COS 570: AI Homework 2 Write-up

Nicholas Soucy

October 9, 2021

Contents

1 Introduction

With everything from self drive cars to Dota 2, artificially intelligent agents are everywhere we look. This Homework 2 assignment for COS 570 teaches us the basics of these agents and how to implement their algorithms.

In this write-up we will be discussing the methods of all our agents and the algorithms behind them, in two separate domains, the robot world and the Towers of Hanoi. Then we will run a series of experiments to demonstrate the effectiveness and efficiency of the agents.

2 Methods

2.1 Agent Programs

In this section we will discuss the implementation of each type of agent: reflex, model, hill climbing, uniform-cost, and A*.

2.1.1 Reflex

This agent uses no information on the world and the only percept used is it's forward sensor. Our implementation of the reflex agent is quite simplistic. Our algorithm is as follows:

Listing 1: Reflex Agent

```
input: reflex-agent, percept
output: action
begin
    if there is something in front of us
        action ← :right
    else
        action ← :forward

    return action
end
```

In a world without obstacles, this agent will eventually get into one of the two right corners, however, with obstacles, it will get stuck at local minimum.

2.1.2 Model

The model agent is by far the most difficult agent. This agent has no concept of full world, but can remember previous actions and things solely related to them, like local coordinates. Due to the open-ended nature of the required intelligence level for this agent, we programmed it to get out of false corners with no greater than 3 sides. However, if there is a false corner with 3 sides (not an L, but more like an n) the 'tongs' can not be of size ¿ 1, else it would be trapped at that local minimum.

We achieve this by keeping track of our previous action and implementing two stacks when we get to a possible corner.

Our algorithm is as follows:

Listing 2: Model Agent

```
input: Model-agent, percept
output: action
begin
    action ← nil
    if we are already in a corner
        action ← :nop
    else if there is something on the action-stack
        action ← pop(action-stack)
        previous ← action
    else if there isn't something in front and something behind
        action ← :forward
        previous ← action
        push :backward back-stack
        push :left action-stack
    else if there isn't something in front and behind
        action ← :forward
        previous ← action
    else if back-stack is not null and nothing to the left
        action ← :nop
```

```
20          previous ← action
21          back-stack ← nil
22      else if back-stack is not null and something to the left and nothing forward
23          action ← :forward
24          previous ← action
25          push :backward back-stack
26          push :left action-stack
27      else if back-stack not null and something is in front and left
28          push :nop action-stack
29          for i in back-stack:
30              push (pop back-stack) action-stack
31          action ← :nop
32          previous ← action
33      else if there is something in front
34          action ← :turn-right
35          previous ← action
36          push :backward action-stack
37      else if there isn't something in front
38          action ← :forward
39          previous ← action
40  end
```

We essentially have our agent check to see if a corner is valid by turning and going around, if it can't get around the corner, we back track back to the true corner, if it can get around it, we restart our logic.

2.1.3 Reflex Hill Climbing

For our hill-climbing agent, we decided to implement a reflex agent. Therefore, this agent has no information on the world nor it's previous moves.

Our algorithm for Reflex Hill Climbing is as follows:

Listing 3: Reflex Hill Agent

```
1   input: reflex-hill-agent, percept
2   output: action
3   begin
4      c ← heuristic(:current)
5      f ← heuristic(:forward)
6      b ← heuristic(:backward)
7      l ← heuristic(:left)
8      r ← heuristic(:right)
9
10     action ← min_cost(c f b l r)
11
12     return action
13  end
```

For the heuristic, we decided to do calculate Manhattan distance while ignoring obstacles in the calculation. Our heuristic inputs a direction, like :forward, then calculates the Manhattan distance for the square in front of the robot relative to the goal coordinates. However, once our orientation can be :north, :south, :east, or :west, the calculation for :forward would be orientation dependent. Therefore, we check orientation and have a different calculation for each direction depending on the orientation.

In a world without obstacles, we expect the hill climbing agent to preform perfect and find the optimal path. However, with obstacles, we expect this agent to get stuck at local minimum as it is a reflex hill agent without information on the world.

2.1.4 Uniform-Cost

To perform a search function, we need complete knowledge of our world. Therefore we have our search have access to the goal location, world size, and obstacle locations. Our agent program only has access to the list of actions output by the uniform cost search.

Our algorithm for Uniform-Cost is as follows:

Listing 4: Uniform Cost

```
1   input: uniform-cost-agent
2   output: list of actions that is the optimal path
3   begin
4      visited ← []
5      actions ← []
6      fringe ← Priority Queue
7      fringe.add(start_state)
```

```
8       while fringe is not empty:
9           curr_path ← dequeue(fringe)
10          actions ← curr_path.actions
11          if curr_path is at goal location:
12              return actions
13          else
14              push curr_path visited
15              branches ← get_children(curr_path)
16              for route in branches:
17                  if route is not in visited or fringe:
18                      route.actions ← actions + curr_path.action
19                      route.cost ← curr_path.cost + 1
20                      fringe.add(route)
21                  if route is in fringe and has greater cost than current route node:
22                      fringe.remove(old_route)
23                      route.cost ← curr_path.cost + 1
24                      route.actions ← actions + curr_path.action
25                      fringe.add(route)
26
27      return action
28  end
```

Listing 4: Uniform cost

In our implementation each node has an action stack that holds all the actions that have led us to said node so far. This way, when our priority node is at the goal location, we can pass the action stack to our agent program, then reverse, and pop off each action one at a time.

Once we have to use this algorithm in a different problem later on, we tried to make this as independent as possible from the robot space. Therefore, we made a get-children method that gets all possible nodes we can go to from said node.

Get-children takes a current node as input. We have to make this orientation independent, so we have four different cases for each orientation. In each case, we check to see if each direction is in bounds, and push it on the list of children if it is, if not, we ignore it. We check if something is in bounds to see if the theoretical coordinate is within the grid and if there is not an obstacle there.

The priority queue we used is based off of the simple-pqueue class, we added a few methods, is-empty, get-node-from-state, is-in-queue, length-queue and remove-from-queue.

Is-empty checks if the queue is empty. Get-node-from-state returns the node based solely off of a coordinate. Is-in-queue checks if a node is in the queue based on a coordinate. Length-queue returns the length of the queue. Remove-from-queue removes the given node from the queue.

2.1.5 A*

A* is incredibly similar to Uniform cost, the only change is the cost that each node receives.

The algorithm for A* is as follows:

Listing 5: A*

```
1   input: A*-agent
2   output: list of actions that is the optimal path
3   begin
4       closed ← []
5       actions ← []
6       open ← Priority Queue
7       open.add(start_state)
8       while open is not empty:
9           node_current ← dequeue(fringe)
10          actions ← curr_path.actions
11          if node_current is at goal location:
12              return actions
13          else
14              push node_current closed
15              branches ← get_children(node_current)
16              for node_successor in branches:
17                  if node_successor is not in open or closed:
18                      node_successor.actions ← actions + node_current.action
19                      node_successor.cost ← (node_current.cost + 1) + heuristic(node_successor)
20                      open.add(node_successor)
21                  if node_successor is in open and has greater cost than current node_successor node:
22                      open.remove(old_node_successor)
23                      node_successor.cost ← (node_current.cost + 1) + heuristic(node_successor)
24                      node_successor.actions ← actions + node_successor.action
25                      open.add(node_successor)
26
27      return action
28  end
```

Thankfully, the logic for get-children is the same as Uniform cost as we are in the same world space. However, the cost of each node changes by adding a heuristic cost. We use two heuristics, Manhattan for A*1 and Euclidean for A*2.

2.2 Towers of Hanoi

We choose the Towers of Hanoi because we were interested in possible heuristics for A* and we felt creating the world representation would be fun. However, the assignment encouraged us to choose a domain where we can change the initial and goal states easily, and Towers of Hanoi has very defined initial and goal states. Therefore, we decided to allow for custom number of disks. This way we can also scale up the problem to see how well our search algorithm performs.

2.2.1 World Representation

Once we aren't given a simulator file for the second domain, we need to create everything from scratch.

We start by creating our towers-hanoi object. This has a cost (including a g-cost and h-cost for A*), parents, A, B, and C attributes. Parents is treated as a stack of tower-hanoi objects that led to the current state. A, B, and C are treated as stacks that represent the three pegs.

We extended and slightly modified the simple-pqueue for working with this new world space. We change the print function and all of the methods we added as described in the Robot Uniform Cost section to work with states (tower-hanoi objects) as input instead of coordinates.

When we initialize our first state, we create a list of n integers and set our A stack to that list. For example if $n = 3$, $A = 1, 2, 3$. Each disk is represented by an integer, this makes it easier to compare to see if one disk is larger than the other. Our goal state is to have $C = 1, 2, 3$ if $n = 3$.

2.2.2 Uniform-cost

Our algorithm for Uniform-cost is the same as our Robot World one. However, once we represent our world a little differently, we change a few things.

The algorithm is as follows:

Listing 6: Uniform Cost

```
1  input: uniform-cost-agent
2  output: list of actions that is the optimal path
3  begin
4     visited ← Priority Queue
5     parents ← []
6     fringe ← Priority Queue
7     fringe.add(start_state)
8     while fringe is not empty:
9         curr_path ← dequeue(fringe)
10        if curr_path is at goal location:
11            return curr_path
12        else
13            enqueue visited curr_path
14            branches ← get_children(curr_path)
15            for route in branches:
16                if route is not in visited or fringe:
17                    fringe.add(route)
18                if route is in fringe and has greater cost than current route node:
19                    fringe.remove(old_route)
20                    fringe.add(route)
21
22    return action
23 end
```

There are two main differences in our implementation here compared to the robot world.

1. The visited list is changed to a priority queue. The only reason we do this is because now visited holds tower-hanoi objects instead of just coordinates. Once it is a priority queue, we can use our is-in-queue class to check if a state has been visited already.

2. We also change get-children once the world space changes. Get-children now returns a list of tower-hanoi objects with their cost pre-calculated, so we can just add them directly to the fringe without calculating cost within this function.

For get-children, We first take the top of the stack of A, lets call it *current*. Then, if what is on top of B or C is greater than the value *current* or if B or C is nil, then create a new tower-hanoi object with *current* moved to those possible locations and push those tower-hanoi objects on a list. We then do the same thing for the top of the stack for B and C. Then, we return the list of all possible children states according to the rules of the puzzle.

2.2.3 A*

Our algorithm for A* is the same as our Uniform-cost, to see the algorithm look at the above section.

The only thing that changes between A* and Uniform Cost is the cost for each state. Due to our implementation, we set the cost of each node in get-children. Therefore, get-children had to be slightly modified. When we set the cost of a child node, we set the total cost to be: $f(state) = (1 + state.parent.gcost) + heuristic(state))$. The challenge was developing a heuristic that would work in Towers of Hanoi.

A good heuristic is simple and represents how close a current state is to the end state in some form. To do this, we explore two possible heuristics.

1. The goal state is achieved when n disk are stacked on peg C. Therefore, for each state, we take our heuristic cost to be $n - C.length$. This is one way to measure how far each state is from winning, as the start state is at our maximum, n, and the goal state would measure as 0.

2. Another way to measure our goal state is when both pegs A and B are empty. Therefore our second heuristic is $A.length + B.length$. This way our start state would be our maximum at $2n$ and our goal state would measure as 0.

2.3 Statistical Techniques

For statistical analysis, for each run, we analyze the amount of steps it took for our agent to get there. We also hold the max size of our priority queue for our search algorithms so we can compare the efficiency of uniform-cost versus A*. We then average the results from 20 runs, but we only do this for the robot world, because for Towers of Hanoi the minimum cost will always be $2^n - 1$ where n is number of disk. Therefore, for same n, the cost and the amount of nodes expanded will remain constant.

3 Experimental Results

Due to the difference in domains for the robot world and Towers of Hanoi, experiments are handled differently between the two domains.

3.1 Robot World Results

For each robot world we run each world simulator 20 times. We have three different worlds. World 1 is a world with zero obstacles. World 2 is a world with set obstacles that do not create false corners, and World 3 is a world with false corner obstacles. All obstacles are randomly placed for only the third world. For the agents who need a goal (Hill Climbing, Uniform Cost, and A*) the goal of $(25, 25)$ was chosen.

3.1.1 Reflex

For the reflex agent we collected the total amount of steps the robot took to reach the goal. If the robot did not reach the goal due to local minimum problems, we express that as a 1000. Full data can be shown in the appendix.

The averaged cost for each world of all 20 runs can be found in Table 1.

World 1	World 2	World 3
23.45	26.3	512

Table 1: Average Cost for Reflex Agent in each world.

The number of times the robot failed for each world is shown in Table 2.

World 1	World 2	World 3
0%	0%	45%

Table 2: Percent of times Reflex Robot failed

3.1.2 Model

For the model agent we collected the total amount of steps the robot took to reach the goal. If the robot did not reach the goal due to local minimum problems, we express that as a 1000. Full data can be shown in the appendix.

The averaged cost for each world of all 20 runs can be found in Table 3.

World 1	World 2	World 3
101.85	104.05	399.65

Table 3: Average Cost for Model Agent in each world.

The number of times the robot failed for each world is shown in Table 4.

World 1	World 2	World 3
0%	0%	35%

Table 4: Percent of times Model Robot failed

3.1.3 Reflex Hill Climbing

For the reflex hill climbing agent we collected the total amount of steps the robot took to reach the goal. If the robot did not reach the goal due to local minimum problems, we express that as a 1000. Full data can be shown in the appendix.

The averaged cost for each world of all 20 runs can be found in Table 5

World 1	World 2	World 3
26.15	460.55	802.6

Table 5: Average Cost for Reflex Hill Climbing Agent in each world.

The number of times the robot failed for each world is shown in Table 6.

World 1	World 2	World 3
0%	45%	80%

Table 6: Percent of times Hill Climbing Robot failed

3.1.4 Uniform-Cost

For the search algorithms, it no longer makes sense to compare cost, once we are guaranteed to always get the minimum cost path from both of these algorithms. Therefore, we wish to compare the average number of nodes expanded for each run. Full data can be shown in the appendix.

The average number of nodes expanded for Uniform Cost in all 20 runs can be found in Table 7

World 1	World 2	World 3
781.5	851.1	842.05

Table 7: Average number of nodes expanded for Uniform Cost in each world.

3.1.5 A*

For A* do the same data collection as Uniform Cost as described above. However, we have two different heuristic functions as well. Therefore, we did the same data collection routine for both heuristic functions. Full data can be shown in the appendix.

The average number of nodes expanded for A* with Manhattan distance in all 20 runs can be found in Table 8

World 1	World 2	World 3
157.05	205.2	222.15

Table 8: Average number of nodes expanded for A* with Manhattan distance in each world.

The average number of nodes expanded for A* with Euclidean distance in all 20 runs can be found in Table 9

World 1	World 2	World 3
219	197.95	251.45

Table 9: Average number of nodes expanded for A* with Euclidean distance in each world.

3.2 Towers of Hanoi Results

For towers of Hanoi, once we implemented only search algorithm, it does not make since to report cost, as both algorithms always output the minimum cost path, which is $2^n - 1$. Therefore, like we did with the robot world, we will compare the difference between the amount of nodes expanded between Uniform Cost and A*. Once Towers of Hanoi is a deterministic system, unlike the random start and obstacle locations in our robot world, the amount of nodes expanded for each algorithm is fixed for each value of n. Therefore, we report the amount of expanded nodes for three n values: 6, 7, and 8.

3.2.1 Uniform-cost

The number of nodes expanded for Uniform Cost for varying n can be found in Table 10.

$n = 6$	$n = 7$	$n = 8$
729	2187	6561

Table 10: Number of nodes expanded for Uniform Cost with varying n.

3.2.2 A*

For A* in Towers of Hanoi, we had two theoretical heuristic functions, however, both produced the exact same amount of nodes and are therefore functionally identical heuristics.

The number of nodes expanded for A* for varying n can be found in Table 11.

$n = 6$	$n = 7$	$n = 8$
652	2006	6193

Table 11: Number of nodes expanded for A* with varying n.

4 Discussion

4.1 Agent Type Differences

There are two broad categories of agents we created for this assignment, informed and uninformed agents. Our uninformed agents include no information about the overall world, these include our reflex, model, and hill climbing agents. Our informed agents can use information about the over all world, these are our search agents Uniform Cost and A*.

The issue with the uninformed agents is that there is no way to guarantee that we find an optimal path, or sometimes, even a path at all. We see that we often get a trade off of space/time complexity and optimally. Even though we always get a optimal path when one exist with Uniform Cost and A*, we have to search all possible paths which take time and space in addition to needing a complete knowledge of the world.

We believe all of these agents can scale up into a larger grid and get the similar results, especially without more obstacles. As long as there is sufficient steps allowed. However, if we add more obstacles, we can probably get even more failures with our uninformed agents, even though our informed agents would be fine.

Based on this evidence, it is pretty clear that when ever possible, it is simpler, easier, and guarantees better results when we use informed search agents over uninformed agents. As long as we have information on the world space, we recommend using it.

4.2 Robot World

4.2.1 Uninformed Agents

Between our reflex agent and Model agent, it appears that the reflex agent performs better in worlds without false corners, which makes sense. The reflex agent is built to act as if there is no obstacles while the model agent searches around the corner to make sure it is an actual corner. That search is not perfect, but it is enough to out perform the reflex agent in the world with false corners with a 10% decrease in failures.

We were quite surprised by our Hill agent though, it seems to do the same as the reflex agent in a world without obstacles. However, there is no built in corner detection in the hill agent as our hill climbing heuristic does not account for obstacles. Therefore, the hill climber gets stuck on any square where the next best state is an obstacle, that is why we see such high failure rates with a 45% failure rate in world 2 and a 80% failure rate in world 3.

4.2.2 Informed Agents

When it comes down to it, A* is the same as Uniform Cost when you set the heuristic cost to 0. Therefore, A* is trying to get an optimal path like Uniform Cost, but expand less nodes by adding a underestimating heuristic cost to each node. We tried two different heuristic functions (Manhattan distance and Euclidean distance) and found success in both.

We see that our two heuristics performed about the same compared to each other. We see a small advantage when we use Manhattan distance. This is to be expected as Manhattan distance is a better estimator of the number of moves to the goal as we cannot move diagonally like Euclidean calculates.

When we compare A* to Uniform Cost it is amazingly successful. We see about a 75% decrease in nodes between A* and Uniform Cost.

4.3 Towers of Hanoi

We only implemented our Informed agents for the Towers of Hanoi.

We can see that our amount of expanded nodes grows exponentially with respect to n which is to be expected. Our heuristic was pretty simplistic, however, was slightly successful on decreasing the amount of nodes expanded when compared to Uniform cost for each n. We can see the the advantage of using A* grows with increasing n. When $n = 6$ we see a 77 node decrease, when $n = 7$ we see a 181 node decrease, and when $n = 8$ we see a 368 node decrease.

5 Conclusion

In conclusion, we certainly learned the depth and complexity that is involved in the field of artificial intelligence. These problems we solved are quite elementary compared to the the AI that is behind self driving cars and Alpha GO.

This tip-of-the-iceberg assessment into the world of artificially intelligent agents cemented our lessons in Agent types, Informed versus Uninformed searches, and most importantly algorithm implementation.

6 Appendix

6.1 Reflex Agent Data

World 1	World 2	World 3
33	23	1000
46	12	21
24	37	1000
16	46	14
20	21	20
29	26	1000
17	42	1000
20	9	38
19	33	18
15	28	16
27	12	32
23	29	1000
16	36	999
25	14	1000
16	38	22
13	20	1000
35	44	1000
24	22	1000
16	19	32
35	15	28

Figure 1: Figure shows all data for each of the 20 runs in each world for the reflex agent.

6.2 Model Agent Data

World 1	World 2	World 3
105	115	58
96	111	51
98	95	1000
91	116	116
113	107	49
111	83	1000
115	76	1000
90	97	1000
97	103	47
97	90	32
116	84	1000
101	104	95
94	112	102
100	99	38
101	121	22
114	109	126
103	105	157
100	111	100
109	107	1000
86	136	1000

Figure 2: Figure shows all data for each of the 20 runs in each world for the Model agent.

6.3 Hill Climbing Agent Data

World 1	World 2	World 3
20	23	1000
33	28	1000
40	1000	1000
30	24	1000
42	1000	1000
30	25	1000
23	1000	1000
10	1000	7
21	1000	11
36	4	1000
31	10	1000
21	1000	1000
24	1000	1000
27	21	17
18	24	1000
6	2	1000
6	1000	1000
43	29	17
28	21	1000
34	1000	1000

Figure 3: Figure shows all data for each of the 20 runs in each world for the hill climbing agent.

6.4 Uniform Cost Agent Data

World 1	World 2	World 3
7	256	917
1091	1073	970
1189	588	68
961	1073	818
412	858	1020
349	1025	1020
1117	974	366
301	280	1017
66	1073	1016
1201	1073	1009
895	635	695
895	1073	1016
808	1037	1011
1201	791	1024
1019	757	1018
691	598	1024
1114	1073	1016
894	1023	1022
1201	689	592
218	1073	202

Figure 4: Figure shows all data for each of the 20 runs in each world for the uniform cost agent.

6.5 A*1 Agent Data

World 1	World 2	World 3
41	106	131
137	521	367
23	138	149
625	569	486
11	181	41
76	211	414
233	149	119
431	19	52
39	316	268
459	331	78
159	86	42
51	300	82
24	125	384
107	140	410
149	472	145
46	22	115
379	96	603
100	129	344
24	148	123
27	45	90

Figure 5: Figure shows all data for each of the 20 runs in each world for the A*1 agent.

6.6 A*2 Agent Data

World 1	World 2	World 3
46	163	413
68	521	499
172	325	186
32	293	104
123	5	531
439	243	120
543	251	76
70	31	174
358	52	113
580	199	340
68	209	165
367	463	170
153	51	65
85	104	186
14	218	337
153	319	94
336	19	310
137	116	215
126	267	470
510	110	461

Figure 6: Figure shows all data for each of the 20 runs in each world for the A*2 agent.