Maze Solving by Learning State Topologies

Derek Tingle
Computer Science Department
Swarthmore College
Swarthmore, PA 19081
dtingle1@swarthmore.edu

Erick Ball
Computer Science Department
Swarthmore College
Swarthmore, PA 19081
jball2@swarthmore.edu

Malcolm Augat
Computer Science Department
Swarthmore College
Swarthmore, PA 19081
maugat1@swarthmore.edu

Abstract

In this paper we present the Maze Solving by Learning State Topologies (MSLST) algorithm for maze localization and solving. The algorithm works in three phases. In phase I the robot wanders through the maze randomly, building up and correcting a set of growing neural gas (GNG) states. Phase II consists of the robot wandering around the maze and creating a graph representation of the maze, with distinctive states it sees as nodes and actions as edges connecting the nodes. Finally, in phase III the robot uses the graph to navigate and solve the maze. We evaluated MSLST on two different mazes and with different error thresholds for the equilibrium-GNG algorithm, and compared it with maze-solving by choosing random directions and by wall-following. In all cases MSLST performed significantly better than the other two methods.

Introduction

The field of autonomous mobile robotics is filled with interesting potential applications. This is because one of the primary motivations behind the field is the possibility of using robots to perform tasks that are dangerous, expensive, or simply impossible for humans. Yet even though autonomous robots are used to perform complex tasks that range from land mine detection to mapping the surfaces of other planets, the fundamental problem of localization underlies all potential applications of autonomous mobile robots. Without the ability to determine their locations within their environment, both autonomous and controlled mobile robots would have severely limited functionality.

Localization, even when an accurate map is available, is not a trivial problem. Robots have two sources of information about their location, sensor inputs and motor outputs, and both of these sources are inherently noisy. In theory, a robot could integrate its motor outputs to determine its location, but in practice small errors in the motor outputs lead to large errors in position. Additionally, using sensor inputs to localize can lead to the problem of perceptual aliasing: multiple distinct locations in an environment might give indistinguishable sensor inputs. These two sources can be combined, however, to minimize error in location estimation.

We developed the Maze Solving by Learning State Topologies (MSLST) algorithm to create a topological representation of a maze that was then used to localize the robot within the maze. The algorithm first quantizes the robot's sensory into distinctive states inputs as it traverses the maze. Using these states, the robot learns a topological representation of the environment. Once the topological representation is built, the robot uses a history of previously seen distinctive states and actions to localize itself within the maze and find the shortest path to a specified exit.

Related work

Provost et al. [2] introduce a method called Self-Organizing Distinctive-state Abstraction (SODA) for navigating a continuous world using a discrete perceptual representation of the world. The algorithm works by first exploring the environment using primitive motor actions, such as turning or moving a fixed distance. During this initial exploration the robot creates a Self-Organizing Map (SOM) of the environment. The authors chose to use a variant of the SOM algorithm that uses Growing Neural Gas (GNG) to quantize sensory vectors. Once the sensory stream has been broken up into distinctive features, the robot constructs high-level actions that are used to transition from one distinctive state to another. These actions are based on trajectory-following and hill-climbing algorithms. The trajectory-following algorithm is used to move from one state to another. Once in a new distinctive state, a hill-climbing algorithm is used to move the robot to the location within the distinctive state that is most similar to the exemplar of that distinctive state. In order to navigate through the environment, the authors used reinforcement learning on high-level actions to train a simulated robot to move to a desired location in the environment. In the experiment, the authors trained the robot to move from one arm of a T-maze to the base of the maze. They found that the time needed to train the robot was significantly lower when using high-level actions compared to learning without the high-level actions. The authors did note two problems with the SODA algorithm. The first is that the hill-climbing algorithm requires the robot to make several small movements at each location to determine the direction of the distinctive feature maxima. In the experiment, a robot using low-level actions was able to complete the maze much faster than the robot using high-level actions. Another potential problem is the issue of perceptual aliasing. The experiment conducted by the authors used a maze that had limited possible aliases by design, but for the SODA algorithm to function in real world environments or larger simulated environments with many possible state aliases, a method to disambiguate aliases would be necessary. The authors refer to the following paper for possible alias disambiguation methods:

Kuipers and Byun [3] describe a topological map-building approach that closely resembles Provsost et al. [2], (which Kuipers co-authored). The main difference between the two papers is Kuipers and Byun provide a more general and theoretical outline for constructing and navigating a topological map. Whereas previous approaches typically used sensor inputs to create a geometric map and then created a topological map from the geometric map, the authors argue that metrical information should be used to augment a topological map rather than vice versa. The authors refer to an unspecified distinctiveness measure that can be used to locate local maxima instead of using GNG. The authors also include two methods for identifying previously visited locations. First, metrical information such as distance traveled and net change in orientation is maintained throughout the exploration phase. A new location is only considered a possible alias of a previously seen location if the metrical information is similar, in addition to the distinctiveness measure. If the metrical information is similar for two identical distinctive states encountered, the rehearsal procedure is activated. The rehearsal procedure assumes the robot is in the previously observed distinctive state. The robot then attempts to travel down a previously taken edge to another previously seen distinctive state. If the observed state matches the expected state, the prior state is considered to be the previously known state, otherwise the prior state is considered to be new.

Fritzke [1] presents the Growing Neural Gas (GNG) algorithm for vector quantization. He extends the work of Martinez and Schulten [4] who first described a neural gas network for learning topologies. While Martinez and Schulten's method adapted a fixed number of units to create a topology of an input space, GNG incrementally adds units so that the number of units needed to approximate a distribution in the input space does not need to be known in advance. The algorithm works as follows:

0: Select two random units and connect them with an edge for each input signal, γ, observed.
1: find the two closest units, s1 and s2, to γ. If not connected, connect them; set age to 0.
2: Increment the age of each edge connected to s1.
3: Add the squared distance between s1 and γ to s1.
4: Move s1 toward γ and each of s1's neighbors slightly toward γ.
5: Remove each edge in the graph with an age over a threshold; remove units that have no edges.
6: Decay the error on each node in the graph.
Every λ steps, add a unit between the node with the highest error and its neighbor with the highest error.

Provost et al. [2] used a variation of GNG known as equilibrium GNG. The only difference between equilibrium GNG and the original GNG algorithm described by Fritzke is that at a new node is only added if the average error of all of the nodes in the graph is above some threshold. This way new units are only added when needed.

Another topological approach is described by Mataric [5]. Unlike Kuipers and Byun [3], Mataric uses a distributed data structure to represent the graph. The graph is linear with each node connecting to two other nodes except the first and last nodes which only connect to one other node. As the robot wanders through the environment each node in graph simultaneously receives data from a landmark detector, a compass, and sonars. Nodes can pass expectation, deactivation, and wakeup messages to their immediate neighbors. At any given time the node with the most recently seen landmark is considered to be the active position. When a landmark is detected, it is broadcast to every node. Each filled node then compares the observed landmark and compass heading to its own landmark and compass heading. If no nodes match, the new landmark is assigned to the node following the current active node. A linear graph cannot handle cycles in the environment, so the author uses a system in which a switchboard can pass messages to non-neighboring nodes if a potential cycle exists. A running integration of compass heading and velocity is maintained to determine approximate location. This way it is possible to determine if a landmark has previously been seen or if it is a new landmark. To navigate to a goal, a message passing equivalent of breadth first search finds the shortest topological path to the goal.

Experimental Setup

The MSLST algorithm uses a three-stage process to solve a maze, first using GNG to abstract perceptions of the maze into discrete states, then by mapping the topology of those states within the maze into a graph, and finally using that graph to localize a robot within the maze and find the most direct path to the destination.

MSLST is written for the Pyrobot Stage Simulator, and can allow a simulated robot to navigate any maze consisting of corridors of the appropriate width.

The Robot

We used a pioneer-like robot, capable of rotating in place and equipped with a SICK laser range finder. The range finder is configured to give 360-degree coverage with 72 evenly spaced sensors, each of which gives a range value between 0 and 2.5 robot units at every time step. The robot also has a simulated compass, which allows it to rotate its sensor inputs so that a particular location is distinct no matter which way the robot is facing. The compass also allows the robot to rotate itself to a specified heading.

The Mazes

We used two different mazes to test MSLST. The first is a simple J shape (the J-maze), while the second is much more complex, containing several dead ends, a four-way intersection, and a loop. Each maze also has a defined destination zone in the bottom right corner, which is the robot's goal during navigation.

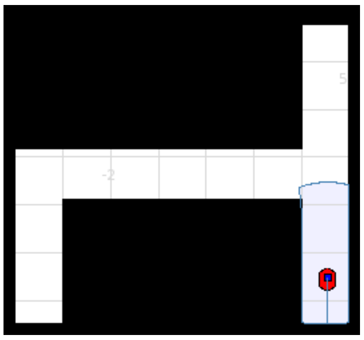


Figure 1: The J-maze, with robot in the destination zone.

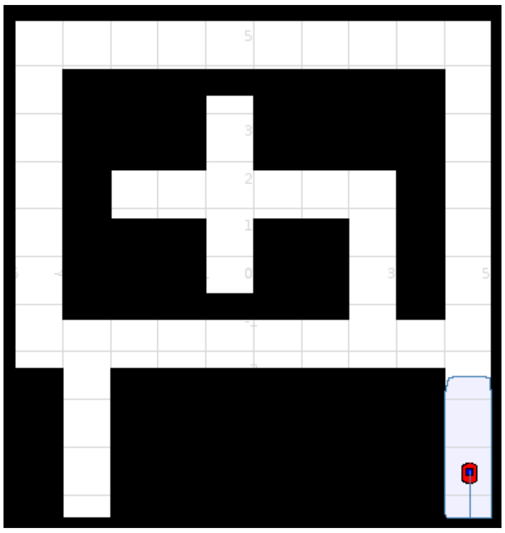


Figure 2: The large maze, with robot in the destination zone.

The Experiments

In each test, the robot is placed at a random starting location within the maze. It then navigates its way to the destination zone. The number of time-steps it takes to reach the destination is recorded. If it fails to reach the destination within a specified number of time steps (200 for the J-maze and 500 for the large maze), the experiment times out. We ran each test 25 times. The tests compared three brains on each of the two mazes, and for MSLST on the large maze we tested two different GNG error-values, for a total of eight experiments. The three brains, all of which use the layered corrections architecture described below for their movement, are MSLST, an Avoid brain that wanders randomly (but does not backtrack unless it reaches a dead end, and does not repeat failed actions), and a Wall-Follow brain that always sticks to the wall on its right.

Robot Operation: Layered Corrections Architecture

The brains each operate the robot using a very simple interface that consists of high-level actions. There are four possible high-level actions: north, south, east, and west. The brain specifies one of these actions, and the robot moves approximately one unit in approximately the specified direction, and requests another action from the brain. Each high-level action can take several time steps, since the robot may have to turn before it can move and needs multiple time steps to move far enough. If the specified movement would take the robot into a wall, it is not carried out and the brain is notified that the action failed. The MSLST brain receives one additional input–the number of the GNG state that most closely matches the current input vector from the range finder.

The advantage of using high-level actions is twofold: first, it reduces the number of states GNG encounters, since it need only deal with them at the end of a high-level action; second, the path-finding algorithm can focus on navigation instead of worrying about low-level motor control. It acts as an alternative to the trajectory-following and hill-climbing approaches in SODA [2], in that they both use sensory data to correct the enormous inaccuracies inherent in tracking position with the cumulative motor outputs. Hill climbing does so far more explicitly, but it also takes more time and requires having reversible motor primitives.

These high-level actions specified by the brain are carried out using a set of layers that make corrections to the desired action, similar to a subsumption architecture as is used by Kuipers and Byun [3]. The purpose is to provide a simple interface to the brains while preventing the robot from getting stuck and only allowing it to move in directions where there is empty space. This system is hard-coded, but it could also theoretically be built using a neural network and/or evolutionary algorithms–none of this specific implementation is crucial to the working of MSLST. Each layer is a class with a correct() method that takes in the action output by the previous layer. The top layer picks the motor actions it thinks most appropriate to accomplishing the high-level action from the brain, then each subsequent layer checks the validity of that action and modifies it. This system provides more flexibility than subsumption; for instance, if the Orient layer decides on a necessary amount of rotation to get the robot facing straight, and then Center decides it needs to be facing at a small angle, it can add the two values instead of simply overriding Orient. The layers are:

Move:
The top layer, Move, expands the high-level action into the necessary rotation and subsequent translation components. It translates the direction into a heading (0, 90, 180, or 360) and compares that to its current heading to calculate how much it needs to turn. It makes no distinction between forward and backward motion; it does whichever requires the least amount of turning. First, it outputs a rotation proportional to the total amount it needs to turn (difference in degrees * .0105); because of the simulator's high noise level in the motor controls, this almost always overshoots or undershoots the desired amount of rotation, so it repeats until the actual heading is within 4 degrees of the desired heading, at which point it switches to translation (with a value of .25) for two time steps. It has no way of checking whether it moves the correct distance. After two steps of translation, it considers the high-level action complete. The action may also end earlier if a lower layer determines that it fails.

The next two layers, Orient and Center, both operate only when the robot appears to be in a corridor, rather than an intersection. It determines this by using the sums of opposite range sensors (the total distance between the opposite walls at each angle). Inside a corridor, it will find these distances to be long in one direction (either north-south or east-west, depending on the orientation of the corridor) and short in the other. In an intersection, it will find long distances in multiple directions. So of the 36 sensor pairs, if it can find 14 consecutive pairs with summed ranges less than 1.625 times the width of the maze's hallways, it decides it is not in an intersection.

Orient:
The Orient layer keeps the robot from turning perpendicular to the corridor it is in, since that would just allow it to run into walls. Orient determines the orientation of the corridor relative to the robot by finding the opposite-sensor pair with the minimum summed range (the one pointing perpendicular to the two walls). If that range is noticeably less than the width of the maze hallways, the robot is stuck in a corner instead of in an open hallway, so Orient adds a small random translation to help it get out. Otherwise, it rotates the robot to face more directly along the corridor (value .04*number of sensors between where the minimum is and where it should be); if this contradicts the rotation Move requested, then the high-level action fails and a new one will start on the next step.

Center:
This layer also operates only in corridors, and its goal is to help keep the robot centered in the corridor instead of near the walls. It only works when the robot is moving. It checks the ratio of the distances to the left and right walls, and if greater than 1.25, it adds a rotation of $\pm .05$ (depending on the direction of translation) that makes the robot move toward the center of the corridor.

Avoid:
Avoid is a last-resort system to keep the robot from running into walls, and operates whenever the robot is trying to translate. It checks the sensor values in the direction the robot wants to move against a buffer value (.9 robot units). For the three sensors directly in front, any range less than the buffer causes the robot's translation to be zeroed and the high-level action to fail (although if it has already moved in that direction for one time step before getting too close to the wall, MSLST will not record it as a failed action). It also checks the other eight sensors next closest to the center, which cause it to fail only if much less than the buffer distance; otherwise they add a small rotation to move the robot away from the obstacle.

Phase I: Building GNG States

During phase I, the robot wanders the maze at random, using the Avoid brain, to gather sensor data that GNG can use. It records its input vector at every time step (for approximately 4000 time steps, on the large maze). We then ran equilibrium GNG on the sensor data to categorize these vectors into discrete states, using several values for the error parameter (20, 50, and 100). For the subsequent phases, we used the error-100 states for the J-maze and both the error-20 and error-50 states for the large maze (we decided on a lower error threshold to account for its greater complexity).

Phase II: Mapping the Topology Into a Graph

The graph that represents the maze consists of nodes that contain the number of a GNG state, connected by edges that have the numbers of high-level actions as their values. Opposite high-level actions have opposite values, so if node A has an east connection to node B (which means that going east from node A will take you to node B), then node B also has a west connection to node A. Nodes can have multiple connections associated with the same action, if that action can lead to multiple states depending on the exact conditions–for instance, from a particular spot associated with one node, going east might sometimes lead to another state and thus another node, but other times it could fail if the robot is facing in a slightly different direction. Nodes can also have connections to a dummy node, which represent failed actions. The graph contains the list of nodes with their connections, and also a list of which nodes are associated with the destination zone (ones that were created while the robot was in that zone).

To build the graph, the robot wanders at random as in Phase I, but this time instead of recording sensor data it determines the closest-matching GNG state at the start of every high-level action. If the last high-level action succeeded, then normally it adds a node to the graph with the current state, and connects it to the previous one with the action that it just completed. If the last action failed, then it adds a connection from the previous node to the dummy node. When it reverses an action or string of actions and ends up in the same state it was in before, instead of adding a new node it traces the graph back to the old node and when it eventually reaches a new state it adds on from there, so that the graph can (in theory, at least) follow the topology of the maze instead of being linear. When the graph-building is complete, we calculate the length of the path from each node to the nearest destination node (as described below) and store it.

Phase III: Navigation Using the Graph

MSLST navigates by matching its current and past sensory states and actions with part of the graph, then following the actions encoded in the graph that lead to the nearest destination node (first simplifying that set of actions by compressing the path). Once it reaches a destination node (and thus believes it has solved the maze), it acts randomly (like Avoid) for eight actions, so that it will have a chance to get out of states aliased with the destination and recalculate its location. If it can't match its current state with any node in the graph or if the actions the graph suggests fail, it chooses an action randomly.

Localization:
After it is placed randomly in the maze, the robot has to determine which node on the graph best corresponds to its location in the maze so that it can find a path to the destination. It also recalculates its location after every high-level action. At every action, it stores its current state and the action that got it there into a list (leaving out failed actions and the states they resulted in), which it can then compare against the graph to localize itself. We used the last eight state/action pairs to localize when solving the large maze, and six for the J-maze. The localization algorithm first makes a list of all nodes within the graph that match the current state of the robot. It then recursively compares the previous action and state in the robot's memory with the neighbors of each of the nodes in the list to see if the node could still match that string of states and actions. For instance, if the robot sees sequentially states A, B, C, and D, all while traveling east, the algorithm first makes a list of all the nodes with state D (the most recent state). It then checks each of them to see if it has a "west" connection to a node with state C, and discards those that do not. It checks each of those state C nodes for a "west" connection to a state B node, and so on. It stops when it reduces its list of nodes to one, in which case that node is the best match, or when it runs out of states or actions to match to, in which case it chooses the node from the list with the shortest path to a destination node.

Path-Finding:
To find a path from the current node to the nearest destination node, we use a simple breadth first search along the graph, then make a list of the values of the edges connecting the source node to the destination node. This is the path, which we then compress. The robot takes the first action in the compressed path. If it fails, we recalculate the path, not allowing the first action to be equal to the one that failed (this results in a longer path).

Path Compression:
Because the path is often unnecessarily complex, we perform two types of simplification on it. The first is simply to remove any pairs of consecutive opposite actions–ones that just cancel each other out, e.g. "east" followed by "west". The second is to remove uneven strings of opposite actions–ones that almost, but not quite, cancel each other out. If a string of actions is followed by a string of opposite actions, each of length two or more and with lengths not differing by more than one, we remove all of them instead of just canceling out pairs. The reason for this is that motor noise causes high-level actions to cover varying amounts of distance, so a robot that goes down a north-south hallway and then back up it, ending where it started, often takes different numbers of north actions and south actions. They should nonetheless cancel each other out in the path. These two simplifications are repeated alternately until neither is possible, after which if the first action in the path is one the robot just failed at, we remove it from the path and begin with the next action. We then consider the path fully simplified.

Results

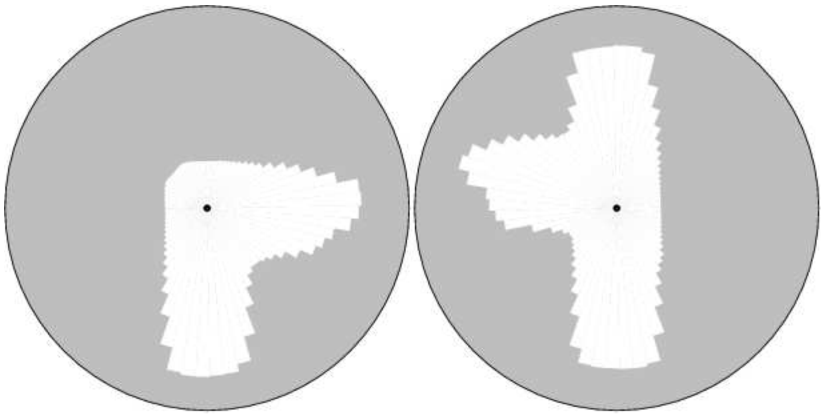
GNG states

The number of GNG states created was highly correlated with the error threshold, and the complexity of the maze (see Table 1). The larger maze was more complicated and had more different patterns of sensor readings than the J-maze, and more GNG states were created for the large maze. Also, when the error threshold was lowered many more GNG states were used.

As can be seen in Figure 3, as the error threshold increased the number of model states increased and states represented more specific sensor readings.

Maze	Error threshold	Number of GNG states
J	100	16
Large	20	43
Large	50	26

Table 1: The number of GNG states created for the two mazes with various error thresholds.


(a)

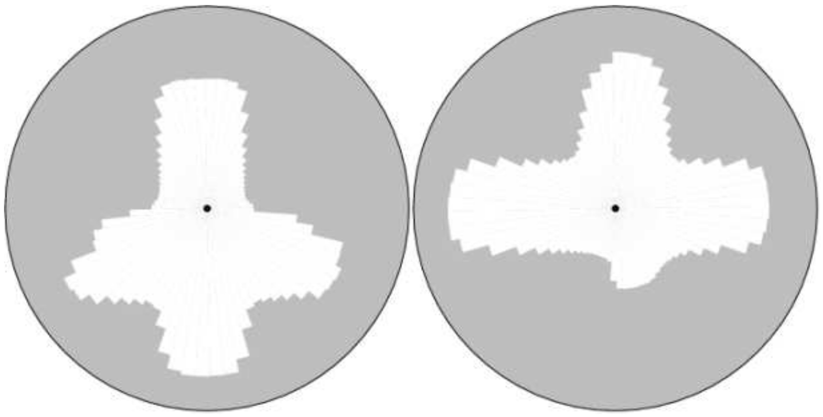

(b)

Figure 3: Examples GNG model states. **(a)** GNG states from the J-maze with an error threshold of 100. **(b)** GNG states from the large maze with an error threshold of 20.

Graph maze

The graphs produced as maps from phase II are all highly linear, with very little branching. Also, as can be seen in Figure 4, the robot would often take an action, then take the reverse action and end up in a different state. It is unclear, and our algorithm cannot distinguish, whether the two nodes created in these cases should actually be merged or whether they should remain separate, and our assumption that they should remain separate contributes greatly to the linearity of the map.

J-maze

The results of using a simple wall avoidance algorithm, a wall-following algorithm, and MSLST to solve J-maze are presented in Table 2. MSLST had the highest percentage of successful runs, and reached the exit within 200 time steps for all twenty-five of its runs.

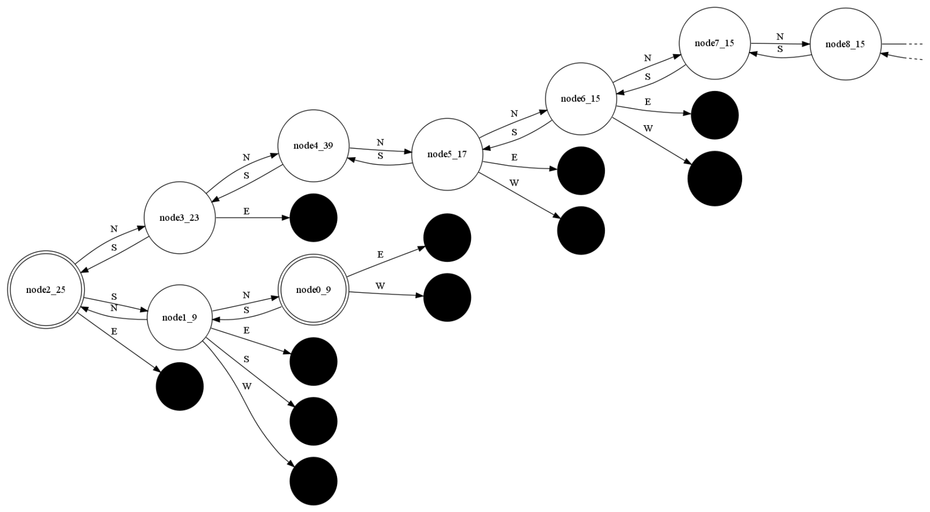


Figure 4: A subsection of the graph created in phase II for the large maze using GNG states made with an error threshold of 100. Destination nodes are doubly circled, and the filled in circles represent walls.

Wall-following had the next highest success rate and the avoidance algorithm had the lowest. Comparing the mean amount of time and actions it took the three algorithms to complete the maze reveals a similar trend: MSLST has the shortest mean runtime, Wall-follow the next shortest, and Avoid the longest. The distributions of the times taken to solve the maze for the three algorithms can be seen in Figure 5. The runtimes for all three algorithms show relatively high variance and are highly skewed for all but the Avoid algorithm.

Algorithm	**Percent success**	**Mean±std dev actions**	**Mean±std dev steps**
Avoid	68	73.16 ± 70.231	109.48 ± 78.521
Wall-follow	92	43.68 ± 26.24	91.76 ± 56.421
MSLST	100	17.08 ± 16.088	38.12 ± 34.003

Table 2: J-maze results. The percent of the 25 runs which were successful (found the exit within 200 time steps), mean number of high-level actions taken, and mean number of steps taken for the Avoid, Wall-follow, and MSLST algorithms applied to solving the J-maze.

Bonferroni-corrected Mann-Whitney U tests suggest that the amount of time in steps taken by MSLST is significantly different than the time taken by Wall-follow and Avoid ($p = 0.0018$ and $p = 0.0036$, respectively), and that Wall-follow and Avoid do not take significantly different amounts of time at the ninety-five percent confidence level.

Large maze

Results for the large maze show similar trends as for the J-maze (Figure 3 and Figure 6). In addition, of the two MSLST varaints tested, one using GNG states created with an error threshold of 20 and the other using GNG states created with an error threshold of 50, the latter was successful more often and took less long on average to solve the maze.

Using Bonferroni-corrected Mann-Whitney U tests again reveals that the two MSLST variants are not significantly different at the 0.05 level, nor are the results from Avoid and Wall-follow, but the MSLST20 and MSLST50 are both significantly different from Avoid and Wall-follow (MSLST20-Avoid: $p = 6.66 \times 10^{-5}$, MSLST20-Wall-follow: $p = 9.22 \times 10^{-4}$, MSLST50-Avoid: $p = 1.88 \times 10^{-5}$, and MSLST50: $p = 2.93 \times 10^{-4}$).

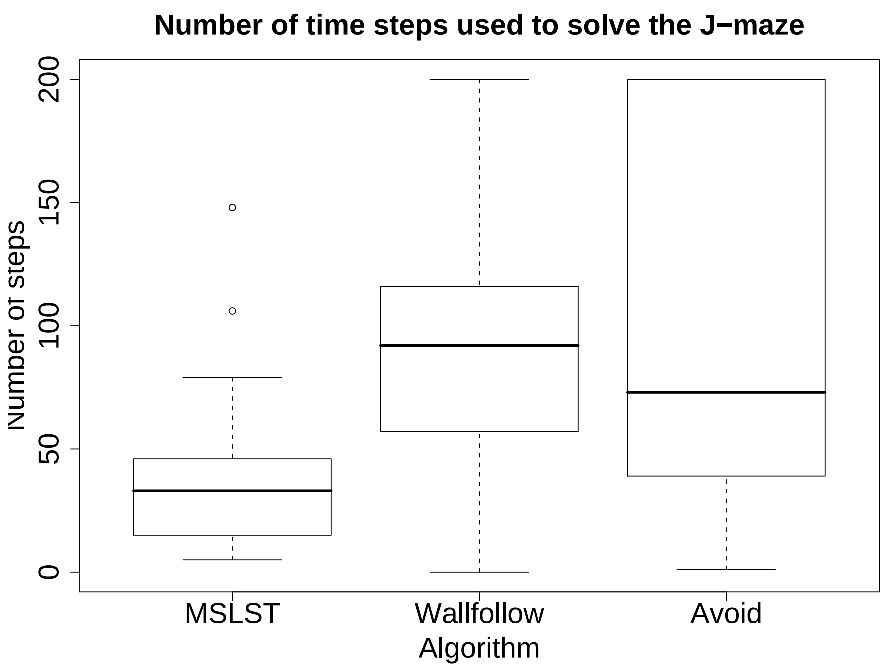


Figure 5: Boxplots for J-maze runs with different algorithms. The number of time steps taken to complete the maze. Note that after 200 time steps the run was halted and marked as a failure. Empty points indicate outliers.

Algorithm	**Percent success**	**Mean±std dev actions**	**Mean±std dev steps**
Avoid	40	247.32 ± 166.916	383.72 ± 173.098
Wall-follow	72	130.04 ± 83.414	278.52 ± 162.169
MSLST 20	92	69.72 ± 130.205	117.08 ± 130.452
MSLST 50	100	42.72 ± 41.733	103.2 ± 99.839

Table 3: Large maze results. Results of using the Avoid, Wall-follow, and MSLST algorithms to solve the larger, more complicated maze. MSLST was evaluated with graphs built using gng states with error thresholds of 20 and 50 (MSLST 20 and MSLST 50, respectively). The percent of the 25 runs which were successful (found the exit within 500 time steps), the mean number of high-level actions taken, and the mean number of steps taken are given.

Discussion

When compared with the baseline maze-solving strategies of wall-following and wall avoidance, MSLST performed significantly better on both the J maze and the large maze. This result was to be expected considering the vast difference in the complexity of the algorithms. While the MSLST algorithm built up two layers of perceptual and topological representation of the maze, the other two baseline algorithms were functionally oblivious to the task. The two baseline algorithms were only superior to MSLST in that they required no extra time to train. This difference is negligible compared to the discrepancy in performance on the task of maze navigation in cases where the maze needs to be navigated repeatedly in the performance of the robot's primary task. Furthermore, there was no significant difference in the performance of MSLST when error thresholds of 20 and 50 were used by the equilibrium GNG algorithm to quantize the input space. This is somewhat surprising given the difference in the number of GNG states between the two approaches. Because MSLST20 contained more GNG states, we expected its performance to be slightly worse than the performance of MSLST50. Without the hill-climbing algorithm used in[2][3], minor

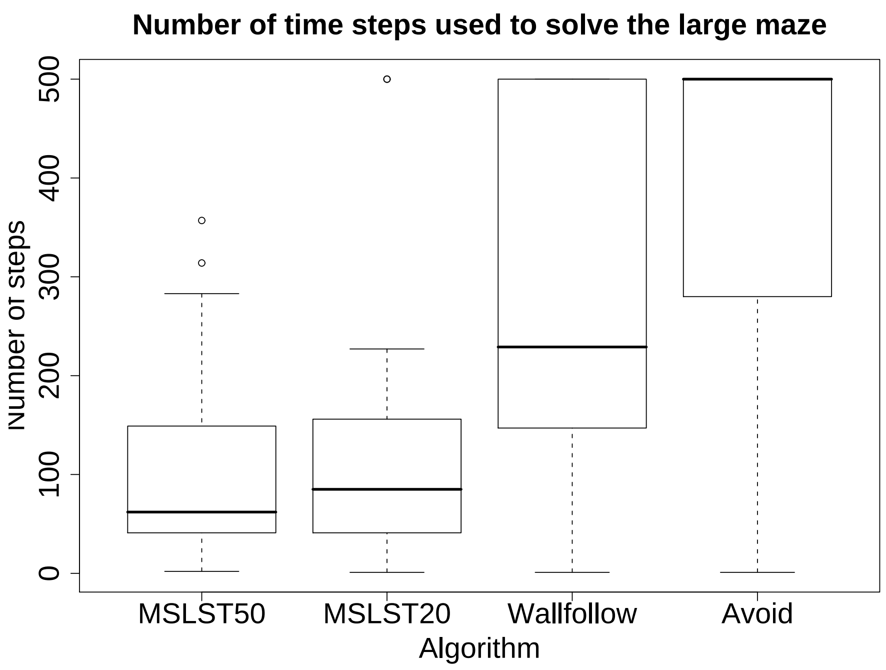


Figure 6: Boxplots for large maze runs with different algorithms. The number of time steps taken to complete the maze. Note that after 500 time steps the run was halted and marked as a failure. Empty points indicate outliers.

errors in the motor outputs should be expected to take the robot along a different physical route through the maze even when the high-level actions are unchanged. If the input space is divided into too many distinctive states by GNG, it is possible that while building the graph the same corridor could be traversed many times without creating identical, or perhaps even similar, topological paths. If this were to happen, the robot could stray from a topological route heading one physical direction to another topological route heading the opposite physical direction (for instance, going the other way around the loop). This could continue indefinitely, preventing the robot from reaching the destination. This process of bouncing between two possible routes to the destination was observed infrequently in the large maze, but each time the robot managed to break out of the loop. We suspect that the outliers in Figure 6 for both MSLST50 and MSLST20 were caused by this error.

Future work

Although MSLST has so far been fairly successful at solving mazes, there are many directions for possible improvement and generalization: several fairly minor tweaks could improve the effectiveness of the navigation algorithm, parameters in equilibrium GNG could be adjusted to find the best state set and learning times for phases I and II could be tested to see how much is necessary, performance could be improved dramatically by reducing the linearity of the graph through localization during the graph-building process, the algorithm could be generalized to work in open spaces as well as mazes with fixed-width corridors, and, of course, the system could be ported to real robots in addition to simulation.

Path-finding:
Along with a few bug-fixes, the biggest pending improvement is in how we use our graph to localize the robot. Currently, MSLST makes use only of a linear string of successful actions in matching the window of recent experiences to a section of the graph. Ideally, it could instead build up a new graph during navigation, using the same techniques used to build the graph during Phase II, and overlay it on the existing graph to find places that match.

This would help deal with the remaining state aliasing issues, in that the matching could be done with much higher certainty. It would also make it possible to specifically seek out paths that will distinguish between aliased states, where they exist.

Parameter Adjustment:
Our decisions about error thresholds for equilibrium GNG were somewhat arbitrary, since we lacked the time to compare many different values. Our results for the large maze showed that the error threshold does have a noticeable effect on the ability to navigate, so a comparison of several different error values in different mazes could be advantageous. Likewise, the number of steps spent gathering data to build GNG states and the length of the graph constructed to represent each maze were chosen largely based on how much time was available. In these cases, we expect that more learning time would probably provide only small improvements, but it is unclear from these experiments how much learning time is really necessary to achieve the results we show.

Graph-building:
The original concept behind MSLST was to build a non-linear graph that matched the geographical arrangement of the maze. If successful, this would result in a small and highly interconnected graph that would stop growing after the exploration had gone on long enough to discover all the locations of all the states within the maze. Instead, we found that the system generated very linear graphs, and so to encode all the relevant parts of the maze requires a very large graph which still may not include the most direct path from many locations to the destination; it also would not scale up effectively to very large mazes. The cause of the problem is that the graph does not recognize a location it already has a node for on subsequent visits unless the actions it takes returning to it are the exact reverse of those it took leaving; the solution is to use both metric and topological localization methods during the construction phase. The topological localization used during navigation can find places where two sections of the graph might overlap, but it cannot distinguish reliably between aliased regions; for instance, two long north-south hallways or two right turns could appear very similar for the length of the window. So to tell whether they are the same, it can also keep track of approximate position metrically by adding up its successful motor actions in each direction; if the actions since leaving the old node add up to approximately zero, and the states experienced in the surrounding area also match up well, then the two sections of the graph could be merged. The graph-building algorithm could then also seek out areas of the graph that are too linear or disconnected and return to them to fill them out.

Generalization:
The main reason this system is restricted to mazes with fixed-width corridors is that it allows the high-level actions to have more predictable results, since the layered corrections architecture can use the walls of the maze to adjust the robot's position and orientation more finely. Also, a maze provides more aliased states than do most environments, which makes for a better testing ground for MSLST. In principle, however, it could be extended to more variable environments. The sensors would need longer ranges to make navigation viable in large open spaces, since motor outputs have such a high level of noise. This would result in a much larger number of GNG states, and therefore a longer graph-building process and a more complex graph that is less effective for navigation. The linearity of hallways, corners, and intersections helps keep linear graphs from being disastrous, so extension to open environments would probably have to be combined with better localization during graph-building.

Real Robots:
The SICK laser range finder used in the simulation is expensive and requires a large robot to mount it on, so MSLST as written could probably only be tested with a robot like a Pioneer navigating large hallways. The other difficulty is the inaccuracy of electronic compasses, especially when placed in close proximity with robots. However, a modified version could be run without too much trouble on a Khepera with 360-degree sonars. The reduced sensory input would make it somewhat less effective and would require some rewriting of the layered corrections, but it would likely still be viable.

References

[1] B. Fritzke. A growing neural gas network learns topologies. *Advances in Neural Information Processing 7*, 1995.

[2] B. Kuipers J. Provost and R. Miikkulainen. Self-organizing distinctive-state abstraction for learning robot navigation. *Connection Science*, 18.2, 2006.

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

[4] T.M. Martinez and K.J. Schulten. A "neural-gas" network learns topologies. In O. Simula T. Kohonen, K. Makisara and J. Kangas, editors, *Artificial Neural Networks*, pages 397–402. Springer, 1991.

[5] M.J. Mataric. A distributed model for mobile robot environment-learning and navigation. Master's thesis, MIT, Cambridge, MA, January 1990.