COMP3702 Artificial Intelligence (Semester 2, 2022)
Assignment 1: Search in HEXBOT – **Report Template**

Name: Eskil Pedersen

Student ID: 47613722

Student email: e.pedersen@uqconnect.edu.au

Note: Please edit the name, student ID number and student email to reflect your identity and **do not modify the design or the layout in the assignment template**, including changing the paging.

Question 1 (Complete your full answer to Question 1 on the remainder page 1)

Modularity: Flat. The system cannot be divided into interacting modules that can be understood separately, there is no organizational structure and therefore the modularity is flat. There is only one level of abstraction.

Representation: State. The description of the world, both the agent and the environment with the targets, widgets, and walls has two individual states. The robot and widgets have positions and orientations which are individual, while the walls and targets only have one position.

Planning horizon: Indefinite. The agent looks ahead a finite number of steps, to the goal state. However, this is not a predetermined number of steps, because the agent doesn't know how many actions are required to get to the goal state.

Sensing uncertainty: Fully observable. The agent has access to the complete state of the environment at each point in time, and therefore it is fully observable.

Effect uncertainty: Deterministic. The agent knows with certainty what the next state will be after an action is performed from a prior state.

Preference: Achievement Goals. The task of the agent is to solve the task, which will be the goal for the agent.

Number of agents: Single agent. Because there is only one agent in the environment at the time trying to solve the task.

Learning: Knowledge is given. The designer of the agent gives the agent knowledge, and the agent doesn't learn from data or past experiences.

Computational limits: Perfect rationally. The agent finds the best thing to do at the current state, without considering its computational resources.

Interaction: Offline. The agent is not in a dynamic environment, and there is no penalty for computing too long or interleaving between actions and computation, which is usual for online interaction. Therefore, it is offline interaction, because it reasons what to do before it acts in the environment.

Question 2 (Complete your full answer to Question 2 on page 2)

The game board is a hexagonal grid, where the robot occupies one of the cells in the hex grid and has a direction on this cell. The robot has four different available actions; hence the action space of the robot consists of the four different movements: A = {forwards, backward, spin left, and spin right}.

The state space is all possible configurations of how the environment. There are obstacles in the hex grid, which neither the robot nor the widgets can go into. The border of the hex grid works the same way as the obstacles. The robot can move to different cells and change its orientation. The widgets can also be moved into another position and orientation. However, the robot cannot be in the same cell as the widgets, and the widgets cannot be in the same cells as other widgets.

The transition function takes in a state and an action of the robot, and the output is a new state. The action can move or rotate the robot. The robot can also rotate or move a widget. However, the action and state inputted to the transition function can only output states that are in the state space. See description of state space over.

The utility function takes in a state and outputs a value that indicates how desirable it is for the agent to occupy that state. In the HexBot the output is either solved (value 1) or not solved (value 0). The utility function output is 1 if all target cells are covered by a widget cell, and the state is valid.

Question 3 (Complete your full answer to Question 3 on page 3)

(15 marks)

Compare the performance of Uniform Cost Search and A* search in terms of the following statistics:

a) The number of nodes generated

For the five test cases, the nodes generated for UCS and A* were 1: 1 633 and 1 568, 2: 25 609 and 23 393, 3: 957 362 and 956 208, 4: 1 908 676 and 1 885 629, 5: 6 229 781 and 5 900 695

b) The number of nodes on the frontier container when the search terminates

Test case 1: UCS has 34 and A* 93 in the frontier container. Test case 2: UCS has 466 and A* 693 in the frontier container. Test case 3: UCS has 5 170 and A* 7108 in the frontier container. Test case 4: UCS has 77 969 and A* 104 139 in the frontier container. Test case 5: UCS has 50 103 and A* has 64 648 in the frontier container.

c) The number of nodes on the explored list (if there is one) when the search terminates

Test case 1: UCS has 1599 and A* has 1475 in the explored list. Test case 2: UCS has 25143 and A* has 22700 in the explored list. Test case 3: UCS has 952192 and A* 949100 has in the explored list. Test case 4: UCS has 1 830 707 and A* has 1 781 490 in the explored list. Test case 5: UCS has 6 179 678 and A* has 5 836 047 in the explored list

d) The run time of the algorithm (e.g. in units such as mins: secs). Note that you can report run-times from your own machine, not the Gradescope servers.

(These are run times on my computer) Test case 1: UCS has 00:00.05 and A* has 00:00.05 in run time. Test case 2: UCS has 00:00.77 and A* has 00:00.89 in run time. Test case 3: UCS has 00:46.49 and A* has 01:02.29 in run time. Test case 4: UCS has 01:26.81 and A* has 01:47.30 in run time. Test case 5: UCS has 05:29.98 and A* has 06:26.62 in run time.

e) Discuss and interpret these results. If you are unable to implement A* search, please report and discuss the statistics above for UCS only, and what you would expect to change for A*.

A* generated 4 %, 8 %, 0.1 %, 1 % and 5 % fewer nodes than UCS. The number of nodes generated was between 8 % and 0.1 % lower for A* than UCS. This was expected because the heuristic function makes it preferable to generate new nodes closer to the goal node. We want the generated nodes to be as few as possible, and with this heuristic, it was partly lower.

UCS had 63 %, 33 %, 27 %, 25 % and 22 % fewer nodes in the frontier container than A*. In all the test cases, the number of nodes in the frontier was lower for UCS than for A*, which is expected when a good heuristic is implemented. This shows that the heuristic manages to add costs that make it preferable for the agent to choose new nodes to explore that are closer to the goal node, and in turn, the number of nodes in the frontier will be higher when the agent reaches the goal node.

A* explored 7.7 %, 9.7 %, 0.3 %, 2.6 % and 5.5 % fewer nodes than UCS. The number of explored nodes should be lower in A* than UCS when a good heuristic is implemented, which is true for all test cases in c). We want the agent to explore as few wrong nodes as possible. This heuristic shows that the percent of nodes explored ranges from 9.7 % to 0.3 %.

UCS ran 0 %, 13 %, 25 %, 20 % and 15 % faster than A*. This is not favorable, because A* usually is considered a better algorithm. When the number of nodes expanded, the number of nodes in the frontier, and the number of nodes expanded are lower for A* than UCS, the run time should also be lower for A*. The slower run time can be explained by a slow heuristic. The heuristic function runs each time the agent checks out a new node, and there if it should be computationally inexpensive. If not, the run time will be affected.

Question 4 (Complete your full answer to Question 4 on pages 4 and 5, and keep page 5 blank if you do not need it)

"Some challenging aspects of designing a HexBot agent are the hexagonal grid, the asymmetric cost of actions (pushing is more expensive than pulling a widget), rotation of widgets to avoid obstacles, choosing the order in which to manoeuvre each widget, and determining which target squares to cover.

Describe heuristics (or components of a combined heuristic function) that you have developed in the HexBot search task that account for these aspects or any other challenging aspects you have identified of the problem. Your documentation should provide a thorough explanation of the rationale for using your chosen heuristics considering factors such as **admissibility** and **computational complexity** (maximum of 5 marks per heuristic)."

Heuristic 1

The first heuristic looks at how many of the target cells are filled with a widget cell. For each target cell that is not filled with a widget cell, the cost is added by 1/3. The reason why 1/3 is chosen, is because this is the minimal cost of moving a widget cell into a target. Theoretically, this is possible if WIDGET4 has its center in the correct position, and the robot rotates the widget into the correct position. That movement has a cost of 1 and moves three widgets into position. However, this doesn't imply that the heuristic function never overestimates. The heuristic function favor widgets in the target, but doesn't consider if it is the correct target cells. Which can lead the robot to push or pull a widget into a cell along the way to the actual spot for this widget. Therefore, this heuristic is not admissible. The two for-loops are relative fast, for small and medium size hex grids. The heuristic has a runtime of O(n)+O(m), where n is the number of widget cells, and m is the number of target cells. Since the number of widgets always is larger or equal to the number of target cells, the computational complexity can be written as O(n).

```
def heuristic1(self, state):
        # how many target cells are filled with the widgets
        environment = self.environment
        widget_cells = set([widget_get_occupied_cells(environment.widget_types[i],
state.widget_centres[i], state.widget_orients[i]) for i in range(environment.n_widgets)][0])
        # loop over each target
        tgt_unsolved = 0
        for tgt in environment.target_list:
            if tgt not in widget_cells:
                tgt_unsolved += 1/3
        return tgt_unsolved
```

Heuristic 2

The second heuristic can be seen under. This heuristic makes the robot moves towards target cells that are not covered by widgets. The computational complexity is pretty similar to heuristic 1, but this also enumerates the target cells that are not covered by widgets. The runtime is therefore still O(n), and n is the number of widgets. The heuristic is not admissible. The robot can move back towards a widget that already is in the correct position, and therefore overestimates the distance. The function self.compute_euclidean_distance takes in two cells and computes the distance between the two cells. However, this doesn't look at the robot's orientation.

```
def heuristic2(self, state):
        # total distance from robot to target cells that not is covered by widget
        environment = self.environment
        widget_cells = set([widget_get_occupied_cells(environment.widget_types[i],
state.widget_centres[i], state.widget_orients[i]) for i in range(environment.n_widgets)][0])
        tgt_cells_not_solved_cells = set()
        distance = 0
        for tgt in environment.target_list:  # loops the targets-cells
```

```
            if tgt not in widget_cells:
                # adds the target cell if no widget is in it
                tgt_cells_not_solved_cells.add(tgt)
        for cell in tgt_cells_not_solved_cells:  # loops target-cells not solved
            distance += self.compute_manhattan_distance(
                cell, state.robot_posit) # adds euclidean distance between robot and that cell
        return distance
```

Heuristic 3

This heuristic enumerates all widget centers and finds the Manhattan distance to the closest target. Since Manhattan can overestimate the distance, this heuristic is not admissible. Regarding computational complexity, this heuristic has the highest. The function first enumerates all center widgets, then enumerates all target cells. The Manhattan distance is multiplied by 1.5 because this is the cost of pulling a widget one tile. This heuristic doesn't look at the rotation of the widget, obstacles, or in which order to move the different widgets. Computational complexity is O(n*m), where n is the number of center targets and, m is the number of target cells.

```
def heuristic3(self, state):
        environment = self.environment
        distance = 0
        for center in state.widget_centres:  # loops every widget center
            closest = 9999  # defines a max distance
            for tgt in environment.target_list:  # loops evert target
                curr_dis = self.compute_manhattan_distance(center, tgt) * 1.5
                if curr_dis < closest:
                    closest = curr_dis  # updates closest target
            distance += closest  # adds the closest target-path
        return distance
```