COMP3702 Artificial Intelligence (Semester 2, 2021)
Assignment 1: Search in DRAGONGAME – **Report Template**

Name: Dicky Sentosa Gozali

Student ID: 43976746

Student email: d.gozali@uq.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)

The dimensions of complexity of DragonGame are:

Dimensions	**Values**
Modularity	Hierarchical
Planning Horizon	Indefinite stage
Representation	States
Computational Limits	Bounded rationality
Learning	Knowledge is given
Sensing uncertainty	Fully observable
Effect uncertainty	Deterministic
Preference	Achievable goal
Number of agents	Single agent
Interaction	Offline

- Hierarchical: as they are composed of multiple levels where every level is more complex than the next.
- Indefinite stage: The DragonGame does not provide and predetermined knowledge of the prior actions and it does not consider the next best course of action given the cost of the action as long as they do not exceed the threshold point of the total cost and time limit.
- States: As the DragonGame is composed of various predetermined moveset, each movement/ actions could be identified as a new state.
- Bounded rationality: the precomputation of heuristics take account for the action cost for each branch of movement/ action, hence, it will keep track and eliminate those course of action that would have exceed the threshold point.
- Knowledge is given: as there are threshold and limitation involved in the computation, the knowledge can be considered given to uphold to these thresholds.
- Fully observable: In the DragonGame, the agent is very aware of the state it is in after taking the movement/ action from the successor computation function.
- Deterministic: each update of a state is determine by the action it take to validate if the action is to be expected.
- Achievable goal: The goal is to collect all the available gems and get to the exit tile. This shows that the effect uncertainty is determine by the state action to collect the closest gems then once all gems has been collected, get to the exit tile.
- Single agent: the agent involves in DragonGame is a single player. The gems and exit tile are part of the environment.
- Offline: agent reasoning is determined by the course of action it takes, the cost of the selected action and eliminate course of actions that have exceed the predetermined cost.

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

The components of my agent design for DragonGame are:

As aforementioned in Q1, the environment is fully observable, hence, the percept space is the same as the state space and percept function is simply maps each state to itself.

- Action space: all valid movement set/ action set taken by the player

$$A = \{WR,\ WL,\ J,\ GL1,\ GL2,\ GL3,\ GR1,\ GR2,\ GR3,\ D1,\ D2,\ D3\}$$

- State space: all the possible actions executed given the player (agent) current position

$$S = \{a: a\ is\ an\ valid\ action \times \{location\ of\ the\ player\ surrounding\} \times \{no.\ of\ moves\ taken \leq \max moves\}\}$$

- World dynamics/ Transition function: The $perform\_action$ function is conducted to check the validity of taking an action given the player position. It will then be stored in dictionary for the visited gamestate with the path_cost. As the DragonGame is deterministic, the result of perform_action would lead to multiple possible new states.

$$T: S\ x\ A = S'$$

State (S)	Action (A)	New_state (S')
(1, 3, tuple((4,2))	WR (1)	(1,4, tuple(4,2))
(1,4, tuple(4,2))	J (2)	(2,4, tuple(4,2))

- Utility function: Since the goal is to collect all the gem and then go to the exit tile, the agent has to collect all the presence gem in the given level. For the UCS, the search algorithm finds the shortest distance to the closest gem and travel its way to collect the next shortest distance to the closest gem until all the gem has been collected (x-, y- coordinates of gems position visited (self.gem_positions)).

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

Level 1

Comparison Description	**UCS**	**A_star**
The number of nodes generated	**24**	**24**
The number of nodes on fringe when search terminates	**4**	**6**
The number of nodes on explored list (if there is one) when the search terminates	**20**	**18**
The runtime of the algorithm	**0.019s**	**0.01434s**

Level 3

Comparison Description	**UCS**	**A_star**
The number of nodes generated	**4369**	**4383**
The number of nodes on fringe when search terminates	**21**	**47**
The number of nodes on explored list (if there is one) when the search terminates	**4588**	**10129**
The runtime of the algorithm	**1.0876s**	**2.494s**

Level 5

Comparison Description	**UCS**	**A_star**
The number of nodes generated	**92734**	**92784**
The number of nodes on fringe when search terminates	**38**	**502**
The number of nodes on explored list (if there is one) when the search terminates	**96499**	**602877**
The runtime of the algorithm	**20.886s**	**200.698s**

e) Discuss and interpret these results.

With UCS, the quantity of nodes explored is determined by the minimal cost there is and the branching factors. Whilst, in A_star, it combines both the minimal cost to travel in the certain path p with $f(p) = g(p) + h(p)$. The heuristic value has contributing factor in determining the path of the node travel and the path cost. Hence, the number of nodes explored to find the minimum $path\_cost$ in a_star is much greater in a_star compared to the UCS by almost 6 times more. In a_star algorithm, it discovers all its neighbours and tends to find the closest gems to the exit whilst, the UCS tends to find the closest to the player position.

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)

The three heuristics that would best used in this DragonGame are Euclidean distance, Manhattan distance and Neural Network Heuristics.

1. Euclidean Distance Heuristic

This is calculated by the simple straight forward equation of Euclidean distance between two coordinates.

$$distance(src, dest) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

```
def manhattan_dist_heuristic(src, dest):
    # Euclidean distance
    return math.sqrt(math.pow((dest[1] - src[1]), 2) + math.pow((dest[0]-src[0]), 2))
```

Figure 1 - Euclidean Distance Application

For Level 3, comparison between Euclidean vs. Modified Manhattan Distance does not seem to show significant difference. Using Euclidean distance, the number of visited nodes 92784, expanded_nodes 602930 and nodes in container is 504, which is compared to Q3 (Level 5), less than Manhattan Distance for expanded nodes and nodes in container. However, the difference may not be as significant in terms of the nodes and actions chosen, however, by comparing the duration/ time of calculation as Manhattan seems to perform much faster than Euclidean distance (223.814s).

```
A*:
Visited Nodes: 92784,          Expanded Nodes: 602930,        Nodes in Container: 504
Cost of Path (with Costly Moves): 281.9999999999997
Num Actions: 198,
Actions: ['wr', 'wr', 'j', 'j', 'gr3', 'gr3', 'j', 'gr3', 'j', 'wr', 'gr3', 'wr', 'j', 'gr3', 'j', 'gr3', 'wr', 'wr', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'wl',
 'wl', 'd2', 'd1', 'd3', 'd3', 'wr', 'wr', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'wl', 'wl', 'gl2', 'gl3', 'd2', 'd1', 'd1', 'wr', 'wr', 'j', 'd1', 'wl', 'wl', '
wl', 'd2', 'gr3', 'j', 'gr3', 'wr', 'wr', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'wl', 'wl', 'gl3', 'gl3', 'j', 'gl3', 'wl', 'wl', 'j', 'gl3', 'j', 'gl2', 'j', 'g
l3', 'wl', 'wl', 'j', 'gl3', 'd3', 'gr1', 'wl', 'wl', 'j', 'j', 'j', 'j', 'j', 'wr', 'gr3', 'j', 'gr3', 'd2', 'gl2', 'd2', 'gr3', 'gr3', 'j', 'gr3', 'j', 'wr', 'gr3', '
wr', 'j', 'gr3', 'j', 'gr3', 'wr', 'wr', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'wl', 'wl', 'gl3', 'gl3', 'j', 'gl3', 'j', 'gl3', 'd2', 'gr2', 'd3', 'gr3', 'wr',
'j', 'gr3', 'j', 'gr3', 'wr', 'wr', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'wl', 'wl', 'gl3', 'gl3', 'j', 'gl3', 'wl', 'wl', 'j', 'gl3', 'wl', 'd2', 'd2', 'gr1',
'gr3', 'wr', 'gr3', 'wr', 'j', 'gr3', 'j', 'gr3', 'wr', 'wr', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'j', 'wl', 'wl', 'gl3', 'gl3', 'j', 'gl3', 'wl']
Time: 223.81413555145264
```

Figure 2 -Euclidean Distance Heuristic Application for L5.txt

2. Manhattan Distance Heuristic

Initially, the design of the program was to simply use the minimal distance between the player to the closest gem without consideration of the exit tile after collecting all the gems. As a result, this causes a much slower effect due to last gem collected may not be closer to the exit tile compare to other time when the program is being run. This causes the runtime to run much longer time and the action cost tend to reach beyond the threshold. However, there are times when the action cost path reaches optimal solution path, however, these paths are not ideal as the last heuristic distance of the last gem collected to the exit seem to stretch along the grid map.

```
    def get_heuristic(self):
        player_pos = (self.game_env.init_row, self.game_env.init_col)
        dist1 = manhattan_dist_heuristic(player_pos, self.exit_tile)

        player_gem_min_dist = max(self.game_env.n_rows, self.game_env.n_cols)
        for tile in self.game_env.gem_positions:
            d = manhattan_dist_heuristic(player_pos, tile)
            if player_gem_min_dist < d:
                player_gem_min_dist = d
        dist2 = player_gem_min_dist + self.gem_dist_min

        player_ladder_min_dist = max(self.game_env.n_rows, self.game_env.n_cols)
        for tile in self.ladder_tiles:
            d = manhattan_dist_heuristic(player_pos, tile)
            if player_ladder_min_dist < d:
                player_ladder_min_dist = d
        dist3 = player_ladder_min_dist + self.ladder_dist_min

        return min((dist1, dist2, dist3))
```

Figure 3 - Minimal Heuristic Distance Computation

3. Modified Manhattan Distance Heuristic (inc. storing gem minimal distance to exit)

The heuristic that I have decided to choose is the Manhattan Distance Heuristic. The Manhattan distance helps in finding the shortest path between source tile to the destination tile with minimal cost path.

```
def heuristic_precomputation(self):
    self.ladder_tiles = []
    self.exit_tile = (self.game_env.exit_row, self.game_env.exit_col)
    for r in range(self.game_env.n_rows):
        for c in range(self.game_env.n_cols):
            if self.game_env.grid_data[r][c] == GameEnv.LADDER_TILE:
                self.ladder_tiles.append((r,c))

    gem_dist = max(self.game_env.n_rows, self.game_env.n_cols)
    for gem in self.game_env.gem_positions:
        dist = manhattan_dist_heuristic(gem, self.exit_tile)
        if dist < gem_dist:
            gem_dist = dist
    self.gem_dist_min = gem_dist

    ladder_dist = max(self.game_env.n_rows, self.game_env.n_cols)
    for ladder in self.ladder_tiles:
        dist = manhattan_dist_heuristic(ladder, self.exit_tile)
        if dist < ladder_dist:
            ladder_dist = dist

    self.ladder_dist_min = ladder_dist

    self.data = (self.ladder_tiles, self.exit_tile, self.gem_dist_min, self.ladder_dist_min)
```

Figure 4 - Heuristic Precomputation storing values in the constructor of StateNode

The StateNode class stores the tuple of gems positions coordinates, the coordinates of the exit tile, list of the ladder tiles, the minimal distance between the ladder to the exit tile, and the closest distance of the gem to the exit tile. This way, the visiting nodes will be to expand its nodes horizontally towards all the gems and then the exit tile once all the gems has been collected. Using the Manhattan distance, it will prioritize those gems that are closest to the player, then it will go through the gems that are gems closest to the goal (exit) tile. The Manhattan Distance works the best for collecting gems away from the exit as it minimizes the overall path cost the player to the exit. Based on the action cost, it always finds the gem closest to the exit. It does not seem to go to the exit node first before getting all the gems.

The heuristic calculation in my code however, emphasizes the shortest heuristic distance by computing the minimum cost between the player distance to the exit tile, the distance between the player to the closest gem relative to the player with an additional with the minimal distance between exit tile with the closest gem relative to the exit tile and the distance between the player to the ladder tile with an additional of the minimum distance between the exit tile and the ladder tile. This way, it will choose the minimal action cost for player to take in order to minimal the path_cost.

However, from implementing the Manhattan distance, it seems that it is not the most ideal heuristics due to its shortcomings of generating non-optimal successors when the exit is closer to the player whilst the gem is distant away from you relative to the exit. As aforementioned in the question, one of the challenging aspects of this DragonGame agent are the asymmetric movement dynamics. The code does not seem to take account for such action, it does find the minimal cost based on the threshold value provided. It will keep updating a new self.data in which contain the new set of data position relative to the player coordinates. Another limitation and this could be the most major drawback of the current design is its time-consuming computation. It seems that it does not seem to meet the cutoff time limit provided testcases. Therefore, it would be much better to run the computation in parallel.