Artificial Intelligence 1
Lab 1

Roeland Lindhout (s2954524) & Name2 (student number 2)
AI2/AI3

May 2, 2016

Theory

Exercise 1

Reversi The performance measure is the ability to win, or on a smaller scale, the ability to turn over as many of the opponent's discs as possible. The environment is a board, divided into 8x8 squares, all of the same color. The actuators are the hands and arms of the player's, and the sensors are the senses the players have, e.g. the eyes and the nerves in their fingers. The environment can be described by the following features:
Fully observable, because the entire playing board can be seen.
Deterministic, because the next state is a consequence of the current state and the action.
Sequential, because the players should have some sort of tactic to win, therefore some form of future planning should take place.
Semi-dynamic, because the player could have a planned action, which has to be altered when the opponent makes a different move than expected.
Discrete, because the game is played move by move, and therefore not continuous.
Multi-agent, because two players are involved.
The Reversi agent architecture is a utility-based one, because there are often multiple options to place the discs, and therefore the player has to decide which option is best to reach to goal, i.e. which option has the highest utility.

Robotic Lawn Mower The performance measure is the length of the grass. The environment is a field of grass, with borders indicated by a wire. The actuators are the robot's wheels and the knives to cut the grass. The sensors change, depending on the way in which the borders are indicated. It could be some pressure sensor, that is activated when it hits the border. Sophisticated lawn-mowers can have rain sensors and sensors to detect the length of the grass, but with less sophisticated types, the robot is just always cutting, independently of the length of the grass. The environment can be described by the following

features:
Partially observable, because the robot only senses the wire used to indicate the borders when it reaches the line. It does not always know where the line is, so it is not completely aware of the environment.
Deterministic, because the robot moves on and/or cuts grass. The relevant parts of the environment state, i.e. the length of the grass, are therefore determined by the robot's action. Someone could walk into the field, but that is not relevant to the robot.
Episodic, because the robot moves somewhere, then decides to either cut or not cut the grass, and then moves to the next place, without any memory of some kind. It just systematically moves over the field. The decision to cut grass does not depend on where it has been or where it will go.
Dynamic, because objects could be placed or removed from the field, and the borders can be moved during the cutting of the grass.
Continuous, because the robot is always moving around the field until it is shut down.
Single-agent, because the only one involved is the grass-mower itself.
The best agent type for a robotic lawn mower, is a simple-reflex architecture. If the grass is long, cut it and if a wire is reached, turn are basically all it needs to be able to successfully move over a simple field.

Exercise 2

1. $mazeDFS()$ is unable to find a path from the yellow to the red square because of the order in which the actions are taken. The order is currently [N, E, S, W], which results in that particular maze in an infinite loop at position 6. Each time that position 6 is visited, position 5 will be the last to be pushed, therefore the first to be popped in the next iteration. At position 5, position 6 is the last position to be pushed, making it the first position to be popped in next iteration. This continues until the stack is full and no goal will be found.

2. A way to make sure that $mazeDFS()$ always finds a solution is to keep track of the positions that have already been visited. If the neighbour position is in an array called visited, then this neighbour should not be visited again. In the pseudocode this part will be added after checking if the move is possible. The already visited positions won't be pushed to the stack.

3. The path that the algorithm will take is: 1-2-6-5-9-13-14-10-7-3-4-8-12-11-15

4. The path for this algorithm is: 1-2-6-7-3-4-8-12-15

5. If the algorithm is fitted with a FIFO queue to turn it into an BFS version, the algorithm will turn into a complete one. If correctly implemented, this algorithm will always return a solution simply because BFS is a complete

algorithm and will search through all possible states. Whilst BFS is not the fastest algorithm, it will always find a solution if one exists.

6. The path that this algoritm will take is: 1-2-6-7-5-3-9-4-13-8-14-12-10-11-15

7. Again, implementing a visited array to check for visited states. This was already done for the previous question. Further reduction of the number of visited states is not possible for BFS.

8. For extremely large mazes, a DFS algorithm would probably be the better choice. BFS will probably run out of memory to store all the possible states and will never return a solution. DFS might end up in an infinite loop, but it also might not. In this case DFS would have a higher chance of returning a solution. In that case I'd choose the DFS approach.

Exercise 3

1. The program finds a path from 0 to 99 and from 0 to 102, using BFS. The found paths are respectively 28091 nodes and 29325 nodes. The program also finds a path from 1 to 0, using DFS. This path has a length of 2 nodes. The other paths are not found, for these the program returns a fatal error, because it does not have enough memory.

2. BFS uses a lot of memory, therefore the solution to why it did not work, is to increase the allocated memory. We multiplied it by a factor of 10. The DFS problem is solved, by making sure that the program does not continuously add 0's and 1's to the end of the stack, because that would mean that all new states would also use 0. The way to solve this, is by making sure that the multiplication uses values bigger than 0, and division only uses values other than 0 and 1.

3. To tackle the problem of finding a path in BFS, we need to change something in the struct State. We've added some integers called index, length and cost. The index is to keep track of which node we are in, length is to keep track of the length of the path and cost is to keep track of the cost of each action. To find the path that BFS explored, we need to backtrack its steps. The function findPath does this. It gets the goal node from the function search and traces each step back by figuring out which move was taken. We've added an index to each move to backtrack which move it actually made, so if one was added, subtracted or that the value was multiplied by three or two etc. We calculated each move by taking the modulo 6 of the index. The value that is returned shows which action was performed on that value. We put the values first in a stack and popped them back in an array for the right path. In the end we print the cost and length of the path as well.

4. We implemented a heap to get the lowest cost to the front of the fringe. Unfortunately, somehow the program ends up in infinite loops when the path is longer than 2 nodes. The loop occurs in the insertFringe function. It is unclear to us why this happened.

5. We did not find this path, because of the flaws from exercise 4.

6. To program an IDS-algorithm, we first made an Depth-limited algorithm. This algorithm first checks if the limit is reached, and if the current position is the goal position. If this is not the case, the children of the position are produced, according to the steps given in the exercise. These are then continuously reproduced, with each recursion decrementing the limit. This is done for all children. If the goal position is found, the path cost is printed and this function returns 1. **Important note:** The path cost is printed multiple times, because the result is calculated with the "and/or" function, instead of the "XOR", therefore multiple costs are given through to the function. After the path cost print, the function returns 1. Here comes the iteration into play. Each iteration has a deeper depth, which is passed on to the Depth-limited function as the limit. If the Depth-limited function does not return 1, the depth is increased by 1, until infinity. When it does return 1, the length of the path is equal to the depth. This is then printed, and the function goes out of the loop. The program then terminates.

7. The iterative deepening program and the UCS (if implemented correctly) find the most optimal solutions. UCS does this quicker than IDS, but both are fast. DFS only finds a solution when the path is very short, because else it uses too much memory. It can be made more effective by reducing the maximum depth. BFS looks at every node, and is therefore very slow, but it does find solutions.

Programming

Program description

Problem analysis

Program design

Program evaluation

Program output

Program files

Main.c

```
        Your code here
```

SomeFile.c

```
        Some other code here
```