On-Line Searching With Quagents

Harry Glaser
with Team Members Leland Aldridge & Tom O'Neill

February 9, 2006
CSC 242: Artificial Intelligence
Professor Chris Brown

Abstract

Three algorithms for on-line searching are implemented as Quagents and tested in the University of Rochester's URQuake environment. RandomWalker selects directions at random, but will only turn completely around when no other options are available. WallHugger follows the wall on its right all the way around the maze. The final agent is based on Korf's Learning Real-Time A* algorithm, which assigns a heuristic, the Manhattan distance to the goal, to each node in the maze until it learns of a better estimate through exploration. This agent chooses the direction that leads to the node with the least estimated cost, updating cost estimates as it goes. In a deviation from Korf's algorithm, the agent flags and refuses to revisit nodes with only one successor and the nodes that must lead to single-successor nodes.

Introduction

The goal of this project is to design an agent that can "solve" a maze with no *a priori* knowledge other than the (x, y) location of the goal. The only way for an agent to gather more information about the maze is for the agent to move around in the maze. In that way, the search is "on-line." The agent must decide upon an action in constant time, execute that action, perceive its new state, and then use that information in deciding upon its next action. In particular, the only information that the agent can use in its decision-making process is information that it has perceived while in the environment.

The agents exist in the simulated URQuake environment, which was developed from the popular "Quake II" video game at the University of Rochester. The environment allows simulated "Quake Agents" (Quagents) to connect to the environment, to request information that represents simple sensory perception, and to execute simple actions. For the purposes of this project, the agents are permitted the following behaviors:

- Request absolute position (in Cartesian coordinates) and heading (in degrees).
- Request the distance that can be "seen" in four cardinal directions (i.e., the distance until some object, presumably a wall, is encountered). These directions are relative to the agent.
- Turn any number of degrees.
- Walk forward for a given distance.

The agent is also given the (x, y) position of the goal.

Four algorithms, of increasing complexity, were designed and implemented for this project.

Algorithms

Drunkard

Drunkard is an extremely simple agent that selects a direction and length at random and attempts to walk in that direction for that length. In general, due to bumping into walls and frequently walking back over old paths, Drunkard is lucky if it leaves the square it started in. It is more useful as a framework for creating more sophisticated agents than as a viable agent in itself, and it has not been included in testing. The Python code for Drunkard is available on the University of Rochester Computer Science Undergraduate network at `/u/hglaser/pub/Drunkard.py`.

RandomWalker

RandomWalker represents a merely incremental improvement on the Drunkard algorithm, though RandomWalker's results are far better. Specifically, RandomWalker has been given the ability to reason about its surroundings as states with possible successors. Hence, RandomWalker identifies the possible moves from its current location, chooses randomly among them, and walks in that direction. This decision-making process occurs each time a previous action has completed.

Early testing showed that RandomWalker, like Drunkard, was likely to remain very near to the location in which it started. Precisely, as the walking distance from the starting node to a node *n* increases (and the number of opportunities to turn around grows), it becomes exponentially less likely that RandomWalker will visit *n*. To combat this issue and encourage exploration, RandomWalker was modified so that it would only turn around and walk back to where it had come from if no other options were available to it.

RandomWalker is not complete, but it can randomly generate an optimal solution. The Python code for RandomWalker is available on the UR CSUG network at `/u/hglaser/pub/RandomWalker.py`.

WallHugger

WallHugger has a four-step structure, which is repeated indefinitely:

1. Perceive the immediate environment.
2. Decide which way to turn.
3. Turn in the selected direction.
4. Walk forward.

In fact, WallHugger shares this structure with RandomWalker. The only difference between the two agents is in the second step: whereas RandomWalker chooses a direction at random, WallHugger maintains an internal preference regarding the choices. Right turns are most preferable, followed by forward marches, and then left turns, and finally about-face turns. After perceiving its current state, WallHugger selects the highest-ranked successor that is available to it. In this way, WallHugger simulates an agent that is "keeping its hand on the right wall." In some sense, WallHugger is not a "search" at all, but rather a methodology for proceeding through the maze.

WallHugger is not optimal, and it is only complete for certain types of maps, as will be seen later. The Python code for WallHugger is available on the UR CSUG network at `/u/hglaser/pub/WallHugger.py`

Learning Real-Time A*

The final search agent attempts to "learn" the path to the goal by filling out a table of cost values. This algorithm is based on R. E. Korf's Learning Real-Time A* (or

LRTA*) algorithm, which he published in 1990 [1]. The cost table is persistent, and it is empty at the beginning of the algorithm. It is indexed by node locations (x-y tuples), and it contains the most recent estimate of the cost to the goal node from the index node. Initial cost estimates are based on a heuristic function. As the agent visits nodes, it fills out the table, using actual knowledge gained about the environment to make its original heuristic estimates more precise.

Real-time decisions about which successor to pursue are made using the best current information. When a table estimate is unavailable for a successor (presumably because that successor has not been visited), the minimum possible value is used – namely, the heuristic estimate for that node. This encourages exploration of new nodes rather than retreading old paths. The LRTA* agent selects the successor with the smallest cost estimate. Before leaving a node, the agent updates the cost estimate for that node to be the cost estimate of the minimum successor that was chosen, plus the cost of travel to that successor. As all single-node movements are equal in this environment, that cost is a constant. A higher value for this constant will discourage revisiting old states, which can be advantageous or disadvantageous, depending on the situation.

The precise LRTA* algorithm that we used is as follows:

1. Perceive the current position and available movement directions.
2. Compute current state from current position and possible successor states from available movement directions.
3. For each successor state *s'*: If `cost_table[s']` exists, set `cost_to_s' = cost_table[s']`. Otherwise set `cost_to_s' = h(s')`, where *h* is the heuristic function.

4. Select the successor *s'* with the least cost. For the current state *s*, set

   `cost_table[s] = cost_to_s' + movement_cost.`

5. Translate successor state *s'* into movement instructions from current state *s* into *s'*. Move (into *s'*) according to movement instructions.

For LRTA* to be complete, the heuristic function h must satisfy $h(n) \leq c(n)$ for all nodes n, where c is the actual cost function. That is, the heuristic function must always underestimate the actual cost from a node to the goal. For our heuristic function, we chose the "Manhattan distance" h_m, where $h_m(n_x, n_y) = |n_x - goal_x| + |n_y - goal_y|$. This satisfies the underestimation requirement.

The LRTA* agent has one significant deviation from Korf's algorithm. When a node has a single finite-cost successor, its cost estimate is set to an infinite value. It is assumed that in a state with a single successor, the single successor must be the node that was visited previously. With this modification, we ensure that the agent will never visit a dead-end state twice. Since only successors with finite costs are counted as successors in this modification, the entire path to a dead-end state in which there are no options but to go to the dead end or to turn around will be recursively marked as infinitely costly.

LRTA* is not optimal, but it is complete. With the modification for infinitely costly dead end paths, the algorithm is only complete for maps that can be represented by undirected graphs. Specifically, if a state has one transition out, then it must lead to the only state that can transition into it. This will be true for most "real-world" simulations. The Python code for the LRTA* agent can be found on the UR CSUG network in `/u/hglaser/pub/LRTAStar.py`.

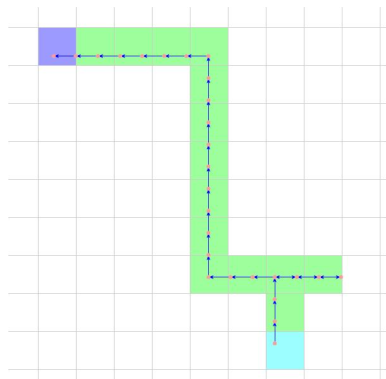
Results

Each of the agents was run in two mazes. One is 10x10 grid squares, and the other is 20x20 grid squares. During testing, the agents were used a graphical maze output program that enabled testers to observe the agent's progress in the maze and to capture images of that progress. Those are the images included here. Squares colored blue are start squares, squares colored purple are goal squares, and squares colored pink are observed but unexplored squares.

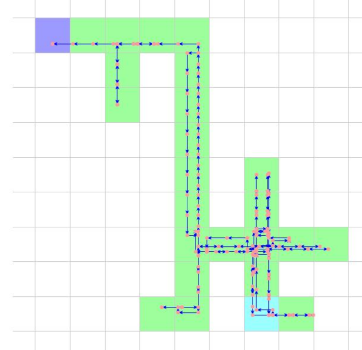
10x10 Maze

Ten trials of **RandomWalker** were conducted in the 10x10 maze. The number of steps to completion for each of these trials is as follows:

Trial #	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Time to Completion ("walk" operations)	209	296	30	182	124	253	34	162	155	322



Trial 3: 30 steps to completion



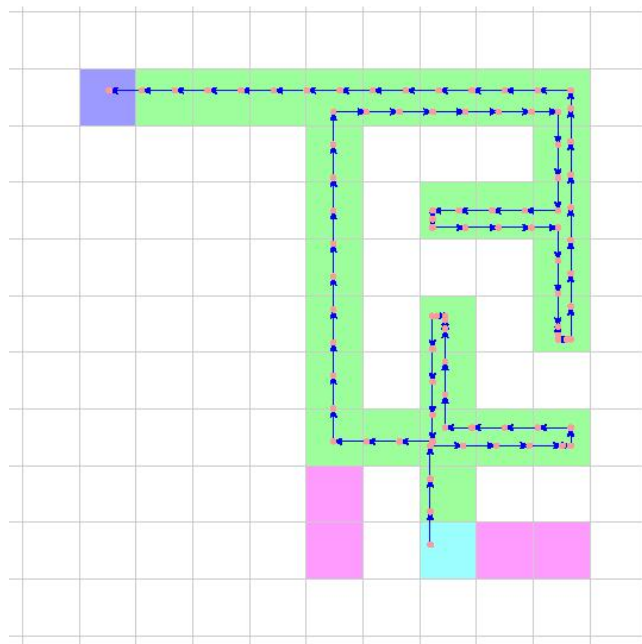
Trial 1: 209 steps to completion

The average time for RandomWalker to complete the maze is 176.7 steps. Additionally, in one test, RandomWalker walked for 519 steps without completing the maze at all. Factoring that number in, the average time to completion becomes about 209 steps.

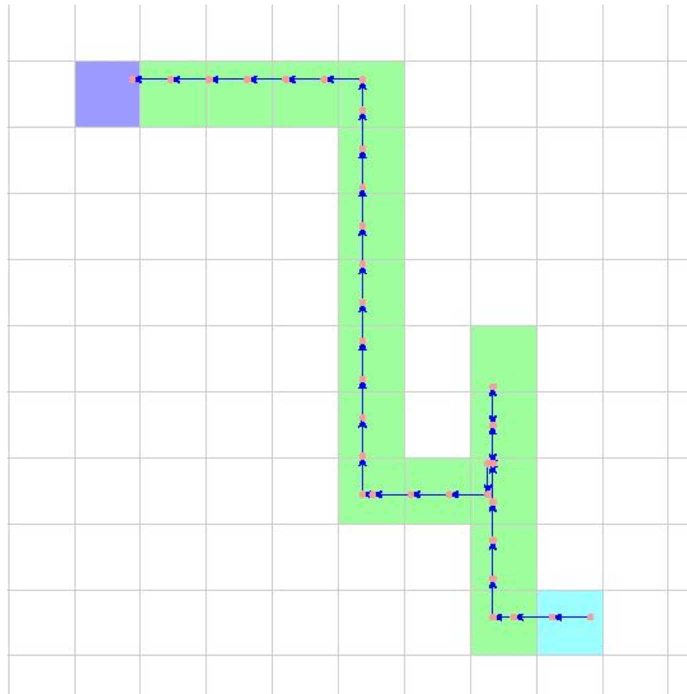
Since RandomWalker will not turn around unless it is forced to, there are four decision nodes (including the starting node) in the most direct route from the initial node to the goal node. One of them has three choices, and the rest have two. Thus, there is a 1/24 probability that RandomWalker will walk straight to the finish line, which we estimate would take approximately 27 steps. This is even faster than LRTA*, as will be seen momentarily. Even when this 1/24 chance is not reached, there are paths which take only slightly more steps, as in Trial 3, depicted above. Of course, it is also possible for RandomWalker to take a very long time, as in Trial 1, also depicted above.

Since WallHugger and LRTA* are deterministic, they performed the same route in every trial.

WallHugger completed the 10x10 maze in 121 steps. Its route is depicted here:

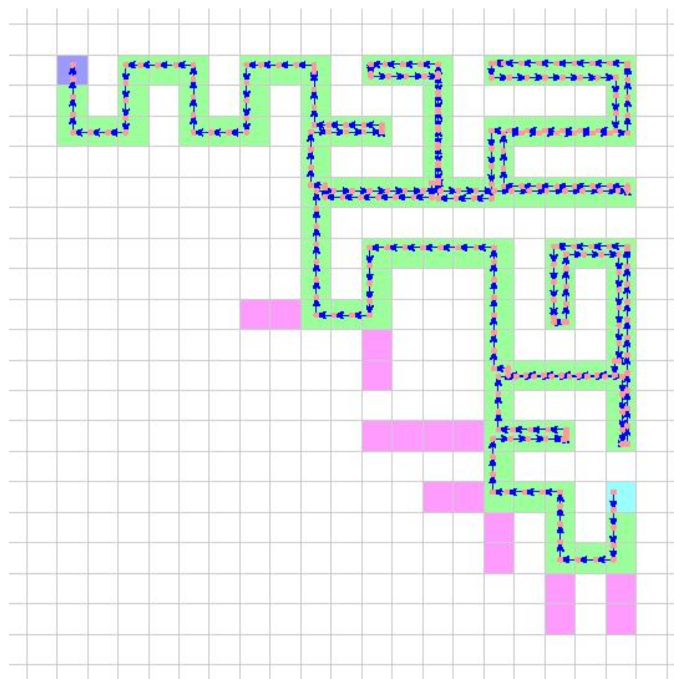


LRTA* completed the 10x10 maze in 35 steps. Its route is depicted here:

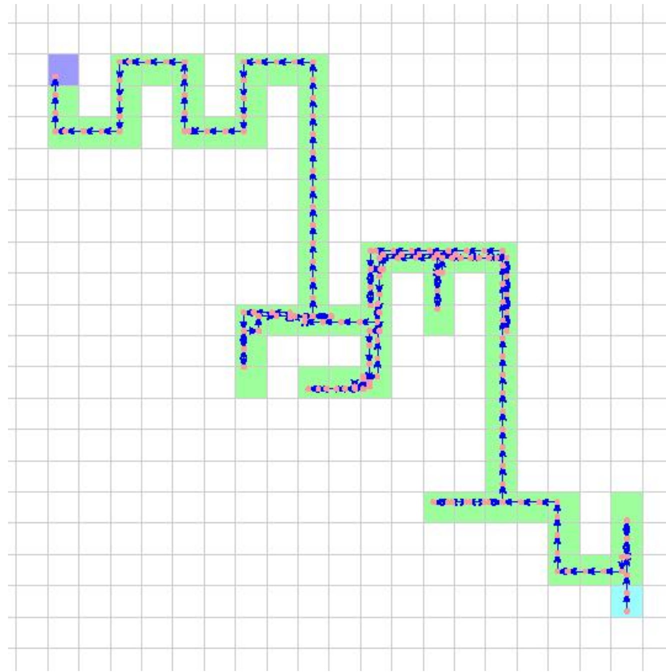


20x20 Maze

WallHugger completed the 20x20 maze in 386 steps. Its route is depicted here:



LRTA* completed the 20x20 maze in 196 steps. Its route is depicted here:



In ten trials, **RandomWalker** never completed the 20x20 maze.

	RandomWalker (avg.)	WallHugger	LRTA*
10x10 Maze	209 steps	121 steps	35 steps
20x20 Maze	Never	386 steps	196 steps

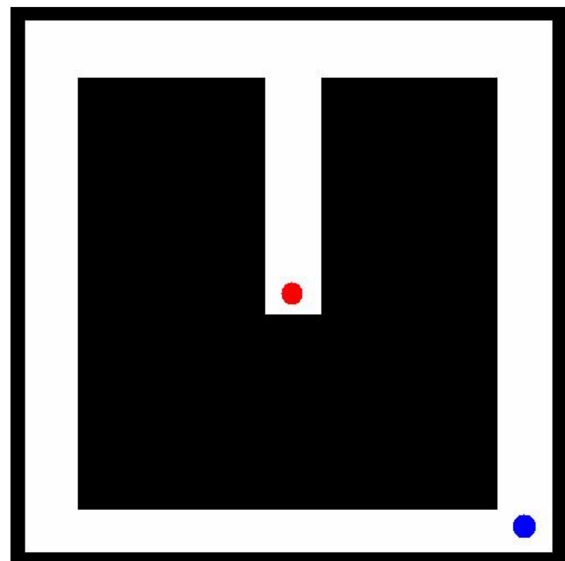
Summary of results for three algorithms over two mazes

Discussion

Though RandomWalker is inconsistent in its performance, it often does respectably well in a small maze. As shown above, in our 10x10 maze, there is a 1/24 probability that RandomWalker will march straight to the goal without any missteps. Though this did not crop up in any of our trials, paths with only one small "misstep" did occur. In these cases, RandomWalker made one wrong turn, encountered a dead end, turned around, and proceeded to make only correct turns until the goal was reached. In

these cases, RandomWalker matched or edged out LRTA*, and soundly defeated WallHugger. Of course, many times (eight out of our ten trials), RandomWalker did significantly poorer than the other two algorithms. And when the map grows to 20x20 cells, RandomWalker's performance diverges, to the point where it did not complete the maze within 600 steps.

This seems to suggest a modification to RandomWalker in which RandomWalker is killed and a new RandomWalker is spawned after a certain amount of time has passed. However, this new agent could not be considered "on-line." An on-line version in which RandomWalker simply returns to the start after a certain amount of time seems not to be worth it: there is no reason to believe that the original starting point is any closer to the goal than the currently-occupied node, much less worth the cost of returning.



A maze which WallHugger, starting at the blue point and moving upwards, would not complete.

WallHugger performs as expected. With the goal placed in the opposite corner of the maze from the starting location, as it was in these trials, WallHugger traversed fully half of the maze, plus a significant amount of backtracking. It is possible to design a maze in which WallHugger would be forced to traverse the entire maze twice before reaching the goal. In fact, WallHugger is not guaranteed to be complete: it is possible to design a maze in which WallHugger never finds the goal. One such maze is

depicted here. However, barring any of these special cases, WallHugger can generally be relied upon to find the solution in a time proportional to the area of the maze.

The LRTA* agent effectively cuts out the half of the maze that WallHugger explores. Generally speaking, the LRTA* agent attempts to draw a straight line from the starting point to the goal. It will make a mistake when the path that most closely resembles a straight line to the goal is not the correct path to take. In this way, it might be said that the LRTA* agent makes roughly the same choices that a human with a compass might make. However, the LRTA* agent does not do so well when the only successors both lead equally far away from the goal. The agent will set off in one of those directions, quickly discover that the estimates are getting larger than the heuristic estimate from the successor in the other direction, and so it will turn around and try that direction until it encounters the same problem. This can lead to the zig-zagging behavior that was observed in the 20x20 maze.

Some of this undesired behavior can be ameliorated by raising the cost of an individual move, effectively “punishing” the agent for retracing its steps. However, a move cost that is too high will result in the agent refusing to try a path again, even when that is the desired behavior.

Despite these problems, the LRTA* algorithm performed admirably. On the smaller maze, it walked straight to the goal with only one minor misstep. On the larger maze, it did zig-zag slightly, but then discovered the correct route, and confidently marched to the goal from then on.

Future Work

Further work in the direction of optimizing LRTA* so it does not reach these "indecisive" meta-states would certainly be fruitful. In [1], Korf mentions that giving unexplored states an estimate equal to its heuristic value is intended to encourage exploration of new states. This seems worthwhile, but in a situation when two directions take the agent equally far away from the goal, our LRTA* agent still backtracks after a few steps in one direction when the heuristic from the immediate unexplored successor in the other direction starts to look enticing compared to the diminishing returns it is getting in the current direction. On the one hand, that might be valuable, as the other direction might immediately turn towards the goal. On the other hand, often it does not, resulting in this "zig-zagging" behavior. Some exploration of increasing the cost estimate of unexplored states seems worthwhile.

Additionally, RandomWalker has shown good potential. One idea for leveraging the correct guesses is, at each node, to give each successor node a ranking based on the heuristic (Manhattan distance) used for the LRTA* agent. Then the RandomWalker chooses a successor s' with probability p proportional to $h(s')/H$, for h being the heuristic function and H being the sum of heuristic values of all the successors of the current node. The hope is that this sort of "StochasticWalker" might leverage the advantages of RandomWalker, while introducing a note of sanity by giving choices that take the agent farther away from the goal a low probability of being chosen.

Finally, Professor Brown's idea of spawning more agents at decision nodes and having each agent explore one of the succeeding paths seems very attractive. This type of search resemble a sort of non-deterministic Breadth-First Search, in which all the successor nodes are expanded simultaneously, and if one of them leads to the goal, then

that is the path that was taken. This method is not on-line unless the environment allows for instantaneous asexual breeding (and children that follow parental instructions to the letter). However, it might be considered on-line for some “Puppeteer” agent that is using the “physical” agents to explore the map. In this case, the new agents would not have to inherit the “memory” of the parent agents in order to produce a path at the end. Rather, the Puppeteer would be following the progress of all of its sub-agents, spawning new ones (and perhaps killing agents that reach a dead end) as needed.

Division of Labor

This project was truly a team effort. The divisions below are general at best: we all participated in planning development; drawing out, critiquing and revising algorithms; tweaking code; and fixing bugs.

- This paper was written by Harry Glaser.
- The trials were run, and pretty pictures generated, by Tom O'Neill.
- RandomWalker was written by Harry.
- WallHugger was written by Leland Aldridge.
- The LRTA* agent was written by Harry (though the algorithm is due to Korf [1]). The modification for infinitely costly dead end paths was dreamed up and added by Leland.
- The real-time graphical map display program was written by Tom.
- Leland is responsible for a couple of key solutions to environment-specific problems, notably discovering the width of a grid square, and correcting the path when a Quagent catches its shoulder on a corner.
- The back-end library to provide Quagent functionality in Python was written by Harry.

References

[1] R. E. Korf "Real-Time Heuristic Search", *Artificial Intelligence*, Vol. 42, No. 2-3, March 1990, pp. 189-211. 1990.