Project B

Students:
Bernd van den Hoek (5895391), Maaike Galama (5987857), Jesper Steensma (5750458)

The following report contains the results of Project B of the pacman projects. The files edited are:

- *searchAgents.py*
- *search.py*
- *util.py*

The reason for editing util.py is explained in the answer to exercise 3. Since this was listed on the Berkeley website under 'files you might want to look at', we figured this was ok. The best place for the code from exercise 3 is in util.py, so that is where we put it, to keep the code clean.

*All changes in code are preceded by a line of commentary, starting with characters ####**
This way, by doing a global search for #### in our code, all changes can be found relatively easily.*

Exercise 1

a. The Game state class is an interface in which all data can be retrieved. It has methods, which search in the other classes for data you want to have. The state of the game updated each move by calling the generateSuccessor function, which checks all rules and conditions for a move to be legitimate. This function will raise an exception when a move can not be made or tell you if you are game over. When all the checks have been passed it will return the current gameState.

b. The agent state class is a class that stores all the data for each agent and handles the creation of the agent in the beginning. The information about the movement is stored in the configuration class. The configuration class stores position, direction and the speed of the agent. This configuration class also has a generateSuccessor class, which creates a new configuration class for every move. This replaces the old configuration.

c. A. Stack ---------- III. Pallets of beer crates
B. Queue ---------- II. supermarket check-out line
C. Priority Queue ---------- I. Hospital waiting room

Exercise 2

a. Each type of search from Q1 - Q4 searches in the same way. It keeps an open list which record the nodes that are seen but not yet visited, and a closed list which records which nodes have been visited. The order in which the nodes in the open list are visited depends on the search method, and this order can be implemented by using a different data structure for the open list.
The search pops the next node from the open list data structure, gets all this node's successors, and if these successors are not on the closed list, they are put back into the same

data structure. This goes on until the goal test returns a positive result. Then, each state has a record of his parent state and the action it took to get to the state, and this can be read backwards to create the path needed to get to the goal state.
Thus, the only variance between these search methods is the data structure for the open list, and the application of a search heuristic (if any).
To exploit this, we adapted the push functions in the data structures in Util.py, so that they all take a node, priority, and a heuristic as arguments. By default, this heuristic is the nullHeuristic. The queue and stack do not use the priority argument, but by adding this to all push() methods we were able to create one genericSearch() function in search.py, which is used by all search functions to execute the search. The different implementation of the search all call genericSearch with a different data structure as argument for the open list, and, in the case of A*, a heuristic function.

b. For tree search, the search methods are essentially the same, except for the fact that we don't need to keep track of our visited nodes in a closed list. We could implement this by adding a boolean isTreeSearch to the genericSearch method, and then changing the line `if successor not in closedList:` to: `if isTreeSearch or successor not in closedList:`
However, this would mean that we are performing unnecessary work when doing a tree search, namely keeping track of nodes in an open list, which is exactly where the tree search could give us a performance benefit. Therefore, we would prefer to make a distinction between the two search methods by using a genericTreeSearch and a genericGraphSearch function.

Exercise 3

a. A search strategy is complete when it always finds a solution if one or more solutions exist.

b. The search will only reach completion when there is a limited amount of nodes he can reach, else it is possible for him to follow a path for unlimited length, while the end goal is not on there.

c. The solution is most likely not the most cost efficient path, this is because we do not explore all other possible paths, but only the one we found first, which is completely dependant on the order of actions returned by the getSuccessors() function. There might be a lot of shortcuts in the paths we did not explore.

d. The getSuccessors() function in the positionSearchProblem returns actions in the order north, south, east, west. Since the west action is the last one to be added to the openlist, and the openlist is a stack, we expect that the squares west of each state are first to be explored. When we look at the search pattern for the BigMaze, we see this is exactly what is happening.
The search first explores all squares west of the parent state, then all squares east, then south, and finally north. This means on each intersection, the squares that can be reached

from going west on this intersection, are explored. If the search reaches a dead end after looking west, DFS backtracks, and all squares east of the intersection are visited, and so on. Since the square to the North is the last to be explored, all other directions must have lead to dead ends. This also means that whenever pacman turns North on an intersection (and there exists a path to the goal state), this North action will lead to the goal. Or, alternatively, pacman will never turn North in an intersection if that direction does not lead to the goal Pacman only visits the squares that are on the found path to the goal, because when DFS reaches a dead end, it will not push more Nodes to the stack. This is because we keep track of the closed list. The next node taken from the stack will be a node that can be reached from the last intersection that was explored. Thus, the path to each Node does not contain any squares that lead to a dead end.

Exercise 4

a. Breadth-first search is complete. If we have a search tree with a solution at depth D, it will find that solution after BFS has checked all D - 1 layers in the search tree. After that it will check every node in layer D and therefore it will always find the solution

b. It will be the shortest path if we do not care about the cost between nodes or all costs are equal. If this is not the case BFS might find a solution which is suboptimal.

c. Our pacman BFS will return a solution to the eight puzzle if it is properly defined as a search problem. This means that it will need to have a successor function and a goal state. If this is true (which it is for the given eight puzzle), our BFS will be able to find a solution for it if one exists.

Exercise 5

a. *SearchAgent:* The SearchAgent is a generic search agent that solves the positionSearchProblem(by default). The search problem will give a certain cost to each action and this cost of action is used by the search method. Since all actions have the same cost, the order in which nodes are explored depends on the search method used.
StayEastSearchAgent: This search agent is a derivation of the SearchAgent, which passes an extra cost function to the search problem. The intention here is to keep the agent as long as possible on the east side of the map. This is done by prioritizing the states with the most eastern coordinates, by implementing the cost function. By assigning the cost value $½^x$, the larger the x value, the smaller the cost of that state. The coordinates on the board increase from left to right, so the larger coordinates are on the east side, meaning the search will assign lower costs to squares on the east.
StayWestSearchAgent: This works on the same principle as the StayEastSearchAgent, but now for the west side of the board. The cost function returns 2^x, so when the value for x is lower (which it is on the west side of the board), the cost function returns a lower value.

b. For bigMaze:
SearchAgent: 620 Nodes
StayEastSearchAgent: 639 Nodes
StayWestSearchAgent: 497 Nodes

The Maze runs from east to west, so the west prioritizing algorithm does not take many turns to the east, which would be wasted energy. The east prioritizing does this and that costs him the extra nodes, where the generic search agent expands an amount of nodes in between the go-west- and stay-east- agents, as expected.

Exercise 6

An overview of the number of nodes expanded and found path length per search algorithm for openMaze is shown in the table below:

Search type	Nodes expanded	Found path length
BFS	682	54
DFS	576	298
A*	535	54
UCS	682	54

Depth first search:

All the solutions find the shortest path except for dfs. Dfs always expands the last node added to the openlist. First, this is always the node going east. Is this has been visited already, the next node added to the open list is the one going west. Because DFS has this preference for first going east, and otherwise west (and only then south and finally north) we can see pacman zigzagging down through maze. This takes a lot of useless steps. Because DFS first expands nodes deeper down the tree, and the first path to the goal is returned, there might be a path higher up the search tree which is a lot more efficient. Indeed, we can see this is the case with openMaze.

Breadth first search & uniform cost search:

The BFS and the UCS both expand the same amount of nodes; 682 nodes. UCS acts like a BFS search in this problem, because the cost for each step is constant (1). UCS uses a priority queue as its open list, but because of the constant step cost this turns into a normal queue. So UCS and BFS both expand all successors of a node, and then all successors of the successors, and so on. Varying the cost per step can change the behaviour of UCS. Both of these search algorithms find the shortest route, because all nodes in depth *d* in the search tree are reached with the same path length (*d*), so the first path found to reach the goal must be the fastest route.

Put differently: if there had been a shorter path to the goal of length *n*, it would have been reached by the search algorithm already at depth *n*, since all nodes at every depth are expanded.
It is still possible that here is another path with the same length and cost, but you will not reach that part of the node tree once you got your answer.

A* with Manhattan-distance:

A* expands the least nodes, 535, to find the shortest route in openMaze. This method saves a lot of node expansions, because it first expands the nodes that are closer to the search goal. The manhattan distance to the goal state is calculated for each node. The search engine then prefers expanding the nodes with the lowest distance.

Exercise 7

In the state representation for the corners problem we encode our current position, and the list of visited corners. Is is obvious why we need this information: we need to know our current position to find successor nodes, and to construct the path from the list of states once we have found our goal. We need to know which corners we have visited to know whether we have reached our goal, and to calculate the heuristic value. For example, if we have already visited the top-left and bottom-left corners, we need to know these have been visited, in order to know that it is now beneficial to move right.
Our heuristic for the corners problem did not need any more information than the abovementioned, so we did not encode any more information into our state.

Exercise 8

For the corners problem we calculate Manhattan Distance for visiting the unvisited corners in all possible orders. We choose the shortest possible manhattan distance from these calculation. This means that for each permutation of the set of corners, we calculate the distance from our current pacman position to the first corner, then from the first corner to the next, and so on. The complexity of this calculation is *O(n!)*, which is bad, but since there are always only four corners, complexity is *O(1)* in practice.
This heuristic will work, because the manhattan distance is the shortest possible distance between two points on a grid, thus the heuristic can never return a value higher than the actual cost.

Admissability:

To proof admissibility of our heuristic, we must proof that the value returned by the heuristic is always lower than or equal to the actual cost. It is clear that the manhattan distance is the shortest possible path between two points on the grid.
Lets call *P(c)* the set of all permutations of unvisited corners.
When we calculate the minimum manhattan distance over *P(c)*, we must end up with the shortest possible path between all the corners.
This can be proven by looking at a contradictory situation: imagine we somehow found a shorter path that visits all these corners, but was not in *P(c)*. The order in which the corners are visited

however, is a permutation of our corners, so therefore it must be in *P(c)*, contradicting this hypothetical situation.
Of course the distance from pacmans position to the first corner in the permutation must always be travelled, so this is added to the permutation distance. But this distance can never be more than the actual distance, so the heuristic is admissible.

Consistency:

To proof consistency we must proof that the value returned by our heuristic function *h(n)* must be smaller than or equal to *h(n') + C(n, n')*, where *n'* is a descendant of *n*, and *C(n,n')* is the true cost between *n* and *n'*.
Now we consider what happens when Pacman moves.
If Pacman moves in the direction of a corner which is predicted to result in the shortest path, the path between Pacman and this corner (the first unvisited corner) is reduced by one. The path length between that corner and the rest of the unvisited corners will remain the same, thus the total path is reduced by at most one, so $h(n') \geq h(n) - 1$, and since $C(n, n') = 1$, this means $C(n, n') + h(n') \geq h(n)$.

If Pacman arrives at a corner, this corner is removed from the list of unvisited corners and no longer considered in the heuristic. Let's call the distance from this corner to the next corner *d*. Now, $h(n') = h(n) - d$. However, pacman is now positioned in the corner, so the distance from pacman to the next corner is also *d*. This gives us $h(n') = h(n) - d + d = h(n)$.
Since the distance between the corner and pacmans previous position was 1, $h(n') = h(n) - 1$.
Adding to this the path cost $C(n, n') = 1$, we again get $(n, n') + h(n') \geq h(n)$.

Nodes expanded:

950 nodes in mediumCorners.

Exercise 9

For the food search problem we calculate the minimum spanning tree of a graph consisting of all unvisited food items, and the position of pacman. The nodes in this graph are the coordinates of pacman and the food items. The edges are the manhattan distances between these nodes. This algorithm is similar to what we used for Q6, the corner problem, but implemented in a different way. In our implementation for the corner problem, we calculated the minimum cost for all possible permutations of the search goals. Essentially, we ended up with a minimum spanning tree. However, because of the O(n!) complexity of this method, we needed something more efficient for the food search problem. Here we construct the minimum spanning tree over all unvisited nodes by using Prim's Algorithm. Using a priority heap, this algorithm runs in O(E log N) complexity, where E is the number of edges and N the number of nodes (since our graph is a complete graph, we have N(N-1)/2 edges, which is why unfortunately this algorithm will prove to be inefficient).

Admissibility:

To quote wikipedia; "A minimum spanning tree [...] is a subset of the edges of a connected, edge-weighted (un)directed graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.".
A graph without any cycles is fine in our scenario, since each node is connected to every other node, so we never need any cycles in our graph to visit all nodes.
Prim's algorithm has been proven to find the minimum spanning tree in a graph from an arbitrary starting point. After we calculate the minimum spanning tree, we find the shortest possible path from our current pacman position to any unvisited food in the grid, and we add this to our minimum spanning distance. This value is what our heuristic function returns.
It is easy to see that this heuristic is admissible, since:

- Manhattan distance gives us the shortest possible path between two nodes.
- The distance from pacman to the graph can never be less than the manhattan distance from pacman to the nearest food item.
- The distance that pacman needs to travel to visit all nodes can never be less than the distance to travel the minimum spanning tree, since the minimum spanning tree has been proven to be, well, the minimum spanning tree.

Consistency:

The proof for consistency for this heuristic is similar to the proof given in exercise 8. The A* algorithm ensures that, when using this heuristic, Pacman will move towards states containing less pieces of food.
Let's again call the current heuristic value h(n), and let the descendant of n be n'. Again, we need to proof that h(n) <= C(n, n') + h(n')
If pacman does not eat a piece of food, he will at least move towards the nearest piece of food. This reduces the heuristic value by at most 1, because the distance from pacman to the nearest food is decreased by one. So, h(n') >= h(n). Since C(n, n') is always 1, this means h(n') + C(n, n') >= h(n).
If pacman eats a piece of food, the minimum spanning tree of food consists of one less piece of food. The distance from this food to other elements in the tree must be 1, because Pacman was one step removed from eating the food, so h(n') = h(n) - 1
However, since we are removing a node from a minimum spanning tree, path costs to other nodes might increase, so h(n') will be *at least* h(n) - 1.
This gives us h(n') >= h(n) - 1. Again the step size is 1, so h(n') + C(n, n') >= h(n), which is consistent.

Nodes expanded:

Unfortunately our heuristic proved not to be effective. We expanded 17196 Nodes for trickySearch. During our attempts to find a consistent heuristic, we found several heuristics which expanded around 9000 nodes. However, these heuristics were not consistent.

Exercise 10

a. findPathToClosestDot will use breadth-first search with a AnyFoodSearchProblem. This problem has a isGoalState method which returns true if we are at the nearest food. The nearest food is calculated with the manhattan distance. This will be effective because the goal of the search is now always the closest food. Once this has been eaten, the next closest food is now the closest, thus pacman will continue on his way eating all the closest foods until all have been eaten.

b. Going to the closest dot will not be optimal in the following case (food indicated by o):

```
%%%%%%%%%%%%%%%%
%                %
% o    P oooooooo %
%                %
%%%%%%%%%%%%%%%%
```

A greedy algorithm will go all the way to the right first and then has to go all the way back to pick up the last piece of food. It is clearly better to pick up the piece of food on the left first

c. The problem with greedy search is that it only decides which dot is the best one to go to first. It does not care about the total solution, therefore it will not always find the best solution.

Exercise 11

Q1 - 3/3
Q2 - 3/3
Q3 - 3/3
Q4 - 3/3
Q5 - 3/3
Q6 - 3/3
Q7 - 1/4
Q8 - 3/3
Total = 22/25