**Exercise 1**

1. Game State

The game state is the current state of the game, where pacman is, what he’s doing, where the ghosts and food are and in what state they are. It also contains the world of the game, the layout of the level, the score at a certain point in time.

The game state contains a lot of accessor methods to acces a GameStateData object. If one of the things in the level has to make a certain action, the corresponding method in an object gets called.

1. The Agent state describes the properties of an agent. Where an agent is in the level, its speed and its ‘configuration’, which is how it behaves (this can return pacmans behaviour or that of a ghost). It also includes if an agent is ‘scared’ or not, with a scared timer. This is for the ghosts to run away. The agentstate has a getPosition and getDirection method, which return these properties. There is a method def \_\_eq\_\_( self, other ):, which takes a self and an other, and replaces the self.configuration with the other.configuration, and does this for the scaredTimer as well. This is likely how the agentstate is updated.
2. A – III : similarly to a stack, the crate on top of the pallet is removed first, and new crates will be added on top. It is not possible to remove a crate from the middle of the pallet either.

B – II : a queue in code form works like the line in a supermarket (provided no-one is being polite and letting someone go before them), where the first one to get in line is the first one who will be helped by the cashier.

C – I : a hospital room works like a priority qeue, the person with the worst injury will be helped first.

1. StayEastSearch is intended to prefer a path through the eastern part of a level. This is represented in the code by using uniform cost search and setting the cost function costFn used to calculate the cost of a step to lambda pos: .5 \*\* pos[0]. CostFn is given (x,y) as input. This formula take the x component and increases the cost by calculating ½ to the power of x. The result for a higher x (which is the east side of the level) is much lower than the result for a lower x. Uniform cost chooses the step with the lowest cost, and will thus go for the fields in the east side of the level.

StayWestSearch is intended to prefer the west side of a level. Very similar to StayEastSearch, the difference is in the costFn. For StayWestSearch, it is lambda pos: 2 \*\* pos[0]. This achieves the opposite of StayEastSearch because for higher values of x the cost 2 to the power of x will be a lot bigger than for lower values of x.

Both these agents have an exponential increase in cost, so the differnence between costs in one half of the level and the other will be absolutely huge and both will stick to their preferred side of the level.

**Exercise 2**

Our general structure is pasted below.

**(4)** We use a fringe/frontier, which can be any of the things found in util.py, depending on the a search algorithm.

**(5-6)** For the solution and the tracking of the explored nodes we use lists.

We fill the solution with Directions which pacman can use to move around.

For the explored nodes, we use a list named closed. A list comes with a lot of useful functions, such as checking if a node is already in there or not.

**(9)** We represent nodes as (currentLocation, (previousLocation, directionFromPrevToCurr)). This makes it easier for us to construct a solution when the algorithm finds the goal. For something as uniform cost search, cost would be included as well, making nodes look like :  
(currentLocation, (previousLocation, directionFromPrevToCurr, costForThisStep)).

**(11)** There’s and empty tuple named goal. This will later be used to store the goal node, and it is also used to construct the solution list.

**(13-24)** Our program then enters a while loop, executing its steps as long as the fringe is not empty.

The pop() action is called for the fringe.

A check is made to see if this node is already in the closed list. If not, it is added. closed is changed into a dict here, which Kevin is changing

A second check is made to see if the node is the goal node. If this is the case, the goal is set to be current node we are working on. The while loop is now interruped and the solution list will be build.

If not, we get the successors of the current node. These are added to the fringe using the push() action. This means there will probably be duplicates in the fringe. Because we check the closed list before working on a node, these will be discarded when they are popped from the fringe.

**(26)**  The closed list is converted to a dictionary

from game import Directions

s = Directions.STOP

fringe = ???

closed = []

solution = []

startLocation = problem.getStartState()

node = (startLocation, (startLocation, s))

fringe.push(node)

goal = ()

while not fringe.isEmpty():

node = fringe.pop()

if node[0] not in dict(closed):

closed.append(node)

if problem.isGoalState(node[0]):

goal = node

break

successors = problem.getSuccessors(node[0])

for successor in successors:

fringe.push((successor[0], (node[0], successor[1])))

closed = dict(closed)

while goal[0] in closed and goal[1] != s:

solution.append(closed[goal[0]][1])

goal = closed[goal[0]]

solution.reverse()

solution = solution[1:]

return solution

**Exercise 3**

1. Our solution is complete. Because of the graph search, infinite depth in trees is not a problem. The algorithm will explore the entire tree of nodes that it generates. A solution will be found.
2. It is not a least cost solution. Because the algorithm ‘dives’ into the tree as far as it can before trying a different way, it might return a terribly long, but working, path. A shorter, more efficient path might exist. The cost of steps is also ignored, which is also a problem.
3. **Looking at the expanded states in this solution, it is what we would expect of dfs. With its FIFO queue, it should keep going further down the same path (out of the list of options it chooses the first option at each branch here).** Pacman does not visit all the explored tiles on the way to his goal if the goal was not found in one try. Otherwise, there is a straight path of explored nodes through the tree of nodes.

rev\_solution:

"0:A->B1

0:B1->C

0:C->D

0:D->E1

0:E1->F

0:F->G"

rev\_expanded\_states:

"A B1 C D E1 F"

start\_state: A

goal\_states: G

A 0:A->B1 B1 1.0

A 1:A->C C 2.0

A 2:A->B2 B2 4.0

B1 0:B1->C C 8.0

B2 0:B2->C C 16.0

C 0:C->D D 32.0

D 0:D->E1 E1 64.0

D 1:D->F F 128.0

D 2:D->E2 E2 256.0

E1 0:E1->F F 512.0

E2 0:E2->F F 1024.0

F 0:F->G G 2048.0

B1 E1

^ \ ^ \

/ V / V

\*A --> C --> D --> F --> [G]

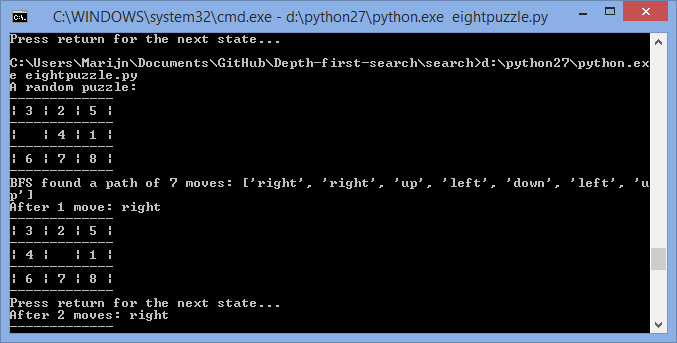
\ ^ \ ^

V / V /

B2 E2

**Exercise 4**

1. Our solution is complete. If an answer exists, it will be found. Because of the graph search an infinite tree does not pose a problem.
2. Our solution is a least cost solution if all steps have an equal cost. Because bfs expands all nodes at a certain depth before continuing its search deeper in the tree, the shallowest solution will be found. If all costs are equal, this is also the least cost solution.
3. Our code works for the eightpuzzle as well, we did not have to change anything to get this working

****

**Exercise 5**

1. SearchAgent is intended to run a given search algorithm such as dfs or bfs. It will find the right functions and problems based on the algorithm it has to employ. SearchAgent sets no costFn function, so it will likely use the standard function. SearchAgent can run A\*, and thus can handle heuristics and will use these to calculate costs.

StayEastSearch is intended to highly prefer a path through the east side of the level. Its cost function achieves this by using an exponential function which return very small values for high x values, the east side of the board, and higher values for small values of x, the west side of te board. The cost function is

0.5 ^ x.

StayWestSearch is intended to prefer a path through the west side of the board. It achieves this by using an exponential cost function which returns very large costs for high x values and lower costs for lower x values. The cost function is:

2 ^ x.

1. python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

Has a path cost of 68. With a step cost that is always 1, pacman has made 68 steps before he reached his goal.

python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

Has a path cost of 1. The step cost in this algorithm, for steps in the east side of the level, is very small. The exact cost of 1 was very surprising though.

python pacman.py -l mediumScaryMaze -p StayWestSearchAgent

Has a path cost of 68719479864. With an x that goes over 30 and a cost formula that has ~2^30 in it a few times, a cost like this is not a surprise.

The huge differences in values is not a surprise since we are dealing with exponential functions here.

**Excercise 6**

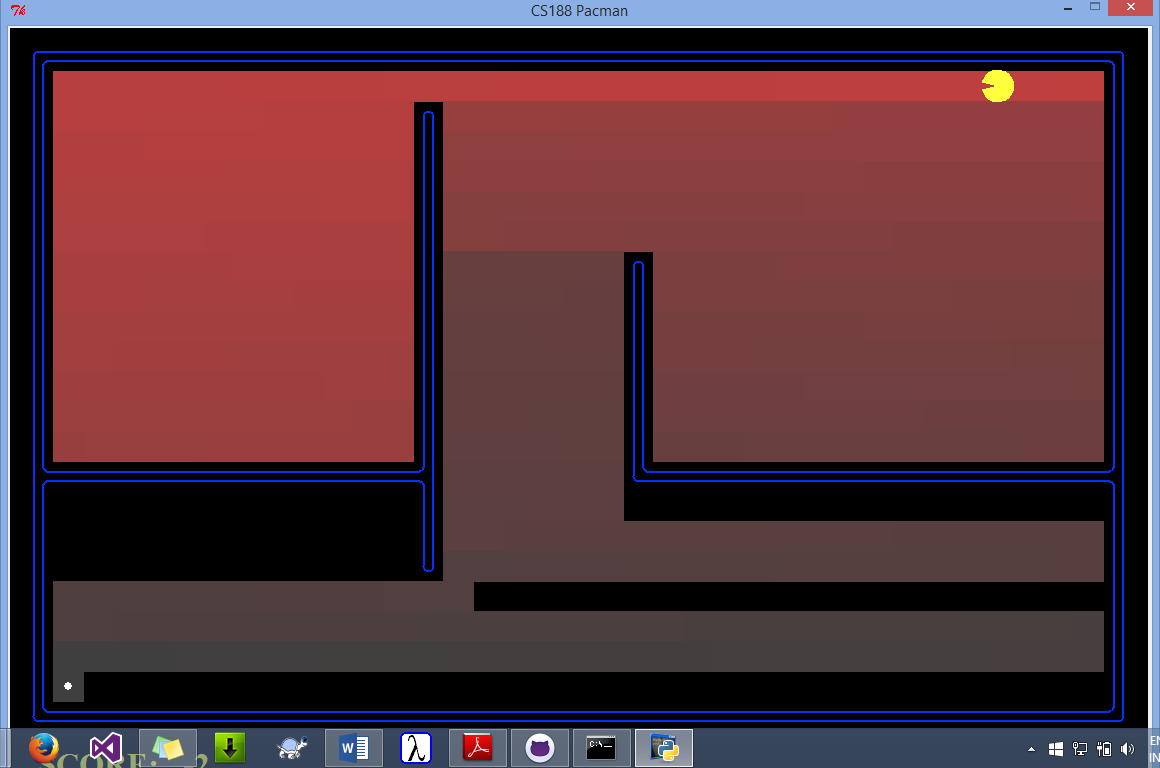
Dfs moves down in a zig-zag pattern, and has a path cost of 298. It expanded quite a few nodes. It seems to go left first, than right, moving down as the last option. It seems the successors are passed in this order.

Bfs costs 54, expand almost all nodes, and the coloring shows that it expanded the nearby nodes first, as expected. It found a very efficiënt path through the maze though.

Ucs shows the same expansion and path as bfs. Since all costs are 1, it expands in the same way as bfs does. Bfs also finds the shallowest path, which is also the cheapest path here, which ucs also finds.

A\* follows the same path, but expands less nodes. The Manhattan heuristic is the cause of this. It ignored the lower-right side of the level. The heuristic has it search in the directinos left and down.

It’s worth mentioning that there is a section in the top left corner of the level, which is a huge empty cube which leads nowhere. The way algorithms treat this cube is interesting. Dfs searched this cube before going anywhere else, because it tried going left at first. Bfs and ucs also searched throughout this cube, but they expanded everything else as well. A\* examined the first cube it was in first, but did eventually check out the cube in the top left as well.



**Exercise 7**

Our state contains an extra tuple of four bools, which is (False, False, False, False) by default. These bools are used to keep track of which corners have been visited.

The heuristic we uses the unseen/unvisited corners in its calculations, thus the state keeps track of this information.

**Exercise 10**

1. We use bfs to find the path to the closest dot. We simply call the bfs we implemented for question 2.