A Search And Rescue Agent: CS 63 Final Report

Ryan Carlson, Andrew Stromme, Dougal Sutherland

Contents

1 Introduction

1.1 The Task

We created an agent for the disaster recovery simulation described in Eaton (2009). The goal of the task is to create a high-level AI for an autonomous search-and-rescue robot in a public building-based disaster situation, which can seek out victims in the environment and bring them safely to an exit.

The provided simulator is based on a two-dimensional grid world, with walls, various kinds of objects, and victims scatted around the environment. The victims have health data associated with them and die over time; while alive, they wander around the map. Several agents are placed in the environment competing to rescue the most victims, though in the spirit of saving lives, explicit adversarial reasoning against other agents is prohibited.

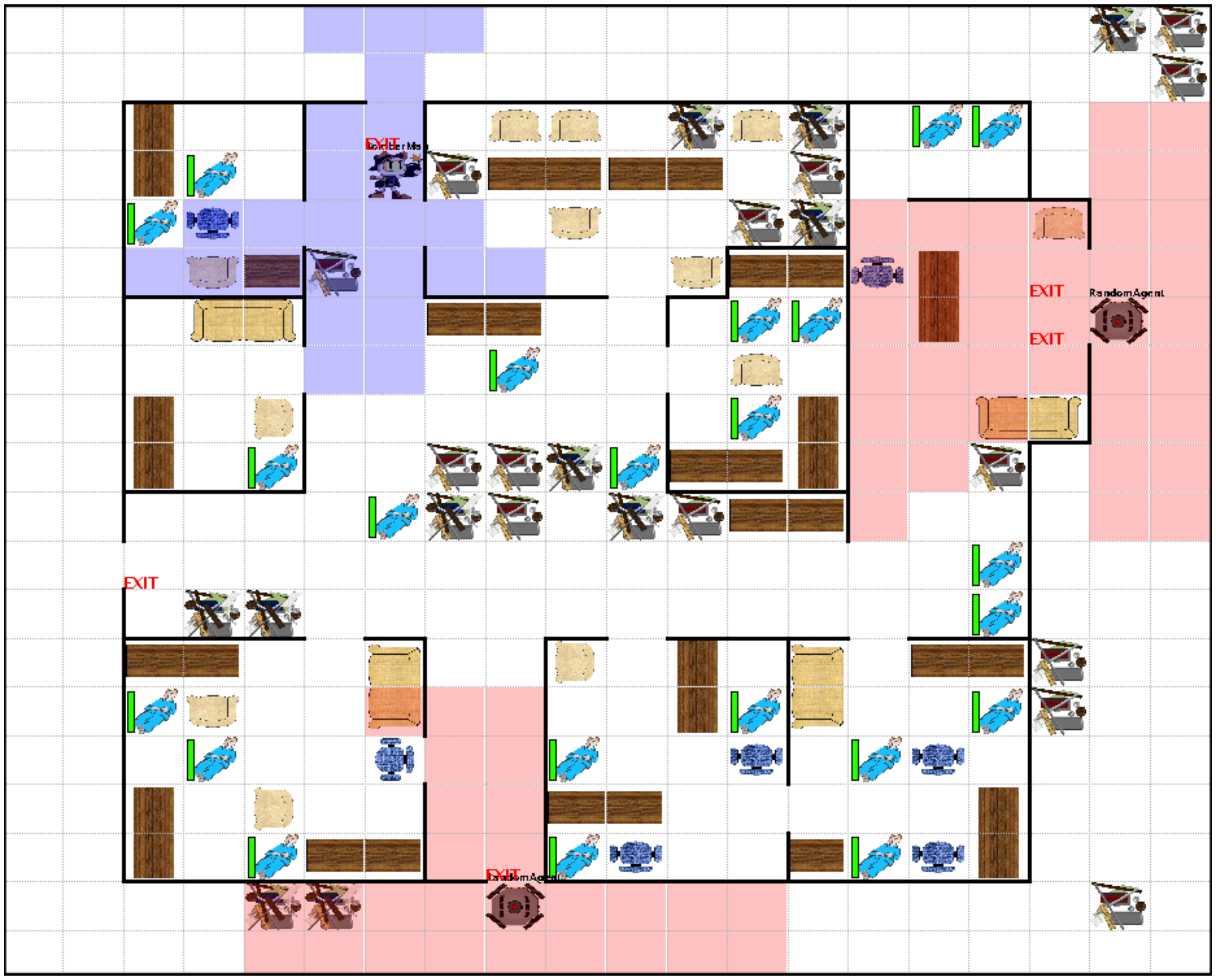


Figure 1: A screenshot of the simulator's display.

Agents can pick up and move victims and certain classes of objects (e.g. chairs), while others (e.g. sofas) are immovable. Victims are rescued by carrying them to an exit. Agents are given the layout of the map (that is, the walls) in advance, but not the locations of debris, the total number of victims, or any other such information. Each agent is equipped with a long-range visual perception device of some kind, which is always on and perceives the presence and type of objects, as well as short-range scanners which give health information about victims.

1.2 Our Agent

The overall framework for our agent's AI is a finite state machine (described in section 2). The state machine chooses goals based on a decomposition of the gridworld into "areas" (section 4) and a probabilistic victim memory (section 5), which associates victims with a probability distribution for their location and health data. If the agent knows about victims who it thinks could be healthy enough to save, it chooses one; otherwise, it "wanders" according to its prioritization of areas. The decision of whether to attempt saving a victim is made based on an ensemble of linear thresholds on the health data, derived by our implementation of AdaBoost (section 6). Pathfinding is done by A* search (section 3).

Complete code is given in an appendix, as well as provided with the `handin63` program by the user `dsuther1`.

The agent was implemented mostly in Common Lisp.[1] The AdaBoost implementation is in C++ and relies on the Boost program options library and `cmake`; its input is provided by running a Python script on the training data provided. The `AreaView` utility (which is essentially a debugging tool) is implemented in C++ using Qt libraries. The simulation server, from Eaton (2009), is implemented in Java; we provide convenience scripts for running both the server and our agent in `bash`. Most of the code has been tested only on Linux, though it should run on other operating systems, probably even Windows, provided all the dependencies are available.

2 Finite State Machine

2.1 Introduction

Our agent uses a finite state machine for its top-level reasoning about its actions. Each state has predefined transitions that lead the agent to other states; when the conditions for a transition from the currently active state hold, the transition will fire. This design allowed us to work on the individual AI implementations separately, managing their interactions through simple rules. It also provided an easily-understood representation for what happens next, given a current state and the conditions of the environment.

2.2 The States

The transitions between states are shown in Figure 2.

Wandering

In the `Wandering` state, the agent does not have a current victim to save; it is thus exploring the map looking for one. This state makes heavy use of the area representation of the map (see section 4). It prioritizes the areas based on an area sorting function (currently set to prefer the least visited area which is close to the agent) and chooses the 'best' area as its destination. From this point, each move is chosen by the result of an `a*-search` to a specific coordinate with the area.

The agent will transition to `Moving-Towards-Person` when it discovers a person who has not previously been marked as not savable.

Moving Towards Person

In the `Moving-Towards-Person` state, the agent uses `a*-search` to move towards the closest victim in its percepts. In the case where the victim is obscured from the agent's perceptions, the agent will attempt to

[1]Code specific to our agent is portable across implementations, but the provided base framework for agents uses `clisp` extensions.

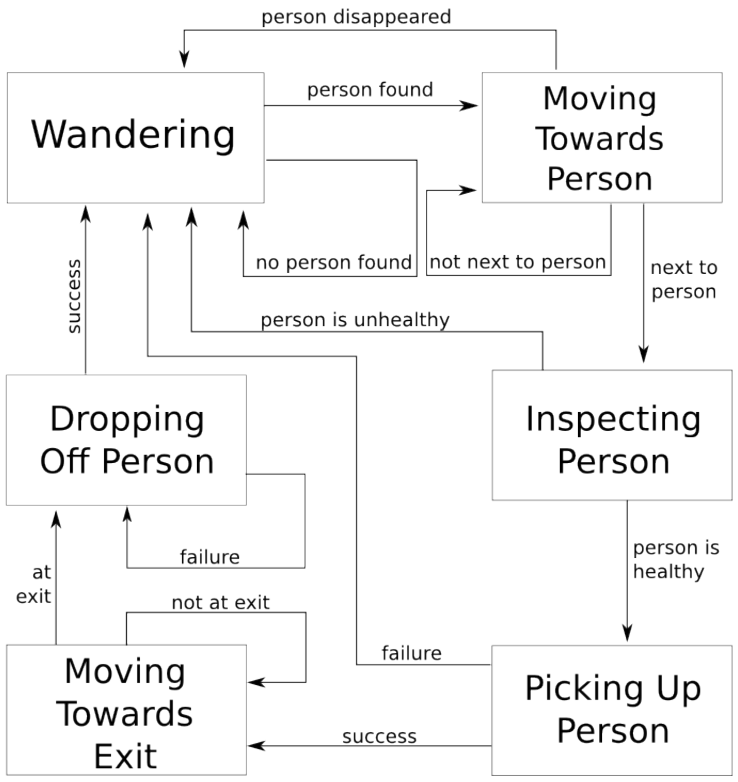


Figure 2: A visual representation of the finite state machine.

continue to the square where it last saw the victim. If it reaches this square without finding the victim again, it will revert back to the `Wandering` state.

When it reaches the victim it will transition to the `Picking-Up-Person` state if the person has already been scanned (according to the victim memory); otherwise the agent will enter the `Inspecting-Person` state.

Inspecting Person

In the `Inspecting-Person` state, the agent simply performs a sense action. The results of this sense action are associated with the relevant victim object and examined using the AdaBoost-derived health classifier. If the classifier determines that the victim is healthy enough to be saved (based on the `a*-search` distance to the nearest exit), it will transition to the `Picking-Up-Person` state. Otherwise, it will revert back to the `Wandering` state.

Picking Up Person

In the `Picking-Up-Person` state, the agent performs a pickup action. If this action fails then the agent reverts back to `Wandering`, otherwise it moves on to `Moving-Towards-Exit`.

Moving Towards Exit

This state is very much like the `Moving-Towards-Person` state, except that in this case it chooses the nearest exit based on `a*-search` pathlengths and moves towards it via the same `a*-search`. When the agent is adjacent to an exit, it transitions into `Dropping-Off-Person`.

Dropping Off Person

In this final state, the agent performs a dropoff action in the direction of the adjacent exit. If this fails (possibly due to another agent being on the same exit) the agent attempts to perform it again. If the dropoff has succeeded then the agent returns to the `Wandering` state.

2.3 Oh Shit Framework

During the dry run of the tournament we found that our agent was often getting stuck. This could happen for a number of reasons, but two in particular showed up repeatedly. The first case happened when our agent got stuck trying to take an action (e.g. move or dropoff) into a square that contained another agent. The second emerged when we added the assumption to A* that adjacent agents were immovable objects. Because of this extra reasoning, our agent could sometimes get stuck in a loop of choosing the same action as the other robot.

To deal with these problems, the "Oh Shit" Framework was introduced. It is implemented as a kind of state machine, where the two possible states are normal operation and "Oh Shit" operation. In the former, the previously described finite state machine determines the agent's actions. In the latter, the "Oh Shit" conditions determine the agent's actions. When the agent has finished with "Oh Shit" mode, it returns to its prior state in the normal state machine. This process is shown in Figure 3.

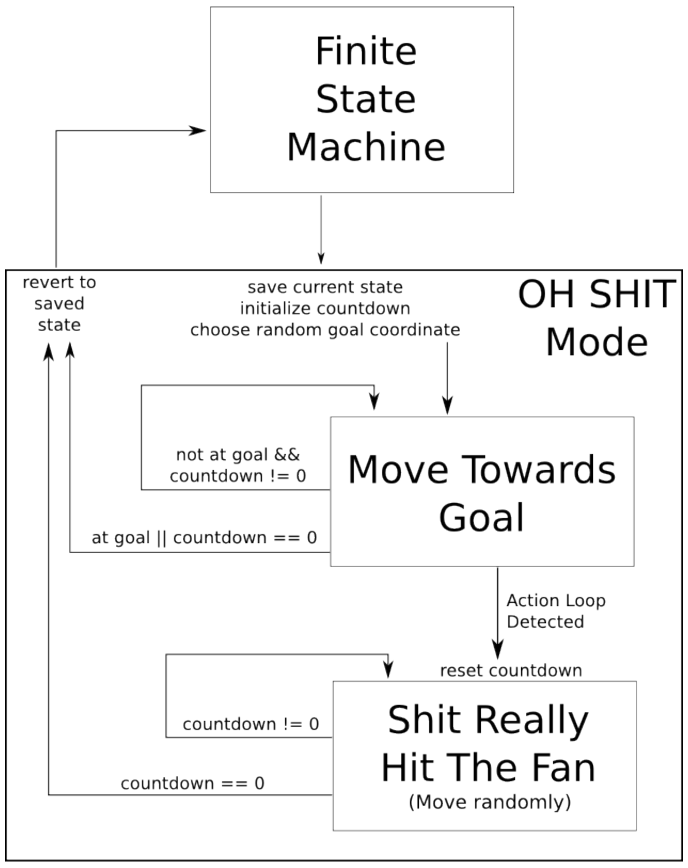


Figure 3: A visual representation of the "Oh Shit" framework.

Normal "Oh Shit" Mode

This mode is entered when a loop is detected or the state machine returned a `NOOP` action. It sets a countdown timer for the panic state and forces a new goal position, similar to wandering but based on random selection among those coordinates reachable via `a*-search`. The agent then goes directly to this location, returning to normal operation when it reaches the destination or the countdown expires, whichever is sooner.

Extended "Oh Shit" Mode

Sometimes the above panic operations aren't enough to successfully get the agent out of a trouble spot. In the case that we are already in "Oh Shit" mode and either the goal position is unreachable or the agent returns a `NOOP`, we enter the more extreme `Shit-Really-Hit-the-Fan` state. In this mode the agent returns a random movement action until the countdown has expired. After termination, the agent reverts to the `Wandering` state.

Trapped "Oh Shit" Mode

Even with these two modes there was still a chance for things to go very wrong. In the case where the agent was carrying a victim and had to move objects out of the way, there was no way to tell the agent not to pick up a second object — which, due to a bug in the simulator, was possible but not allowed in the rules. In this case the agent goes into a modified version of "Oh Shit" mode where it first puts down the object/person it is currently carrying and then proceeds to move to the nearest exit for 10 timesteps. When it reaches the exit or runs out of time it reverts to the `Wandering` state.

3 A* Search

3.1 Overview

Once we have decided on a destination, we need some way to get there. This service is provided by A* search. A* is a complete heuristic search algorithm that is optimal when given an admissible heuristic. It uses an evaluation function $f(n)$ given by

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the true distance from the start node to the current node n and $h(n)$ is the estimated distance from n to the goal node. As our heuristic we used the Manhattan distance from the current node to the goal node, modified to include an increased cost of moving objects (described in 'Implementation' below). Let $h^*(n)$ be the true distance from a node n to the goal node. We see that our heuristic is admissible since, for all n, clearly $h(n) \leq h^*(n)$; we will as such have an optimal search. Since our task is to rescue as many victims as possible in a given time frame, not wasting time in taking suboptimal routes is very important.

3.2 Implementation

Luckily, we had already implemented the search algorithm as part of the Romanian search assignment, based on the presentation in Russell and Norvig (2003). Most of the code (which is in *a-star.lisp*) stayed intact, though a new node structure and `expand-node` function was necessary.

We note that the agent almost never calls our `a*-search` function explicitly. Instead, the agent generally calls one of two functions:

- `get-next-move`: Given start and goal coordinates, returns the next action required to get to that goal
- `get-next-move-nearest-exit`: Given only starting coordinates, returns the next action required to get to the nearest exit

Each of these functions invokes `a*-search` to decide on a path, and then calls `path-to-action` which extracts the correct action to take. In the basic case, we want to `MOVE` to a square. But sometimes we need to `PICKUP` objects and, importantly, `DROPOFF` in a cell that will not hinder future movement. We have a number of checks that ensure the cell we drop an object is not in our current path and also will not be in the path from the current goal to the nearest exit (i.e. the return path). Additionally, it is worth noting that upon deciding to `PICKUP` an object we set a flag `pickup-object-attempt` that indicates what type of object we're attempting to interact with. When we get back the return status ("success" or "fail") of that action, we can decide whether that object type is movable or immovable.

We should also take a moment to consider the heuristic function, $h(n)$. Initially, of course, we don't know about any of the variables in the environment (debris, chairs, and the like), so our heuristic just assumes every path is unobstructed. As we explore and find these objects, we update our map to reflect our findings. For moveable objects, going "through" them requires picking them up and dropping them somewhere else, so we give moveable objects a path cost of 3.

3.3 Issues

We noticed that `a*-search` was taking a remarkably long time to return a path in certain situations. Specifically, when our agent tried to get to a location that was around a corner, the Manhattan-distance heuristic was a very poor guess. The search tried at first to make a bee-line to the goal, but with a wall in the way, we had to find some other way around. In Figure 4, we see that our agent must move from coordinate $(5, 1)$ west to $(0, 1)$ before it can move past the wall and continue towards its goal.

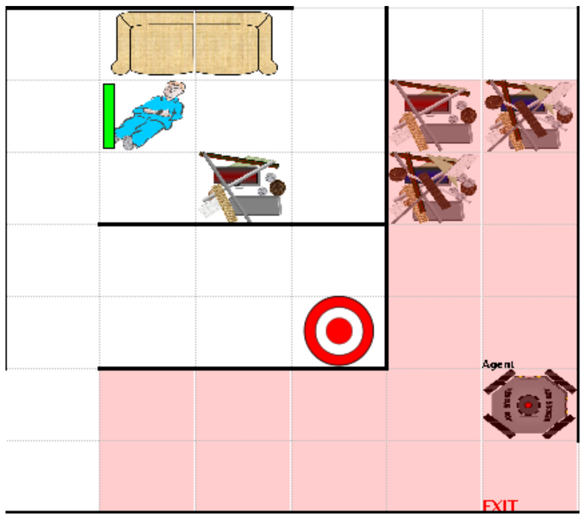


Figure 4: A situation in which the original A* implementation performed particularly poorly.

As `a*-search` cannot blindly check for repeated states while maintaining optimality, coordinates that were clustered around the corner point were being checked repeatedly, since they were being placed at the front of the queue due to their low heuristic value. To fix this, at the start of each search we create a hash table containing each coordinate paired with the length of the shortest path we have found to that node. When deciding whether we should expand a node, we check if we can reach the current coordinate in fewer moves, rejecting the expansion if a shorter path has already been found.

Since there is no situation in the simple pathfinding case in which we would want to expand a node with a longer path than one that has already been found, we have concluded that this does not change the optimality of our algorithm. Should `a*-search` be expanded to deal with moving objects in a more sophisticated manner, this assumption might need to be addressed by changing the hash to key not just on coordinate but also on the position of objects in the environment.

Another problem also came up after deciding that `a*-search` should be in charge of moving objects to clear a path: when trying to rescue a victim in a situation where there was no clear path to an exit, we ran into trouble. Since our agent cannot pick up multiple objects, we needed some emergency procedure. As a bandage we enter `get-me-out-of-this-trap` mode, in which we drop our victim and try to get to an exit. In general, this has the effect of clearing a path such that the next time we return we will have a clear path to an exit. However, as we discovered during the tournament, when we have to move more than one object out of the way to get to an exit, this mode fails. This problem may have been better served by methods outside of just `a*-search`. Since the algorithm has a very narrow view of the problem at hand (i.e. start and goal coordinates), it is difficult to describe more complex problems without having a higher level element to help direct the flow.

4 Areas

4.1 Overview

Areas are meant to provide the agent with high level information about the map it is operating in. Some limited reasoning can be performed with basic coordinates and adjacency lists, but it is far more powerful to have a more natural representation of the environment. In the task at hand, given an office building or another similar man-made space, distinctions based on rooms are an easy way to construct this higher level structure. Buildings are partitioned into more-or-less rectangular rooms, and these can give a more accurate guess as to where the victims in a search and rescue scenario might be.

4.2 The Area

An area is defined as a semi-enclosed rectangular region. Similar to coordinates, adjacencies between areas are stored and accessible via utility functions which allow for traversal of the areas as an undirected graph. This graph of areas is created in the map-to-graph component of our agent (*map-to-graph.lisp*).

4.3 Implementation

4.3.1 Growing an Area

The area detection algorithm starts by choosing a random coordinate on the map. It then grows the cell for that coordinate outwards, starting with one set of directions, for example East/West. On each side this growth stops when it would add cells that would not be adjacent to the cells already present in the line. The result of this initial growth is a line of cells. Because non-adjacencies are really just walls, this line of cells has walls on either side.

The second step of the process is a growth in the other set of directions (in this example North/South) until the stopping condition is reached. Because there is an entire line of cells the stopping condition is more complex, but is similar to the simple case. The condition used was non-adjacency in any of the created block's cells. This means that for each line/row added to the block the algorithm checked to make sure that there were internal adjacencies from each cell to each possibly adjacent cell (i.e. each cell North/South/East/West of the given cell).

The result is a group of coordinates that are all internally adjacent. Because there are many cases where starting with the other set of directions first would have created a better area, the algorithm is run again, swapping East/West first for North/South first. The two resulting areas are compared in their length-to-width ratio and the more square of the two is selected. The idea behind this more-square-is-better selection is that the agent's perceptions are square in nature, and thus the agent will have a better chance of perceiving the entire area at once if it is square as well. This process is illustrated in Table 1.

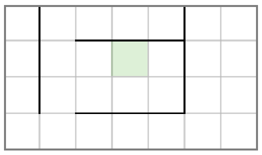
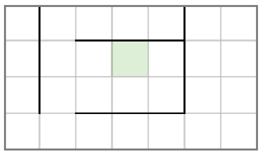
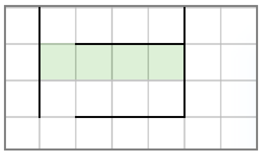
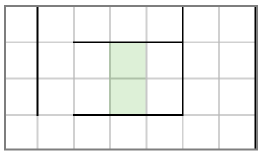
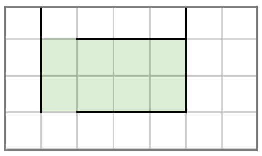
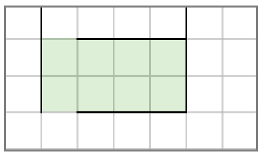
Operation	East/West First	North/South First
Expand Cell		
Expand Block		
Final Area		

Table 1: The area generation process.

4.3.2 Creating Additional Areas

This process repeats itself, starting with a new random coordinate from the pool of coordinates that are not already contained in an area, until all coordinates have been included in an area.

4.3.3 Adjacent Areas

Two areas are adjacent if any of their coordinates are adjacent. These adjacencies are reflected in a hash map of adjacent areas, just like the hash map for adjacent coordinates. During the growth of areas described above, we keep track of the adjacent areas. Once the algorithm considers which of the two possible areas is better (East/West vs North/South), it updates the global hash map.

4.3.4 Walls and Immovable Objects

Initially immovable objects were represented by removing the adjacencies on the map surrounding their squares. While this provided an easy way to inform the area generator of inaccessible coordinates, it was found to be undesirable when there were many objects in a room because of the fundamental differences between objects and walls. Most importantly, walls block the agent's perceptions while objects do not. Because of this, the agent performed better when it populated its areas ignoring any objects in those areas. This resulted in larger and more complete areas that matched up better with the agent's perceptions and the locations of victims in the simulation.

A second pass was done after area population that removed unreachable coordinates from their parent areas. This ensured that the agent would be less likely to choose as its wandering destination a coordinate that was unreachable.

4.4 The Room Issue

During our project design, a complication was raised in the definition of a room and hence the stopping cases for rooms in the algorithm for area (then called room) detection. This issue was solved by modifying the definition of an area from an enclosed space with one or more doors to the more flexible interpretation of a semi-enclosed rectangular region. This means that the algorithm has to consider neither bends in the walls nor the definition of a door.

There are still cases in which physical rooms are split up into smaller areas that are undesirable, but that is covered in a later section. For more information see 'Merging Areas.'

4.5 The AreaViewer

After the initial algorithm was written, we felt that there was no easy way to visualize the results of the area population. It is not easy to conceptualize a list of coordinates into something that can be deemed as correct or incorrect classification of an area. Therefore we felt that there was a need for a small helper application that let us view the areas the algorithm had created.

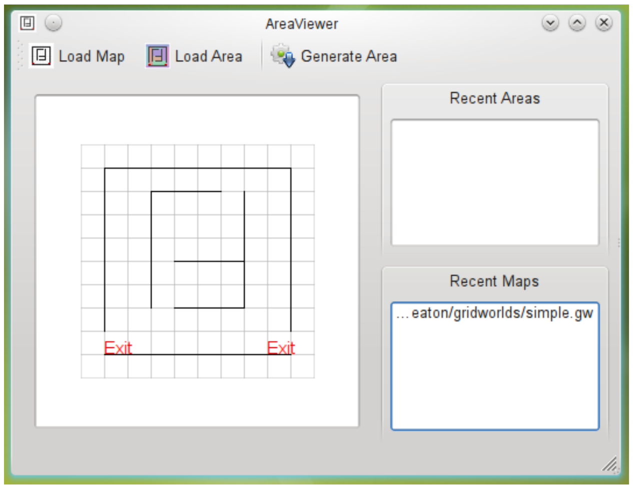


Figure 5: A recent build of the `AreaViewer` application.

The result of this need is the `AreaViewer` application, written in C++ and using the Qt software libraries for GUI painting. `AreaViewer` loads up a map file and a dump of the generated areas and overlays the areas on the map. For example, Figure 6 shows a first iteration of the program and the area generation code for the *simple.gw* map.

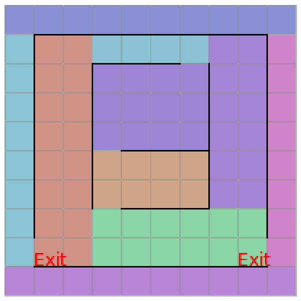
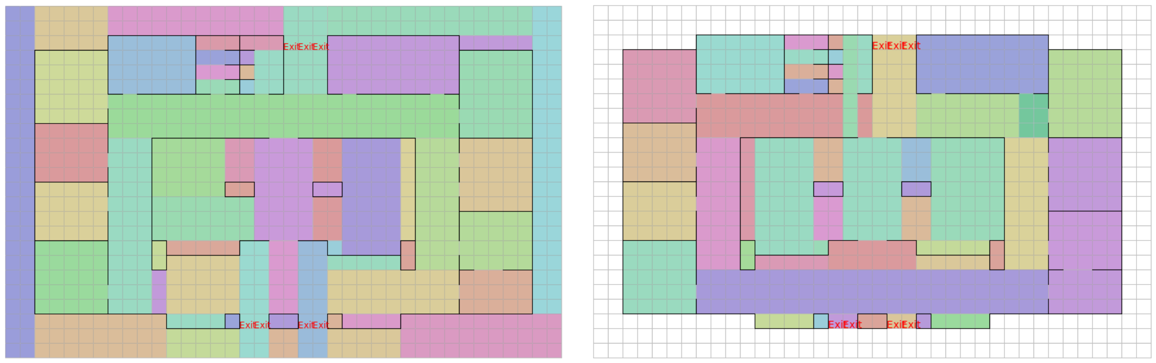


Figure 6: `AreaViewer` showing the areas generated for the simple gridworld under the initial algorithm.

4.5.1 The Outside Area

One issue that was immediately apparent was the problem of outside areas. There is no reason to go explore outside; passing through the outside of a building is only useful when it is necessary to get from one point to another (in the case that there are immovable objects or walls blocking the internal path). As such, areas outside of the main map, as shown in Figure 7(a), are useless.

The second iteration of the area generator first computes the outside area by growing the coordinates (0, 0) as far as possible. The outside area detection uses the exits as the borders of the building and does not allow this growth to continue inside of the building. Applying this process to the *hard-largescale1.gw* map resulted in the areas shown in Figure 7(b).



(a) Before outside area detection. (b) With outside area detection.

Figure 7: The effect of outside area removal on the largescale gridworld.

4.5.2 Unreachable Areas

Another problem observed with the AreaViewer was that of unreachable coordinates/areas. These are undesirable because they interfere with pathfinding, and are useless to our agent because it can't ever do anything with that coordinate. Initially the pruning of inaccessible areas was done by area adjacencies, but this has problems in the case where there are large amounts of inaccessible squares that might get populated into multiple areas. Therefore `a*-search` (described in section 3) was used to quickly identify areas that the agent could not reach. These areas were discarded during the area population at the start of the simulation, as shown in Figure 8.

4.5.3 Merging Areas

A final problem that was observed with the AreaViewer has to do with the small areas detected when there are angles or walls within a room. This produces undesirable sub-partitioning of rooms into too many areas. This is a problem because the agent must go visit each of these areas in turn, and thus these extra areas slow down the agent's exploration of new parts of the map.

We tried several different methods for merging these areas together, but none produced good and consistent results and so in the tournament the code was disabled. Upon later review of the code, the area merging algorithm was improved to a point where it is usable by default. It works by sorting the areas so that the areas of either width or height one are put first. It then iterates through these areas, merging all single width/height areas with their smallest adjacent area. It updates area adjacencies and the global areas list while it works, so that the other users of this list need not worry about whether areas have been merged or not.

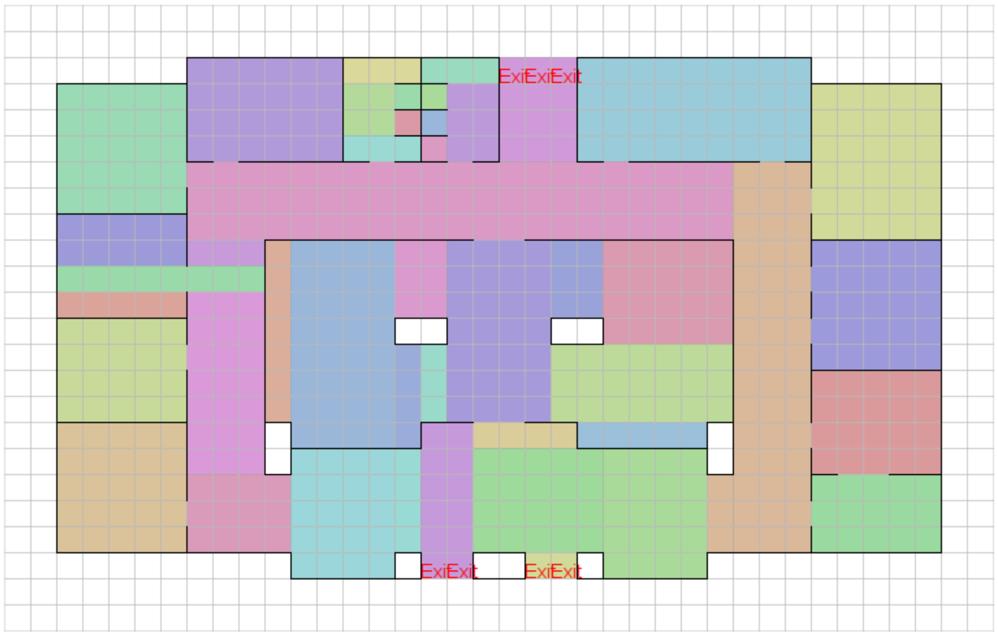
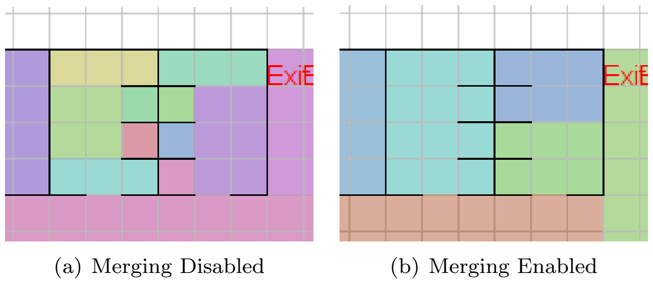


Figure 8: Areas generated for the largescale gridworld using inaccessible areas detection.

.



(a) Merging Disabled (b) Merging Enabled

Figure 9: The effect of area merging on a portion of the largescale gridworld.

4.5.4 Visualizing Adjacent Areas

When working with the areas in the `AreaViewer`, it became apparent that just viewing the areas wasn't sufficient to tell if all of the area generation code was working correctly. The `AreaViewer` was improved to show the adjacent areas for a given area when that area was hovered over with the mouse. In the following screen capture the area in the middle has been selected by hovering over it. The areas with the circles in them are adjacent to this selected area.

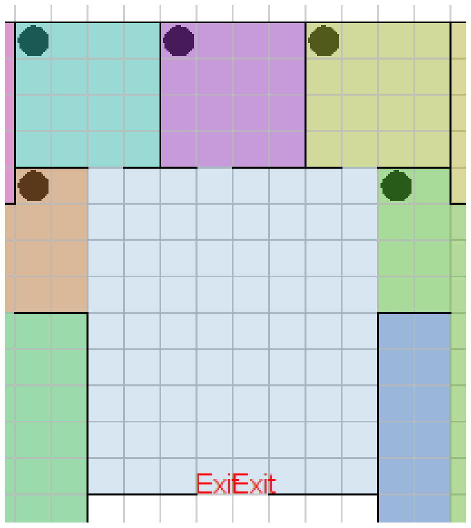


Figure 10: Visualization of adjacent areas in the `AreaViewer`.

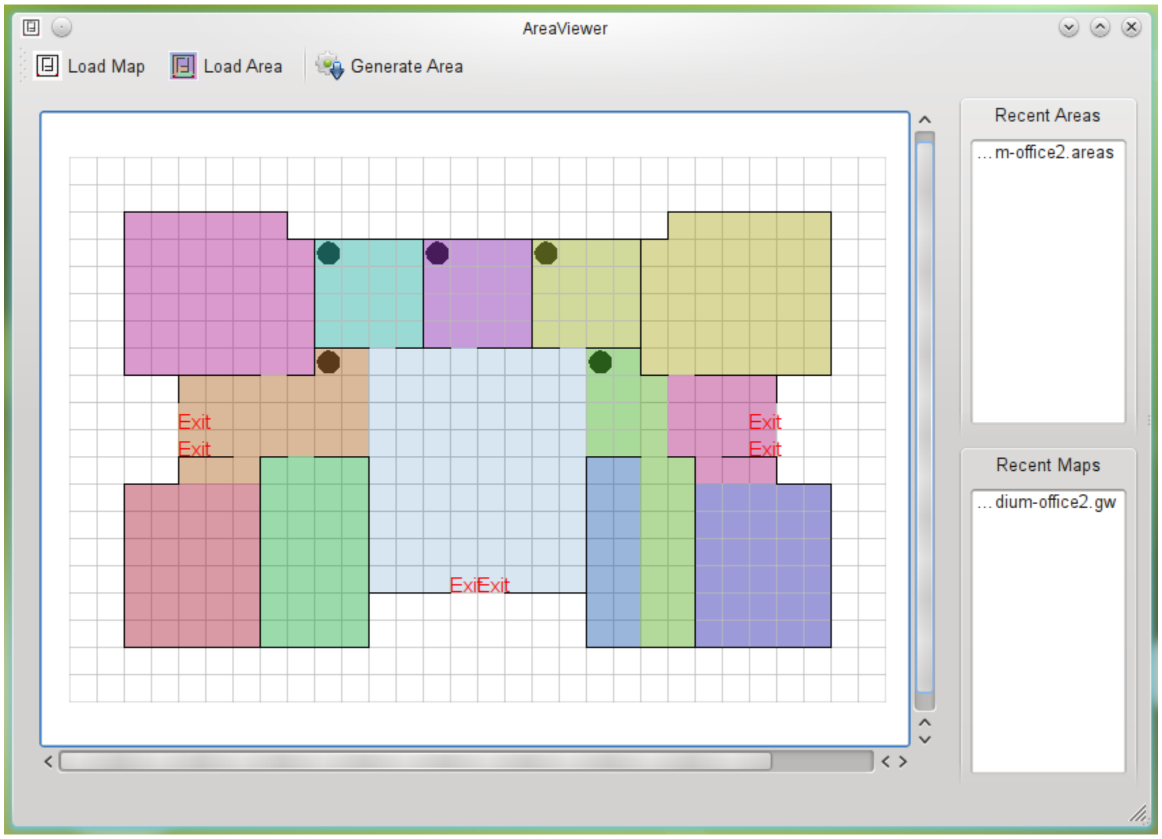


Figure 11: Areas detected (after outside-removal, inaccessible area culling, and merging) in *medium-office2.gw*. Note that the two green areas on the right side are distinct, though their shading is similar.

5 Probabalistic Victim Memory

5.1 Overview

A key part of our agent is the ability to remember where it has seen victims, and how healthy they were when we last checked. Without this, we would be able to do little more than wander randomly through the environment and hope to run into new victims; we would also be unable to distinguish between those victims we have already classified as unsavable and others whom we already know to be healthy.

Because victims do move around on their own, however, we need to be able to reason about where they may have gone in the meantime. This is accomplished by calculating a probability distribution for each victim's location and reconciling it with our perceptions at each timestep.

5.2 Implementation

Most of the code relating to the memory is in *memory.lisp*.

The core of the victim memory is the `*victims*` list, which contains a `victim` struct for each victim of which we are aware. The `victim` struct has associated with it a probability distribution (in the form of a hash table mapping coordinates to the numeric probability that the victim is in that square), as well as optional health information and a counter for how long it has been since the health information was obtained.

At each timestep, our agent considers its perceptions and updates its victim memory via the following algorithm:

1. Diffuse the probability mass for each victim from the previous turn, according to the rule

$$P(v,x,\ t+1) = (1-\alpha)\,P(v,x,t) + \sum_{q\,\in\,adj(x)} \alpha\,\frac{P(v,y,t)}{|adj(y)|}$$

   where $P(v,x,t)$ represents the probability that victim v is in coordinates x at time t, α is the probability that a victim will wander, and $adj(x)$ yields the list of coordinates adjacent to x.

2. Remove each square known to be empty from the probability distributions of all victims, then renormalize those distributions.[2]

3. For each person we see, attempt to resolve it with a victim in memory: if a person is in square s and $P(v,s,t) \geq$ `*victim-resolution-threshold*`, we say that person is victim v by setting $P(v,s,t) \leftarrow 1$, $P(v,q,t) \leftarrow 0$ for all other coordinates q. If there is no v in `*victims*` for which this is true, we create a new victim v' with a probability distribution that reflects its known location. In either case, we set $P(x,s,t) \leftarrow 0$ for all other victims x and renormalize x's probability distribution.

4. If we have any health data in our perceptions (which would occur when our last action was to scan a victim), we associate it with the given victim.

This algorithm is far from perfect. In step 1, we assume that victims have a set probability of moving (which does not vary according to their health data), and that if they choose to move they have an equal probability of moving to any adjacent square. This is not quite what appears to occur in the simulator, where victims have a certain probability to move in each direction, and if they choose to move in an impossible direction they simply stay in place. Additionally, it ignores the effects of the presence of debris, other victims, and particularly other agents within the environment.

Our value for α (`*victim-wander-prob*` in the code) was also imperfect, in that we merely guessed it was around 0.1. We planned to write a script to wrap around the controller for the simulation and learn at least an overall value for how often victims moved, ignoring any health effects that might be present, but the potential benefit for the small increase in accuracy seemed not worth our limited time.

[2]This is a valid operation because we are doing no "backwards" reasoning about victim locations, and each of the probability cells are completely independent at any given timestep.

We also maintain the total probability of a victim being anywhere in the environment as 1 (plus or minus floating-point errors) until we determine that it is impossible for them to be on the map at all, in which case we remove them from our memory. This ignores the actions of other agents, who might save victims, as well as the (admittedly unlikely) possibility that agents save themselves by wandering onto an exit. We had intended to reason about this, but our intuitions about other agents' actions proved difficult to formalize enough to code, particularly given our agent's limited perspective on the world. Additionally, as shown in the tournament, other agents do not always perform in a predictable manner.

Another interesting issue is that when we know a victim has died, we still shift their probability distributions around the map as if they could wander. This could be solved within our framework relatively straightforwardly — either by establishing that victims whose AdaBoost evaluation is sufficiently low are dead, or by implementing some simple decision tree on the health data.[3] When we attempted to do so, however, we introduced a bug in which the total probabilities of victims who have been considered dead skyrocketed above 10^{100} within two or three turns of their being scanned as dead, and so our system currently allows dead victims to wander. In practice, this is not a big issue, as the victim will almost always remain the most likely victim to be in that square.

The resolution phase (step 3 above) could also be improved. Consider the case when a given person is about equally likely to be one of two victims; we simply choose the first one in the `*victims*` list. It might be advantageous to introduce another layer of probabilistic reasoning to the memory component, reasoning based on the chances of a person we see being a given victim. This significantly complicates the task, however.

One potentially intriguing alternative approach, especially had we chosen to deal with resolution probabilistically, would be to initialize the victim memory with a number of victims equal to the number of squares in the environment, each having a probability distribution beginning at $P(v, x, t) = \frac{\text{expected number of victims}}{\text{number of squares in map}}$ for one x and $P(v, y, t) = 0$ for all other coordinates y. As we moved around, we would update victims according to the algorithm above, though we would never need to create new victim objects, rather associating people probabilistically with each of the victims. This would introduce some practical problems in victim resolution, but would have the advantage of being able to reason about where victims could possibly be on the map. Getting an accurate estimate of the number of victims on the map would not even be all that important, as it would merely scale the probability of each victim in proportion with one another. (The main issue would be with predefined thresholds on the probability; our overall reasoning would have to be much more complex.)

5.3 Usage and Results

We use the health data associated with victims to make decisions about who can and cannot be saved. We remember the number of turns since the health data has been obtained and add that to the number of turns required to save the victim to determine whether we can or cannot save him. The most important aspect of that is for dead victims, who we remember as being dead and so waste no time in attempting to save.

We also use the memory in determining our next goal position within the finite state machine. When we have no other goal set and we know of any victims who we think are savable, we move to them.

Despite its simplicity, our victim memory seems to have performed quite well with our agent. A formal evaluation of its accuracy would require significant coding effort, but in examining it, it seems to almost never "forget" victims, especially dead ones. The main area in which it could use improvement is an improved ability to guess where other victims might be, one approach towards which was mentioned above.

Another related task which we did not take on would be remembering the locations of objects, both movable and immovable. This could gain us significant pathfinding advantages in certain situations, though dealing with it properly would require a more sophisticated probabilistic search framework.

[3] Just deciding based on a pulse-rate of 0 is almost completely accurate; in the training data, all dead victims had a pulse of 0, and the only living victims with that property were unconscious with an extremely low time-to-live.

6 AdaBoost

6.1 Overview

Once we have located a victim in distress, we need to know how long they have to live and whether we can save them. For this task we use AdaBoost, a machine learning algorithm developed by Freund and Schapire (1997), using single-dimensional thresholds as our sub-classifier. We start off with classified training data – the more the better. We must then classify positive and negative training examples. Each of these training examples is represented as an n-dimensional feature vector. Initially, each is assigned an equal, normalized weight. We then feed these examples into the algorithm, which sorts the feature vector by each dimension and finds the best dimension on which to threshold. After each iteration, each feature vector's weight is updated. If the weak classifier misclassified the vector, the vector gets a higher weight and is prioritized for the next round. This process gives us a set of weak classifiers, which are then combined into an ensemble strong classifier that can accurately identify victims we can help and those we cannot.

Now that we have covered the general algorithm, we can consider AdaBoost in the context of our search and rescue mission. All the training data we need is contained in the *victimdata* folder. The training data consists of all the statistics we will have when sensing the characteristics of a victim (call this s), and additionally we will have their time-to-live (TTL). To create our positive and negative training examples, we note that if the victim is an A* distance k from the exit, with $k < TTL$, we can save him. Likewise, if $k > TTL$, the victim will probably die if we try to save him. So, for all $k < TTL$, we will create a positive training example of the form $(s \,.\, k)$, with $s \in \mathbb{R}^n$ a feature vector characterizing the victim. For all $TTL < k < MAX$, where MAX is some reasonable maximum distance that a victim could realistically be from an exit, we will create a negative training example. For example, suppose $TTL = 5$. We might have positive and negative training data as follows:

$$\text{POSITIVE: } \{(s \,.\, 1),\ (s \,.\, 2),\ (s \,.\, 3),\ (s \,.\, 4)\}$$
$$\text{NEGATIVE: } \{(s \,.\, 6),\ (s \,.\, 7),\ (s \,.\, 8), \cdots,\ (s \,.\, 15)\}$$

We then feed these data into the algorithm to get out a set of weak classifiers: thresholds on one dimension of the feature vector s from above. For example, we might consider pulse-rate, such that any pulse-rate value below some rate r will be considered negative, and any rate above r is positive. These weak classifiers are grouped together into a strong classifier that enables us to identify victims that we can help. Now suppose we have scanned a victims' health data. We apply the strong classifier (which has already been generated, and will remain static while search and rescue is underway) to the victim, getting a single return value. If this value is above some threshold (i.e. zero or one) we can be confident that the victim can be rescued, so we should go ahead and rescue him.

6.2 Implementation

The AdaBoost implementation is broken into three main parts, discussed in turn below:

6.2.1 Python Parser

The parser is a small Python script called *victim_parser.py*. It extracts all of the data from *victims.arff* and, for each entry, creates the training data as described above. We used a k value of 40, which is to say we tracked whether or not each victim would live for up to 40 timesteps. Each training datum is tagged as positive or negative and written to disk. Thus, from the ∼166,000 data points we extracted 6.5 million training samples.

6.2.2 C++ Strong Classifier Generator

This component is based on the research Ryan did this past summer. The program (*victim_adaboost.cpp*) takes as input the output from the python parser and stores the data as vectors of `FeatureVector` objects, which consist of a POS/NEG flag and a vector of floats representing the actual feature vector. After the training data has been internalized, the algorithm sets to work finding the best weak classifiers and stores them in a vector (our strong classifier). This vector is then written out to disk.

6.2.3 Lisp Evaluator

Finally, it comes time to use the strong classifier to classify our victims. This code is stored in *adaboost.lisp*. The evaluator function takes in a list of values that represent the senses and the A^* distance to the nearest exit. Our evaluator then tests each weak classifier in turn. For each weak classifier that evaluates to a negative result, we assign the evaluation $h = -1$. Positive evaluations are assigned $h = +1$. Since each weak classifier has a weight $\omega \in \mathbb{R}$, we calculate the final evaluation H by combining these features and summing over each weak classifier:

$$H = \sum_i h_i \omega_i.$$

The evaluator returns this H value indicating whether or not our agent can save the victim. We decided to save only victims with an evaluation $H > 1$, since we have noticed that victims with $H \leq 1$ tend to (barely) expire before we can reach the exit. Using this buffer allows for detours caused by objects and other agents that the agent cannot know about until they come into the percepts.

6.3 Results

#	Error (%)	Precision (%)	Recall (%)	$\mathbf{F_1}$ (%)
1	3.58	97.60	98.60	98.10
2	3.58	97.60	98.60	98.10
3	3.58	97.60	98.60	98.10
4	3.37	98.11	98.28	98.19
5	3.37	98.11	98.28	98.19
6	3.37	98.11	98.28	98.19
7	3.32	98.62	97.80	98.21
8	3.38	98.34	97.04	97.69
9	3.47	98.67	97.59	98.13
10	3.22	98.80	97.74	98.27
11	3.17	98.82	97.77	98.29
12	3.23	98.90	97.63	98.26
13	3.01	98.62	98.15	98.38
14	3.18	98.88	97.70	98.29
15	3.24	98.93	97.59	98.26
16	2.91	98.84	98.04	98.44
17	3.18	98.94	97.64	98.29
18	2.99	98.90	97.89	98.39
19	2.97	98.86	97.95	98.40
20	**3.07**	**99.06**	**97.64**	**98.34**

Table 2: Results for our health data classifier.

In an effort to gauge how effective our strong classifier is, we implemented cross-validation in C++. Some of the statistics-extraction part of CV was already ready to go from the summer. The actual partitioning and validation, however, was written for this project. We experimented with a number of k values (discussed above), but ultimately decided on using 40. Below we have listed the performance of the ensemble classifier up to each of 20 weak classifiers; thus the final row is the performance of the strong classifier. Our first classifier had an error of 3.58%, which we were able to reduce to 3.07% after further boosting. We were briefly nervous about how low the error rate was after just the first classifier, but we realized that there is very little (or no) noise in this data, so it makes sense that we can get excellent results. The classification error rate, precision, recall, and the F-measure with $\beta = 1$ for 10-fold cross-validation are presented in Table 2.

7 Concluding Remarks

Overall, we felt that our agent performed quite well. Strategies for improving each of the individual parts have been mentioned in the presentation of each of those parts. A few issues stood out as particularly important, however.

Our agent needs to be able to better handle debris and other agents. The simple strategy discussed in the A* section works for simple cases, but as we discovered during the tournament, it breaks down in more complicated situations. We do particularly poorly in the case when our agent attempts to move into the same square as another agent. The debris issue would require some form of higher-level planning or search in the state machine; our dealings with other agents could be much improved through the addition of a few more rules to the reflex agent.

In general, the agent would ideally do more consideration of the long term than it does now. Replacing much of the state machine with some kind of search or planner – possibly just A* through a much higher-level space, or perhaps some form of partial-order planner – would allow us to look further in the future and, hopefully, perform better based on that.

In lieu of such a major effort, however, the state machine could be extended in such a way as to do some more long-term consideration. The debris issue noted above would be one such way; another might be to scan all of the victims in an area first and then prioritize their rescue based on those results. (This could lead to saving victims nearer to death first, as well as saving long trips back to a room containing only corpses.)

The inherently uncertain nature of the environment could also benefit from more probabilistic reasoning. This was touched on in the memory section, but the ideal overall framework for our agent might be some sort of probabilistic planner instead of the state machine, still making use of the other components developed here. Probabilistic planners are, of course, far more complicated to develop, debug, extend, and even run than are finite state machines.

References

Eaton, E. (2009). *Building an intelligent agent for search and rescue.* Available from `http://cs.swarthmore.edu/~eeaton/teaching/cs63/project/description.html`

Freund, Y., & Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, *55*(1), 119–139.

Russell, S. J., & Norvig, P. (2003). *Artificial intelligence: A modern approach.* Upper Saddle River, N.J.: Prentice Hall/Pearson Education.