Robot world

1. Simple Reflex Agent

This agent only uses the percept information before making any move.

Methodology

To find the corner my algorithm goes forward if there's nothing ahead using the forward sensor. This means it goes forward whenever the front-sensor is nil. If the front-sensor is t, it moves to the right because it means that there is an obstacle right in front of the robot.

This algorithm successfully finds the corner when the world is empty and when the world has obstacles that don't act like false corners or false edges. When the robot faces a false corner or edge made using obstacles, it fails to find the corner because it needs to at least have some memory to track what its actions are for it to avoid such obstacles.

Result

In the simulated results for 0, 10, 20, 30, 40, and 50 numbers of normal obstacles and fake corners, with random as well as a chosen location and the orientation of the robots, it can be seen that the agent does well in the empty world and the world with normal obstacles, however, it fails when obstacles act like corners. In this case the robot stays there thinking that it is a corner. It would behave the same when the simulator size increases or when the number of obstacles are increased.

2. Model-Based Agent

The main information this agent uses to make the next move is also the percepts. However, this agent has a bit of memory that can be used to keep track of its own actions.

Methodology

To find the corner, in normal cases when there are no obstacles that act like false corners or edges, the same two conditions as for the simple reflex agent, work for the model-based agent too i.e. go forward when front-sensor nil and go right when front-sensor is t. To handle the false corner, I added a case where every time it finds a corner, it tries to go around the corner. It knows it is in the corner from its percept values, front-sensor t and right-bump t. It tries to go around by first going back, at this time the "back-flag" is set to 1 to denote that it went back. And then when the back-flag is greater than 1 and both the front-sensor and right-bump are nil(denoting that it is in the process of going around), it attempts to go right. If this action is successful it means that the corner was in fact false. When it goes right, it decreases the back-flag by 1 to denote that it tries going around. Here the false corners are handled. Furthermore I added another case that was necessary to detect a false edge or a stack of obstacles. If the front-sensor is nil but the back-bump is true, the robot goes back. In this way, the robot keeps going back and right, trying to go around the false edge for 5 times. After the 5th time, it keeps going forward, convinced that it found the real edge and reaches the corner. The same flag back-flag keeps the count of the process. The count is maintained because the back-flag is decreased once when it goes right (after back) but it is increased twice when it goes back (when front-sensor nil and right-bump t).

This algorithm successfully found the corner while avoiding many obstacles. It was able to avoid false corners as well as false edges in the simulator. However, as evident by the algorithm, it cannot detect false edges that are more than 5 levels high. Although this can be changed to a big number and it can avoid false edges that are very long, it will still fail in other cases. I think it wouldn't be possible to cover all cases for this agent as there are numerous possible ways an obstacle can replicate a corner and an edge of the world. For instance, in an extreme case, the obstacles may make an enclosed grid inside the world.

Result

In the simulated results for 0, 10, 20, 30, 40, and 50 numbers of normal obstacles and fake corners, with random as well as a chosen location and the orientation of the robots, it can be seen that the agent does well in almost all cases. It is able to find the corners even when there are 30 obstacles among which a number of them act like false corners and edges. It only fails when the obstacles are stacked up too high, i.e. more than 5 levels. Which again, can be increased, but there is no way to increase that and avoid absolutely all possible structures that the obstacles can make. This is because I could increase the going around movement to 24 times, one less than the current simulator. But this wouldn't work when the simulator size increases.

3. Hill-climbing agent

This agent can keep track of its actions and the world to some extent. It starts from a random position and it knows where its goal is and it knows the heuristic values of its nearest neighbors. In the maze created by the simulator, it can detect its immediate neighbors that are forward, backward, left and right. It will then compare their heuristic value and choose the best among the four positions and move. The next move is again determined by the best neighboring positions available at that position. If there is no better position than the current position, it stays in its current position. In this case it never reaches the goal.

Methodology

This needed a lot of change compared to the model-based agent. I made a class node with attributes, parent (to keep track of the parent of the node) , ro-col (to keep track of the row and column coordinates of the node), and g (to store the heuristic of the nodes) which will be useful for the hill-climbing algorithm. Almost all the code is written in the agent-program method itself. It uses functions to generate children and choose the best child. The function that chooses the best child uses a least heuristic function to choose the least heuristic among the children. The heuristic chosen for this agent is the manhattan distance of the node to the end node. In this way the agent chooses the neighboring position that is closest to the goal node based on their manhattan distance.

The algorithm starts by choosing a random location as the start node and defining the end node. The current node is chosen as the start node and its heuristic is calculated. The children are generated and if there is a better position among the children, the agent moves to the better position. Before moving, the agent changes its orientation so that it is always facing north when making any moves. Once it is facing north, the agent moves either front, back, left or right as per the positions calculated. The action is chosen by subtracting the agent's current position from the next position. For instance, if the result of the subtraction is (-1 0) this means the agent needs to move left.

I tried avoiding the obstacles by checking the percepts and if any one of them is true, removing the best child from the children list so that the algorithm chooses the next best child. However, This did not work in all cases as most of the times, the current node has better heuristic than the remaining child nodes and so it stays in its current position and fails to reach the goal node. This is to be expected from the hill climbing agent. This issue would be removed if it were given the map of the world like in the other search agents.

Result

In the simulated environments 0, 10, 20, 30, 40, and 50 numbers of normal obstacles and fake corners, with random as well as a chosen location and the orientation of the robots, the robot did well in finding the goal position as long as it didn't run into any obstacles in its way. Whenever there is an obstacle, it is removed from the robot's child list so that it chooses a different path. This worked in some cases but in others, when there are more than 10 obstacles concentrated around the goal node, it stays in its place because it has no other better option than to be on its own or on the obstacle's position. This could be solved by not comparing the robot's own position with the children's so that it takes another path but it would not be the original hill climbing algorithm then.

4. Uniform Cost Search Agent

This agent has the information of the locations and actions of the world and itself. Here, finally the agent uses the map of the world, where it can detect and avoid obstacles. It not only has information about its immediate neighbors but it also stores information about all the neighbors it has ever come across and organizes them into neighbors that were visited(closed-list) and neighbors that are yet to be visited(opened-list). It uses the same node class as the hill-climbing agent. Here it chooses the children based on its distance from the start node i.e the textbook g value. Once its children are generated and added to the opened-list, it chooses the child with the least heuristic value g and makes the next move.

Methodology

For this, the search of the optimal path using the uniform cost search algorithm happens in the agent's head. The actual search algorithm code is in a function called ucs that takes the maze(replica of the simulated environment), the start node and the end node. The maze is created using the create-map function that extracts the location of the obstacles from the simulator. The maze is initially a 2D array which is the same size as the simulator and whose elements are all 0. This maze is updated by the create-map function by setting 1 at all the obstacle locations. The current node is first selected as the start node. We then enter the loop over the opened list nodes. Inside this the current node is added to the closed list and it is expanded. The children are generated with the generate-children-maze function where the next children positions are added to the opened list if the position is within the walls of the maze ((25 25) in this case) and if there are no obstacles in the maze (1 stands for obstacles and 0 stands for empty). The children are added if it is not in the closed list and if it is already in the opened list, it is added if and only if the heuristic value of the child is better than the heuristic value of the same child already in the opened list. If the current node is equal to the end node, the goal has been found and the function returns a complete path.

The path generated from the ucs function is used by the agent-program of this agent to find what action to take next. First, like in the hill climbing algorithm, the robot is made to face the north. Then, the action is chosen as per the next position in the path. The robot keeps making the next move by extracting the first position in the path, removing it from the path, and in the next call, extracting the first position of the path, and so on until the path is empty or null. In this way the robot optimally moves towards the goal node and stops once it reaches it.

Result

In all the 20 simulated environments with 0, 10, 20, 30, 40, and 50 numbers of normal obstacles and fake corners, with random as well as chosen locations, goals and the orientations of the robots, as shown in the results generated, the agent successfully reaches the goal in all cases. This agent like all search agents in this assignment would certainly scale up both when increasing the world size and the number of obstacles.

5. A* search agent using Manhattan distance

Methodology

This agent is very similar to the uniform cost agent. The only difference is the heuristic value used. Here the heuristic value is $f = g + h$, where g is the distance of the current node from the start node and h is the estimated distance (Manhattan distance) of the current node to the goal node. The Manhattan distance is computed by the function of the same name. This distance is more appropriate for our robot world because it represents the world and the robot's actions better. This is due to the constraints like the robots can only move in right angles and the world is a grid structure. The rest of the algorithm is the same. The search happens in a function called astar that takes the parameter maze, start-node, end-node and the heuristic choice variable. The function is called with the heuristic variable 1 if we want to use the Manhattan distance as the heuristic, and any value other than 1 if we want to use the other heuristic function mentioned below.

Result

In all the 20 simulated environments with 0, 10, 20, 30, 40, and 50 numbers of normal obstacles and fake corners, with random as well as chosen locations, goals and the orientations of the robots, A* search agent finds the goal successfully in even the worst cases of obstacles. This agent like all search agents would certainly scale up both when increasing the world size and the number of obstacles.

6. A* search agent using Euclidean distance

Methodology

This is the same A* agent but with the heuristic function computed using the Euclidean distance. Although the euclidean distance doesn't estimate the distance better than the Manhattan distance in this specific world, it always underestimates the actual distance which is important for the A* algorithm to be optimal. The distance is calculated by the function euclidean-distance. The rest of the algorithm is the exact same as the A* search agent using Manhattan distance. This uses the same function astar with the heuristic variable choice parameter as anything other than 1.

Result

In all the 20 simulated environments with 0, 10, 20, 30, 40, and 50 numbers of normal obstacles and fake corners, with random as well as chosen locations, goals and the orientations of the robots, A* search agent finds the goal successfully in even the worst cases of obstacles. This agent like all search agents would certainly scale up both when increasing the world size and the number of obstacles.

Discussion

Simulated environments

As mentioned in the results section of all the agents, I ran all the agent programs in simulated environments with 0, 10, 20, 30, 40, and 50 random obstacles. In addition to these obstacles, I also added a few obstacles in positions (1 1), (25 1), and (25 25) that would create fake corners. For each of these environments, I ran the robots with 4 different goal positions of (25 25), (13 13), (25 10), and (17 23). The combination of all these configurations accounted for 20/21 total environments. I used 50 as the number of simulator sketches for all the runs.

Generating children and next moves

Since the world is a rectangular grid, the children were generated by adding the coordinates of the current location of the robot by (-1 0) to get the left child, (1 0) to get the right child, (0 1) to get the top child and (0 -1) to get the bottom child. Similarly the robot moves were calculated by subtraction the path location from the robot location. Ofcourse, I needed to convert all these into the (row column) coordinates since I used a 2D array to represent the simulator world.

Unavoidable cases

Since we are using randomly generated environments, there were some cases where the robots seemed to fail. For instance, when the robot, in its starting position, was surrounded by obstacles. In this case, the path returned by the algorithm is null and the robot keeps performing the :nop operation. Similarly, in another case when the selected goal position has the obstacle, the path returned is again null and the robot does nothing. There were some days when no matter how many times I ran the create simulator with various number of obstacles, the robot would always end up being surrounded by obstacles. Also, when using obstacles more than 100, there was always a high probability that the robot would end up in one of these unavoidable cases.

Nodes created by the search agents

Along with the environment with no obstacles, there were 21 total simulated environments that the search agents were tested on. By using the closed-list of these algorithms, I was able to extract the list of nodes created for those 21 environments.

It is seen that Uniform Cost agent creates considerably more nodes than A* agent. When there are no obstacles, both these agents create equal number of nodes. As the number of obstacles increase, A* creates lesser number of nodes than Uniform Cost. Similarly, the second A* creates more node than the second A* agent. This is most likely because the manhattan distance used by the first agent as the heuristic was a better estimator than the euclidean distance used by the second A*.

I have pasted the table that has the number of nodes for each scenario. The default start position is (1 1) and the default goal position is (25 25). The goal position if the default for 6 environments but it is updated in the remaining environments. For the cases where there are fake corners at the position (1 1), the robot is added in a random position, so in that case its starting position is not (1 1). This is evident in the table below where, when there are fake corners, the robot starts from a random position which is why it reaches the selected goal positions while creating very few nodes.

#obstacles	Uniform cost	astar1	astar2
No obstacles			
0	625	625	625
Random obstacles, Goal: (25 25)			
10	615	598	613
20	605	579	599
30	101	18	18
40	585	512	564
50	574	532	561
Random obstacles, Goal: (13 13)			
10	305	157	162
20	297	149	156
30	296	157	157
40	286	139	152
50	288	157	157
Random obstacles and fake corners, Goal: (25 10)			
10	133	21	24
20	473	105	137
30	247	59	72
40	541	188	255

50	531	180	239
Random obstacles and fake corners, Goal: (17 23)			
10	75	16	18
20	180	46	46
30	199	41	41
40	387	66	86
50	319	77	96

Paired t-test of Uniform Cost agent and the first A* agent

	ucs	***astar1***
Mean	364.8571429	210.5714286
Variance	33052.82857	45515.65714
Observations	21	21
t Stat	6.248208369	
P(T<=t) one-tail	2.11E-06	
t Critical one-tail	1.724718243	

For this test, our null hypothesis is that there is no mean difference in the number of nodes created and the alternative hypothesis is the upper tailed version that says that the mean difference is greater than 0. I have taken the number of nodes created by Uniform Cost as the first variable and the nodes created by A* as the second variable. The alpha is set as 0.05. I performed the test using excel that gave me a p value of 2.10857E-06, which is much smaller than the alpha value of 0.05. This means that the null hypothesis is rejected.

The test statistic shows a compelling result that Uniform Cost agent created more nodes that the first A* agent with the p-value of 2.10857E-06.

For this specific problem, the Uniform Cost search was almost expanding all the nodes available because the cumulative cost of going from one position to another was the same for all possible children nodes. This is why uniform cost ended up creating that many nodes than A*.

Paired t-test of Uniform Cost agent and the first A* agent

	astar2	***astar1***
Mean	227.5238095	210.5714
Variance	48080.8619	45515.66
Observations	21	21
t Stat	3.810004295	

P(T<=t) one-tail	0.000548311	
t Critical one-tail	1.724718243	

The same was done for the two A* agents. The A* that uses the heuristic Euclidean was taken as the first variable in this case likely because it makes more sense if the variable with the higher mean is the first variable in this test. With alpha as 0.05 again, the test statistic shows that A*(euclidean) agent created more nodes that the first A*(Manhattan) agent with the p-value of 0.000548311.

Run-time Comparisons

As per the run-time computed using the *time* macro, on average, Uniform Cost takes longer than both the A* searches and the second A* search takes longer than the first one. This macro only computes runtime for a given form in the environment and simply returns the value of the form so I ran all the simulations on the slime repl and extracted the run times that are printed. The run times for the 3 agents in start location (1 1) and goal position (25 25) are shown below. This data isn't enough for the paired t-test.

Obstacles	ucs	astar1	astar2
10	0.329	0.25	0.222
20	0.224	0.223	0.222
30	0.248	0.265625	0.244
40	0.212	0.237	0.153
50	0.171	0.017	0.15625
Average	**0.2368**	**0.198525**	**0.19945**

Other Domain

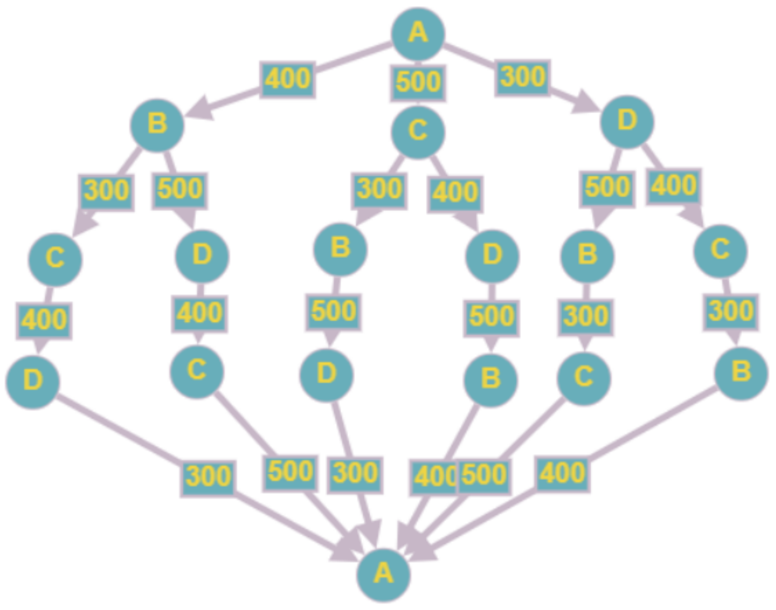
Traveling Salesman

I chose this domain because it was closest to the approach I took in solving the robot world. I already had the node class which I used to create the nodes of the graph. I used a graph with 4 cities A, B, C and D. For this problem, A is both the starting node and the end node of the graph. Here, path should be the optimal path that visits every city once and returns back to the starting city A. Once the graph is formulated, this can easily be solved using either Uniform Cost Search or A* search.

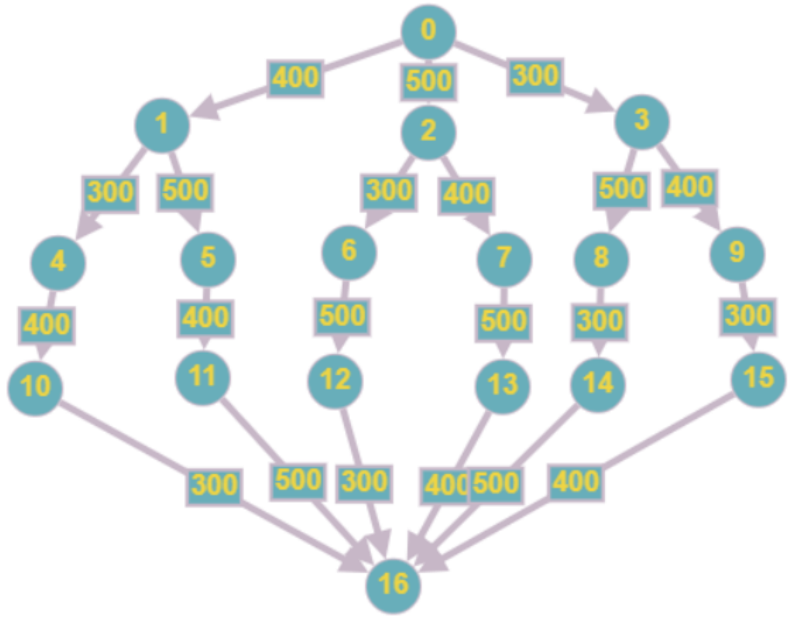
I took a simple case of 4 cities with a matrix that shows its cost of going from one city to the other. Here the cost of going from city A to B is the same as the cost to go from city B to A. The matrix I used is as follows

$$\begin{array}{ccccc} & A & B & C & D \\ A & (0 & 400 & 500 & 300) \\ B & (400 & 0 & 300 & 500) \\ C & (500 & 300 & 0 & 400) \\ D & (300 & 500 & 400 & 0) \end{array}$$

Then I prepared a graph for the problem as follows. Here all the directed paths visit every city once and ends at city A. There are 6 total paths for the 4 cities i.e. (n-1)!



The graph nodes were adjusted so that the names are unique. Since there are 17 nodes in total, the nodes are named from 0 to 16 as follows.



This graph was converted into an adjacency matrix using the make-array. This representation is now ready to be solved by the search algorithms. Once the path was returned, the node names were converted back to the city names. Hence the optimal path for the traveling salesman was computed as **(A B C D A)** .

The nodes were expanded using the adjacency matrix. The heuristic used for UCS was the distance of the current node from the start node(i.e the g value) and in A*, the heuristic was the f = h + g value, where h is the distance of the current node from the end node. For the h values of the A* search, I provided a list of estimated distances for all the corresponding nodes of the graph.

Evidently, this would be hard to scale up, i.e. if we add more cities, the adjacency matrix will blow up and so will the space complexity. Just adding one vertex will increase the size by $(|V|+1)^2$. I chose to represent the graph in this way because due to the time constraint I was unable to figure out how to make the adjacency list representation work in lisp. I chose this simple representation of the problem after everything seemed to fail.

Conclusion

I learnt a great deal about all the different AI agents and their search algorithms. The agents with the least intelligence would be the reflex agents which can be considered as blind agents although they probably are cost efficient in terms of memory and algorithm(software). We can achieve a better performance with the near-sighted hill-climbing agent by using some memory and the ability to track its environment and actions. These agents may or may not find the goal depending on the world they are in. With considerable memory and an efficient algorithm of the uniform cost search, the agent will always reach the goal optimally. Furthermore, when the heuristic is used, as in A* searches, the agent has the practical benefit of having the information that guides it to the goal.

In my implementation of these agents, all the agents behaved like they were supposed to. The search agents performed very well in all the environments and the algorithm performed as expected for the traveling salesman problem too. The agent that needs tweaks and tricks to improve behavior in variable environments is , in my opinion, the model-based agent. Other than that, I feel like I could probably use these algorithms to design a smart roomba.