KI Project B

Martijn Dekker 6013368
Egor Dmitriev 6100120
Cody Bloemhard 6231888

December 16, 2018

1 Exercise 1

1.a ex1-a

The game state holds all the data that represents the game in the current time frame. This includes data like the structure of the level and the agents. The game state can give all legal actions for a given agent. The game state can be updated by giving a action for a agent. The game state will copy itself and edit the data of the copy. This results in a new game state, and it is returned.

1.b ex1-b

The agent state holds all data that describes an agent in the current time frame, like position and direction. You can ask it the position and direction, and it can copy itself. The agent state does not update itself. The copy method is used by other classes to gain a copy, with intention to modify that copy and use it form there.

1.c ex1-c

- A: II, Stack: Pile of dishes. You can put items on top of the stack, and remove items from the top. The bottom plate is last to be removed.
- B: III, Queue: Roller coaster waiting line. You can add items at the back, and remove items at the front. The last person to enter the waiting line is the last person to leave it.
- C: I, PriorityQueue: Emergency room waiting line. You can put items in, and the item witn the highest priority leaves first. The person bleeding to death will be helped before the guy with the broken arm, even if he came in first.

2 Exercise 2

2.a ex2-a

For all the exercises we have used following generic code structure:

fringe For all the different search algorithms we have used a fringe to store the unexpanded nodes. Depending on the exercise/search algorithm we have used different fringe datastructure. (Ex. A* uses a priority queue and DFS uses stack).
Before the expansion loop is run we store the starting node in the fringe so expansion can start from there.

closed list Closed list is used to store all the visited nodes to prevent loops from happening. For this we use a "set" datastructure.

road Can be compared to a policy. It holds data on which node should be visited from which node. This is used to reconstruct path once a goal node has been reached.

expansion loop Is a loop which keeps calling our node expansion code until the fringe is empty.

closed check When a node is expanded, we first check if it is closed/visited. If it is, we can safely skip it. If not, then we add it to the closed list.

goal check At expansion of a node we check whether the node is the goal state. If it is we reconstruct the path and return it.

retrieve neighbours When a node is not the goal node, we loop through each if its neighbours and add them to the fringe. In some of the algorithms costs and heuristics need to be calculated first.

not path found This is an edge case where there is no path found. This means the expansion loop has finished without reaching the goal node. In this case we return an empty path.

2.b ex2-b

Tree is a special kind of graph where there is exactly one path between each pair of nodes. This implies that there are are no cycles in a tree. Therefore we can scrap the closed list in tree-search.
Assuming the tree is correctly used by the generic structure, no additional changes will be needed.

3 Exercise 3

3.a ex3-a

That means that if there exist a path to the goal you are searching for, it will return that goal.

3.b ex3-b

It is complete, assuming that is has a finite depth, because the algorithm will keep searching till it found the path to the goal location or till there are no more possible paths to stroll, which haven't been visited before. This would happen when the stack is empty and that can only occur if it has tried every possible path.

3.c ex3-c

It will not always be the least cost solution. This is because it tries out a path and continues it till it ends, which can lead to the goal location. This first path it finds does not have to be the path with the least cost. And because it stops after finding a path it doesn't always find the least cost solution.

3.d ex3-d

Yes, the order is as we expected. Pacman does not visit all the explored squares on his way to the goal. This is because the first path he explores may or may not be a dead end. If this is the case then he will not go there, because his objective is to reach the goal.

4 Exercise 4

4.a ex4-a

It is complete because the algorithm will keep searching till it found the path to the goal location or till there are no more possible paths to stroll, which haven't been visited before.

4.b ex4-b

Yes, because every step pacman can take is equal in cost the algorithm needs to find the path with the least steps. And because it looks into every path of depth k before looking into paths of depth $>$ k it will always find the least cost solution.

4.c ex4-c

Yes, it returns a solution, because the algorithm was implemented so that it can be generalized on any problem. Also because BFS is complete a solution guaranteed to be found if there is one. Assuming b is finite.

5 Exercise 5

5.a ex5-a

For the first agent: it's intended behaviour is to find the shortest path to the goal, because there is no reason not to cause there are no obstacles, therefore every path costs the same. It is achieved by sorting the currently visible paths based on their cost so far, with the cheapest sorted at the front. That way the algoritm will constantly pick the cheapest path to further explore. And because every path costs the same it basically works like breadth first search. For the second agent: it's intended behaviour is to take the east route because that path has some food on it. The food makes for a cheaper path than a normal path. It is achieved almost the same way as the first agent, only now not every path costs the same. The path with the food costs less, so that path will be explored first instead of the other paths. For the third agent: it's intended behaviour is

to take the most west path, because the other paths contain ghosts which drive up the cost of a path. This is because the paths with the ghosts have a high chance of death. It is achieved the same way as with the other agents, only now the paths with the ghosts have a high cost and thus they will be sorted at the end. This way the algorithm explores the path without ghosts first.

When the algorithm finds a path to the goal it stops looking because it chose the cheapest path at all time, therefore the first path must be the cheapest.

5.b ex5-b

mediumDottedMaze: UCS: 68, 208 StayEast: 1, 186 StayWest: 17183894840, 169 The difference can be explained by the different cost functions. UCS uses a normal cost function which calculates a cost of 1 per move, no matter the direction. StayEast and StayWest use a formula to calculate the cost. The formula for StayEast is $0.5^{x-coordinate}$, so the farther pacman is to the east, the lower the cost of one move. The goal is the most western point the agent goes to, and so the most expensive move has a cost of $0.5^1 = 0.5$. The formula for StayWest is $2^{x-coordinate}$, so the farther pacman is to the east, the higher the cost of one move. Therefore the agent wants to take the most western path. So if pacman makes a move with an x-coordinate of 20, that move will already cost 1048576. And a move that is 1 to the west will cut the cost in half.

6 Exercise 6

6..1 Results

A* score of 456, 535 nodes expanded, cost of 54, found in 0.1 seconds, chooses down over left.

DFS score of 212, 576 nodes expanded, cost of 298 in 0.0 seconds, goes as far left then down, then as far right and down.

BFS score of 456, 682 nodes expanded, cost of 54, found in 0.0 seconds, chooses down over left.

UCS score of 456, 682 nodes expanded, cost of 54, found in 0.0 seconds, chooses down over left.

6..2 Behaviour

BFS searches on a level to level basis, it first looks at every path with length k, if it has not reached the goal it will expand every path in every possible way that has not been visited yet and gets new paths of length k+1.

DFS searches on a path to path basis, it tries to extent a path as far as possible, constantly going in the same direction and only when it can't do that any longer it will try a different direction or go one step back and try a different direction there.

UCS has a list with paths he has found till that point. It takes the path with the lowest cost, removes it from the list and expands that path in every direction, then it calculates the cost for these new paths and puts these

paths in the list. Then it takes the path with the currently lowest cost and keeps repeating this till it finds the goal.

A* works like UCS, only doesn't just look at the cost so far, but it looks at a combination of the cost so far and the heuristic. This heuristic is an estimate of how much it will cost from that point to the goal.

6..3 Differences and similarities

The difference in nodes expanded between BFS and the other is because BFS will also search in the lower right corner. This happens because BFS does not know that it is useless to look here since the goal is in the lower left corner.
BFS is in this way a dumb algorithm and just searches in every direction it can, but it does always find the shortest path. The terrible performance of DFS and the number of nodes expanded in this case is mainly caused by which direction it tries first. The order it expands is most likely Left, Right, Down and Up. This causes Pacman to go from right to left and left to right instead of going straight down. The algorithm will thus first try the path of only left, till it hits a wall and then right is not an option so it goes down one, and then continues right till it gets stuck in the lower right corner of the upper left "chamber" in the maze. Then after that it will go to just before it enters that "chamber" and goes down and tries Left, Right, Down in that order. And that causes it to expand a lot of useless nodes. You could change the order in which DFS tries to find a path but then it will perform terrible in other mazes. As follows from the cost, DFS does not find the shortest path.
UCS works just like BFS because the cost of every move is the same and therefore the priorityqueue of UCS is the same as the queue of BFS.
A* expands the fewest nodes because it does not try every possible path. Because the goal is Left-Down from the start state it will only go left and down, and that is why it does not have to expand as many nodes.
A*, UCS and BFS all find the same path because they try to go down before they go left.
There is in this example not one shortest path, the algorithm could choose to go left first, then all the way down to the bottom and then left to the goal and this would give the same cost. If the order of Left, Right, Down, Up was random at every level of BFS it would probably get a path with a lot of 90 degree turns instead of a few straight lines like it gets now, but this would not affect the cost of the path it finds.

7 Exercise 7

7.a

Both files contain a class which can be extended to implement an agent. Both of these classes implement a different base class / interface.
The one in file *valueIterationAgents.py* is meant to be implemented for agents which estimate the Q-values and values for an environment using a Markov Decision Process, which is done before acting. The other in file *qlearningAgents.py* is meant to be implemented for agents that estimate Q-values from policies rather than from a model.

7.b

- In reinforcement learning transition probabilities are unknown. Which is one of the reasons why MDP can't be used in such cases.
- In reinforcement learning transition rewards are unknown. These are learned with experience and can differ every iteration.

7.c

GridWorld implementation differs in how terminal state and rewards is handled. In contrary to slides the rewards are set per grid cell basis. Which means a cell can give a reward and not a be a terminal state (although only action possible in a cell with a reward is a terminal state).
Terminal states do not belong to any cell, nor do they give any rewards. They are implemented as actions. These are only to indicate when the iteration should break. In the slides they behave as a cell, but a final one.

7.d

"A terminal state is a state that once reached causes all further action of the agent to cease."
Source: `http://burlap.cs.brown.edu/tutorials/bd/p1.html`.

Both slides and implementation use this definition. Since no actions can be taken once a terminal state is reached.

8 Exercise 8

8.a ex8-a

Goal-directed task or goal oriented behaviour.

8.b ex8-b

Yes, assuming the terminal state gives a positive reward it is strongly preferred to reach that terminal state as soon as possible. If you look at planning a couple of actions, you would like to perform as few actions as possible. This preferred behaviour can be stimulated by using a bigger discount, this way one extra action will lead to a significant lower reward, and therefore the algorithm will choose the plan with the fewest actions. If there is not even a single terminal state, the use of a discount is absent. The algorithm can take action after action for infinite time and so the total reward will be infinitely great.

9 Exercise 9

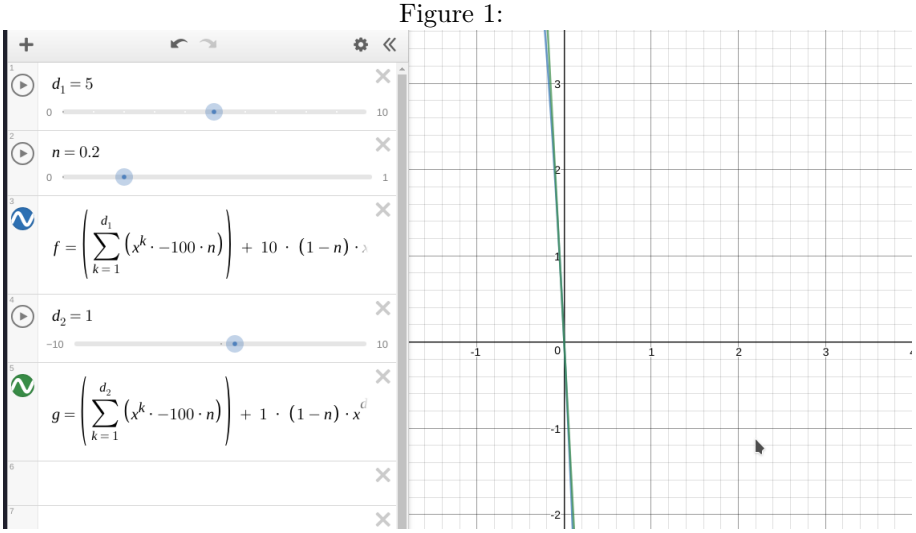
9.a

To solve these questions we have defined a few functions which represent the values of the 2 given options. Go left and don't cross the bridge and cross the bridge. We have chosen X axis to represent the discount and Y axis to represent

value / score of the given function / option.

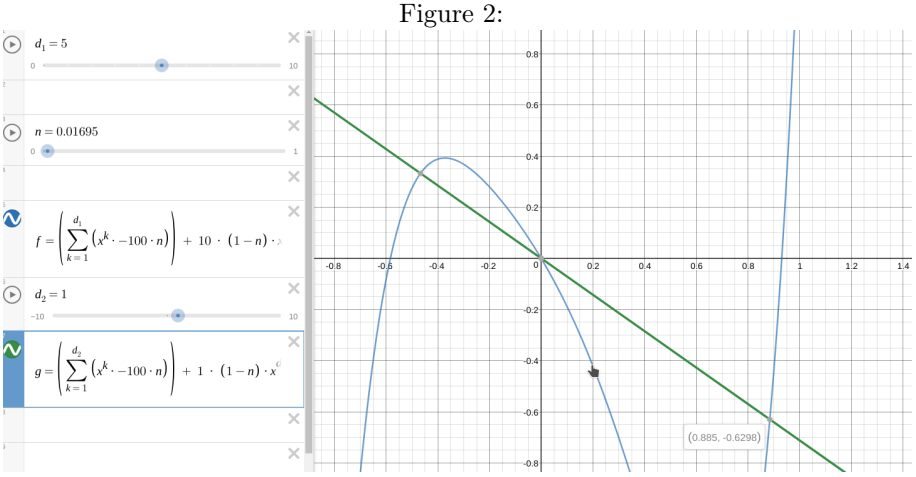
- Agent will never cross the bridge is only discount is changed. See Figure 1. The green lines is always on top.
- AgentAgent is able to cross the bridge when discount is 0.9 and noise is lower than 0.01695. This is because the the value functions cross below discount of 0.9 around this value. Figure 2

Figure 1:



Blue = Cross the bridge values
Red = Take reward on the left
Noise = 0.2

Figure 2:



Blue = Cross the bridge values
Red = Take reward on the left
Noise = 0.01695

9.b

We have changed the noise parameter, since it is not possible to do this by only changing the discount. See explanation above. We have chosen 0.01695 as noise value since it lies right on the border of possible noise values which allow agent to cross the bridge. Values lower than this one will give same results and values above* won't.

10 Exercise 10

10.a

a ($\gamma = 0.2$, n = 0, r = 0)

b ($\gamma = 0.5$, n = 0.1, r = -1)

c ($\gamma = 0.5$, n = 0, r = 0)

d ($\gamma = 0.5$, n = 0.25, r = 0)

e ($\gamma = 0$, n = 0, r = 0)

10.b

For this exercise we have plotted formulae to compute the value / score of different path options. We have chosen X axis to represent the discount and Y axis to represent value / score of the given path option.
Legend for the following figures:

n Noise parameter

l Living reward parameter

Red line Prefer the close exit (+1), risking the cliff (-10)

Blue line Prefer the distant exit (+10), risking the cliff (-10)

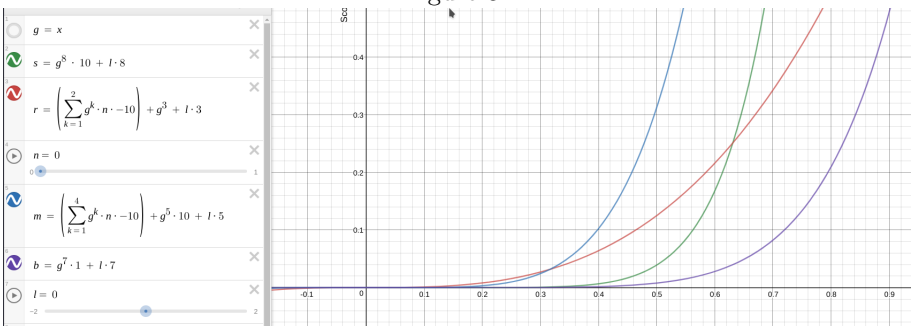
Green line Prefer the distant exit (+10), avoiding the cliff (-10)

Purple line Prefer the close exit (+1), but avoiding the cliff (-10)

10.b.1 a

Because noise is 0 this path is considered safe. Therefore agent would go along the bridge. If discount is low enough the shorter path will be preferred. This preference is amplified if living reward is negative.
We have chosen for for discount of 0.2 since that allows other parameters be unchanged. See Figure 3 the red line.

Figure 3:



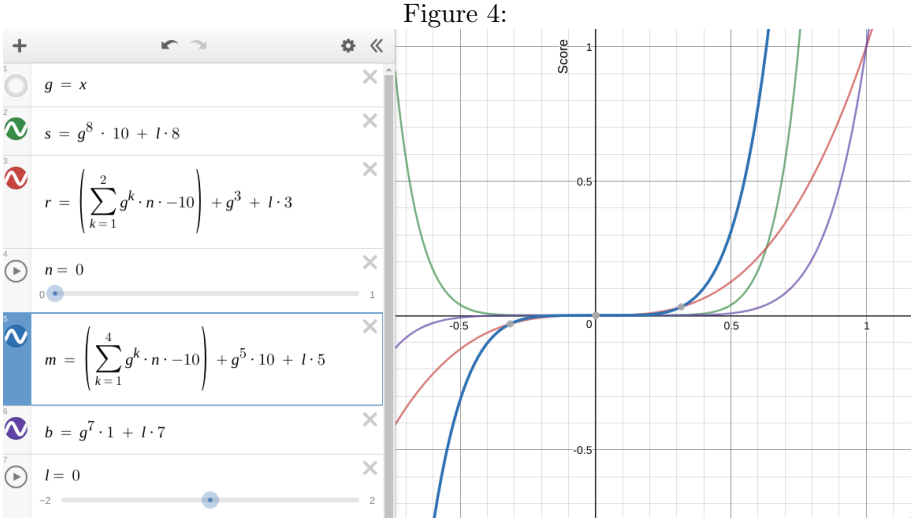
10.b.2 b

Here we make living reward negative to discover long routes. Along with this we make noise big to discourage risk. With this we see that purple line has a higher value then other paths with discount of 0.5, noise of 0.1 and living reward of -1.

10.b.3 c

Here we set noise to 0 to remove any risks and we make long term reward (discount) tempting enough to pick the longest path along the bridge. We can see with noise and living reward set to 0 that the blue line has the higher score than other paths at discount of 0.5. See Figure 4 the blue line.

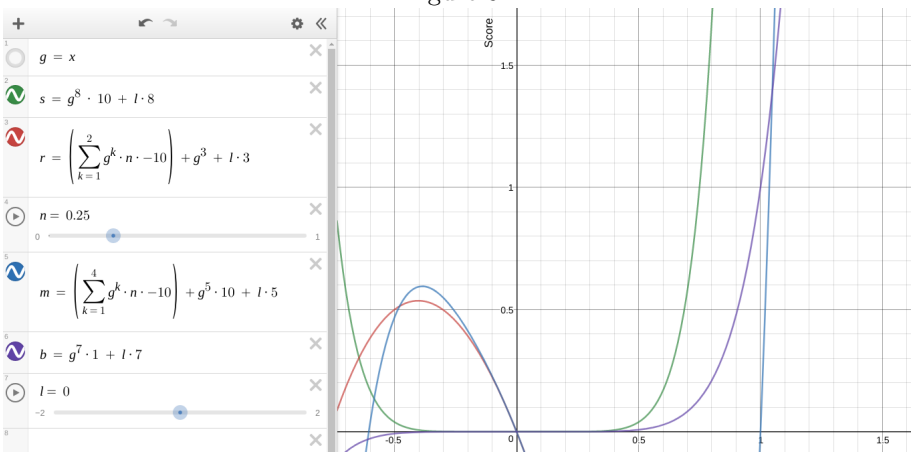
Figure 4:



10.b.4 d

Here we set noise high enough to discourage risk and pick the right discount to encourage long term reward. At discount of 0.5 green line has the highest value. See Figure 5 the green line.

Figure 5:



10.b.5 e

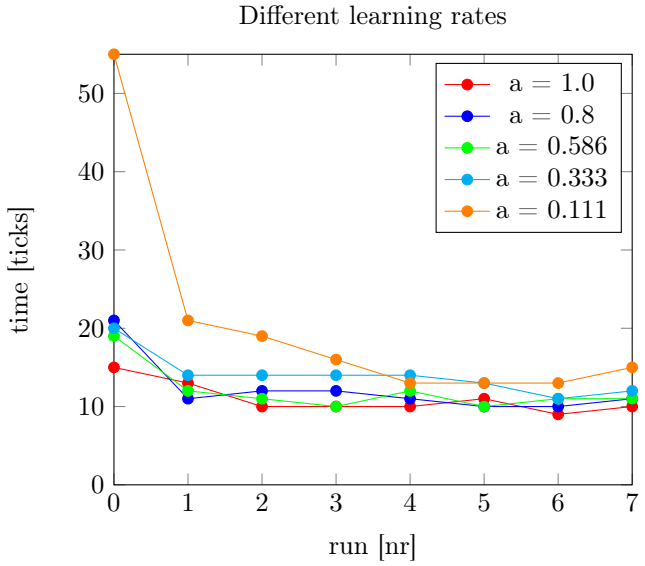
Here we put everything at zero. Because reward is getting multiplied by gamma and gamma is set to 0, every Q-value will also be zero and thus the episode would never terminate.

11 Exercise 11

notes a = learningrate e = exploration/mutation rate y = discount rate
newQ = (1 - a)oldQ + (a)sample sample = reward + y(futureQs) if random < e: random step else best step

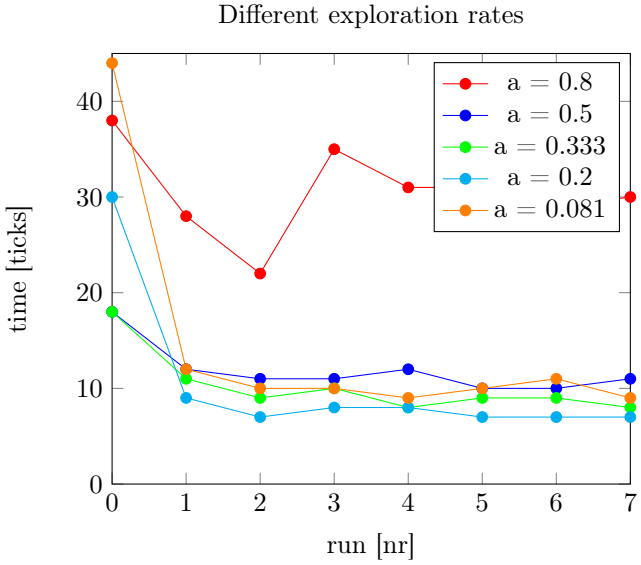
11.a

Testing method(also for 11.b): I let the the crawler walk until he disappears from the screen at the end. That is one run. I measure the run time in ticks. I take 8 runs. The runs are consecutive.



The higher the learningrate, the faster the crawler's policy converges. But, lower learningrate's do not slow the process of learning down much, till a certain point. After that the learning speed is affected significantly, as you can see with the orange line in the graph. Eventually they will converge, sooner or later.

11.b



The exploration rate let the crawler take random actions to discover new things. A high exploration rate will never be fast. Even if the agent has tried many things and learned the optimal policy from it, it will still perform random actions instead of the learned ones. As you can see with the red line, the time of completion stays high. Lowering the exploration rate let the crawler move faster due to less random moves, but if we drop it to low it will learn less fast. We can see that lower rates do better, for all except the lowest value here. We

can see that the orange line is consistently above the cyan line, breaking the trend. Many more tests may reveal a optimal learning rate somewhere between cyan and orange.

11.c

high a(learningrate), low e(explorationrate) = learns fast from actions performered, but tries little new things. Since he does not know anything to start with, he needs to randomly come across a good action at the right time. The low e value holds the crawler back from learning as fast as he could.

high a, high e = learns fast from actions performered, and also tries many things(random) out. This combination lets the crawler learn rapidly at the beginning. However, when he has learned to walk, the high e holds him back from performing at his best. Because a high e brings a high probability for a random action, his behaviour is quite random. Now he knows the right actions, this is not needed anymore. Many steps are now wasted, the random actions do not contribute walking faster and could even push him back at bit.

low a, low e = This is a very slow learning combination. There is a low chance that he will randomly do something. And if he randomly does something right, it picks it up slow because of the low a. But when it, eventually, learned how to walk he performs great. He will not disrupt his walk with randomness, and random bad actions do not influence him because of the low a.

low a, high e = This one is weird. He tries many things, but learns from them slowly. When he learned to crawl, randomness still disrupts his speed. But this time it will not affect the learned policy so much.

The best thing is to have a high e and a high a, so he learns fast. He will start walking in no time. Then you can decrease the e and a slowly as he gets better. At the end set them to zero as the learning has finalised. Then there will be no randomness to disrupt the crawl, and the crawler will be at his fastest.

11.d

11.e

12 Exercise 12

13 Exercise 13