KI Project B

Martijn Dekker 6013368
Egor Dmitriev 6100120
Cody Bloemhard 6231888

December 12, 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

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 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 well that it does not matter what problem it is used on.

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

6 Exercise 6

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, 535 nodes expanded, cost of 54, found in 0.0 seconds, chooses down over left. 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. 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. A* and UCS expand the fewest nodes because they do not try every possible path. Because the goal is Left-Down from the start state they will only go left and down, and that is why they do 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

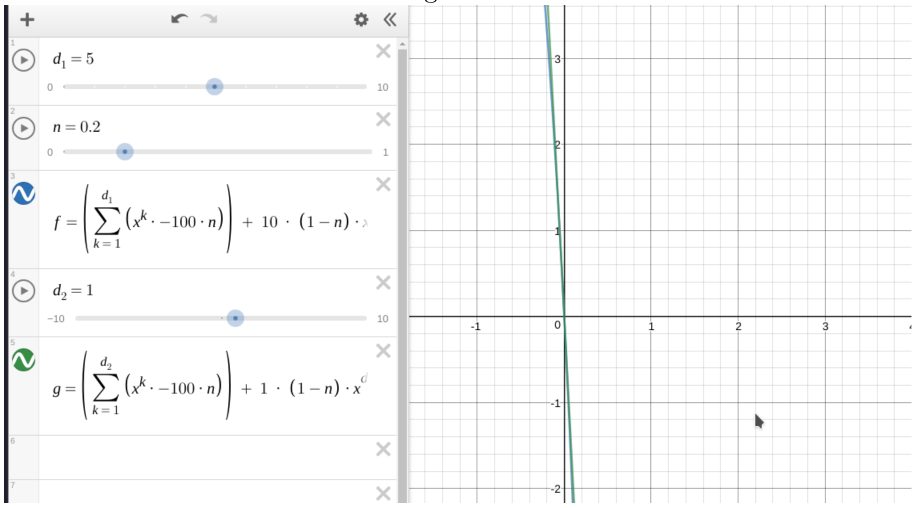
9 Exercise 9

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:



$d_1 = 5$

$n = 0.2$

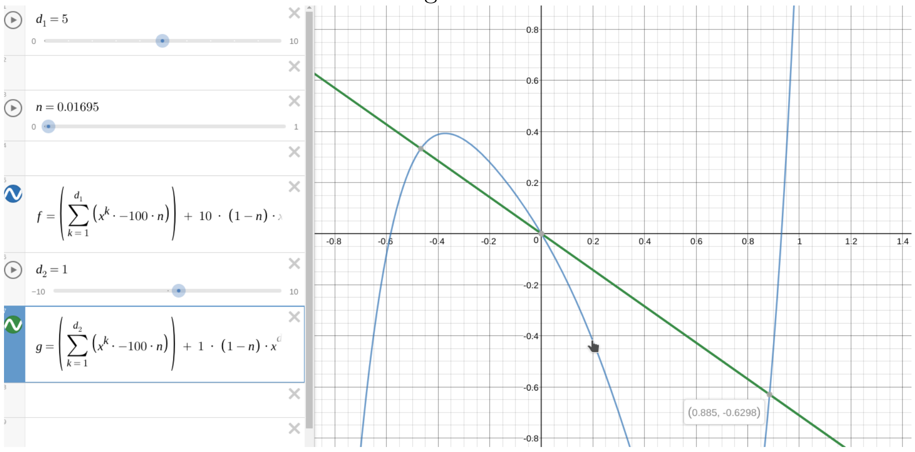
$f = \left(\sum_{k=1}^{d_1} \left(x^k \cdot -100 \cdot n \right) \right) + 10 \cdot (1 - n) \cdot$

$d_2 = 1$

$g = \left(\sum_{k=1}^{d_2} \left(x^k \cdot -100 \cdot n \right) \right) + 1 \cdot (1 - n) \cdot x^{d}$

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

Figure 2:



$d_1 = 5$

$n = 0.01695$

$f = \left(\sum_{k=1}^{d_1} \left(x^k \cdot -100 \cdot n \right) \right) + 10 \cdot (1 - n) \cdot$

$d_2 = 1$

$g = \left(\sum_{k=1}^{d_2} \left(x^k \cdot -100 \cdot n \right) \right) + 1 \cdot (1 - n) \cdot x^{d}$

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 (γ = 0.2, n = 0, r = 0)

b (γ = 0.5, n = 0.1, r = -1)

c (γ = 0.5, n = 0, r = 0)

d (γ = 0.5, n = 0.25, r = 0)

e (γ = 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)

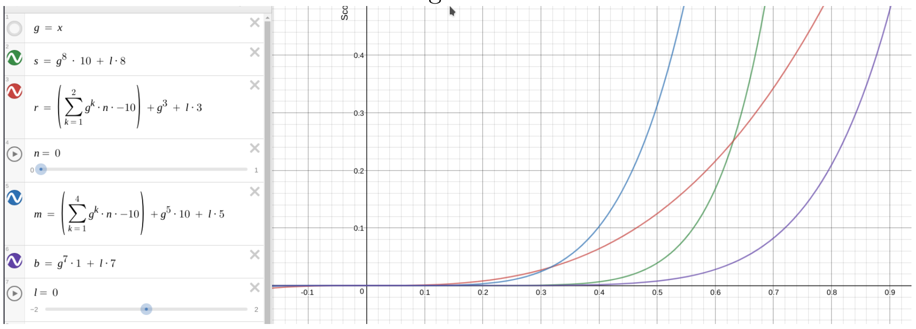
Green line Prefer the distant exit (+10), avoiding the cliff (-10)

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:



$g = x$

$s = g^8 \cdot 10 + l \cdot 8$

$r = \left(\sum_{k=1}^{2} g^k \cdot n \cdot -10 \right) + g^3 + l \cdot 3$

$n = 0$

$m = \left(\sum_{k=1}^{4} g^k \cdot n \cdot -10 \right) + g^5 \cdot 10 + l \cdot 5$

$b = g^7 \cdot 1 + l \cdot 7$

$l = 0$

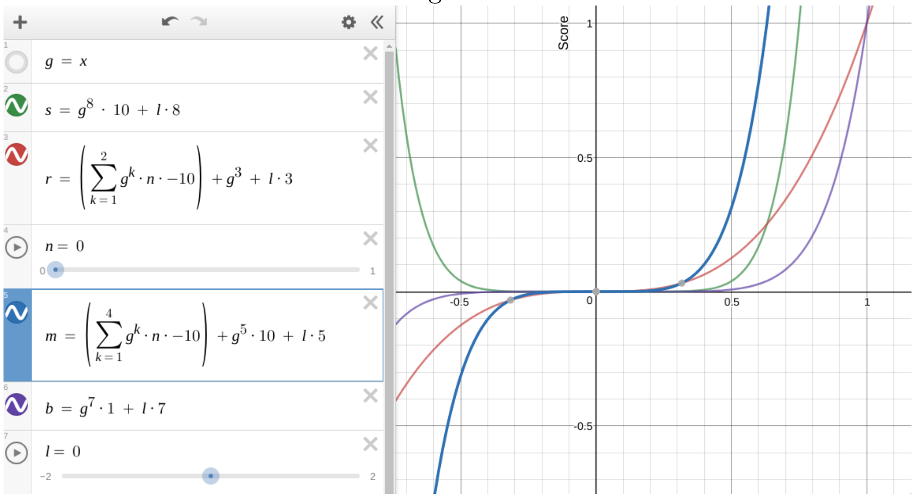
10.b.2 b

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:



$g = x$

$s = g^8 \cdot 10 + l \cdot 8$

$r = \left(\sum_{k=1}^{2} g^k \cdot n \cdot -10\right) + g^3 + l \cdot 3$

$n = 0$

$m = \left(\sum_{k=1}^{4} g^k \cdot n \cdot -10\right) + g^5 \cdot 10 + l \cdot 5$

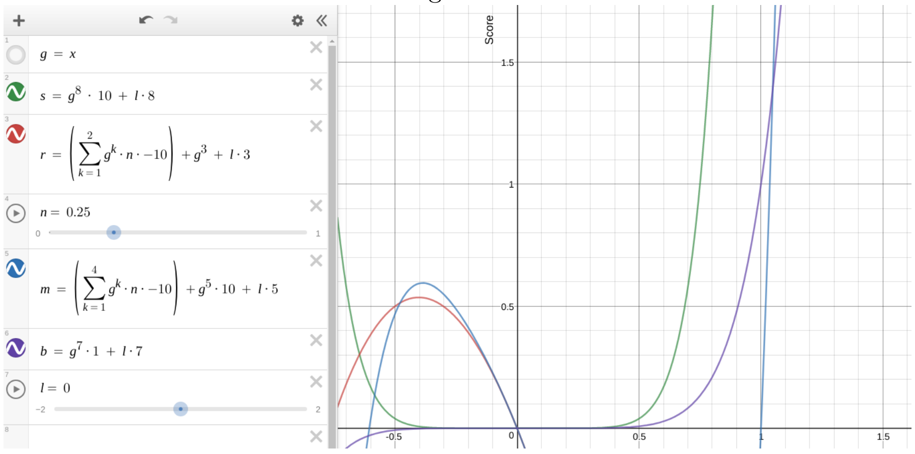
$b = g^7 \cdot 1 + l \cdot 7$

$l = 0$

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:



$g = x$

$s = g^8 \cdot 10 + l \cdot 8$

$r = \left(\sum_{k=1}^{2} g^k \cdot n \cdot -10 \right) + g^3 + l \cdot 3$

$n = 0.25$

$m = \left(\sum_{k=1}^{4} g^k \cdot n \cdot -10 \right) + g^5 \cdot 10 + l \cdot 5$

$b = g^7 \cdot 1 + l \cdot 7$

$l = 0$

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

raw:steps to complete/100 dif:|raw[i] - raw[i - 1]|

all std, a = 1.0 raw:15,28,38,48,58,69,78,88 dif:15,13,10,10,10,11,9,10 all std, a = 0.8(std) raw:21,32,44,56,67,77,87,98 dif:21,11,12,12,11,10,10,11 all std, a = 0.586 raw:19,31,42,52,64,74,85,96 dif:19,12,11,10,12,10,11,11 all std, a = 0.333 raw:20,34,48,62,76,89,100,112 dif:20,14,14,14,14,13,11,12 all std, a = 0.111 raw:55,76,95,111,124,137,150,165 dif:55,21,19,16,13,13,13,15

all std, e = 0.8 raw:38,66,88,123,154,185,214,244 dif:38,28,22,35,31,31,29,30 all std, e = 0.5 raw:18,30,41,53,65,75,85,96 dif:18,12,11,12,12,10,10,11 all std, e = 0.333 raw:18,29,38,48,56,65,74,82 dif:18,11,9,10,8,9,9,8 all std, e = 0.2 raw:30,39,46,54,62,69,76,83 dif:30,9,7,8,8,7,7,7 all std, e = 0.081 raw:44,56,66,76,85,95,106,115 dif:44,12,10,10,9,10,11,9

high a, low e = 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.

12 Exercise 12

13 Exercise 13