An Agent-based Approach to Robot World and N-Queens

Connor Langlois

Computer Science Department, University of Maine
Orono, ME 04469

connor.langlois@maine.edu

***Abstract* — This paper describes the simulation of different implementations of agents in robot world and n-queens. It includes write-ups of the different types of agents and their behavior in the robot world and n-queens domains as well as some statistical analysis of the results gathered.**

***Keywords* — Simulator: the main orchestrator of the agents and the world. Agent: an autonomous entity that makes decisions in its world. Agent program: an implementation of the agent function, which maps percept sequences to actions. World: the environment in which agents act.**

I. Introduction

The purpose of this paper is to describe the simulation of different implementations of agents used in two different domains: the robot world and n-queens. In each domain, an agent, with its agent program, attempts to act on the percepts it senses from the world in order to reach its goal. Performance measures and statistical analysis are run on simulations of the agents to gather insight into their behavior.

II. Architecture

The main architecture of the simulation models a component-based system consisting of the simulator, the world, the agent, and the agent program. Each is represented as a class and implements a strong separation of concerns to avoid information leaking and to divide the work done by each. This was done to model how the agent would act in the real world where it would have no access to the state of other objects, such as the world. Only the world knows its own state.

A. Simulator

The simulator is the orchestrator of the agent and the world. It contains the world and agent and thus is able to control the flow of information from the agent to the world and vise versa. It continuously loops to gather the agent's percepts from the world and pass them to the agent. It then receives an action from the agent and passes it to the world to update the state of the agent. The simulator monitors the agent's behavior and terminates after noticing the agent's state has stayed constant for ten iterations. Ten was chosen for simplicity and for the assumption that if an agent's state has not changed for ten iterations, it must be stuck or believe it has reached its goal. The agent's state is then checked against its goal to determine success or failure. Finally, a performance measure is run on the agent's behavior to analyze its performance relative to optimality.

B. World

The world is the environment in which the agent acts. It contains the state of the agent and any other information relative to the problem domain, such as obstacles in the robot world. The world interacts solely with the simulator, having no interaction at all with the agent. This creates a separation of concerns between the world and the agent. Its main communication with the simulator is to receive an action indirectly from the agent. It then uses this action to update the agent's state (if applicable, of course).

C. Agent

The agent is the main actor in the world. It contains the agent program. Its job is to receive percepts from the simulator, which were received from the world and/or prior actions, and produce an action based on those percepts. The agent, however, does not perform the actual work to produce that action.

Rather, it is up to the agent program to receive percepts from the agent and to produce an action. Thus, the agent can seem to act as a middleman between the simulator/world and the agent program. Therefore, in this simulation, the agent is assumed to receive percepts and produce an action deterministically, meaning it does not have faulty sensors or actuators that would manipulate the intended percepts or action.

D. Agent Program

The agent program is where the real work is done. It receives percepts from the agent and produces an action based on those percepts. The agent program thus is what will vary the most between agent types and domains. It can be as simple as a simple reflex agent program or model-based agent program, or more complex as a goal-based agent program or utility-based agent program. In this simulation, goal-based agent programs were implemented as hill-climbing local search algorithms, and utility-based agent programs were implemented as the two classical search algorithms, breadth-first search and A*.

III. Robot World

The first domain simulated was the robot world. The robot world consists of an agent, the robot, inside of a grid of coordinates. The agent's goal is to travel to a specific coordinate (or one of many possible coordinates) while avoiding obstacles that are in its way. It is assumed the goal is reachable. The agent has a front sensor and four bump sensors, allowing it to detect if an obstacle is in front of it and if its prior action caused it to bump into a wall or obstacle, along with which side of the agent bumped. The agent's state is its location within the world and its heading, i.e. the direction it faces. To implement the agent program of the agent, four different agent programs were developed: a simple reflex agent program, a model-based agent program, a goal-based agent program, and two utility-based agent programs.

A. Simple Reflex Agent Program

The first agent program implemented was the simple reflex agent program. The simple reflex agent program's goal is to reach any of the four corners of the world. It is a rather simple agent program. Its only ability is to receive percepts from the world (from the simulator in this case) and produce an action according to its set of if-then conditions. It cannot store a percept sequence (history of percepts) or action sequence (history of actions). It also cannot know about the world, what the world contains, or in what its actions will result. Thus, it is contained in an unobservable and nondeterministic environment. Due to these conditions, the simple reflex agent program essentially has only a few methods to effectively move to its goal. The method implemented is as follows:

1.) Move forward until the front sensor fires.
2.) Move right.
3.) Repeat 1 and 2 in order until the front sensor and right bump sensor fire.
4.) Stop.

This method allows the agent to move through the world and reach one of the corners using the limited available information it has. It works because it must be in a corner if its front sensor and right bump sensor fire due to the agent not having faulty sensors or actuators.

There is, however, one major downfall. The simple reflex agent program cannot determine whether it has reached a true corner of the world or a fake corner produced by two diagonal obstacles or by an obstacle and a wall. Thus, the simple reflex agent program is not one to choose for completeness.

B. Model-based Agent Program

The next agent program implemented was the model-based agent program. The model-based agent program's goal is, again, to reach any of the four corners of the world. It is rather similar to the simple reflex agent program. Its key difference, however, is its ability to store a history of percepts received and actions taken, known as the percept sequence and action sequence, respectively. Similar to the simple reflex agent program, the model-based agent program cannot explicitly know about the world, what the world contains, or in what its actions will result, existing again in an

unobservable and nondeterministic environment. It may, on the other hand, derive implicit information about the world based on its percept sequence and action sequence. Thus, theoretically, the agent program should be able to use this additional information to guide its way to the goal more effectively than that done by the simple reflex agent program.

Disappointingly, however, the model-based agent program implemented did not use this information and instead exactly mimicked the behavior of the simple reflex agent program, leading again to the trap of fake corners. Thus, the model-based agent program implemented is not one to choose for completeness.

C. Goal-based Agent Program

The third agent program implemented was the goal-based agent program. The goal-based agent program's goal is to reach the specified coordinate. This is different than that of the simple reflex and model-based agent programs. Those agent programs' goal was to reach a corner of the world; the goal-based agent program's is to reach a particular coordinate specified. Thus, to deterministically reach this goal it must have some additional information alongside that of the simple reflex and model-based agent programs. The goal-based agent program knows the location of itself and of its goal. This is the key difference between it and the prior agent programs; it has explicit knowledge about the world. Thus, it would behoove it to utilize this information to find a path to its goal.

In fact, the goal-based agent program implemented does just this. The implementation uses the hill-climbing local search algorithm to find a path to its goal. Hill climbing in this domain carries a step cost of one, as a move of the agent is the same as any other move, and utilizes the Manhattan-distance heuristic function to determine which successors are more likely to reach the goal in the shortest distance. The method implemented is as follows:

1.) Get the neighbors of the agent.
2.) Remove those neighbors that have already been attempted.
3.) Calculate the Manhattan distance of each one to the goal.
4.) Take the minimum.
5.) Add it to the list of attempted coordinates.
6.) Move to it.
7.) Repeat 1 through 6 until reached goal.
8.) Stop.

This method allows the agent to move greedily through the world until it reaches its goal. It works because it moves to the neighbor coordinate that minimizes the estimated distance to the goal and does not attempt those coordinates that it already has attempted. Keeping track of attempted coordinates helps the agent avoid repeatedly hitting a wall or obstacle and helps prevent infinite loops of the agent moving back and forth between two coordinates due to their heuristic cost being the minimum each attempt.

There is, however, one downfall of the goal-based agent program implemented. Due to the application of hill climbing and history of coordinates attempted, the goal-based agent program implemented will get stuck in tunnels, 1-by-n areas surrounded by obstacles or walls, where n represents the length of the tunnel. The agent will not backtrack out of the tunnel and therefore remains in the same state in which it was last. Thus, the goal-based agent program implemented is not one to choose for completeness.

D. Utility-based Agent Programs

The last agent programs implemented were the utility-based agent programs. The utility-based agent programs' goal is, again, to reach the specified coordinate. This time, however, the utility-based agent programs have significantly more information at their disposal. In fact, they have all the information about the world. Thus, they know where they are located, where their goal is located, and where obstacles and walls are located. Once again, it would behoove the agents to utilize this information to find a path to their goal. This time, however, it turns out that, with this information, the utility-based agent programs can find optimal paths to their goal. Two implementations of this type of agent program were developed using classical search algorithms: a

breadth-first-search agent program and an A* agent program. Both were implemented to generate successor nodes based on valid neighbors, those neighbors not being walls or obstacles.

1) *Breadth-first-search Agent Program*: The first utility-based agent program implemented was the breadth-first-search agent program. Breadth-first search is a search algorithm that expands all nodes breadth-first repeatedly. Thus, it is guaranteed complete; however, it is not optimal in all domains. In the robot world domain, however, it is optimal due to the constant step cost of one. The method implemented is as follows:

1.) Make a graph with a root node whose state is the starting location of the agent.
2.) Make a frontier, a queue to hold all discovered but not yet explored nodes, initially holding the root node.
3.) Make an explored list, a list to hold all explored states.
4.) Pop off a node from the frontier.
5.) Adds its state to the explored list.
6.) For each of its successors whose states are not in the explored list:
    a.) If the successor's state is the goal state, return the path to that successor.
    b.) Otherwise, push the successor onto the frontier.
7.) Repeat 4 through 6 until termination.
8.) Pop off a state/coordinate from the path.
9.) Move to that coordinate.
10.) Repeat 8 and 9 until the path is empty.
11.) Stop.

This method is much more complicated than those of the prior agent programs. However, the increase in complexity is well worth the benefit, that is, guaranteed completeness. Additionally, due to the characteristics of the domain, this method is optimal relative to the solution as well. Thus, this agent program performs very well in finding a path to the goal and is a viable option to choose.

2) *A* Agent Program*: The second utility-based agent program implemented was the A* agent program. A* is a search algorithm that expands only the lowest-cost nodes. It does this by using a cumulative path cost as well as a heuristic function to estimate the remaining path cost. Thus, it is guaranteed complete as well as optimal if the heuristic is consistent (underestimates the true remaining path cost). The method implemented is as follows:

1.) Make a graph with a root node whose state is the starting location of the agent.
2.) Make a frontier, a priority queue, ordered based on path cost plus heuristic cost, to hold all discovered but not yet explored nodes, initially holding the root node.
3.) Make an explored list, a list to hold all explored states.
4.) Pop off a node from the frontier.
5.) If the node's state is the goal state, return the path to that node.
6.) Otherwise, adds its state to the explored list.
7.) For each of its successors whose states are not in the explored list:
    a.) If the successor's state is not equal to the state of a node already on the frontier, push the successor onto the frontier.
    b.) Otherwise, if the path cost of the successor is less than that of the node on the frontier, replace the node on the frontier with the successor.
3) Repeat 4 through 7 until termination.
4) Pop off a state/coordinate from the path.
5) Move to that coordinate.
6) Repeat 8 and 9 until the path is empty.
7) Stop.

This method is much more complicated than those of the prior agent programs and even that of breadth-first search. However, once again, the increase in complexity is well worth the benefit, that is, guaranteed completeness and optimality relative to the solution. A* also has the interesting property that it is guaranteed to be optimally efficient, meaning that "for any heuristic function, no other optimal algorithm is guaranteed to expand fewer nodes." Thus, this agent program performs extremely well in finding a path to the goal and is the best option to choose.

IV. N-Queens

The second domain simulated was n-queens. N-Queens consists of an agent, which can be abstractly thought of as a machine, whose goal is to place n queen pieces onto an n-by-n chessboard such that no queen is under attack. A chessboard can be represented as a grid of coordinates, just as the robot world domain. In this domain, the agent has no sensors and therefore no percepts. The agent's state is the current state of the chessboard, consisting of the locations of the queens placed. To implement the agent program of the agent, one type of agent program was developed: a utility-based agent program. This was further broken down into two classical search algorithms: a breadth-first-search agent program and an A* agent program.

A. Breadth-first-search Agent Program

The first utility-based agent program implemented was the breadth-first-search agent program. Just as with breadth-first search in the robot world, breadth-first search here performs the same algorithm. The only difference is the choice of the successor generator function and goal test function.

The generator function chosen was one in which successors were states with one more queen placed. The queen placed in these successor states are found to not be under attack, and thus they are chosen to be in the states of the successors. To find the next queen to place that is not under attack, a simple property of the problem can be realized. For all n queens to be safe (not attacked), there must not be any queen in the row, column, or diagonal of any other queen. Thus, to find a viable queen to place next, simply search through each row of the column to the right of the last placed queen for positions that are not attacked by any queen currently placed, that is, where the row, column, and diagonal of that position are empty. Doing so presents possible placements of the next queen.

The goal test function need only test whether there are n queens placed and each queen is not under attack. This is done as in the generator function, by checking if the row, column, and diagonal of the queen are empty. If both of these conditions are satisfied, then the problem is solved and a solution has been found.

B. A Agent Program*

The second utility-based agent program implemented was the A* agent program. Just as with breadth-first search, A* here performs a similar algorithm. The only difference is the inclusion and choice of the heuristic function.

The heuristic function chosen was one that determines how many queens are left to place. This accurately estimates the true path cost because if five queens have been placed, then only 3 more need to be placed to possibly reach a goal state. Thus, this heuristic consistently underestimates the true path cost and is a viable option.

V. Methods

To gather data on the simulations run and gain insight into the performance of the different agent types in the different domains, performance measures were run after each simulation. A solution performance measure was run on simple reflex, model-based, and goal-based agent programs, while a search performance measure was run on the utility-based agent programs. The key difference is that the solution performance measure measured the agent's path relative to the optimal path found by an optimal search algorithm, in this case A*, whereas the search performance measure measured two attributes of the search itself, the number of nodes expanded and the maximum size of the frontier during the search. Doing so allows insight into how well the non-search agents performed in their paths, while also allowing insight into the utility-based agent programs due to their optimality in nature.

Additionally, some statistics were run on the agent's performance using the Excel program from the Microsoft Office Suite. Tables and graphs were created to show the relationships between the different agent types and domains.

VI. Results

The results accumulated over thirty simulations for each agent of each domain are summarized in the tables and graphs below:

TABLE I
Robot World, No Obstacles, Averaged Over 30 Simulations

Agent	**Performance of Agents**		
	Runtime (s)	**Path**	**Optimal Path**
SR	0.0041	22.1	11.03333
MB	0.003133	24.2	12.6
GB	0.001433	15.6	15.6
		Expanded	**Max Frontier**
BFS	0.008233	227.0667	24.83333
A*	0.0038	19.03333	35.26667

TABLE II
Robot World, 30% Obstacles, No Fake Corners, Averaged Over 30 Simulations

Agent	**Performance of Agents**		
	Runtime (s)	**Path**	**Optimal Path**
SR	0.003133	24.2	12.33333
MB	0.002233	23.93333	13.53333
GB	0.002533	17	15.73333
		Expanded	**Max Frontier**
BFS	0.007233	204.7333	22.16667
A*	0.002833	28	23.8

TABLE III
ROBOT WORLD, 30% OBSTACLES, FAKE CORNERS, AVERAGED OVER 30 SIMULATIONS

Agent	Performance of Agents		
	Runtime (s)	**Path**	**Optimal Path**
SR	0.0017	15.93333	15.2
MB	0.0019	14.93333	11.93333
GB	0.002967	20.66667	19.8
		Expanded	**Max Frontier**
BFS	0.014033	251.4	25.63333
A*	0.001767	30.53333	27.83333

GRAPH I
ROBOT WORLD, NO OBSTACLES, NO FAKE CORNERS, FAKE CORNERS, AND AVERAGED OVER 30 SIMULATIONS

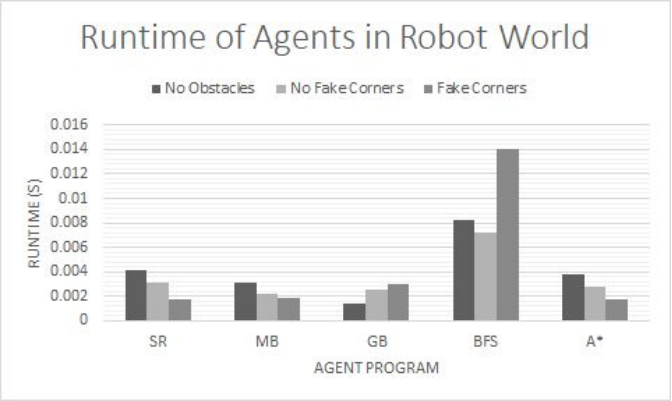


TABLE IV
N-QUEENS, 7X7, AVERAGED OVER 30 SIMULATIONS

Agent	Performance of Agents		
	Runtime (s)	**Expanded**	**Max Frontier**
BFS	0.019533	419	165
A*	0.000633	10	14

TABLE V
N-QUEENS, 8X8, AVERAGED OVER 30 SIMULATIONS

Agent	Performance of Agents		
	Runtime (s)	**Expanded**	**Max Frontier**
BFS	0.156667	1665	573
A*	0.004	114	19

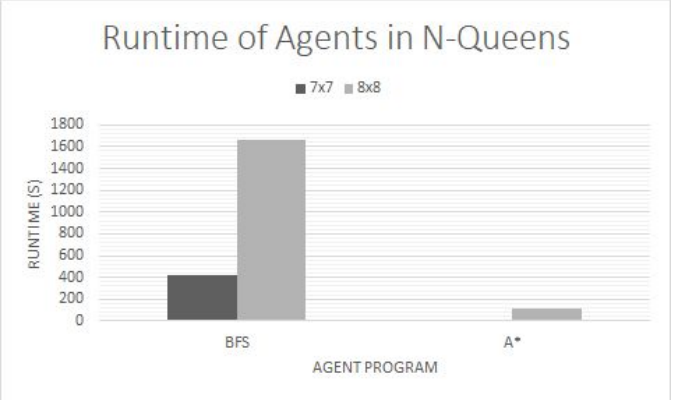
TABLE VI
N-QUEENS, 9X9, AVERAGED OVER 30 SIMULATIONS

Agent	Performance of Agents		
	Runtime (s)	**Expanded**	**Max Frontier**
BFS	1.701933	6977	2295

TABLE VII
N-QUEENS, 16X16, AVERAGED OVER 30 SIMULATIONS

Agent	Performance of Agents		
	Runtime (s)	**Expanded**	**Max Frontier**
A*	5.344133	10053	76

GRAPH II
N-QUEENS, 7X7 AND 8X8, AVERAGED OVER 30 SIMULATIONS



Overall, on average, in the robot world domain, it seems that adding obstacles to the world decreases the runtime of the agents. This is counterintuitive as it would seem that obstacles would add difficulty to the problem and thus require more time to solve. Perhaps obstacles prune off potential paths and therefore decrease running time.

Additionally, it seems counterintuitive that breadth-first search performed significantly worse than the other agents. This can be attributed to simplicity of the domain and size of the grid. The robot world is simple in that a simple reflex agent program can easily maneuver and find its goal. Although not complete, if it is reachable given the simple reflex agent program's logic, then it will find the corner efficiently. Breadth-first search, on the other hand, will search the grid space until it finds its goal, a task that certainly requires more computation. Thus, its runtime is higher.

Furthermore, in all simulations of both domains, A* expanded fewer nodes than breadth-first search. This is expected as A* has its heuristic, allowing it to prune paths that it knows will not be optimal.

Overall, on average, in the n-queens domain, increasing the number of queens / chessboard dimensions increased the path size,

number of nodes expanded, maximum size of the frontier, and the running time.

VII. Conclusions

Changing between agent types and domains told that the implementations could be written better in a more modular way. As different types of agent programs share functionality, such as receiving percepts and producing actions, abstracting this knowledge away from other agent programs allowed for easier use and a more plug-n-play modular style.

One flaw found was that between domains, even more information could be shared than was at the time. Thus, using a modular design across the domains allowed for n-queens to be built rather quickly, and it is suspected that this quick behavior would continue if more domains were developed.

Furthermore, in terms of scaling up, the agents implemented should be fairly stable in higher grid sizes. In fact, if there is a larger grid size with the same number of obstacles, the problem becomes easier for the non-search agents due to fewer collisions. However, as the number of obstacles increases, the non-search agents have a much more difficult time finding their goal due to more collisions and the possibility of fake corners. Of course, this does not matter if fake corners are disallowed. If that is the case, a large number of obstacles will not significantly affect the agents.

Overall for the second domain, N-Queens was chosen due to familiarity with n-queens from reading the book *Artificial Intelligence: A Modern Approach* and with chess in general. Additionally, having already implemented a grid-based world allowed for reuse of the same structure.

In simulating these different agent types in both of these domains, a great deal of information has been learned. It is now known how GPS systems work and how ubiquitous search algorithms are used in the real world.

Acknowledgment

Connor Langlois wishes to acknowledge Roy M. Turner for his teachings and guidance in Artificial Intelligence.

References

[1] Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed., Upper Saddle River, New Jersey, United States: Prentice Hall Press, 2009.

[2] Roy M. Turner, *MaineSAIL*, http://mainesail.umcs.maine.edu/COS470/schedule/slides/heur19-slides.pdf.