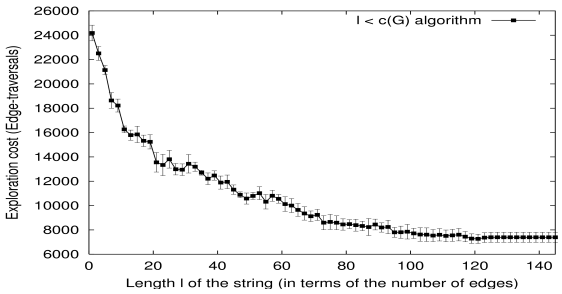


Fig. 5. Performance of mapping on non-homogeneous lattice (20×20 vertices lattice with 20% missing vertices) using strings of different lengths. Results are averaged over 30 graphs each has randomly generated holes. Error bars show standard errors.

The $l < c(G)$ string algorithm has $O(mn)$ cost. Note that the movable $l = \epsilon$ and $l = c(G)$ algorithms discussed earlier can be considered special cases of this general algorithm.

C. Empirical evaluations of the power of different strings

Here we present empirical comparisons of the performance of mapping with the different string algorithms discussed above. We first compare the relative power of a fixed short ($l = \epsilon$) string, a movable short ($l = \epsilon$) string, a long ($l = c(G)$) string, and a very long string ($l \gg |E(G)|$). Experimental results are reported for both homogeneous and non-homogeneous lattice graphs of varying sizes (see Fig. 4). For these environments, exploring by carrying the short ($l = \epsilon$) string which has $O(mn)$ cost, obtains a reduced exploration cost over exploring with a fixed short string which has $O(m^2n)$ cost. Exploring with $l = c(G)$ string also has $O(mn)$ cost. Exploring with a very long string (with knots) which has a (optimal) linear cost $O(m)$, provides the lowest cost over all the other strings considered.

The results for the $l < c(G)$ algorithm with varying length l are shown in Fig. 5. The algorithm demonstrates exploration cost reduction as l increases. Note that when the string is sufficiently long ($l \geq c(G)$), fixed exploration cost is produced (the algorithm acts as the $l = c(G)$ algorithm).

IV. Summary and Discussions

Given an embedded topological representation, it is not, in general, possible to map the world deterministically without resorting to the use of some type of oracle to solve the 'have I been here before?' problem. Given the simplest form of string, deterministic mapping is possible with cost $O(m^2n)$. The minimum cost for mapping is $O(m)$ and this can be obtained with a sufficiently long string. Overall the longer the string the lower the cost. Performance bounds of the various strings reviewed in the paper are summarized in Table I.

TABLE I

Solvability and cost bounds of different strings.

String oracles		Exploration cost
a short $l = \epsilon$ string	fixed	$O(m^2n)$
	movable	$O(mn)$
a long string $l < c(G)$		$O(mn)$
a longer string $l = c(G)$		$O(mn)$
an infinite string $l \gg \|E(G)\|$		$O(m)$ optimal

Acknowledgment

The financial support of NSERC Canada is gratefully acknowledged.

References

[1] Theseus by Plutarch, Written 75 A.C.E. Translated by John Dryden.

[2] X. Deng and A. Mirzaian. Competitive robot mapping with homogeneous markers. *IEEE Transactions on Robotics and Automation*, 12(4):532–542, 1996.

[3] A. Dessmark and A. Pelc. Optimal graph exploration without good maps. *Theoretical Computer Science*, 326(1-3):343–362, 2004.

[4] G. Dudek, M. Jenkin, E. Milios, and D. Wilkes. Robotic exploration as graph construction. Technical Report RBCV-TR-88-23, Department of Computer Science, University of Toronto, 1988.

[5] H. Durrant-whyte and T. Bailey. Simultaneous Localization and Mapping (SLAM): Part I. *IEEE Robotics and Automation Magazine*, 13(2):99–100, 2006.

[6] P. Fraigniaud, D. Ilcinkas, G. Peer, A. Pelc, and D. Peleg. Graph exploration by a finite automaton. *Theoretical Computer Science*, 345(2-3):331–344, 2005.

[7] J. M. O'Kane and S. M. Lavalle. Comparing the power of robots. *International Journal of Robotics Research*, 27(1):5–23, 2008.

[8] S. Thrun. Robotic mapping: a survey. In *Exploring Artificial Intelligence in the New Millennium*, pages 1–35. 2003.

[9] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, USA, 2005.

[10] H. Wang. Exploring topological environments. Technical Report CSE-2010-05, Department of Computer Science and Engineering, York University, 2010.