Abel van Beek SNR: 5978831
Maurin Voshol SNR: 6042279
Wouter Castelein SNR: 5936608

Project B

Exercise 1

a. The game state class keeps track of the entire state of the game. It holds the functions necessary to modify the game state, such as a function to generate the successors of a certain coordinate given and a function to check whether the game has a victor or not. In the game state there are multiple agents. These various agents can be distinguished by the agentIndex: zero means Pac-Man himself, higher values mean ghosts.
On top of that, there are a lot of functions in the game state that allow us to fetch information, such as getting the position of the ghosts, the position of the food in the level, the position of the walls etcetera.

   Upon action, the game checks if the game has come to an end yet. If not, the game state is updated by applying the action given to the agent given in the current game state. The agent is updated according to the rules that apply to that certain agent.

b. In the agent state class the initial game state is defined. The world is given as an argument and the location of Pac-Man is kept on top of that. In order to display the world, a 2D array is created from the data that represent the current state of the game.

   The actions available are defined by the Directions class. We have the option to go north, east, south, west or to stand still for one iteration.

   Upon action, a new configuration will be produced with a new position of Pac-Man. This new configuration will be returned and then this new configuration is displayed. This configuration is kept in a 2 dimensional grid, which offers some functions itself, such as a function to convert the grid to a list, which makes it easier to traverse.

c. Stack > Pallets of beer crates
Queue > Supermarket check-out line
Priority Queue > Hospital waiting room

Exercise 2

a. First initialize the startstate, the data structure that is used (stack/queue/priority queue) and a list to keep track of the visited states.
Next go into a while loop that loops until the goal is found or the data structure is empty. In the loop, check if the current state is the goal state; if so return the list of moves that led to that state. Otherwise, add every possible successor of that state to the used data structure, excluding the states that were already visited. Each state keeps track of the moves that led to that state.

b. We could remove the check on whether we already visited the state, and therefore we wouldn't need the global list of visited states anymore. This way we get a tree-search version that does expand nodes that were already visited multiple times. To reconstruct the path

towards that node we would still need a list of all the expanded nodes and from which direction we came to that node, this way we could ‘walk back’ to find the desired path.

Exercise 3

a. For a search strategy to be complete, it has to always find a solution if one exists in the current game state.

b. In Q1 we implemented depth-first search. This search strategy expands the deepest unexpanded node. If we assume that we apply the graph search method of depth-first search in a finite state space, the depth-first search implementation will be complete.

c. The solution found by depth-first search will not per definition be the optimal solution. The found solution is the left-most solution in the tree, since depth-first search first expands the left-most node at each split. Eventually it will either find a solution there and return it, or go up one level if it has not found a solution and try the righthand side there. The tree will be traversed in this manner until a solution has been found or the end of the tree has been reached.

d. Pac-Man is firstly exploring all the ‘West’-options. Eventually it will be unable to go left, then it will go south and try west again. All the states until a solution is found will be visited in that manner. This way, Pac-Man does not visit all the explored squares on his way to the goal, because if the solution is not found in a certain subtree of the entire tree, the particular way that the nodes in that subtree are visited will not be used, thus removed from the tree. The search algorithm will then build a new list of actions for a new subtree and repeat the proces.

Exercise 4

a. The BFS is complete if there is a finite number of branches. Otherwise we’d never go to the next step since we’re infinitely adding new directions.

b. The solution is least cost if every successor step is equal in all situations. This search algorithm does not look at for example the Pac-Man food. So if those need to be considered as well this algorithm isn’t least cost.

c. Yes, it works for the eightpuzzle. It works the same basically. From the starting position the algorithm checks all possible moves, and checks if they lead to the desired goal, and if not, the algorithm goes on with their successors.

Exercise 5

a. 1. StayEastSearchAgent
The StayEastSearchAgent calculates the total cost of the path by using the formula 0.5^X. Using this formula, the costs of going east, e.g. increasing the value of X, leads to a lower outcome of the total cost function. Namely, 0.5^1 = 0.5, 0.5^2 = 0.25 and 0.5^3 = 0.125. In order to keep the cost at a minimum, Pac-Man tries to be as far east as possible, because that path is cheaper.

2. StayWestSearchAgent
   The StayWestSearchAgent calculates the total cost of the path by using the formula 2^X. Using this formula, the costs of going west, e.g. decreasing the value of X, leads to a lower outcome of the cost function. Namely, 2^3 = 8, 2^2 = 4 and 2^1 = 2. In order to keep the cost at a minimum, Pac-Man tries be as far west as long as possible, because that path is cheaper.

3. Regular uniform-cost search
   The intended behaviour of the uniform-cost search strategy is to find the path to the goal state that has the lowest total cost. The way this is achieved in this algorithm is by using a priority queue on the costs to reach a certain node in the graph. This way, the cheapest node to reach is expanded first. If the solution is not found in the cheapest node, the second-cheapest node is expanded and so on, until the solution is found.

b. Maze: mediumMaze
   1. StayEastSearchAgent
      Path costs: 74
   2. StayWestSearchAgent
      Path costs: 152
   3. Regular uniform-cost search
      Path costs: 68 -> BFS

   The costs for the StayEastSearchAgent are, compared to the StayWestSearchAgent, very low. This is due to the fact that the maze offers a relatively direct route on the east of the field: the costs are here low for the StayEastSearchAgent. The StayWestSearchAgent tries to go and stay west as long as possible, however, in this maze Pac-Man is forced to go to the east near the bottom to be able to reach the exit. The best search algorithm here is the regular uniform-cost search strategy: this is basically breadth-first search, because the cost of all step s are equal, which means that the path calculated by the ucs strategy is the optimal, thus shortest, path.

Exercise 6

1. DFS will always go left first if possible, if that is n no longer possible it will choose a different direction in the following order: East, South, North. After that, it will try going left again. Because of this the Pac-Man will walk back and forth (west and east) and one step down whenever it reaches a wall. This is obviously a very slow route (cost 298) The upside is that it expands less nodes than the BFS and UCS, 576 in total. This is also slightly better than the Astart algorithm

2. The BFS algorithm is optimal in the openMaze map and thus is finds the shortest route (cost 54). The algorithm looks first at each successor of the start state, then at each of their successor etc. This does, however, cover a big part of the openMaze, requiring 682 nodes to be expanded.

3. The UCS algorithm behaves identical to the BFS algorithm because there are no cost differences for each move. Therefore it also finds the shortest path with cost 54 and 682 nodes expanded.

4. The Astar algorithm will guess which nodes are closer than others and expand these first. In the open maze level this results in 556 expanded nodes, which is slightly worse than DFS, but a lot worse than the 682 of DFS and BFS. This is because the way the maze is designed. The Astar algorithm first tries to go left and then down, but that path leads to a death end, which means a lot of nodes are expanded without result. The algorithm does however find the optimal path of cost 54. This probably means that it's the best algorithm. The DFS only expands 2 less nodes which is neglectable, and the other algorithms expand a lot more nodes to find a path just as optimal.

Exercise 7

The information provided in the initialisation of the problem will be the same throughout the (BFS) algorithm. This means we cannot store a value self.cornersVisited in the problem itself, because this should be different for each path in the search algorithm. We came to the realisation that we can use the abstractness of our BFS, to modify the representation gamestate itself! Instead of the normal gamestate (x,y) for other problems, we can now represent the gamestate as ((x,y), cornersVisited). This way, Pac-Man can pass one position twice, because the cornersVisited changes after touching a corner. The result of this extra list is that it creates multiple layers of the level where Pac-Man can go. You could think of it as a 3D Pac-Man game where a corner is a ladder to another level. We figured the problem doesn't need to store any other values (except for the costFn function copied from PositionSearchProblem).

The getStartState function defines the gamestate as explained above, where (x,y) is the starting position of Pac-Man and the cornersVisited is an empty list (unless Pac-Man is already standing in one of the corners).

The getSuccessor function is almost the same as that from PositionSearchProblem. Besides the wall check, the function also checks if the next step also includes one of the corners. If so, add that corner to the cornersVisited list.

The isGoalState function checks if the length of self.corners is the same as the length of cornersVisited. You could also check all the tuples in the list on equality, but since the corners can be visited in different orders and the list cannot contain a corner twice or any other coordinate whatsoever, it's safe to say that checking the length is the correct and quickest way of checking the goal state.

Exercise 8

a. For the cornersproblem we used a heuristic that returns the sum of the manhattan distances from corner to corner. For example, we start with the manhattan distance to a closest corner X from the player position, then we add the manhattan distance to the closest corner from X, etc. We keep track of each corner that was already visited and remove those from the list of

corners we still need to go to. This way we'll keep expanding nodes in the general direction of the closest corner, which seems like a decent strategy.

b. The resulting sum of the manhattan distances would be the exact cost along the corner points if there are no obstructions on the way. So this heuristic returns the minimum possible cost. Which means that the actual cost will always be equal or higher. Therefore the heuristic we use is admissible. Because we used the manhattan distance this heuristic is also consistent. If we walk on step closer the heuristic will always go down by 1, and one step further means it will go up by one. e.g. we will never get a higher heuristic if we get closer to the goal.

c. 901 Nodes

Exercise 9

a. The heuristic we implemented as answer to Q7 is the following. For every piece of food currently on the field, we calculate the distance from Pac-Man. The nodes that lead to the nearest piece of food will be expanded first. This method is continued until all the food has been eaten.

   We expect this to be an effective method, because Pac-Man will try to move to the nearest piece of food available. Continuing this trend leads to a minimized travel time between the different pieces of food, finding the optimal path from food to food. Thus, this will eventually lead to the optimal path to find all the food in the level.

b. This heuristic is admissible, due to the fact that the Manhattan function calculates the costs of the direct path to a piece of food. The costs of the direct path are always smaller or equal to the costs of the real path used by Pac-Man, because in the game there can be walls in the way of the direct path.

   The heuristic we have implemented is also consistent, because the calculation of the costs is done by using the Manhattan function. The costs of the path calculated by this function will always be the same as 1 + the calculated costs from the by-one-move-updated old position of Pac-Man. This leads to the consistent property holding for our heuristic.

c. 13898 Nodes

Exercise 10

a. We use breadth-first-search to find the path to the nearest food. So basically we walk to the closest food, then from that food again to the closest until all foods have been eaten by Pac-Man. We expect this method to be effective, because eventually all the food will be eaten. In general, it is usually a good strategy to get the food close to you first. Even though this leaves us with a sub-optimal path, it is the best we can with our current search strategies.

b. If for example we have a wall, and behind it a piece of food, all nodes towards that dead end will be expanded first, because the algorithm thinks that is the quickest path, while in fact the Pac-Man may need to go way around.

c. It is for an AI not possible to instantly see whether or not a path leads to a dead end, or that it is much longer than another path. Since it can only look one tile ahead, it will have to check for dead ends by walking the entire way, which is a waste of time and computing power.

Exercise 11

Provisional grades
==========
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 3/3
Question q6: 3/3
Question q7: 2/4
Question q8: 3/3

Total: 23/25