Knowledge Representation

Assignment 1 – Agent-Based Representation of Knowledge

Michael McAleer (R00143621)

Part 1 & 2– Getting Started with Agent-Based Representations & Extending the World

For this assignment, an environment has been created to simulate the 'Wild West' with the agent being a representation of an 'Outlaw'. The aim of the outlaw is to first find the gold in the environment, and escape via a horse which is also found in the environment, all whilst avoiding the sheriff or snakes which can arrest or kill the agent. The game will be considered complete if:

- The agent finds the treasure and then finds the horse to escape on,
- The agent is bitten by a snake and dies (health reaches zero),
- The agent is locked up by the sheriff for having no gold to attempt a bribe, or if the bribe was unsuccessful because the agent wasn't lucky enough.

PEAS – Performance Measure

For performance measurement, statistical data includes the number of steps taken by the agent between starting and the game over scenario, the amount of gold found (if any), and the health of the agent which is measured between 0 and 100. Although the amount of gold found would normally dictate the overall performance of the agent within the environment, the number of steps taken by the agent is of greater interest in this context as the faster the agent finds the primary and secondary goals means a more successful run of the game.

Also returned as a performance measurement is the agent state at the time of the game ending, either successfully or unsuccessfully. These states include the various actions the agent may or may not have carried out during the run of the game:

- If the agent escaped the environment on a horse after finding the treasure,
- If the agent found one or more chests in the environment,
- If the agent found the dog in the environment,
- If the agent found the horse in the environment,
- If the agent got bitten by a snake in the environment,
- If the agent was saved by the dog,
- If the agent was successful in bribing the sheriff,
- If the agent was locked up by the sheriff.

There has been an element of random determination built in to the agent and environment, with the agent's luck determined on agent initialisation and luck/random choice determining if the agent is successful in bribing the sheriff, finding a snake or gold in a chest, or if the dog finds anti-venom when the agent is bitten by the snake. This higher the agent's luck, the more chance of success they will have in bribes and the dog finding anti-venom. Finding a snake or gold in the chest is a set probability, with 20% chance of finding a snake and 80% chance of finding gold. An element of random choice/probability was built in to the environment to represent a non-static chance of perceptions and actions, with various agent states or past actions determining the outcome of future states and actions.

PEAS – Environment

The environment is made up of a number of various 'things' which has a different impact on the agent performance or state. Random choice comes into effect here also as the location of each of these things within the environment is entirely random and determined when the environment is initialised. This

ensures that the likelihood of two runs of the agent in the environment are very slim. These things which the agent may encounter are:

- Treasure (Gold)
- Chest (Snake or Gold)
- Dog
- Horse
- Snake
- Sheriff

PEAS – Actuators

The agent actuators are driven by the environment state, that is, what the agent 'percepts' from their current location within the environment. Each of the perceptions of the agent are representative of the real-world actions of a similar use-case, for example, finding gold, or opening a chest. The aim of the actuators in the environment is to simulate, where possible, real-world scenarios and actions. When the agent encounters a 'thing' in the environment and acts on it, the 'thing' is then deleted/removed from the environment to prevent acting on it a second time in the same location.

The agent actuators, dependent on the related environment 'thing' are:

- Agent 'finds' treasure
- Agent 'escapes' on horse
- Agent 'opens' chest
- Agent 'rescues' dog
- Agent gets 'bitten' by snake
- Agent 'bribes' or gets 'arrested' by sheriff

As mentioned previously, the agent actions for these may be influenced by the agent's current state or probability determined by a luck value. The result of an action may be entirely random however, such as the amount of gold found, or the outcome of opening a chest (*note: a move is constituted as a move from one location to another, turning or changing direction is not classed as a move*).

- When the agent finds treasure, there is a random number selection between 1000 and 10000 which determines the amount of gold the agent will get.
- When the agent has found the treasure, their bag will rip and a hole will result in the agent losing 100 pieces of gold per move, the quicker the agent finds the horse the more gold they will escape with.
- When the agent finds the horse, if they have not yet found the treasure they need to keep searching.
- If the agent has already found the gold but has dropped it all by the time they find the horse, the escape scenario is still enacted on and the agent escapes with zero gold
- When the agent opens a chest, they have a 20% chance of finding a snake and an 80% chance of finding gold. If the agent finds a snake they get bitten by it, if they find gold it is added to the agent's gold attribute
- If the agent is bitten by a snake, either be encountering a snake on a random location within the environment or as a result of opening a chest, their lift attribute will decrement from 100 to 0 in

intervals of 10, with each interval of 10 decremented per agent move. When the agent's health reaches 0 the game is over

- If the agent has rescued the dog in the environment, if the agent encounters a snake in a random location or in a chest, the dog will kill the snake every time and save the agent from being bitten thus not affecting agent health
- If the agent has been bitten before finding the dog, for each subsequent move before the agent dies, the agent has a 50% chance of the dog going to find ant-venom, the success of the dog then finding anti-venom is then dependent on the agent's luck determined when the agent is initialised. If anti-venom is found, the agent will no longer lose heath.
- If the agent encounters the sheriff, the agent can attempt to bribe the sheriff if the agent currently has gold. The success of bribing the sheriff is dependent on the agent's luck, if the agent is successful the sheriff will take half of the agent's gold and disappear from the environment, else, the agent is not successful in the bribe and gets arrested and thrown in jail. This is considered a game-over scenario.
- If the agent does not have any gold with which to bribe the sheriff, then the agent will automatically be thrown in jail and the game-over scenario triggered.

PEAS - Sensors

In this basic implementation of a simple goal-based reflex agent, the agent sensors are meant to simulate that of a real-life person. Sensors would then include:

- Hands with which to act on perceptions
- Eyes with which to percept

The agent percepts what is in the current location and what is in all adjacent squares (top, bottom, left, and right).

Part 3 – Inferring Knowledge from an Agent-Based Representation

The two search-based approaches implemented in the environment for the agent to follow are 'breadth-first search' and 'depth-first search. The first part in implementing these search algorithms was to incorporate a method which for every cell in the environment, returned all adjacent cells (top, bottom, left, and right) of a given cell. With this 'grid-map' it would then be possible to determine the links between each node in the environment so the search algorithm could proceed correctly. This 'grip-map' algorithm was implemented as follows:

```
@staticmethod
def get_grid_links(width, height):
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    def neighbors(x, y):
        for (dx, dy) in directions:
            (nx, ny) = (x + dx, y + dy)
            if 0 <= nx < width and 0 <= ny < height:
                yield (nx, ny)
    return {(x, y): list(neighbors(x, y))
            for x in range(width) for y in range(height)}
```

The method `get_grid_links()` takes two parameters, the width and height of the environment, then for a given cell in the environment, determine the neighbouring cells as dictated by the directions provided (top, bottom, left, and right). This will be repeated for all cells in the environment and returned in a dictionary, where each cell is the key and corresponding values are all neighbouring cells.

To implement breadth-first or depth-first search into the environment, it was adapted to fit into a sub-class of the Environment class method `run()` which would not use the `steps()` method as previously but would let the search algorithm drive the movement. Implementing the search algorithms in this way meant a large `run()` method, but this method is well documented at all stages in the code and is included as an appendix to this document. More basic implementations of breadth-first and depth-first search algorithms were implemented in the code for use within a `path_to_target()` method, where the agent could return to the horse if the horse location was already known when the treasure was found. These algorithms match the core functionality of the `run()` method so are more compact to include here for comparison.

The search algorithms were implemented as follows:

```
def breadth_first(start, goal, neighbors):
    frontier = deque([start])
    previous = {start: None}
    while frontier:
        s = frontier.popleft()
        if s == goal:
            return path(previous, s)
        for s2 in neighbors[s]:
            if s2 not in previous:
                frontier.append(s2)
                previous[s2] = s

def depth_first(start, goal, neighbors):
    frontier = deque([start])
    previous = {start: None}
    while frontier:
        s = frontier.pop()
        if s == goal:
            return path(previous, s)
        for s2 in neighbors[s]:
            if s2 not in previous:
                frontier.append(s2)
                previous[s2] = s
```

Both search algorithms are almost identical, except for how the queue (frontier) is handled. In the `breadth_first()` method, `popleft()` is used for determining the next cell, in `depth_first()` `pop()` is used. The core difference here is that `popleft()` in the breadth-first algorithm results in a FIFO (first-in first-out), meaning new nodes which are always deeper than their parents go to the back of the queue, and old nodes which are shallower than the new nodes get expanded first. The depth-first search algorithm in using `pop()` instead uses LIFO (last-in, first-out) meaning that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is a level deeper than its parent, which when it was selected was the deepest node at the time. This is repeated until there

are no further levels of depth available. There is also a conditional check to determine that the next cell targeted by the agent to move to has not already been visited, this prevents infinite loops so the agent will always progress through the environment based on the next cell in the queue which has not been visited previously.

With the cell grid mapping constructed when the environment is initialised, the search algorithms are run by inputting the start location (0,0) and the goal. In our environment because uninformed search is implemented, it is not possible to know the location of the goal when the agent starts on (0,0), so instead the environment determines that the goal has been reached when the agent *percepts* the goal, in this case the treasure. When the agent finds the treasure, if the agent has found the horse previously, the search algorithm is called again to determine the best path to the goal, if the horse has not been found, the search algorithm is reset and the new goal is finding the horse, with the start location being the current agent location.

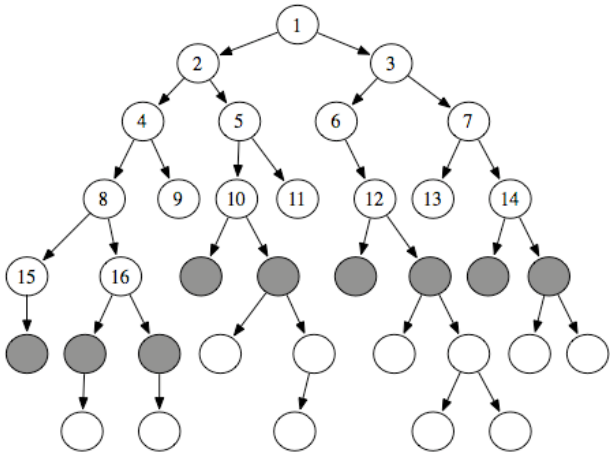
Performance of Random vs Search

To judge the performance of the game, part 2 and part 3 implementations were run 10 times each and the results compared. Part 1 was not included as it is a 1D implementation and so not a valid comparison. Both parts 2 and 3 are 2D implementations, with the difference being part 2 is entirely random and part 3 is driven by a search-based algorithm. All tests were run in an environment measuring 7x7, for part 3 both breadth-first and depth-first were run.

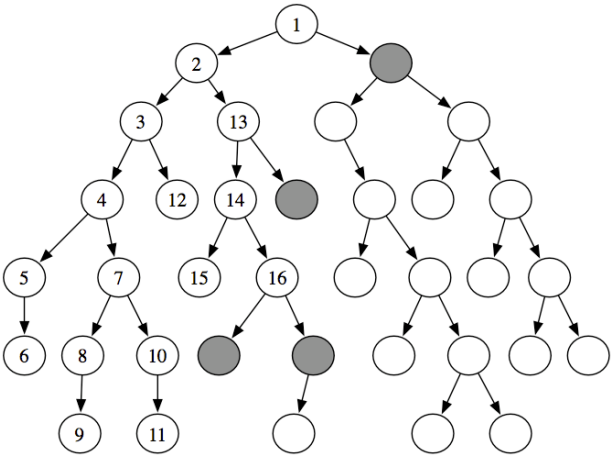
	Random Performance		BFS Performance		DFS Performance	
Run #	**Steps**	**Escaped**	**Steps**	**Escaped**	**Steps**	**Escaped**
1	35	No	42	Yes	47	No
2	17	No	42	Yes	47	No
3	135	No	36	Yes	46	Yes
4	70	Yes	18	Yes	36	Yes
5	44	No	30	Yes	27	Yes
6	76	Yes	47	No	56	Yes
7	77	No	47	No	46	No
8	20	No	22	Yes	44	Yes
9	1	No	30	Yes	47	No
10	28	Yes	47	No	46	No

When comparing the results between random and search-based progression through the environment, having the agent move to locations determined by the search-based algorithms is a lot more successful than randomly moving through the environment. When progressing randomly, the agent only had a 30% escape rate, compared to 70% escape rate for breadth-first and 50% for depth-first. Both search-based algorithms had a more consistent step-count also to meet their 'end-game' scenario, breadth-first had a range of 18-47 steps and depth-first 27-56 steps compared to the random range of 1-135 steps. Even though only run a limited amount of times, the results do accurately show that progressing through an environment using uninformed search algorithms for movement have a far higher success rate and require less moves than those progressing randomly.

While the above results are promising, there was a major flaw spotted with both search algorithms in how the progress through the environment. In this scenario, the environment is a 7x7 grid, where ***legal*** moves are the agent moving to cells directly adjacent to its current location. When using breadth-first search or depth-first search to dictate the next cell the agent should move to, the agent was making what can be considered illegal-moves by following exactly the progression of these algorithms. For example, consider the following progression of a breadth first search (the node numbers dictate the order of progression):



When progressing through an environment as dictated by the sequence of a breadth-first search (FIFO), the agent will move from node 3 to node 4, what this translates to is an illegal move from the position (1,0) to (0,2). In the rules of the environment this is not possible as (0,2) is not directly adjacent to (1,0) but the algorithm dictates that this is the next cell in the expansion of the binary tree. The same can be found with the depth-first algorithm:



When progressing through the environment using depth-first, there are many occasions when the agent makes illegal moves, in the above example moving from 7 to 8 is an illegal move because they are not adjacent to each other.

To get around this and investigate the performance of these search algorithms where the agent could only move to the next node in the search algorithm via a path consisting of legal moves, an additional block of code was added. This additional block of code could determine if the next node was a neighbour of the current node, if not, a path to the target node would be found using the same search

algorithm that was being run for the overall environment progression. In this example, the check for `default_moves` determined if the agent would move to the next node via traditional search-algorithm progression or would move via a path consisting of legal moves:

```
# Move back to previous cell before going to next node, not
# possible to traverse between nodes on the same level, have
# to move back first
if not default_moves:
    current_neighbours = neighbors.get(tuple(agent.location))
    # If the target cell is not a neighbour of the current
    # cell...
    if s2 not in current_neighbours and s2 != (0, 0):
        print("No path to next cell {}, finding next best "
              "route".format(s2))
        path_to_s2 =breadth_first(agent.location,
                                  s2,
                                  neighbors)
        # Delete first and last in path, currently on first
        # cell and the last cell is the destination
        if len(path_to_s2) >= 3:
            del path_to_s2[0]
            del path_to_s2[-1]
        else:
            del path_to_s2[0]
        print("Route to {} found: {}".format(s2, path_to_s2))
        for cell in path_to_s2:
            # Move agent to target cell via adjacent cells
            execute_action(agent, 'move_forward', cell)
            agent.steps += 1
```

Running the same test again for both search algorithms but with only legal moves in the environment permitted, the results were significantly worse than those where illegal moves were carried out:

	BFS Performance		DFS Performance	
Run #	**Steps**	**Escaped**	**Steps**	**Escaped**
1	210	No	20	No
2	70	No	79	No
3	250	No	33	No
4	70	No	119	Yes
5	254	No	127	No
6	47	No	175	No
7	60	No	36	No
8	225	Yes	33	No
9	72	No	163	Yes
10	230	No	69	No

Breadth-first search had very poor performance, escaping only 10%, depth-first was not much better only escaping 20% of the time. When looking at the amount of steps in each run, breadth first ranged from 47-254 steps compared to depth-firsts range of 20-175 steps. Interestingly, these results are the opposite

of what was seen when illegal moves were permitted, in this instance depth-first is the better performing in terms of success rate and the total amount of steps taken. When these results are compared again against the random progression through the environment, random progression had an escape rate of 30% with a range of steps from 1-135. So whilst it can be said that breadth-first and depth-first search algorithms to present better results for successfully reaching an environment goal or state, it needs to be taken into consideration that progression to this end-scenario might not be within the rules of the environment the algorithm is being run. This can be a case-by-case basis however, this may be acceptable in some scenarios, but not where there is an agent following a set of rules imposed by the environment in which it exists.