COMP3702

Assignment 1

Group 666
Linfeng Liu 44563376
Yunwen Wu 44816371
Fan Yang 44594213

1. Define the agent design problem

An agent refers to a computer program that'll act autonomously to achieve the desired goal by gathering and deducing the information based on the environment. The problem when there's a need to create a specific agent to achieve particular output is what we called agent design problem. The agent should be able to learn from the provided data and act rationally to achieve the goal.

In this assignment, an agent which can play game Sokoban and autonomously find the solution (optimal path) to the problem is required. The agent is fully observable and there's no hidden information. Hence the percept space is the same as state space (P = S) and the perception function Z will match each state to itself. So, we only need to define A, P, T, U.

1.1 Action space(A):

The player can only be moved to left, right, up or down.

1.2 Percept space(P):

A character array where free space is represented by '', wall '#', box 'B', target 'T', player 'P', boxer on top of target 'b', player on top of target 'p'.

1.3 Transition Function (T: S x A → S')

We can visualise the transition function as a lookup table with a row for each (state, action) pair and a row for next state (the stage after performing the action). The game is deterministic so the result of performing the action will always lead to one state.

1.4 Utility function (U: S → R)

We consider giving equal cost to every movement which means no matter a box is moved or not it's always rewarded the same.

2. What type of agent is it?

2.1 Discrete/ Continuous

It is a discrete agent in this game since the agent can only go four directions step by step and the cost will be in integer.

2.2 Deterministic / Non-deterministic

After performing an action (moves the player) from a state, the next stage is determined and agent knows exactly what stage it'll be in. Hence, it's deterministic.

2.3 Fully observable/ Partially

The percept function in this case is one-to-one mapping so the agent always knows exactly which stage it'll be in. Hence, it's fully observable.

2.4 Static/ dynamic

If a move is not performed (when agent is calculating for the next action), the stage will always stay the same so we can say it's static.

3. Heuristic definition

The heuristic function in our algorithm consists three parts: the cost that agent have already taken, the heuristic to the goal and the closest distance between player and boxes. Here is how the heurist function is defined.

The heuristic function is shown below:

$$\mathrm{h(n)} = \sum_i D_i$$

where D_i is the distance between one box-target pair.

h(n) is calculated for all permutations of box-target pair, where the permutation is given by:

$$permutation = \binom{\mathrm{Target\_position[\,]}}{\mathrm{len(Box\_pos}ition)}$$

then H(n) can be calculated as:

$$\mathrm{H(n)} = \min\,(h_1,\ h_2, h_3\ \ldots)$$

Once the minimum H(n) is obtained, lowest estimated cost can be calculated as:

$$\mathrm{f(n)} = H(n) + g(n)$$

where g(n) is the cost from root to node n.
In our solution, the player moving steps from initial position to the closest box position p(n) is considered as part of g(n), p(n) can be calculated as:

$$\mathrm{p(n)} = \min(d_1, d_2, d_3\ \ldots)$$

where d_n is the distance between player and targets.

It is a fairly good heuristic function since it can consider all distance combinations of targets and boxes and output the lowest combination. On the other hand, it also finds the closest box to the player so that the program knows to go which box first efficiently. However, if some of boxes have already been pushed on the targets, this heuristic function would still calculate the distance between player and all boxes as consideration. This could generate more nodes and affect running time.

The heuristic function is admissible. Since we are getting the lowest cost between targets and boxes from all combinations, the actual cost will be greater than the heuristic.

4. Comparison of Uniform cost search and A* search

This section is aimed to compare Uniform cost search and A* search in terms of the number of nodes generated, the number of nodes on fringe, the number of visited nodes and the run time of the algorithm. at Details will be listed in the tables.

4.1 Generated Node

The key feature of search algorithm is to expand existing state to reach other potential states. In terms of efficiency, the size of generated node is closely related to the running speed and a better algorithm should be able to find a solution with least nodes generated. By adding one to a counter each time a new node is drawn from queue, the total number of expended nodes is recorded and used for comparison.

Table 4.1 Generated node size

	Uniform Cost Search		A* Search	
Test Case	Node Number	Optimal Solution?	Node Number	Optimal Solution?
1box_m1	70	Yes	34	Yes
1box_m2	174	Yes	136	Yes
1box_m3	400	Yes	259	Yes
2box_m1	43	Yes	39	Yes
2box_m2	14093	Yes	11903	Yes
2box_m3	27299	Yes	22558	Yes
3box_m1	1660	Yes	1106	Yes
3box_m2	356198	Yes	175725	Yes

The table shows that A* algorithm always generates less nodes than UCS while both outputting optimal solution. By analysing the property of each given test case, it is found that A* has more outstanding performance when the map is more spacious. This is because heuristic can 'push' player toward box, rather than trying to move away from box.

4.2 Fringe Node

Fringe, also known as frontier, is a queue containing nodes to be expended. A smaller fringe size means a smaller branch width, displaying the algorithm's ability to solve problem when it takes a lot of steps to reach an optimal solution. To find the fringe size, program prints the length to terminal when the algorithm finds an optimal solution.

Table 4.2 Fringe node size

	Uniform Cost Search		A* Search	
Test Case	Fringe size	Optimal Solution?	Fringe size	Optimal Solution?
1box_m1	22	Yes	36	Yes
1box_m2	24	Yes	27	Yes
1box_m3	14	Yes	30	Yes
2box_m1	0	Yes	3	Yes
2box_m2	150	Yes	399	Yes
2box_m3	253	Yes	671	Yes
3box_m1	78	Yes	141	Yes
3box_m2	16752	Yes	18162	Yes

Upon fringe node, A* has worse performance in all cases. This is because A* algorithm sometimes expends node which is at wrong path, then leave nodes there realizing the path is not optimal. This may cause problem when the case is extremely complicated.

4.3 Explored node

Like the size of expended node, explored nodes shows the size of game states which has been reached and recognized by the program. The number is printed through the length of all states program has explored.

Table 4.3 Explored node size

	Uniform Cost Search		A* Search	
Test Case	Explored node size	Optimal Solution?	Explored node size	Optimal Solution?
1box_m1	98	Yes	76	Yes
1box_m2	204	Yes	168	Yes
1box_m3	421	Yes	296	Yes
2box_m1	44	Yes	43	Yes
2box_m2	14624	Yes	12681	Yes
2box_m3	27967	Yes	23627	Yes
3box_m1	1788	Yes	1286	Yes
3box_m2	383853	Yes	200052	Yes

The trend of data is similar to what has been found in 4.1. A* does much better in most cases but it doesn't see significant improvement in case 2box_m1, because there are many narrow paths.

4.4 Run time

Run time is the most straightforward and convincing indicator showing the efficiency of algorithm. A shorter run time is what the whole team working on. When program starts and ends, time is recorded, which will then be used to find the total run time.
However, the fastest algorithm may not be the best, considering that some algorithm may generates more nodes and thus takes more memory. A comprehensive conclusion will take both node size and run time into consideration.

Table 4.4 Run time

	Uniform Cost Search		A* Search	
Test Case	Run time	Optimal Solution?	Run time	Optimal Solution?
1box_m1	0.032	Yes	0.030	Yes
1box_m2	0.074	Yes	0.068	Yes
1box_m3	0.174	Yes	0.121	Yes
2box_m1	0.032	Yes	0.022	Yes
2box_m2	7.514	Yes	8.194	Yes
2box_m3	17.822	Yes	19.412	Yes
3box_m1	1.277	Yes	0.770	Yes
3box_m2	308	Yes	683	Yes

A* runs faster when the case is relatively simple, but when it comes to problem with larger map, more boxes and more steps to reach the solution, UCS works it out in shorter period. This is confusing because A* always generates less node than UCS.
We think A* itself slower the process because it takes too much time to find 'which node to extract next' in the fringe, especially when the fringe size is extremely large. (over 18000 nodes on 3box_m2, see table 4.2)

4.5 Conclusion

By putting all parameters into consideration, the team finds that A* algorithm always has better performance when the problem case is simple and solution can be reached within less than 50 steps. When the test case is complicated, A* can still output optimal solution with less nodes explored and less memory used, but runs much slower. Notice that this conclusion is based on the A* algorithm designed by the team, which is still in development. The performance may be improved if a better heuristic logic is found

5. Deadlock check

For game Sokoban, there are several common deadlocks which can be systematically summed up. Below is the description on the most frequently seen deadlocks and how to implement them in the code.

For game Sokoban, there are several common situations which will lead to the stage unsolvable. Below are the deadlock stages we consider in our deadlock check function.

1. When a box is in the corner (with two directions facing the wall) and it's not on a target yet, there's no possible way to find a solution
2. If a box's position is found near the wall, consider below arguments to check whether is deadlock or not:

- Find the If n targets are detected and m boxes are detected on that row but m is larger than n, then no solution could be found.
- If the number of targets is larger or equal to the number of boxes and not all the boxes are on targets, check whether two boxes are near each other. It'll lead to deadlock if two boxes are near each other's.

To implement the function in code, algorithm is shown as below.

```
For BOX in ALL_BOXS {
  If BOX in the corner & not on Target {Return deadlock = True}

  If BOX near the WALL {
      total target on that row=n
      total boxes on that row=m
    If n<m {Return deadlock = True}
    If n>m and not all the boxes on targets {
        If two boxes are detected near each other {Return deadlock = True }
    }
  }
```

Below table shows the difference in expanded node, visited node and run time whether the deadlock check function is implemented or not:

For box1_m1

	With deadlock checker	Without
Expanded node	70	89
Explored node	98	133
Run time (s)	0.04	0.06

For box1_m2

	With deadlock checker	Without
Expanded node	174	307
Explored node	204	343
Run time (s)	0.19	0.3

For box3_m1

	With deadlock checker	Without
Expanded node	2612	5292
Explored node	2847	5839
Run time (s)	3.18	5.43

We can see the significant improve in the efficiency of the code. Not only the run time is reduced, but also the node explored is lesser.