Map validation and robot self-location in a graph-like world

G. Dudek,
Centre for Intelligent Machines
McGill University, 3480 University St.
Montreal, Quebec, Canada, H3A 2A7
dudek@cim.mcgill.ca
(514) 398-4325

M. Jenkin and E. Milios
Department of Computer Science
York University, 4700 Keele St.
North York, Ontario, Canada, M3J 1P3
{jenkin,eem}@cs.yorku.ca
(416)-736-5053

D. Wilkes
Department of Computer Science
University of Toronto
Toronto, Ontario, Canada
wilkes@cs.toronto.edu
(416)-978-7726

Abstract

This paper deals with the validation of topological maps of an environment by an active agent (such as a mobile robot), and the localization of an agent in a given map. The agent is assumed to have neither compass nor other instruments for measuring orientation or distance, and, therefore, no associated metrics. The topological maps considered are similar to conventional graphs. The robot is assumed to have enough sensory capability to traverse graph edges autonomously, recognize when it has reached a vertex, and enumerate edges incident upon the current vertex, locating them relative to the edge via which it entered the current vertex. In addition, the robot has access to a set of visually detectable, portable, distinct markers. We present algorithms, along with worst case complexity bounds and experimental results for representative classes of graphs for two fundamental problems.

The first problem is the *validation problem*: if the robot is given an input map and its current position and orientation with respect to the map, determine whether the map is correct. The second problem is the *self-location problem*: given only a map of the environment, determine the position of the robot and its "orientation" (i.e., the correspondence between edges of the map and edges in the world at the robot's position). Finally, we consider the power of some other non-metric aids in exploration.

Please address correspondence to Evangelos Milios.

1 Introduction

One of the problems confronting an autonomous mobile robot is that of maintaining an internal description of its environment. Without a useful internal representation (a map) and knowledge of the robot's pose with respect to this map, many robotic tasks become difficult, if not impossible. This is not to say that all mobile robotic tasks require a map. Some tasks may be amenable to behaviour-based approaches, and for such tasks an internal description of the environment may not be required (see, for example [6,7,8,1]). For other tasks, especially complex structured tasks, which are specific to particular landmarks in the environment, a map of the environment is crucial. If the robot is to have a map, what form would this map take, and how should it be acquired?

Several map representations have been proposed in the literature. These include: *metric representations* [2,20,21,9,15], which explicitly model the two or three dimensional position of elements of the environment, *probabilistic representations* [23,25], which retain many metric properties in the representation, but augment the representation with uncertainty information, and *topological or graph-like representations* [19,10,3], which represent locations of interest. Hybrid maps, which combine elements of each of these representational levels have also been proposed [16,1]. Integration of a map within a reactive framework has also been explored [22].

When an autonomous agent explores its own environment the fundamental problem that the robot has to address is the "have I been here before" problem. If the environment is modelled in a metric manner, this problem can be viewed as equivalent to the following: Given the current position and orientation of the robot (known with a particular covariance error matrix with respect to a world-centered coordinate system), has the robot been to this location previously via this or some other path? If the environment is represented in a graph-like manner, the question becomes "have I visited this location before", and, if so, which entrance did I use last time I entered here?

In previous work [14,12] we described and analyzed an algorithm for exploring an unknown environment and building a graph-like map of an unknown world (see also §3). This work assumes that a topological, rather than metric, description of the world is desirable at some level in a descriptive hierarchy [19]. A graph-like map was chosen because it represents the minimal information that a robot must be able to represent in order to distinguish one place from another. The exploration algorithm allows the robot to form a model of its world by exploring the world systematically with the use of one or more distinct markers that can be dropped and picked up at will. These markers can be recognized if they are found on the path of the robot. The exploration algorithm works by incrementally expanding a "known subgraph" of the world by exploring unknown edges incident with it. The algorithm requires the robot to make no more than $5MN - N^2 + 2d_{max}(M - N + 1) + 2N + M$ edge traversals (where the graph-like world W has M edges, N vertices and a maximum vertex degree of d_{max}). This results in a worst case complexity of $O(MN)$ moves.

In the work presented in [14,12] and extended here, we are concerned with the *mechanical time complexity* (mechanical complexity) of the task, that is, the number of discrete motions that the robot must perform. As the time constant associated with moving a robot is considerably longer than the constant associated with a computational step, mechanical complexity is the limiting factor for these types of operations (given that the purely computational problem is tractable, of course). That is not to say that the computational cost is not important – only that for these tasks it is outweighed by the mechanical or locomotive cost.

Note that it is not possible, in general, for the robot to explore its environment without some navigational aid. As an example, all regular graphs of degree k (i.e., graphs in which each vertex has k incident edges) are indistinguishable from each other without markers, because each vertex appears identical to every other vertex.

The difficulty of identifying graph vertices and edges in a real environment depends

on the environment and the type of sensors used. In a 2D polygonal world, a graph model can be instantiated by considering the Voronoi graph of the world [24]. In practice, a robot equipped with a laser range scanner should be able to traverse Voronoi edges fairly reliably. A problem arises when the distance between two adjacent Voronoi vertices is similar to the accuracy of the robot's pose estimate. In this case, the robot cannot reliably recognize the edge connecting the two Voronoi vertices. Kortenkamp and Weymouth [17] have investigated the combined use of sonar and vision sensing for identifying places and paths between them. Using vision to reliably identify places is a very challenging problem.

After briefly reviewing the robot and world models and our earlier exploration results, this paper focuses on two related problems: Given a map of the world, how can an active agent (a robot) determine whether the map is correct in its description of the connections between locations. This is the **map validation** problem. A related problem is that of **self-location**: given a map, how can an active agent determine its current location and orientation with respect to the map, where "orientation" is equivalent to the correct correspondence between the map edges and the world edges incident on the current location?

In addition to providing a worst case analysis of the validation and self-location algorithms, we present empirical expected-case results on particular classes of graphs for the validation and exploration algorithms. For some classes of graphs the exploration algorithm, which has higher worst case complexity, actually outperforms the validation algorithm.

Finally, we comment on the usefulness of markers as the mechanism by which the robot validates, explores, or localizes itself, and consider the performance advantages in using other non-metric mechanisms such as fixed markerd to serve as the reference for both location and orientation.

2 The World Model

The algorithms presented in this paper operate on an augmented graph-like representation of the world, which is defined below. This is an abstraction of real world problems that involve movements of the robot between locations along paths. Metric information about the world is assumed to have been collected and abstracted away by allowing the robot to know how to traverse a path, how to recognize that it has reached a path, and how to recognize paths starting at a particular location.

2.1 Definitions

The World: The world is defined as an undirected graph W: $W = (V, E)$ with set of N vertices V and set of M edges E and an ordering relation on the edges at each node, as described below. Note that in practice such a graph could be defined within a continuous environment based on landmarks or other features. One possible definition of a graph vertex is as a location that maximizes a distinctiveness measure [18]. The vertices of W are denoted by: $V = \{v_1, ..., v_N\}$. We will restrict the world model to graphs W that contain no cycles of length ≤ 2. This definition prohibits the world from having multiple edges between two vertices or between a vertex and itself. Although this restriction is not essential for the operations that follow, it does simplify them considerably.

The definition of an edge in W is extended slightly to include the explicit specification of the order of edges incident upon each vertex of the graph with respect to the other edges. This ordering can be obtained by enumerating the edges in a systematic (e.g., clockwise) manner from some standard starting direction. Such a specification of the cyclic edge ordering is provided by an embedding of the graph; in conjunction with the specification of the exterior face of the graph it is equivalent to a planar embedding for planar graphs.

An edge $E_{i,j}$ incident upon v_i and v_j can be assigned labels n and m, one with respect

to each of v_i and v_j, given a reference edge in each vertex. The labels n and m can be considered as general directions; e.g., from vertex v_i the nth exit takes edge $E_{i,j}$ to vertex v_j. More specifically, a path can be specified as a series of edge labels such that the entry edge at a vertex is always the reference edge and the successive labels specify the exit edges (e.g., take the 3rd edge on the right, then take the 2nd edge on the right, etc.).

Movement and Action: The robot can move from one vertex to another by traversing an edge (a *move*), it can pick up a marker that is located at the current vertex, and it can put down a marker it holds at the current vertex (a *marker operation*). The robot in general has $K \geq 1$ markers at its disposal.

Assume the robot is at a single vertex, v_i, having entered the vertex through edge $E_{i,l}$. In a single move, it leaves vertex v_i for vertex v_j by traversing the edge $E_{i,j}$, which is located r edges after $E_{i,l}$ according to the edge order at vertex v_i. This is expressed by the transition function: $\delta(v_i, E_{i,l}, r) = v_j$. We assume the following invertibility property concerning the transition function:

$\delta(v_j, E_{i,j}, s) = v_k$, then $\delta(v_j, E_{j,k}, -s) = v_i$

where $-s$ refers to the inverse edge enumeration: i.e., the label such that if $E_{j,k}$ has label s with respect to $E_{i,j}$ then $E_{i,j}$ has label $-s$ with respect to $E_{j,k}$. This implies that a sequence of moves is invertible, and can be retraced.

A single move is thus specified by the order r of the edge from which the robot exits the current vertex, where r is defined with respect to the edge along which the robot entered the vertex. Note that in the special case of a planar embedding of a graph, enumeration of edges in a clockwise fashion satisfies the above assumption.

A marker operation is fully specified by indicating for each of the K markers whether it is being picked up, put down, or not operated upon. This is specified by a K-tuple $\Omega^K =$

$(op_1, op_2, ..., op_K)$, where the element op_k has a value from the set $\{pickup, putdown, null\}$, according to the operation performed on marker k.

A *simple action* a is defined as a marker operation accompanied by (followed by) a single move, therefore $a = (b, \delta)$, where $b \in \Omega^K$. The robot performs some action on the markers in the current vertex and then moves to a new location. A *path* $A \in a^+$ is a nonempty sequence of actions. Note that not all action sequences are feasible at all times since the robot is only permitted to pick up markers that reside in its current vertex or to drop markers that it is carrying.

Perception: The robot's perception is of two kinds, marker-related and edge-related perception.

Marker-related Perception. Assume that the robot is at vertex v_i. The marker-related perception of the robot is a K-tuple $B = (b_1, b_2, ..., b_K)$, where b_k has a value from the set $\{present,\ not\text{–}present\}$, according to whether marker k is present at vertex v_i.

Edge-related Perception. The robot can determine the relative positions of edges incident on the vertex v_i in a consistent manner given that it knows it arrived by edge $E_{i,j}$, for example, by a clockwise enumeration starting with $E_{i,j}$. As a result, the robot can assign an integer label to each edge incident on v_i, representing the order of that edge with respect to the edge enumeration at v_i. The label 0 is assigned arbitrarily to the edge $E_{i,j}$, through which the robot entered vertex v_i. The ordering is local, because it depends on the edge $E_{i,j}$. Entering the same vertex from two different edges will lead to two local orderings, one of which is a permutation of the other. Note that if the graph is planar and a spatially consistent (e.g., clockwise) enumeration of edges is used, then two permutations will be simple cyclic shifts of each other. However, this will not hold in general, and in this paper we only

require that the edges can be ordered consistently.

The sensory information that the robot acquires while at vertex v_i is the pair consisting of the marker-related perception at that vertex and the order of edges incident on that vertex, with respect to the edge along which the robot entered the vertex. If the robot visits the same vertex twice, it must relate the two different local orderings produced and unify them into a single global ordering, for example by finding the label of the 0-th edge of the second ordering with respect to the first ordering. Determining when the same vertex has been visited twice and generating a global ordering for each vertex is part of the task of the following algorithms.

3 Map building: exploration

In earlier work ([12,14]) it was demonstrated that in general it is not possible for a robot to explore and map an unknown graph-like environment as defined here without markers or additional sensory information. This is consistent with human intuition: fairy tales, and mythology are full of stories of heroes who avoided becoming lost within a maze by dropping markers or unwinding string as they went. The basic problem is that, once a human or robotic explorer enters an unknown environment, they cannot always determine when or if they have returned to a previously visited location. The mythological solutions typically require a potentially large number of markers, breadcrumbs or a lot of string.

In these earlier papers it was also demonstrated that as long as the explorer had at least a *single* marker which could be dropped and picked up at will it was possible for the explorer to fully map the environment (an arbitrary graph). This could be accomplished with only $5MN - N^2 + 2d_{max}(M - N + 1) + 2N + M$ or $O(MN) \leq O(N^3)$ steps (although complexity for most typical cases appears substantially better than this worst-case bound). The basis of the algorithm is the maintenance of an explored subgraph of the full graph. As new vertices are encountered, they are added to the explored

subgraph, and their outgoing edges are added to the set of edges that lead to unknown places and therefore must be explored.

More formally, the algorithm maintains an explored subgraph S, and a set of unexplored edges U, which emanate from vertices of the explored subgraph. A step of the algorithm consists of selecting a set, E, of k unexplored edges from U, and "validating" the vertex v_β at the unexplored end of each edge $e = (v_\alpha, v_\beta)$ in the set E (where $v_\alpha \in S$ by construction). Validating a vertex v_β means making sure that it is not identical to any other vertex in the explored subgraph. This is carried out by placing one of the k markers at v_β and visiting all vertices of the known subgraph S along edges of S, looking for the marker (and each of the other $k-1$ markers dropped at this step). Note that the other vertex v_α incident upon e is already in the subgraph S.

If the marker is found at vertex v_i of the explored subgraph S, then vertex v_β (where the marker was dropped) is identical to the already known v_i (where the marker was found). In this case, edge $e = (v_\alpha, v_\beta)$ must be assigned an index with respect to the edge ordering of vertex v_β. To determine this, the robot drops the marker at v_α and goes back to v_β along the shortest path in the explored graph S. At v_β, it tries going out of the vertex along each of its incident edges. One of them will take the robot back to v_α, which the robot will immediately recognize due to the presence of the marker. Note that the index of e with respect to the edge ordering of v_α is known by construction. Edge e is then added to the subgraph S and removed from U.

If the marker is not found at one of the vertices of S, then vertex v_β is not in the subgraph S, and therefore must be added to it. The previously unexplored edge e is also added to S, which has now been augmented by one edge and one vertex. Adding the vertex v_β to the subgraph causes all edges incident upon it to be assigned an index with respect to the edge e by which the robot entered the vertex (edge e is assigned index 0) and the new edges are added to the set of unexplored edges U. Note that no other edge of the new vertex v_β has been previously added to the subgraph, because otherwise v_β

would have already been in the explored subgraph. This index assignment establishes the edge ordering local to v_β. The algorithm terminates when the set of unexplored edges U is empty. A formal proof of the correctness of the algorithm is presented in [14].

The cost of exploring the graph in terms of edge traversals by the robot (mechanical complexity) is $O(MN) \leq O(N^3)$, where M is the number of edges and N is the number of nodes. This follows from the need to go back and actually visit all of the locations in the known sub-graph to solve the "have I been here before" problem. In practice, it is not always necessary to explore an environment completely from scratch. A map may already be available and it may suffice to determine if the existing map of the world corresponds to its current state and, if not, to determine where the two are inconsistent.

4 Map validation with known initial pose

The problem we will address in this section is defined as follows [13]. The robot is given a map C of its graph-like world W. The map C is a graph of the same form as the one computed by exploration: it consists of a set of vertices $V_C = \{v_{C1}, v_{C2}, ..., v_{Cn}\}$, and a set of edges E_C with a cyclic edge ordering at each vertex. The world W in which the robot resides is also a graph consisting of vertices $V_W = \{v_{W1}, v_{W2}, ..., v_{Wn'}\}$ and a set of edges E_W with an associated ordering at each vertex. The robot is told which map vertex v_{C0} corresponds to the world vertex v_{W0} that it starts in. The robot is also told which map edge e_{C0k} corresponds to a specific physical exit edge from vertex v_{M0} (this is the robot's initial "orientation"). The robot is then asked to verify the correctness of the map C, i.e., to determine whether C is consistent with the world W, by looking for an isomorphism relationship between C and W that preserves the cyclic edge ordering at each vertex. An affirmative answer to this question automatically establishes a one-to-one correspondence between the elements (vertices and edges) of C and W. We now present an algorithm for solving the above *map validation* problem, which requires at

most $4N^2 + 4M - 4N - 6$ or $O(N^2)$ moves of the robot.

The key idea underlying the validation algorithm is the construction of a spanning tree of the map C rooted at the initial vertex of the map (an operation requiring (for example) $O(N^2 \log N)$ computations using Kruskal's algorithm [5]). The algorithm first verifies the presence of this tree in the world and then verifies the remaining edges of the world with respect to the map (which is akin to an exploration task). We now describe the algorithm for the case in which the robot is equipped with a single movable marker. We will use the notation $\phi(v_{Ci})$, $\phi(e_{Cij})$ and $\phi(subgraph)$ to denote the vertex, edge or subgraph in the world W which corresponds to v_{Ci}, e_{Cij} or $subgraph$ of C, respectively. The symbol $\phi^{-1}(\cdot)$ is used to denote the inverse mapping from W to C. The notation $index(e, v_{Ci})$ indicates the index of edge e incident on vertex v_{Ci} with respect to v_{Ci}. Finally, $edge(l, v_{Ci})$ indicates the map edge incident on v_{Ci} with index l with respect to v_{Ci}. The algorithm is as follows.

1. Validate a spanning tree rooted at the initial robot position.

   (a) Compute a spanning tree ST of the map, rooted at the initial map vertex v_{C0}.

   (b) Validate ST.

   For each vertex v_{Ci} of ST,

   i. Move the robot to $\phi(v_{Ci})$ and place the marker there. Do this by following a path entirely on $\phi(ST)$. Verify consistency between map and world at that vertex by verifying on the map the observed vertex degree and any additional sensor information available.

   ii. Visit all vertices of $\phi(ST)$ by traversing edges of $\phi(ST)$ only. Verify consistency between map and world at each vertex by verifying on the map the observed vertex degree and any additional sensor information available. If the marker is found at a physical vertex v different from

$\phi(v_{Ci})$ as indicated by following the same physical path on C as well, then validation fails.

With the marker at each of N nodes in turn, $2(N-1)$ edge traversals are required to do a complete tour of $\phi(ST)$ (since a complete tour of a tree need only traverse each tree edge once in each direction). At most $2(N-1)-1$ edge traversals are needed to move the marker from node to node. Thus $2N(N-1) + 2(N-1) - 1$, or $2N^2 - 3$ edge traversals are required altogether for step 1.

2. Validate the edges outside the spanning tree, by checking that they connect the correct two vertices. An efficient way to do this is to check all edges that are incident on a single vertex with at most a single physical traversal of the graph. For each vertex v_{Ci} of the map C

   (a) Move the robot to $\phi(v_{Ci})$ and place the marker there. Do this by following a path entirely on $\phi(ST)$.

   (b) Leave the marker at $\phi(v_{Ci})$.

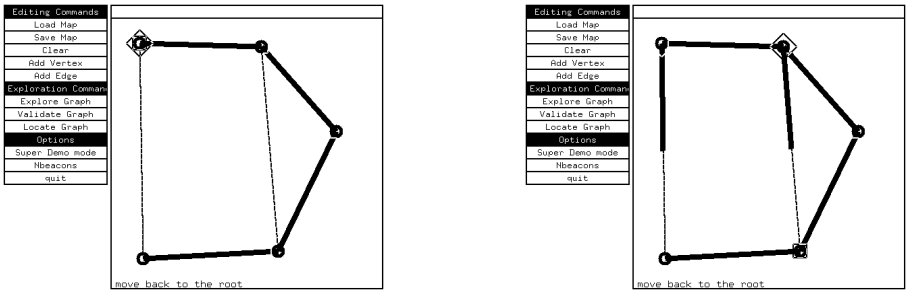
   (c) Visit all neighbours v_{Cj} on the map of v_{Ci} via the ST.

   (d) For each such vertex v_{Cj} (associated with an edge $e_{Cij} = (v_{Ci}, v_{Cj})$ according to the map) check physically whether edge $\phi(e_{Cij})$ exists by traversing the edge $\phi(edge(index(e_{Cij}, v_{Cj}), v_{Cj}))$ and looking for the marker at the other end of that edge.

   Using similar reasoning to that used for step 1, this requires at most

$$2(N-1) - 1 + \sum_{i=1}^{N}(2(N-1) + 2d'(v_{Ci})) \qquad (1)$$

   edge traversals, where $d'(v_{Ci})$ is the degree of map vertex i less the number of spanning tree edges connected to vertex i. Since $\sum_{i=1}^{N} d'(v_{Ci}) = 2(M-N)$, we have



(a) (b)

Figure 1: Stages in validating a graph. The graph was validated with itself starting with the correct location and orientation. Edges that have been validated are drawn with a thicker line. Note that non-spanning tree edges are validated in one direction at a time and these are drawn half thick/half thin with the thicker end corresponding to the validated end of the edge. The location of the robot is indicated with a square and the location of the marker is indicated by a diamond. (a) shows the validation algorithm after the validation of the spanning tree, while (b) shows the validation algorithm after validating half of each of the graph edges. This graph was correctly validated.

that step 2 requires $2(N-1) - 1 + 2N(N-1) + 4(M-N)$, or $2N^2 + 4M - 4N - 3$ edge traversals.

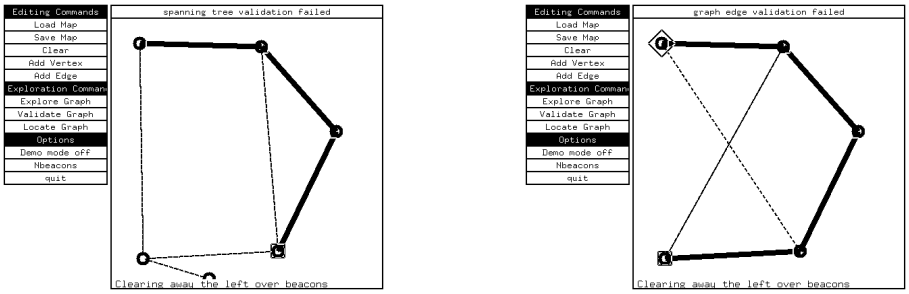
Thus the validation algorithm requires at most

$$F = 4N^2 + 4M - 4N - 6 \tag{2}$$

or $O(N^2)$ edge traversals.

4.1 Special cases

The following example serves to illustrate briefly the functioning of the algorithm.



(a) (b)

Figure 2: Two graphs which fail when being validated by the map given in Figure 1. (a) fails as the spanning tree cannot be validated, while (b) fails because a non-tree edge is incorrect.

Example 1. Figure 1 shows the state of the validation process at two stages in the validation of a simple graph. Figure 1(a) shows the validation process after the validation of the spanning tree of the graph, while Figure 1(b) shows the same graph after half of each of the graph edges have been validated. Treating this graph as a map, two corrupted versions and the point of failure of the validation process are shown in Figure 2. Figure 2(a) is a version of the graph from Figure 1 corrupted by the addition of an extra vertex and edge. The validation process (indicated by the heavy lines) fails in the spanning tree validation stage as the spanning trees do not agree (the degree of one of the nodes on the spanning tree is incorrect). Figure 2(b) shows a corrupted graph in which the spanning trees agree but in which the graph edges are incorrect. Here the spanning tree is verifiable, but the non-spanning-tree edges of the map and the world disagree.

Example 2 (Random Graphs). To compare the performance of exploration and validation, both algorithms were tested on a variety of random graphs [4]. The first set of parameterized random graphs was generated by starting with a complete 2D lattice

(i.e., a grid) and deleting a specified fraction of randomly selected edges such that the graph remains connected. This first family of graphs should be familiar to those who have been forced to drive a car in Montreal (where roads are often under repair in the summer), and are termed *Montreal graphs with deletion factor* p, or $Montreal(p)$, $p \in (0, 1)$. They are also characteristic of the hallway structure of many indoor environments such as offices or warehouses.

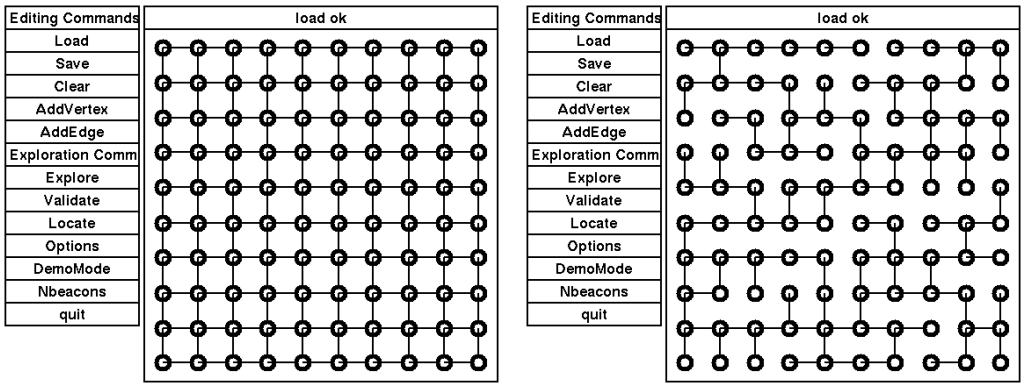
The second class of examples, *random-connection* graphs $rc(p)$, is generated by starting with a tree ($p = 0$) and adding some number of random edges, to the point of having a fully connected graph (when $p = 1$). For each of these sequences we observe how the number of steps required for exploration and validation varies as a function of the number of edges in the graph.

Montreal lattices Two dimensional square lattices (with holes) represent the type of environment that is often encountered in the interior of modern buildings and in warehouses. (Note that it is the connectivity of the graph, not the distance between nodes that is important.) Figure 3 shows two sample input graphs. In order to examine the effect of the connectivity of the graph the performance of the algorithm was evaluated on examples with progressively larger numbers or edges deleted (p decreasing) up to a maximum deletion factor of 20% of all of the edges in the graph ($p = 0.2$). A sample graph with 20% of its edges removed is shown in Figure 3(b).

For the family of Montreal graphs (connected random square lattices indexed by the fraction p of edges present) we have:

$$d_{max} = 4$$
$$M = 2(1 - p)(N - \sqrt{N}).$$

Thus the number of steps required by the validation and exploration algorithm is bounded



(a) (b)

Figure 3: Sample lattice graphs. (a) corresponds to the fully connected case, while (b) has 20% of its edges removed.

by the following for exploration and validation, respectively.

$$\begin{aligned} explore &\leq 9N^2 - 10N^{3/2} + 12N - 18\sqrt{N} + 8 + p(-10N^2 + 10N^{3/2} - 18N + 18\sqrt{N}) \\ &\leq O(N^2) \\ validate &\leq 4N^2 + (4 - 8p)N + (8p - 8)\sqrt{N} - 6 \\ &\leq O(N^2) \end{aligned} \tag{3}$$

Figures 3(a) and 3(b) correspond to the extreme conditions for the graphs which were explored in order to produce the results shown in Figure 4. Figure 4 shows the number of robot steps required for exploration and validation of rectangular lattice graphs with holes. As can be seen from the graph, the cost of exploring the graph (marked by $+$) is consistently less than the cost of validating the graph (marked by $\diamond$), even though the constant scale factor in the complexity above suggests that the opposite should happen.

Note that over all graphs the validation operation takes far fewer robot moves in the worst case ($O(N^2)$ versus $O(N^3)$). If the graph is planar and has no more than one edge

per vertex pair, however, the number of edges M satisfies the inequality $M \leq 3N - 6$. In this case, exploration is also quadratic in the number of vertices N. In the random graph examples shown below, the number of vertices is fixed. In this case, complexity is a linear function of the number of edges, which is observed in both examples.

For sufficiently large N, both algorithms should perform within a constant factor of each other. For a fixed N, exploration and validation should be linear decreasing functions of p. As illustrated by the plot in Figure 4, the curves are approximately linear, however (i) exploration outperforms the validation algorithm, and (ii) the exploration algorithm runs roughly 8 times faster than the bound in equation (3) and the validation algorithm runs roughly 2 times faster than the bound in equation (3).

The bounds in equation (3) are worst-case upper bounds which could be improved for specific classes of graph-like worlds, especially if the algorithm could be correspondingly adjusted. In addition, there are a number of stages in both the validation and exploration algorithms where heuristics are applied based on "likely" scenarios; we believe this improves their performance in typical graphs although the worst-case performance is unmodified. Unfortunately, the difficulty of establishing a suitably general yet meaningful formalization of an "average" graph is a long-standing problem in the field and so these heuristics cannot be readily evaluated in the context of our worst-case analysis. For example, in the validation code the following optimizations occur during the validation of the non-spanning tree edges:

- If there are no non-tree edges out of a node then the node does not require the marker drop and the tree search for edges leading to the node (steps 2a-d in the presentation of the algorithm). This saves $2(N-1)$ steps per node lacking non-tree edges.

- When searching for nodes with edges leading back to the node at which the marker has been dropped (steps 2c-d), it is not necessary to traverse subtrees which do not

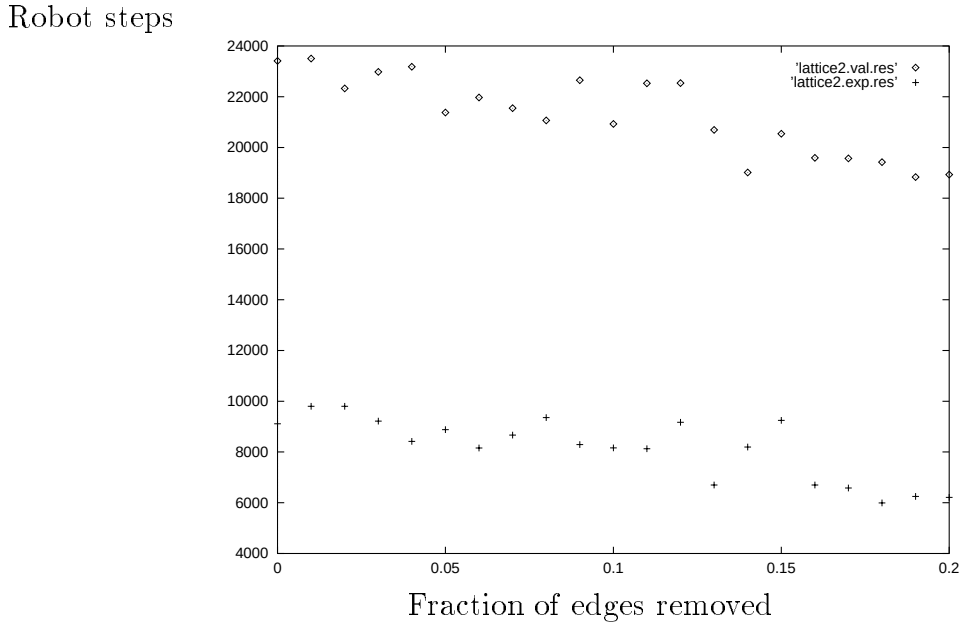
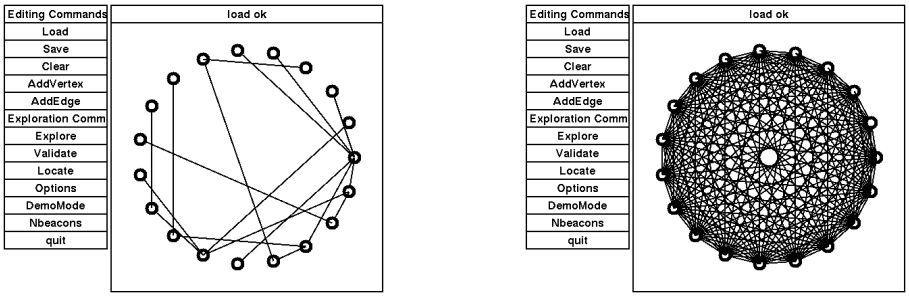


Figure 4: Regular lattice with holes. The exploration algorithm (+) consistently outperforms the validation algorithm ($\diamond$).

have map edges that return to the node at which the marker was dropped.

Similar optimizations have been applied to the exploration algorithm. In addition, in both algorithms there are a number of steps in which nodes can be visited in any order (as long as all of the nodes having a particular property are visited). In particular, nodes can be visited in an order which would tend to reduce motion of the robot. We will return to this point later when we consider the addition of metric data to the validation and exploration algorithms.

Random adjacency matrix: A second class of graphs we consider are those with random node interconnectivity. Figure 5(a) shows a random spanning tree of 19 nodes. Additional edges were added to random spanning trees until a complete graph (i.e., all possible edges) was obtained. This is shown in Figure 5(b).



(a) (b)

Figure 5: Sample random adjacency graphs. (a) corresponds to the tree case, while (b) corresponds to the fully connected (regular) graph case.

The number of moves made by the robot while validating or exploring the random graphs is shown in Figure 6. The horizontal axis shows the fraction of tree edges present in the graph. Random adjacency matrices were generated with M edges. If a given node was not connected to the bulk of the graph an extra edge was added above the M edges to ensure the graph was connected. The left hand side of the graph (0.0-0.04) corresponds to the type of graph shown in Figure 5a while the right hand side of the graph (1.0) corresponds to the graph shown in Figure 5b.

For the family of random adjacency matrices indexed by the fraction p of edges of the complete graph we have:

$$d_{max} \leq N - 1$$
$$M = p(N^2 - N)/2.$$

thus for $p >> 0.05$ (the graph is connected) the number of steps required by the validation and exploration algorithm is bounded by

$$\begin{array}{lll} explore & \leq -3N^2 + 6N + 2 + p(\frac{7}{2}N^3 - 4N^2 + \frac{1}{2}N) & \leq O(N^3) \\ validate & \leq (4 + 2p)N^2 - (4 + 2p)N - 6 & \leq O(N^2) \end{array} \quad (4)$$

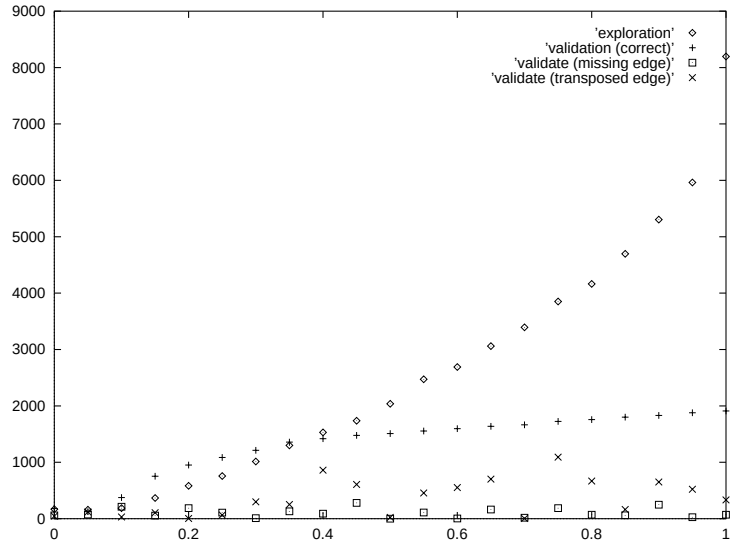


Figure 6: Random graph results. The horizontal axis is the fraction of edges in the graph relative to the complete graph. The vertical axis gives the number of robot moves for the different cases mentioned. We observe that for a sufficiently well connected graph it is confirmed that the validation algorithm outperforms the exploration algorithm.

Note that if the random graph became disconnected, random edges were added to unconnected nodes connecting them to the graph. Thus the bounds above under-estimate the costs for small p.

As the connectivity of the graph increases, the performance of the validation algorithm relative to the exploration algorithm improves. The validation algorithm "wins" in that its basic step is to drop a marker in a particular location and then visit all nodes with unexplored edges to see if they are incident on the node containing the marker. Thus, as the connectivity of the graph increases, validation gets more work done for a given traversal of the tree. On the other hand the exploration algorithm must continually visit the boundary of the explored subgraph S and more and more edges extend from it as the connectivity of the graph increases.

Although validation is a less expensive operation in terms of worst-case mechanical complexity, unless the domain under consideration is highly interconnected, it may be more desirable to re-explore the environment, rather than validate an existing graph.

5 Self-location and map validation with unknown initial pose

Now consider the more general problem in which the robot is given a map of its environment, but is not told its location and orientation with respect to the map. In this section we develop an algorithm for performing two tasks at once: validating the correctness of the map, and, in the case in which the map is correct, finding the correspondence between the map and the world. This is the *self-location problem*, which in our case involves identifying the initial vertex on the map and/or the correct "orientation" (i.e., the mapping between physical exits from that vertex and map edges incident on that vertex).

Two major issues are involved here:

- Under what conditions does the problem have a unique solution? In graphs with symmetries it may not be possible to identify uniquely the robot position and orientation.

- Can the problem can be solved efficiently? This paper gives an algorithm for the *self-location* problem that uses $O(N^3)$ moves.

The idea behind the self-location algorithm is first to form all possible hypotheses using the given map, corresponding to all possible initial vertices and orientations (i.e., their reference edges) in the map, and then to explore the graph, discarding hypotheses which are found to lead to inconsistencies during exploration. The number of such hypotheses is $\sum_{i=1..N} d(v_{Ci})$ where $d_{v_{Ci}}$ is the degree of vertex i on the map C and N is the total number of vertices. If d_{max} is the maximum degree of any vertex, the number of hypotheses is bounded by Nd_{max}. Under our assumption that there are no instances of multiple edges between a single vertex pair, or edges with both ends attached to the same vertex, then $d_{max} < N - 1$ and there will be no more than $N(N-1)$ or $O(N^2)$ hypotheses in total.

The algorithm for self-location and validation is much like the exploration algorithm described in [14] in terms of the physical steps of the robot, except that additional data structures are maintained as the robot moves. For each hypothesis, consisting of an initial pose depicted on the map, a correspondence is maintained between world nodes/edges and map nodes/edges as the robot carries out the exploration algorithm. Whenever the information the robot senses about the real world does not match the information modelled by a specific hypothesis, then that hypothesis is rejected. The following hypothesis rejection conditions correspond to all the instances within the algorithm where sensed information must be associated with the map, and therefore these conditions are both necessary and sufficient.

1. When the robot moves from one location to another, the robot's position on each hypothesis is updated. If the degree of the map node and the world node disagree, then that hypothesis is rejected.

2. When the exploration algorithm finds the marker at a vertex of S which is different from the vertex expected based on the hypothesis, the hypothesis is rejected.

3. When the exploration algorithm discovers that a particular edge index disagrees with the edge's index on the map according to the hypothesized initial pose, then that hypothesis is rejected.

4. When the exploration algorithm fails to find the marker, while at the same time the prediction based on the hypothesis is that the vertex should belong to the explored subgraph, and hence the marker should have been found, then the hypothesis is rejected.

When the exploration process is complete, one of the following cases must hold:

1. No hypotheses remain. In that case no starting pose was consistent with the world model, and the map must be incorrect.

2. One (or more) hypotheses remain. Then for each of these hypothesized starting points, there does not exist a sequence of operations or sensations that the robot can perform that illuminates any inconsistency between the hypothesized initial pose(s), the map, and the robot's true starting pose and the true environment. Thus the map can be used for navigation and path planning, and any one of the starting pose(s) can be assumed to be correct. Figure 7 shows the surviving hypotheses for the validation algorithm with unknown initial pose. Of the 18 possible initial hypothesized poses, 6 remain after map validation.

A detailed description of the self-location algorithm is given in the appendix.

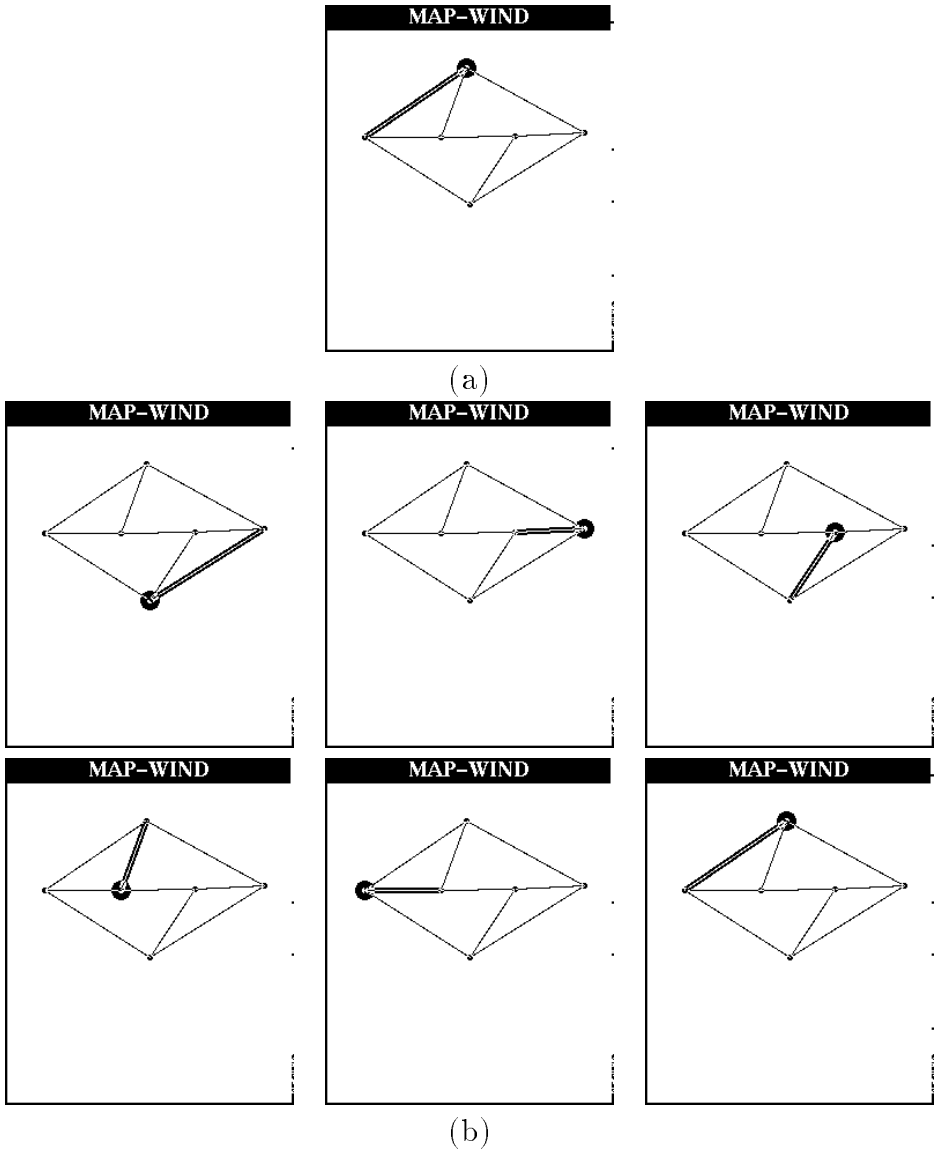


Figure 7: Part (a) shows a graph with symmetries and a reference position and orientation. Part (b) shows all of the possible symmetries of the graph that the implemented self-location algorithm discovered in terms of a reference position and orientation indistinguishable from the one of part (a). The robot's initial pose is indicated by a heavy circle and a heavy line.

6 Discussion and Conclusions

We have presented new algorithms that permit a mobile robot to use *a priori* graph-like maps of a known environment. In [14] we demonstrated how a robot could explore its environment using only a single movable marker. This paper presented *self-location* and *map validation* algorithms which allow a robot to localize itself and validate a previously computed map. We have analyzed the mechanical time (i.e., motion) complexity of the various algorithms and have demonstrated that for certain classes of graphs (with the relatively sparse connectivity characteristic of typical real environments), it may be less expensive to re-explore the environment than to validate the map, even when the worst case complexity analysis indicates that the opposite could be true. We consider this to be an open question, if we seek an analytical answer. Based on our experiments, we propose a heuristic that has to do with the connectivity of the graph. So a simple rule is to ignore the given map and reexplore if the given map has relatively low connectivity. Section 4 provides some insight as to what low connectivity means.

Concerning the choice of the number of markers, the more (movable) markers are available, the faster the exploration, validation and self-location processes. Considerations that may limit the number of markers are practical: the robot may be able to only carry up to a certain number of markers. Making the size of the markers smaller (thus allowing the robot to carry more markers) will make the markers harder to recognize.

In a practical implementation of the algorithm, there is a variety of errors to be considered such as failure to recognize the marker or recognizing a marker where none exists, failure to execute a planned path correctly (taking the wrong exit somewhere). In an exploration context, if such failures are detected before an incorrect explored subgraph S has been constructed, repetition of the operations by the robot can help the process recover from the error. Otherwise, the robot may discover inconsistencies later, if the error is not systematic, and it will then be forced to start the exploration all over again.

If the error is systematic (e.g., it *always* misses a particular exit) then it will become a permanent feature of the map.

Both the self-location and map validation algorithms terminate if the map is found to be incorrect. One problem that is not addressed in this paper is how to deal with changes to the environment or errors in the map. We call this the *map recovery problem*: given a partially incorrect map of an environment, how can one (optimally or near-optimally) determine where the errors in the map are and correct them. There are various ways in which one map can be modified to produce another. If the allowable set of transformations is limited, it may be possible to devise efficient algorithms to fix a "defective" map. Possible constraints could include to assume that the damage is local: only node deletion might be allowed but no node addition; only edge deletion might be allowed, *et cetera*. Many such constraints have real world analogues (for example, only edge deletions might correspond to navigating in an office where some of the doorways had been closed). We leave this as a future research problem.

6.1 The addition of metric aids

The algorithms presented in this paper operate independently of any metric positional information. Metric information can be added in many different ways. One option would be to add the metric location of a node to the "node signature" [18]. Thus nodes would only be confused with each other if they had the correct degree and they were sufficiently close to each other. One potential problem with this approach would be that "close enough" could be quite difficult to evaluate in practice.

A second approach would be to use the algorithms as presented here, but to use metric information as a heuristic to order potential nodes throughout the algorithms. In each of the *exploration*, *validation*, and *self-location* algorithms, there are a number of steps which require traversing a subset of the nodes in the world searching for the

marker. As described above, the order in which these nodes are searched is arbitrary. If metric information is available then this could be used to choose to search "near" the expected node first. One advantage of this second approach would be that "close enough" would not have to be defined, and failure of the metric information would not impair the correctness of the algorithm, only its performance.

6.2 Other non-metric graph exploration aids

In this work and in earlier papers we have assumed that the robot is equipped with one or more unique markers that can be used to help the robot explore (or validate or locate itself) within its environment. Markers are not the only possible aid that a robot can use to explore its environment. There are many others. One alternative is to accept ambiguous world models as inevitable and attempt to retain the simplest model, or use heuristic cues about the world [11]. Other alternatives are sketched below.

Immovable markers: A single uniquely identifiable immovable marker is sufficient to solve the exploration, validation and self-location problem. Each node in the graph v becomes uniquely identifiable by a "door sequence" $P(v)$ back to the "origin" (the location where the immovable marker lies). The validation of a newly visited vertex v_a, for which the movable marker was used, can now be solved as follows. For each vertex v of the explored subgraph S, the robot starts at v_a and executes the reverse $P(v)$ and again the forward $P(v)$ to return to v_a. If the marker is found at the end of the reverse $P(v)$, then v_a must be identical to v. The mechanical cost of this algorithm can become extremely high. A modification to make the operation more efficient is to always pick $P(v)$ to be the shortest path in S from the marker to node v. As S is augmented, $P(v)$ may have to be updated for each node v to take advantage of possible shortcuts. The use of r uniquely identifiable immovable markers can make the method even more efficient, by associating with each node v a path $P_i(v)$, where i is the marker that is

closest to v. When a newly visited vertex v_a must be validated against a vertex v of the explored subgraph S, the robot will choose the reverse path $P_i(v)$. The average savings over the single marker case depends on how "uniform" the coverage is. Finally, we can envision the hybrid case, in which both movable and immovable markers are available. In the hybrid case, the robot has a choice between two vertex validation operations, one that uses movable and the other that uses immovable markers. Cost considerations will determine which one is the more sensible for each vertex.

A single movable directional marker: Consider the exploration algorithm in [14] and also sketched in §3 but in which the marker is directional, that is, it can be placed so as to point in a particular direction. The exploration code can then be simplified. Whenever the robot drops the marker it drops it pointing back towards the explored part of the graph. Then whenever the robot encounters the marker it can trivially determine the edge index of the newly explored edge. Without a directional marker, this process of "validating the back-link" is expensive. As validation of backlinks makes up a large part of the effort in exploration, this minor change in the complexity of the marker will have a major impact on the algorithm's performance.

Acknowledgments

Financial support from NSERC Canada, and the Ontario-Quebec interchange program is gratefully acknowledged. We thank Derek Corneil, Xiaotie Deng and Andy Mirzaian for many helpful discussions, and Jack Snoeyink for pointing out the random graph literature. The anonymous reviewers made useful suggestions for improving the presentation.

References

[1] R. Arkin. Integrating behavioural, perceptual, and world knowledge in reactive navigation. *Robotics and Autonomous Systems*, 6:105–122, 1990.

[2] Nicholas Ayache and Olivier D. Faugeras. Maintaining representations of the environment of a mobile robot. *IEEE Journal of Robotics and Automation*, Vol. 5, NO.6:804–819, 1989.

[3] Kenneth Basye and Thomas Dean. Map learning with indistinguishable locations. In M. Henrion L. N. Kanal J. F. Lemmer, editor, *Uncertainty in Artificial Intelligence 5*, pages 331–340. Elsevier Science Publishers, 1990.

[4] B. Bollobas. *Random Graphs*. Academic Press, London, 1985.

[5] J. A. Bondy and U. S. R. Murty. *Graph Theory With Applications*. North-Holland, New York, 1976.

[6] R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[7] R. Brooks. A robot that walks: Emergent behaviours from a carefully evolved network. *Neural Computation*, 1(2):253–262, Summer 1989.

[8] J. Connell. *Minimalist Mobile Robotics: A Colony-style Architecture for an Artificial Creature*. Academic Press, Boston, MA, 1990.

[9] I. Cox. Blanche: Position estimation for an autonomous robot vehicle. In *Proceedings of the IEEE/RSJ International Workshop on Robots and Systems (IROS)*, pages 432–439, 1989.

[10] Ernest Davis. *Representing and Acquiring Geographic Knowledge*. Pitman and Morgan Kaufmann Publishers, Inc., London and Los Altos, California, 1986.

[11] G. Dudek, P. Freedman, and S. Hadjres. Using local information in a non-local way for mapping graph-like worlds. In *Thirteenth International Joint Conference on Artificial Intelligence*, pages 1639–1645, 1993.

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

[13] G. Dudek, M. Jenkin, E. Milios, and D. Wilkes. Map validation in a graph-like world. In *Thirteenth International Joint Conference on Artificial Intelligence*, pages 1648–1653, 1993.

[14] G. Dudek, M. Jenkin, E. Milios, and David Wilkes. Robotic exploration as graph construction. *IEEE Trans. on Robotics and Automation*, 7(6):859–864, 1991.

[15] A. Elfes. Sonar-based real-world mapping and navigation. *IEEE Journal of Robotics and Automation*, 3(3):249–265, 1987.

[16] S. P. Engelson and D. V. McDermott. Maps considered as adaptive planning resources. In *AAAI Fall Symposium on Applications of Artificial Intelligence to Real-World Autonomous Mobile Robots*, Cambridge, MA, 1992.

[17] D. Kortenkamp and T. Weymouth. Topological mapping for mobile robots using a combination of sonar and vision sensing. In *Conf. of the American Association for Artificial Intelligence (AAAI)*, pages 979–984, 1994.

[18] B. Kuipers and Y-T. Byun. A robot exploration and mapping strategy based on a semantic hierarchy of spatial representations. *Robotics and Autonomous Systems*, 8:47–63, 1991.

[19] B. Kuipers and T. Levitt. Navigation and mapping in large-scale space. In *AI Magazine*, pages 25–43, 1988.

[20] Jean-Claude Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, Stanford University, 1991.

[21] V. Lumelsky and A. Stepanov. Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2(4):403–440, 1987.

[22] M. Mataric. Integration of representation into goal-driven behaviour-based robots. *IEEE Transactions on Robotics and Automation*, 8(3):304–312, June 1992.

[23] A. Meng. Free space modelling and geometric motion planning under location uncertainty. In *Workshop on Spatial Reasoning and Multisensor Fusion*, pages 430–440. Morgan Kaufmann, 1987.

[24] F. Preparata and M. Shamos. *Computational geometry : an introduction.* Springer-Verlag, New York, 1985.

[25] R. Smith and P. Cheeseman. On the representation and estimation of spatial uncertainty. *International Journal of Robotics Research*, 5(4):56–68, Winter 1986.

A Statement of the Self-location Algorithm

THE ALGORITHM

The following is a self-contained statement of the self-location algorithm. The self-location algorithm has the same mechanical steps as the exploration algorithm presented in [14], with the addition of hypothesis generation and testing, which takes place entirely in the robot's mind.

We denote the mapping from the model S that the exploration algorithm is contructing to world W by ϕ; $\phi(v)$ of a vertex v in S is the real-world vertex to which the robot's label corresponds. The additional data structure is a set H_S of hypotheses $\{H\}$. For each hypothesis H another mapping μ_H is established between the model S and the hypothesis H; $\mu_H(v)$ of a vertex v in S is the vertex of hypothesis H to which v corresponds.

A hypothesis H is a possible registration of the given map M with the model S, and therefore with the real world via the mapping ϕ^{-1}.

A "soft" marker is used for each real marker on hypothesis H.

Definitions of the subroutines used by the algorithm are given following the statement of the top-level algorithm. In the top-level algorithm, mechanical steps are in italics, and electronic steps are in roman. Note that all steps related to the maintainance of hypotheses are electronic steps.

Comments are preceded by the symbol ▷.

```
▷
▷ Initially the robot starts at some vertex v_initial with
▷ the markers on the floor
▷ Each hypothesis H is defined by assigning a possible value to the mapping μ_H:
▷ μ_H(v_init) and μ_H(e), for all edges e incident on v_init
▷ As the exploration proceeds, either the mapping μ_H is completed,
▷ or the hypothesis is rejected
▷ At any time, mapping μ_H has been defined for
▷ all vertices and edges within model (explored subgraph) S.
▷
for i from 1 to k
    pickup(marker_i)
S := ({ v_initial}, {}) U := {φ^-1(e) | e is incident with φ(v_initial)}
        ▷ Elements of U are pairs (v, k), where v is the known vertex and k is the
        ▷ index for each edge e
        for each e in U
            index(e, v_initial) := consistent ordering of φ(e)
                    w.r.t. an arbitrary edge incident on v_init
        end for
        v_current := v_initial
        loop
            exit when U = {}    ▷ i.e. no unexplored edge remains
            ▷
            ▷ Select a subset of the unexplored edges in U
            ▷ of up to k elements and call this E
            ▷
            choose(E)
            ▷
            ▷ For each new edge (v_1, l) in U, move the robot to φ(v_1)
            ▷ Move physically along the new edge, drop the marker there φ(v_2)
```

▷ and immediately return to the known graph $\phi(v_1)$.
▷ Note that although vertex v_2 is not yet identified with respect to S,
▷ and hence $\mu_H(v_2)$ is not yet defined, we can determine that
▷ $\mu_H(v_2) =$ otherVertex(edge($\mu_H(v_1), l), \mu_H(v_1)$)
▷ and whether $\mu_H(v_2) \in \mu_H(S)$.
▷
for each $e_i = (v_1, l) \epsilon$ E, with known vertex v_1
    walk($\phi(v_{current})$, $\phi(v_1)$)
    followEdge($\phi(e_i)$) ▷ to $\phi(v_2)$
    ▷
    ▷ Hypothesis test 1. Vertex degrees do not match.
    if (degree (otherVertex(edge $(\mu_H(v_1), l), \mu_H(v_1)$) $\neq$ degree $(\phi(v_2))$
        then reject (H)
    ▷
    drop($marker_i$)
    followEdge($\phi(e_i)$) ▷ to $\phi(v_1)$
end for
▷
▷ Search through the graph looking for the dropped markers
▷
search(S, **markerFound**, **markerLocation**) ▷ and set $v_{current}$
    ▷ **markerFound** is now a k-vector of boolean flags
    ▷ **markerLocation** is a k-vector of vertex numbers
for each i from k down to 1
    if $markerFound_i$ then
        ▷
        ▷ Found a marker at vertex v_2 of S.
        ▷
        ▷ Hypothesis test 2. Marker found at the wrong vertex
        if ($\mu_H(v_2) \neq$ otherVertex(edge $(\mu_H(v_1), l), \mu_H(v_1)$)
            then reject (H)
        ▷
        ▷ Determine index of e_i w.r.t. to its unknown end v_2:
        ▷
        walk($\phi(v_{current})$, $\phi(v_1)$)
        drop($marker_i$)
        walk($\phi(v_1)$, $\phi(v_2)$)
        for each edge f leaving v_2
            followEdge(ϕ(f)) ▷ to $v_{unknown}$
            if $marker_i$ at $v_{unknown}$ then

```
            index(e_i, v_2) := index(f, v_2)
            ▷
            ▷ Hypothesis test 3. Edge index mismatch.
            if (index (e_i, v_2) ≠
                    index (μ_H(f), otherVertex(edge (μ_H(v_1), l), μ_H(v_1))
                then reject (H)
            ▷
            remove e_i from U
            pickup(marker_i)
            v_current := v_1
            exit for
        end if
        followEdge (φ(f)) ▷ back to v_2
    end for
else
    ▷
    ▷ Didn't find the marker. This is a new vertex
    ▷
    ▷ Hypothesis test 4. Vertex expected in S not found.
    if otherVertex(edge (μ_H(v_1), l), μ_H(v_1)) ∈ μ_H(S)
        then reject (H)
    ▷
    walk(φ(v_current), φ(v_1))
    followEdge(φ(e_i)) ▷ to φ(v_2)
    pickup(marker_i)
    add v_2 to S
    add e_i to S
    ▷
    ▷ Hypothesis maintenance action: Set value of μ_H(v_2) and μ_H(e_i)
    μ_H(v_2) := otherVertex(edge (μ_H(v_1), l), μ_H(v_1))
    μ_H(e_i) := edge (μ_H(v_1), l)
    ▷
    index(e_i, v_2) := 0
    for each other edge f leaving φ(v_2)
        index(φ^{-1}(f), v_2) :=
                consistent ordering with respect to e_i
        add φ^{-1}(f) to U
    end for
    for each other marker_j at φ(v_2)
        pickup(marker_j)
```

$markerFound_j := true$
$markerLocation_j := v_2$
end for
end if
end for
end loop

subroutines:

choose(E)
▷ choose up to k edges $e_1, e_2, ...e_k$ from U such that the known
▷ incident vertex of e_1 is closest to $v_{current}$, and
▷ for $i = 1, 2, ...k-1$ we have that the known incident vertex of
▷ e_{i+1} is closest to the known incident vertex of e_i
run shortestPath k times to find edges satisfying the above description.

walk(v_{from}, v_{to})
run shortestPath to get shortest path (e_1, e_2,... e_k)
from v_{from} to v_{to}, through S.
for i from 1 to k
followEdge($\phi(e_i)$)
end for

search(S, **markerFound**, **markerLocation**)
▷ a breadth-first approximation
▷ to a travelling salesman problem solution seems appropriate
▷ since markers are likely to be close to current vertex in S
▷ Do traversal of S, stopping when k markers have been encountered
▷ or all vertices have been visited
run Kruskal's algorithm to get min spanning tree of S
for i from 1 to k set $markerFound_i$ to $false$
do breadth-first traversal, taking "short cuts" across non-tree edges to next vertex where possible. We consider two versions here:
a) only take single-edge short cuts (check if current and next vertex in traversal are adjacent).
b) run shortestPath to find shortest path from current to next vertex at each step.
whenever a marker i is encountered, set $markerFound_i$ to $true$ and set $markerLocation_i$ to the vertex number in S, and

execute $pickup(marker_i)$.

shortestPath(source)

▷ do a breadth-first labelling of vertices starting from vertex "source"
▷ where labels indicate previous vertex in path back to sink. This
▷ is inspired by the Ford-Fulkerson labelling algorithm.
▷ It suffices for finding shortest paths in an unweighted graph,
▷ taking O(n) time. Use Dijkstra's algorithm if there are weights
▷ on the edges (O(n^2)).

```
label source
loop
    for each newly-labelled (i.e. from last loop pass) vertex v
        label all vertices adjacent to v
    end for
end loop
```