Artificial Intelligence 1

Lab $n1$

Elisa Klunder (s5190940)
Julia Belloni (s5049911)
Lisa Overgoor (s5191734)
Learning Group 14

April 28, 2023

Disclosure

To develop the code I (we) used the following resources: lecture slides, stack overflow, geeks for geeks

Exercise 1.1

a) PEAS-descriptions

Reversi/othello computer

- **Performance**: the aim of the program is to interact with the user and carry out the game of Reversi. The performance is optimal if the program manages to win against the user, and chooses the move in a fast way: this depends on the efficiency of the search algorithm. Immagine if for every move the computer took two minutes to move, probably no one would play against a computer program since it would not be enjoyable and the train of thought for the winning strategy might get lost.
- **Environment**: to carry out the task, the program needs an opponent to interact with, an 8x8 matrix to portray the board to represent the game. Lastly, the program needs to be aware of the set of rules that define the game, so that it is clear what the legal moves are.
- **Actuators**: for the opponent to be able to interact with the opponent. The algorithm that chooses the next move is essential for determining the sequence of the game. At every move the algorithm needs to update the current representation of the board to be able to progress with the next move.

- **Sensors**: to detect the changes in the board the program needs to sense the opponent's entry and update the new state of the board.

Performance	Environment	Actuators	Sensors
# of games won time efficiency	opponent 8x8 board	algorithms display of board	opponents entry state of the board

Lawn mower

- **Performance**: the performance of a lawn mower is optimal if the end result can be achieved in a fast way and it consists of an evenly cut lawn. The energy consumption for accomplishing the result should not be too elevated, otherwise, it would not be applicable on a large scale and the automation of the process would not be worth the benefits.

- **Environment**: the robot has to operate in a field where there is grass to lawn, and the working area should be delimited by a wire so that the agent knows where to move around. It is more than possible that in the area that needs to be lawned, there are obstacles that need to be avoided.

- **Actuators**: to accomplish the task a lawn mower needs blades for trimming the field, wheels for moving around the environment, and a motor to coordinate all these activities.

- **Sensors**: to sense the environment the agent needs to perceive its surroundings and this can be achieved with proximity sensors, to avoid collisions with obstacles. The same result can be also achieved with a bump sensor.

Performance	Environment	Actuators	Sensors
lawn evenness energy consumption speed	mowning area obstacles	blades wheels/tracks motor	proximity sensors

b) Domains characterization

Environment type	Reversi/othello computer	Lawn mower
Observable	fully	partially
Deterministic	deterministic	stochastic
Episodic	sequential	episodic
Static	static	dynamic
Discrete	discrete	continuous
Agents	multi	single

Exercise 1.2

a) The sequence of moves from the red square to the yellow square before the program gets stuck is: E N E S N. At this point, the algorithm would perform the actions S N repeatedly without ever getting out of the loop. The problem with the algorithm is that it does not check whether the square has already been visited before moving there. Therefore, this results in an infinite loop since the sequence of moves is N E S W, so after going south in square number 7 the first available move is S. The algorithm at this point is in square 8, and the first move available in N, ending up in an infinite loop.

b) To solve this problem, we could modify the code to turn this open list version of DFS into a closed-list version. To do this we would need to include an array that would store the locations that have already been visited. Before exploring the new location then, we should check whether the location has already been visited or not. Only if the location is not in the array of visited locations, then we should push it on the stack and proceed. The modified pseudo-code would be:

```
1  procedure mazeDFS(maze, start, goal):
2      stack = []
3      explored = [] //Array to store explored locations
4      stack.push(start)
5      while stack is not empty:
6          loc = stack.pop()
7          if loc == goal:
8              print "Goal found"
9              return
10         explored.append(loc) //Add the current location to explored
11         for move in [W,S,E,N]: //in this order
12             if allowedMove(loc, move) and neighbour(maze, loc,
                   move) not in explored:
13             stack.push(neighbour(maze, loc, move))
14     print "Goal not found"
```

c) For the call mazeDFS(maze, 1, 15) the sequence of actions of the algorithm would be E, N, E, S, E, N, N, W, N. The resulting order of visited states would be 1, 2, 6, 7, 3, 4, 8, 12, 11, 15.

d) If we change the order of the list at line 9 into [E,W,S,N], these values would be pushed on a stack and then popped from it in the reverse order. Therefore, the sequence for each square would be N S W E. The resulting order of visited states carried out by the algorithm for the call mazeDFS(maze, 1, 15) would be 1, 2, 6, 5, 9, 13, 14, 10, 7, 3, 4, 8, 12, 11, 15.

e) If we replace the stack with a FIFO queue, the algorithm will indeed

become the open-list version of Breadth First Search (BFS). Since BFS explores all the neighboring locations of the current location before moving on to their neighbors, BFS is guaranteed to find a solution if one exists. Moreover, BFS will always find the shortest path to the goal. However, this BFS algorithm can be very time and memory inefficient. This is because it explores all possible paths by keeping them in the FIFO queue and processing all neighboring locations, even the ones that were already visited and processed.

f) The pseudo-code in which the algorithm is turned into the open list version of Breadth First Search would be:

```
procedure mazeDFS(maze, start, goal):
    queue = [] //a queue instead of a stack
    queue.enqueue(start)
    while queue is not empty:
        loc = queue.dequeue()
        if loc == goal:
            print "Goal found"
            return
        for move in [N,E,S,W]: //order is reversed to mantain the
             same sequence in visiting neighbouring squares
            if allowedMove(loc, move):
            queue.enqueue(neighbour(maze, loc, move))
    print "Goal not found"
```

For the call mazeDFS(maze, 1, 15) the order of the first 10 states visited by this algorithm would be 1, 2, 6, 1, 7, 2, 5, 2, 3, 6.

g) To reduce the number of visited states by the BFS algorithm we could, again, introduce the concept of an explored set, so that the algorithm won't continue to enqueue and process already visited locations. This could be done by storing the visited locations in an array and then checking whether the location has already been visited before enqueuing it.

h) In general, for path searching in large mazes, we would prefer to use a variation of Breadth-First Search (BFS) over a variation of Depth-First Search (DFS).

The main reason is that BFS systematically explores all the neighboring nodes at the current level before moving on to the next level. This ensures that BFS will find the shortest path from the starting node to the goal node, as it explores nodes in order of their distance from the starting node. BFS is also complete, meaning it will always find a solution if one exists in a finite graph or maze with uniform edge costs.

On the other hand, DFS may not be as suitable for large mazes because it can get stuck in long dead-ends, leading to inefficiency in terms of time and may require extensive backtracking. Furthermore, DFS runs until a leaf

node is found, checks whether the solution is reached, and then backtracks to explore the next path. For this reason, if a path is found DFS will not look for other paths along other branches even though they may end up being shorter. Therefore DFS does not ensure that the selected path is also the shortest one.

However, there may be specific cases or requirements where a variation of DFS, such as Depth-Limited Search or Iterative Deepening Depth-First Search, could be more appropriate for path searching in large mazes. For example, if the maze has certain characteristics, such as being very open with few dead-ends, or if the goal is to find any valid path rather than the shortest path, DFS may be more suitable. It ultimately depends on the specific constraints and characteristics of the maze and the desired outcome of the path-searching task.

Exercise 2

a) DFS is not able to find a solution to the maze since the algorithm gets stuck in a loop. The fringe is implemented as a stack, so every room that is visited will be pushed on it and dequeued afterwards. The algorithm starts from position 0 3 0, and the only path that can be taken is to the right of the initial position. Room 1 3 1 is pushed on the stack. Since there are no other items on the stack, the state 1 3 1 becomes the new starting position. At this point three connections are pushed on the stack in the following order: 1 3 1, 1 2 0, and 0 3 0. The first item to be popped is then 0 3 0, which was the original starting position. Since the code does not keep track of the visited locations, the program runs into an infinite loop. The program only stops when the stack becomes full, at which point it is interrupted by an error message.

b) BFS does not find the path toward the goal because the set of all possible paths that the algorithm explores is too big to be contained in the fringe. The program is used without an array of visited locations, so it ends up revisiting the same nodes multiple times. The while loop that iterates through the fringe and expands the states can thus result in an infinite loop if there is a cyclic path in the maze. This can cause the fringe to continuously expand and fill up, resulting in the behavior where the queue storing the fringe becomes completely full. When this happens, the program is interrupted by an error message.

c) This is the only part of the code that was changed:

```
1 #!/usr/bin/env python3
2 from fringe import Fringe
3 from state import State
4
5
6 def solve_maze_general(maze, algorithm):
```

```
7
8       if algorithm == "BFS":
9           fr = Fringe("FIFO")
10      elif algorithm == "DFS":
11          fr = Fringe("STACK")
12      else:
13          print("Algorithm not found/implemented, exit")
14          return
15
16      room = maze.get_room(*maze.get_start())
17      state = State(room, None)
18      fr.push(state)
19      visitedLocations = [] // array for storing visited locations
20
21      while not fr.is_empty():
22          state = fr.pop()
23          room = state.get_room()
24          visitedLocations.append(room) //add to array visited
                locations
25
26          if room.is_goal():
27              print("solved")
28              fr.print_stats()
29              state.print_path()
30              state.print_actions()
31              maze.print_maze_with_path(state)
32              return True
33
34          for d in room.get_connections():
35              new_room, cost = room.make_move(d, state.get_cost())
36              new_state = State(new_room, state, cost)
37
38              if new_room not in visitedLocations:
39                  fr.push(new_state) // push the new state on the
                        stack only if it is not visited
40
41      print("not solved")
42      fr.print_stats()
43      return False
```

d) To implement the UCS algorithm we had to update the value of "priority" so that the priority queue could work. In particular, the value of priority was updated accordingly to the cost of the path, so that paths with a lower cost were processed first. The only two things that were added to the code above were an if statement after line 11

```
1       if algorithm == "BFS":
2           fr = Fringe("FIFO")
3       elif algorithm == "DFS":
```

```
4           fr = Fringe("STACK")
5       elif algorithm == "UCS":
6           fr = Fringe ("PRIORITY")
7       else:
8           print("Algorithm not found/implemented, exit")
9           return
```

and one line between lines 38 and 39 to update the priority

```
1       if new_room not in visitedLocations:
2           new_state.priority = new_state.get_cost() //added line
3           fr.push(new_state)
```

e) BFS implemented with array of visited locations:

- moves: E D S U W
- transition model:
  (0, 0, 1) → (1, 0, 1) cost: 1
  (1, 0, 1) → (1, 0, 0) cost: 3
  (1, 0, 0) → (1, 1, 0) cost: 4
  (1, 1, 0) → (1, 1, 1) cost: 7
  (1, 1, 1) → (0, 1, 1) cost: 8

UCS:

- moves: E E S S W N W
- transition model:
  (0, 0, 1) → (1, 0, 1) cost: 1
  (1, 0, 1) → (2, 0, 1) cost: 2
  (2, 0, 1) → (2, 1, 1) cost: 3
  (2, 1, 1) → (2, 2, 1) cost: 4
  (2, 2, 1) → (1, 2, 1) cost: 5
  (1, 2, 1) → (1, 1, 1) cost: 6
  (1, 1, 1) → (0, 1, 1) cost: 7

Because BFS does not take the costs of the paths into consideration, it takes the shortest path from x to g. The UCS algorithm instead does consider the costs and therefore runs until it finds the path that has the lower final cost.

f) For implementing the greedy algorithm, an extra if-statement is added to read in the algorithm type.

```
1       if algorithm == "BFS":
2           fr = Fringe("FIFO")
```

```
    elif algorithm == "DFS":
        fr = Fringe("STACK")
    elif algorithm == "UCS":
        fr = Fringe ("PRIORITY")
    elif algorithm == "GREEDY":
        fr = Fringe("PRIORITY")
    else:
        print("Algorithm not found/implemented, exit")
        return
```

Besides that, the priority in the queue is now ordered by the heuristic values instead of the cost, therefore extra if-statements are added to distinguish the two algorithm types.

```
    if new_room not in visitedLocations:
        if algorithm == "UCS":
            new_state.priority = new_state.get_cost()
        elif algorithm == "GREEDY":
            new_state.priority = new_room.get_heuristic_value()
    fr.push(new_state)
```

g) Just like the Greedy algorithm, implementing the A* algorithm includes reading in the algorithm type.

```
    if algorithm == "BFS":
        fr = Fringe("FIFO")
    elif algorithm == "DFS":
        fr = Fringe("STACK")
    elif algorithm == "UCS":
        fr = Fringe ("PRIORITY")
    elif algorithm == "GREEDY":
        fr = Fringe("PRIORITY")
    elif algorithm == "ASTAR":
        fr = Fringe("PRIORITY")
    else:
        print("Algorithm not found/implemented, exit")
        return
```

Then, the order of the queue is determined by the cost value added to the heuristic value of the room. Therefore, an additional if-statement is added for this algorithm

```
    if new_room not in visitedLocations:
        if algorithm == "UCS":
            new_state.priority = new_state.get_cost()
        elif algorithm == "GREEDY":
            new_state.priority = new_room.get_heuristic_value()
        elif algorithm == "ASTAR":
```

```
7             new_state.priority = new_room.get_heuristic_value() +
                  new_state.get_cost()
8     fr.push(new_state)
```

h) Greedy:

- moves: N N N E E E S U N
- transition model:
  (0, 3, 0) → (0, 2, 0) cost: 1
  (0, 2, 0) → (0, 1, 0) cost: 2
  (0, 1, 0) → (0, 0, 0) cost: 3
  (0, 0, 0) → (1, 0, 0) cost: 4
  (1, 0, 0) → (2, 0, 0) cost: 5
  (2, 0, 0) → (3, 0, 0) cost: 6
  (3, 0, 0) → (3, 1, 0) cost: 7
  (3, 1, 0) → (3, 1, 1) cost: 10
  (3, 1, 1) → (3, 0, 1) cost: 11

A*:

- moves: E E N E N U N
- transition model:
  (0, 3, 0) → (1, 3, 0) cost: 1
  (1, 3, 0) → (2, 3, 0) cost: 2
  (2, 3, 0) → (2, 2, 0) cost: 3
  (2, 2, 0) → (3, 2, 0) cost: 4
  (3, 2, 0) → (3, 1, 0) cost: 5
  (3, 1, 0) → (3, 1, 1) cost: 8
  (3, 1, 1) → (3, 0, 1) cost: 9

The paths that the two algorithms find are different because the greedy algorithm only considers the heuristic values whereas the A* algorithm considers the heuristic values and the costs.

i) For implementing IDS, again reading in the algorithm name is required first.