

## PROJECT PHASE I IMPLEMENTATION: SINGLE ROBOT IN A KNOWN MAP

### Objective

To design and implement a robot which will traverse a maze and obtain as many prizes as possible, knowing the location of walls, prizes, food, and its own location in the maze.

### Introduction

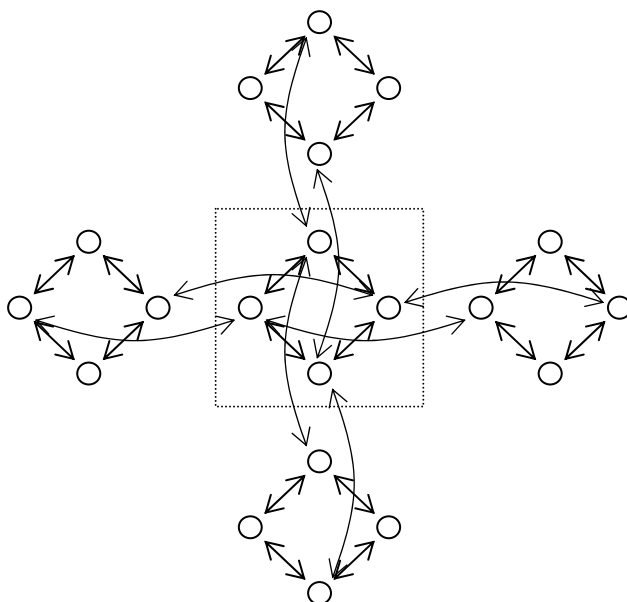
Thomas Coffee and I worked together on most of the project. His specialty is in conceiving different design concepts, and mine is in implementation. Therefore the bulk of this report will be covered from an implementation point of view. The controller code can be referenced from: [web.mit.edu/dongs/Public/16.413/control.scm](http://web.mit.edu/dongs/Public/16.413/control.scm).

### Robot Design

The robot will initially generate a catalogue of all the possible positions it can take on in the maze and take incremental steps towards various goals. At each step, the robot will use A\* search to determine the best path to the location of the nearest prize or food. Once it sets a path, it will try to move according to the path at each step until it feels a baddie, upon meeting which it will zap the baddie and continue with the path. When the robot reaches its goal state, or the end of its assigned path, it will grab whatever is in front, which will either be a prize or food, and plan for a new path toward the next nearest goal.

### Roadmap Setup

The state of the robot is described by its position in the maze in x and y coordinates and the direction that it is facing. A basic node is denoted by a list of these values: eg.  $(x, y, d)$ . Each node has at most three transitions because at any certain point, the robot can only either move forward, turn left, or turn right. Therefore, node  $(x, y, d)$  can transition to  $(x, y, d+1)$ ,  $(x, y, d-1)$ , or one case out of  $(x\pm 1, y, d)$  and  $(x, y\pm 1, d)$ , depending on the direction the robot is facing. Figure 1 shows an example of how the robot might transition to and from a square in the maze.



**Figure 1.** All possible state transitions for the four states in one maze square. The maze square is denoted in dotted line; circles represent states; arrows represent transitions.

The initial catalogue of possible states includes the four directional states at every square that is not a wall. For ease of access for the controller, each node that is added to a path is tagged with the transition command that links it to the previous node. Also for implementation purposes, each node is indexed by an enumeration so that it can be easily and uniquely identified. Thus a full node specification takes on the form of: (index, x, y, d, command).

### **Move Generator**

Each node starting from the robot position will be expanded into the (up to) three adjacent nodes, derived from moving forward, turning left, or turning right. Each of these expansions will be tested to see if it had been previously visited by comparing it to the items in a list of visited nodes. If it is a new node, it will be added to the path as well as to the list of visited nodes.

### **A\* Search**

The program uses A\* search to find the paths for the robot. It first generates the graph to the nearest prize by adding all the immediate transitions from each node, beginning with the node at which the rover sits, until the goal node is found. Each path is preceded with its A\* cost. The A\* cost is the sum of a node's current cost and its heuristic cost, all in terms of energy costs. Since every move and turn decreases the energy by 2, the node's current cost is just the number of steps it took to get to that point from the start, multiplied by 2.

The heuristic was first designed to be just the minimum manhattan distance (MD) to either a prize or food. It has more recently been changed to a calculated value based on the amount of energy left, such that at lower energies, the heuristic will favor food over prizes. Specifically, the heuristic looks like:

$$H = \frac{(MD\ to\ food\ or\ prize) \times (energy) \times (MD\ to\ food)}{(total\ energy\ in\ system) \times (maze\ length \times maze\ height)}$$

The denominator is a normalization factor, where the most amount of energy that the rover can obtain in the system is equal to  $10 \times$  number of food items, and maze-length  $\times$  maze-height is the largest possible value a manhattan distance can take on.

### **Priority Policy**

Of the list of possible paths, a priority queue is generated, placing the paths in ascending order with respect to their A\* costs so that the path with the smallest cost can be used.

A path is determined to be completed if the last node on the path corresponds to a prize or food. In the latest implementation, there is a special case for when the energy is too low. Generally speaking, when the energy falls below 50 times the heuristic cost of getting to food, then the goal node, or last node of a complete path, is necessarily a food node.

### **Dealing with Baddies**

In an earlier implementation, the robot feels for baddies at every move. If a baddie is felt, the robot will zap it. This simple method posed a few areas of improvement. First, since the robot feels for a baddie at every move, it uses 1 energy point for feeling at that move, in addition to the amount of energy it needed to make that move. In order to prevent the robot from feeling so much, a different method was implemented. In this implementation, the robot uses the look function after every turn to see if a baddie is in sight. If it doesn't see a baddie, it will not use the feel function at all. If it does see a baddie, it will then use the feel function at every step in that direction. This implementation is unique to my controller, as my partner's uses the feel function at every step. The performance results from the implementation with the look command are actually slightly less favorable than those from feeling at every step using the maze given to us, but I suspect this is due to the currently suboptimal amount of turning that the robot undergoes. Unfortunately, time constraints did not allow further investigation into the problem, although a better implementation can be expected for Phase II.

Another problem with the baseline strategy is that the robot is designed to always zap upon feeling a baddie, even if a path exists for the robot to go around the baddie. A better implementation would be to have the robot replan a path around the baddie and compare the increase in A\* cost from the original plan; if the increase in cost is less than it would take to zap the baddie, then take the replanned path; if not, zap the baddie. Though straightforward in concept, this method is hard to implement and has not been fully integrated into the program as yet, but can be added to Phase II.

### **Controller**

The controller first retrieves the current and previous commands from the A\* planner. If the previous command was for the robot to turn, then a look command will be used to search for baddies. Following that, a current command of robot-turn will be immediately executed, since it does not interfere with anything in the maze. Next, if the node was the last node of the path, then the robot has reached a goal state, so the only thing it needs to do is grab the item. If the current command is neither to turn nor to grab, then the robot will check to see if it previously saw a baddie; if it had, it will try to feel for baddies and zap them as necessary; if no baddie is detected, it will follow the next move command that the A\* planner has.



## PHASE II DESIGN CONCEPTS

### **Main Objective Changes**

The goal of Phase II is to design and implement a robot which will compete against an opponent for the most number of prizes in a maze whose environment is initially unknown to either robot.

### **Maneuverability Design Approach**

Since the robot starts off completely blind of its surroundings, it will need to dynamically create a map as it goes. The map will be generated relative to the coordinates of the initial rover position—which we will define to be (0, 0, 0), or in other words, at the origin, facing north.

If we make the assumption that mazes have generally similar features, we can create local Bayesian networks by determining a priori the probabilities of all the items (wall, prize, food, baddie, or space), and the probability of transitioning from one to another. For example, since walls are usually formed in large groups of blocks, we can infer that the probability of finding two wall blocks next to each other would be higher than, say, a block next to a baddie. The a priori information can be determined using the existing mazes (the .map files). After the robot senses new information, its local map will be expanded according to an updated Bayesian network.

The robot will conduct path planning using its local map in the same way as in Phase I. Now that the paths are uncertain, the robot may get to the end of a path and not actually find a goal, in which case it will simply replan from that position with the newly acquired sensing information. Another possibility is that the robot reaches a goal state before the end of the path, in which case it needs to accept that goal state and replan. If the robot meets a baddie, it will first try to replan and circumvent the baddie, but if no viable path is found, it will zap the baddie.

### **Adversarial Planning**

Before meeting the opponent, the robot has no idea where the opponent is, and cannot really do any productive adversarial planning. However, once the robot senses the opponent, it can make some guesses as to where the opponent will move by assuming that the opponent employs similar strategies. At that point, the robot should try to place itself in the way of its opponent's path, or put itself between the opponent and goal states such as prizes or food.

## **PROJECT PHASE II IMPLEMENTATION: TWO ROBOTS IN AN UNKNOWN MAP**

Shuonan Dong

### **Objective**

To design and implement a complete agent:

- Select and implement two or more decision making or estimation methods suitable for achieving the robot challenge.
- Develop suitable representations and problem encodings that will allow these methods to perform reactively.
- Design, integrate and demonstrate a closed-loop agent based on these methods and encodings.
- Document the agent design and architecture, highlighting design options considered, and unique innovations.
- Evaluate and report the agent's overall performance analytically and empirically.

Explore sophisticated versions of the basic algorithms taught in class:

- Use the web and library to evaluate and select suitable advanced decision making and estimation algorithms.
- Implement and demonstrate these algorithms within the agent.
- Provide a design rationale and tutorial explanation of the methods employed.

### **Introduction**

Thomas Coffee and I worked together on this project. After the baseline coding was complete, many efforts were spent on adjusting minor things in the code to bring about incremental improvements to the program performance. The controller code that I will be referring to can be referenced from:

[web.mit.edu/dongs/Public/16.413/control2fewsensorm.scn](http://web.mit.edu/dongs/Public/16.413/control2fewsensorm.scn).

### **Robot Design Overview**

Since in Phase II the maze structure is unknown, the robot will dynamically generate a probabilistic map of the surroundings that it becomes aware of. Whenever the robot performs sensing, the map will be updated automatically. The robot can take incremental steps towards various goals. At each step, it will use A\* search to determine the best path to the location of the nearest prize. These paths are generated from a Bayesian network of all of the nodes in the maze for which the robot is not certain is a wall. Once it sets a path, it will try to move according to the path until it reaches a prize, food, or baddie, whereupon it will immediately grab or zap, as appropriate. The planning process is repeated after every step in the controller.

## State Representation Architecture

### State Node Setup

The state of the robot is described by its position in the maze in x and y coordinates and the direction that it is facing. A basic node is denoted by a list of these values: eg. (x, y, d). Since in Phase II the nodes are not known a priori, we need to keep track of all the possible “things” that a particular xy-coordinate could take on. Possible “things” include wall, prize, food, baddie, space, or opponent. The node data structure maintains the probabilities of all of the possible things. Each node is given a unique index. The node structure looks like:

```
(mazeindex (x y) (p[wall] p[prize] p[food] p[baddie]
p[space] p[opponent]))
```

The notation `p[thing]` is the probability that the particular node with the indicated index and x, y positions might be that thing. For example, if the robot knows for certain that the 3<sup>rd</sup> node at (2, 1) is a wall, then the node would look like: (3 (2 1) (1 0 0 0 0 0)).

### Global Data Structures

Two global reference data structures are used. One called `enumnodes`, stores the state graph nodes in the form of (stateindex x-position y-position direction); this data structure is based off of a similar one implemented in Phase I. The other named `maze`, is the dynamic map that stores the nodes of the maze with its respective probabilities.

#### *State graph subspace*

The state graph potentially covers all possible position states that the robot can assume. Each square in the maze encompasses four possible positions for the robot (north, east, south, west).

#### *Dynamic map setup*

The maze map monitors the state of each square in the maze. The maze data structure is a list of the nodes described above in the section State Node Setup.

Initially, the only information that is known about the maze is an estimate of the overall probabilities of different items in the entire maze. These probability values are based on the maze given in `maze.map`, discounting the last two rows of solid wall. The initial probability values are shown in Table 1.

**Table1:** Initial probability values for every node in maze

Thing	Occurrences in maze.map (of 720 total)	Default initial probability
wall	294	0.408333
food	113	0.156944
baddie	39	0.0541667
space	142	0.197222

The initial probabilities for prizes are deterministic, or known to be 1 for all  $x, y$  positions found in `(robot-get-prizes)`. The probability of the opponent in any given square of the maze is initialized to zero in the current implementation. This value can be updated if the robot senses the existence of an opponent nearby.

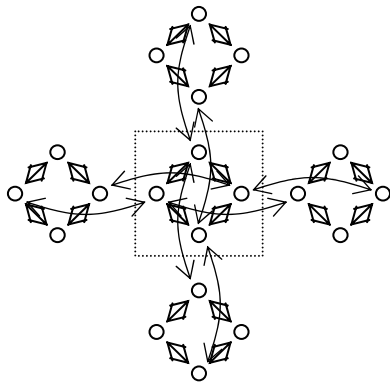
### State Transitions

#### *Precondition*

In order for a `robot-move` function to be successful, the node in front of the current robot position must not be a wall, or in other words, the probability that the front node of the current position is a wall must not be equal to 1.

#### *Transition space*

Each node has at least two transitions: turn left and turn right. It might also have the option of moving forward. The current implementation limits the forward movement to a distance of 1 per turn. Figure 1 shows an example of how the robot might transition to and from a square in the maze.



**Figure 1.** State transitions for the four states in one maze square. The maze square is denoted in dotted line; circles represent states; arrows represent transitions.

For ease of access for the controller, each node that is added to a path is tagged with the transition command that links it to the previous node. Thus a full state node specification assumes the form: `(index x y d (command))`.

*Path planning move generator*

Generally speaking, each node starting from the robot position will be expanded into the three adjacent nodes, derived from moving forward, turning left, or turning right. Each of these expansions will be tested to see if it had been previously visited by comparing it to the items in a list of visited nodes. If it is a new node, it will be added to the path as well as to the list of visited nodes. To prevent circular paths which do not reach a goal,—as this may occur since the goal is not definite, and the length of paths are limited—the back node to the current node (i.e. the state node with the same  $x, y$  coordinates but in the opposite direction) as well as the left, right, and back nodes of the previously visited node are all added to the visited list. This will ensure that a path will not cross over or repeat itself.

*Extension to multiple step moves*

The extension to multiple step moves is actually quite straightforward. When preparing the possible moves to add to a path, all nodes in front of the robot can be considered. The robot can have some number of forward movement transitions between 1 and the distance from the current robot position to the edge of the maze in the direction that the robot is facing. Therefore, node  $(x, y, d)$  can transition to  $(x, y, d+1)$ ,  $(x, y, d-1)$ , or any number of instances out of  $(x \pm n, y, d)$  or  $(x, y \pm n, d)$ , where  $1 \leq n \leq \text{max transitions to edge}$ , depending on the direction the robot is facing.

**Search Design****Fundamental A\* Search**

The program uses A\* search to find the paths for the robot. It first generates the graph to the nearest prize by adding all the immediate transitions from each node, beginning with the node at which the rover sits, until the goal node is found.

*A\* cost and path probability*

Each path is preceded with a pair consisting of its A\* cost and the probability of the existence of the path. The A\* cost is the sum of a node's current cost and its heuristic cost, all in terms of energy costs. Since every move and turn decreases the energy by 2, the node's current cost is just the number of steps it took to get to that point from the start, multiplied by 2. The path probability indicates how likely the path will exist, or in other words, how likely the path consists of nodes which are all spaces. The path probability is calculated as the product of all  $p[\text{space}]$  values for each node in the path.

*Heuristic*

The heuristic is a value that is reflective of the manhattan distance (MD) to the nearest goal of prize or food, the probability that the goal actually is a prize or food, and also the amount of energy that the robot has left. The desired heuristic should have a smaller

manhattan distance, a higher probability that the goal state is a prize or food, and it should value food more than prizes as the robot energy diminishes. Specifically, the heuristic looks like:

$$H = (1 - k p[\text{prize or food}]) \times \frac{(\text{MD to food or prize}) \times (\text{energy}) \times (\text{MD to food})}{(\text{total energy in system}) \times (\text{maze length} \times \text{maze height})}$$

The denominator is a normalization factor, where  $10 \times$  number of foods is the most amount of energy the rover can obtain from the food, and maze-length \* maze-height is the largest possible value a manhattan distance can assume. The probability value denoted by  $p[\text{prize or food}]$  is discounted by a factor  $k$  so that the degree of certainty in the path result can be adjusted in how much it will affect the heuristic.

### Priority Policy

Of the list of possible paths, a priority queue is generated, placing the paths in ascending order with respect to their A\* costs so that the path with the smallest cost can be used first. The priority queue data structure is implemented using a binary tree. A comparator is used to check that the left child node has a smaller value than the parent node and the right child node has a larger value.

### Complete Path Criteria

A path is determined to be complete if the last node on the path corresponds to a prize or food with a probability over some reward threshold, or if the length of the path reaches 10. There is a path length limit because the state space would grow exponentially until it becomes more than what the program can handle, and certainly the computation time would be driven over the limit of 30 seconds per move. Thus limiting the number of nodes in a path to 10 would bound the state space to less than  $3^{10}$ , or about 59000 nodes.

The danger to limiting the path to some value is that the robot may generate paths that do not lead to anywhere. Increasing the path limit will increase the chance that more paths lead to prizes or food, but there is an increasing chance of reaching memory limits. When a trial was run with path limit set at 20, the robot was able to travel much farther than it had previously, but it eventually reached a point of stack overflow. With a path limit of 20, the upper bound on the number of possible paths generated is  $3^{20}$ , or about 3.5 billion nodes.

## Sensor Design

### State maintenance versus observation history approaches

Sensors can potentially be implemented in several ways. One option is that sensor outputs directly update the probabilities in the relevant nodes of the system, but are then immediately discarded. Another option is to keep track of all past sensing observations and infer the states of the maze from reviewing some set of them. The second option reduces the amount of redundant sensing that the robot may perform. However, it is not as efficient to look up the maze states.

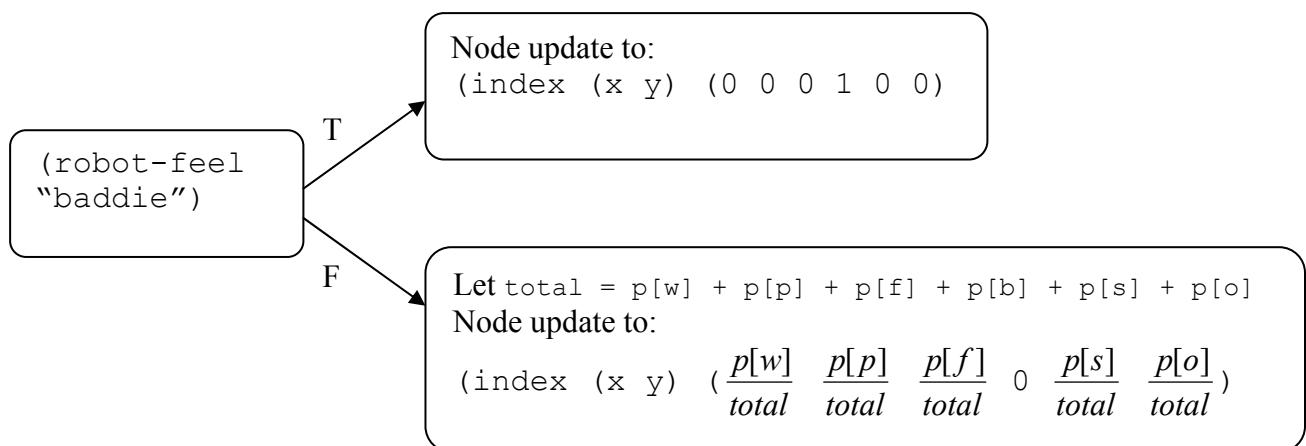
The current implementation uses the first option, so that as soon as the robot uses sensing, the nodes in the maze which are sensed immediately get updated with new probability values before the robot takes the next step.

### Implementation of sensing updates

Currently, the robot uses only the feel and smell sensing functions.

#### *Feel sensor*

Because the feel sensor only focuses on one node, it can exactly determine the probability (either 1 or 0) of that node. If the feel sensor detected `true` for some node being some item (wall, prize, food, baddie, or space), then the probability of that item in that node is updated to 1 and the probabilities of all other items for that node are 0. If the feel sensor detected `false` for some item, then the probability of that item is updated to 0, and the probabilities for all other items in that node are normalized. Figure 2 shows an example of the node updating process for a (`robot-feel "baddie"`) sensing function.



**Figure 2.** Example of node updating for feel sensing

*Smell sensor*

The robot smell function updates all 8 nodes surrounding the current node. Bayes Rule is employed to calculate the new node probabilities. If the robot smells for an item at some node and returns `true`, then the updated probability is the conditional probability of the item at that node given that the robot smelled the item from a neighboring node. This can be calculated using the Bayesian relation

$$P(T_i | S) = \frac{P(T_i)P(S|T_i)}{P(S)},$$

where  $S$  is the condition that `(robot-smell item)` returns `true`, and  $T_i$  is the condition of the item being in square  $i$ . In the above equation,  $P(T_i)$  is the prior probability of the item in the node, and  $P(S|T_i)$  is the probability that the robot would smell the item given that the item is in the neighboring node, which in this case is always equal to 1.  $P(S)$  is the probability that the robot would smell the item. This value can be calculated as follows:

$$P(S) = 1 - \prod_{i=1}^8 (1 - T_i)$$

So the conditional probabilities for each node  $i$  can be rewritten as:

$$P(T_i | S) = \frac{P(T_i)}{1 - \prod_{i=1}^8 (1 - T_i)}.$$

Then in node  $i$ , the probabilities of the other items need to be normalized to  $1 - P(T_i | S)$ .

Now, it is a much simpler story when `(robot-smell item)` returns `false`. If the robot does not smell an item, then it is certain that the item does not appear in any of the eight surrounding squares, in which case the probability of the item in each of those nodes should be 0, and the probabilities of other items in those nodes should be re-normalized.

**Extension to look sensor**

The look sensor will be useful for implementations allowing multiple-step moves. Look sensing will be able to update the probabilities of the nodes in front of the current node. Since the look function does not indicate how far away the object is, we need to use probabilistic inferences. An object is seen is less likely to be farther away because the farther away it is, the more likely that some other item would have existed between the

robot and the item. Therefore closer nodes have higher probabilities of containing the item than farther nodes. The conditional probability can be written as:

$$P(T_j | L) = \frac{P(T_j) \prod_{i=1}^{j-1} (1 - T_i)}{1 - \prod_{j=1}^{\infty} (1 - T_j)}$$

where  $L$  is the condition that the robot looks for and sees the item, and  $T_j$  is the condition of the item being in square  $j$ .

### **Controller Design**

The controller can be described by step-by-step replanning, greedy grabbing and zapping, and hypersensitive feeling of walls and baddies and smelling of food and spaces.

#### **Controller**

The controller begins each turn by a series of sensing functions. The sensing functions will update the maze map with more appropriate probabilities. Then it will conduct planning and retrieve a command from the A\* planner. Before executing this command, a series of checks are put in to determine if the node in front of the robot is a prize, food, or baddie, in which case it will grab or zap, as appropriate, and update the maze map probabilities again now that what had been a prize, food, or baddie is now a space. If the node in front of the robot is a space, then it will execute the command obtained from the planner.

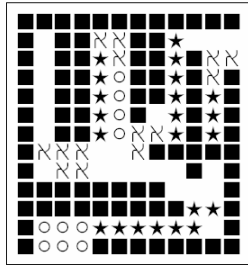
#### **Dealing with Baddies**

Currently, the robot feels for baddies at every move. The robot is designed to always zap upon feeling a baddie, even if a path exists for the robot to go around the baddie. A better implementation would be to have the robot replan a path around the baddie and compare the increase in A\* cost from the original plan; if the increase in cost is less than it would take to zap the baddie, then take the replanned path; if not, zap the baddie.

## **Performance**

### **Probabilistic map evolution**

The evolution of the robot's knowledge of the maze can be seen in graphics attached at the end of the report<sup>1</sup>. Figure 3 shows the top left section of the maze of interest to us (since the current implementation of the robot never traverses outside of this area). Squares represent walls, stars represent prizes, circles represent food, and x's represent baddies.



**Figure 3.** Top left section of the maze

Initially, the robot knows nothing of the maze except where the prizes are, and default probabilities. In the graphics that follow the report, the degree of grayness indicate how certain the robot is of the particular items. The graphics follow the progression of the robot as it traverses the maze. The triangle represents the robot, and the line represents its current path.

### **Time and space limitations**

The current implementation calls for planning at every step of the program. The planning function calls the A\* search algorithm, which takes a while to complete. The moves of the robot may take between 3 to 20 seconds, depending on the path lengths. One way to alleviate the long time requirement per turn is to reduce the amount of planning involved. When knowledge of the maze state is scarce, it is best to dynamically plan at every step in the program, but after the robot becomes more familiar with the environment, it may be reasonable for the robot to follow one path for several steps before replanning.

The Scheme implementation also has limitations on memory space usage during execution. Inefficient functions may lead to stack overflow issues. The current implementation avoids memory overflow by employing tail recursion on functions that are recursively called to large depths.

---

<sup>1</sup> Graphics generated by Thomas Coffee

Output

```
#####
#.##@.##.....#++.....#####@@@$$#
#.#.#.#.#@.#####...$$$$@.....#$$#
#.#.#.#.#@#.###.+$$$$#.#####@###...###
#.#.#.#.#.$#.....+$$$$#.....#+++++#+#
#.#.#.#.#.$#.....#####$#####.$$$$$$$$$#
#.#.#.....#$$#@@.....@@@@@$$@$$#$$$$#####
#.....@#####@@.....#####$#$$$$$#####
#>.....#.#++.#.....++#####+++++++#+
#####.#++.#.....+
#####$#$#+##+.$$$$$$$$$$$$$$$$$$$$.+#
++++$$$$$.#.#++..$$$$$$$$$$$$$$$$$$$$.+#
+++#####.##++.$.+++++++..$.+#
+++#####@.##++.$.+++++++..$.+#
#####@...#@..#++.$$$$$$$$$$$$$$$$$$$$.+#
+++@@@$.#@#$#####$$$$$$$$$$$$$$$$$.+#
+++.$$$#+#+$#$#++++++++#+
#####
#####
#####
```

-----STATISTICS-----

ROBOT ONE

Shields: 100

Energy: 0

Score: 11

\*\* Robot 1 ran out of energy.

## **Possible future improvements**

### **Prize traversal**

One way to ensure that all prizes get picked up is by using a prize traversal algorithm which will initially generate an overall minimal distance plan that traverses all the prizes in the maze. This implementation may lose some efficiency, but will guarantee that the robot will not be stuck in cyclical steps, which often occurs in the current implementation.

### **Food/prize balance given differing information**

The current algorithm generates paths which are prize-seeking and only picks up food when encountered. However, after many trial runs with slightly different implementations, it appears that the greatest limitation of the robot performance is that it runs out of energy faster than it can replenish.