Assignment 2 Description

Assignment 2 reconstructs Optional Homework on Search (based on the Search project brom Berkeley http://ai.berkeley.edu/), with the difference it is in Java and comes without fancy graphical interface.

The code for this project consists of several Java files, some of which you will need to read and understand in order to complete the assignment, and other you will need to modify. You can download all the code and supporting files as a zip archive .

Files you'll edit:	
`GraphSearch.java`	Where all of your search algorithms will reside.
`SearchProblem.java`	Where all of your formalisations of search problems and search heuristics will reside.
Files you might want to look at:	
`Pacman.java`	The main file that runs search algorithms for Pacman search problems.
`Maze.java`	The class describing maze with some useful functionality and a dedicated maze parser.
`Util.java`	Useful data structures for implementing search algorithms.

Welcome to Pacman

Download the code (`pacman.zip`), unzip it and change to the directory. You can compile the code from command line as follows:

```
javac @compile_list.txt
```

Once compiled, you should be able to run the code for a naive search strategy that always tells Pacman to go west and that has been implemented for you. This strategy allows Pacman to successfully arrive at the goal state in `testMaze`.

```
java Pacman -l testMaze -f gowest -a
java Pacman -l testMaze -f gowest -s
```

You can see the list of all options and their default values via:

```
java Pacman --help
```

Graph Search Pseudocode

As a reminder, a high-level pseudocode procedure for graph search is as follows:

```
Algorithm: GRAPH_SEARCH:
frontier = {startNode}
expanded = set()
while frontier is not empty:
    node = frontier.pop()
    if isGoalState(node):
        return solution
    if node not in expanded:
        expanded.add(node)
        for each child in expand(node):
            frontier.push(child)
return null
```

Note: the pseudocode does not make distinction between *search space states* and *search tree nodes* (search space state plus a sequence of actions to get to it from the start state). Your implementation will need to differentiate those. In particular, you will be expanding *search space states* so `expanded` contains *search space states*. But `frontier` contains *search tree nodes*.

Question 1 (5 points): Finding a Fixed Food Dot using Depth First Search

It's time to write full-fledged generic search functions to help Pacman plan routes!

Implement the depth-first search (DFS) algorithm in the `GraphSearch.depthFirstSearch` function in `GraphSearch.java`. To make your algorithm *complete*, write the graph search version of DFS, which avoids expanding any already visited states.

Important note: All of your search functions, if successful, need to return an object of `Solution` class (found in `Util.java`), which contains list of *actions* that will lead the agent from the start to the goal. These actions all have to be legal moves (valid directions, no moving through walls). If no solution has been found, the search functions need to return `null`.

Important note: Make sure to **use** the `Util.Stack`, `Util.Queue` and `Util.PriorityQueue` data structures provided to you in `Util.java`! These data structure implementations provide all functionality you should need for implementing search functions (`pop()` and `push()`).

Hint: Implement the `Node<S,A>` data stucture in `GraphSearch.java` for the objects you are going to store in the frontier. At the least, it needs to store the current state in the search space and the sequence of actions to get to that state. For A* you can store the path cost unless you want to compute it each time.

Hint 2: You can use the method `expand` in `SearchProblem` ro retrieve a collection of all successors of a state, where each element describes a next state, the action to get there

and the cost of this action. For each successor you will need to create a new node according to your data structure `Node<S,A>` and push it onto the frontier.

Hint 3: Each algorithm is very similar. Algorithms for DFS, BFS, and A* differ only in the details of how the frontier is managed. So, concentrate on getting DFS right and the rest should be relatively straightforward. Indeed, one possible implementation requires only a single generic search method which is configured with an algorithm-specific frontier, see the function `GraphSearch.graphSearch` declaration. (Your implementation need *not* be of this form to receive full credit).

Hint 4: If you are not familiar with generics in Java, do not worry, you are not expected to do anything fancy with them. For examples of classes with generics, have a look at `Util.Stack`, `Util.Queue`, `Util.PriorityQueue`, `SuccessorInfo` , `Solution` in `Util.java`. and `PacmanPositionSearchProblem` in `SearchProblem.java`. This is as much as you would need to know about generics in this project.

Hint 5: Make sure to create a new object for actions rather than reusing the old one each time you create a new node. Remember the difference between references and objects.

Your code should quickly find a solution for `PacmanPositionSearchProblem`:

```
java Pacman -l tinyMaze -s
java Pacman -l mediumMaze -s
java Pacman -l bigMaze -s
```

Hint: If you use a `Util.Stack` as your data structure, the solution found by your DFS algorithm for `mediumMaze` should have a length of 130 (provided you push successors onto the fringe in the order provided by `expand`; you might get 246 if you push them in the reverse order). Is this a least cost solution? If not, think about what depth-first search is doing wrong.

Question 2 (5 points): Breadth First Search

Implement the breadth-first search (BFS) algorithm in the `GraphSearch.breadthFirstSearch` function in `GraphSearch.java`. Again, write a graph search algorithm that avoids expanding any already visited states. Use appropriate data structure for the frontier. Test your code the same way you did for depth-first search.

```
java Pacman -l mediumMaze -f bfs -s
java Pacman -l bigMaze -f bfs -s
```

Does BFS find a least cost solution? If not, check your implementation.

Question 3 (5 points): A* search

Implement A* graph search in the empty function `GraphSearch.aStarSearch` in `GraphSearch.java`. A* takes an instance of `SearchHeuristic`. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The `NullHeuristic` in `SearchProblem.py` is a trivial example.

When using `Util.PriorityQueue` data structure for implementing the frontier, you can pass to its constructor an object of anonymous `Comparator` in one of the following fashions:

```
Util.Frontier<Node<S,A>> frontier = new Util.PriorityQueue<Node<S,A>>(
    (node1, node2) -> Double.compare(/* node1 f-value */, /* node2 f-value */)
)
```

or

```
Util.Frontier<Node<S,A>> frontier = new Util.PriorityQueue<Node<S,A>>(
    public int compare(Node<S,A> n1, Node<S,A> n2) {
        // compare n1 and n2 f-values
    }
)
```

Naturally, the comparison should take into account the value given by the heuristic.

You can test your A* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic (implemented already as `ManhattanDistanceHeuristic` in `SearchProblem.py`).

```
java Pacman -l bigMaze -f astar -h ManhattanDistanceHeuristic
```

You should see that A* finds the optimal solution slightly faster than BFS (about 549 vs. 620 search nodes expanded in our implementation, but ties in priority may make your numbers differ slightly). What happens on `openMaze` for the various search strategies?

Question 4 (5 points): Finding All the Corners

The real power of A* will only be apparent with a more challenging search problem. Now, it's time to formulate a new problem and design a heuristic for it.

In *corner mazes*, there are four dots, one in each corner. Our new search problem is to find the shortest path through the maze that touches all four corners (whether the maze actually has food there or not). Note that for some mazes like `tinyCorners`, the shortest path does not always go to the closest food first!

Note: Make sure to complete Question 2 before working on Question 4, because Question 4 builds upon your answer for Question 2.

Implement the `PacmanCornersProblem` search problem in `SearchProblem.java`. You will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached; implement it in the `PacmanCornersSearchState` class. Now, your search agent should solve:

```
java Pacman -l tinyCorners -f bfs -p PacmanCornersProblem
java Pacman -l mediumCorners -f bfs -p PacmanCornersProblem
```

Hint: the shortest path through `tinyCorners` takes 28 steps.

Hint 2: When coding up `expand`, make sure to add each child node to your children list with cost `getCost` and next state `getSuccessor` that you will need to code up as well (cost should be 1 for each valid action).

Hint 3: Make sure that you correctly implement the logic of `PacmanCornersProblem` for all possible starting configurations. Do you handle all starting locations of Pacman correctly? What is Pacman starts in one of the corners?

Our implementation of `breadthFirstSearch` expands just under 2000 search nodes on `mediumCorners`. However, heuristics (used with A* search) can reduce the amount of searching required.

Question 5 (5 points): Corners Problem: Heuristic

Note: Make sure to complete Question 3 before working on Question 5, because Question 5 builds upon your answer for Question 3.

Implement a non-trivial, consistent heuristic for the `PacmanCornersProblem` in `CornersHeuristic`.

```
java Pacman -l mediumCorners -f astar -p PacmanCornersProblem -h CornersHeuristic
```

Admissibility vs. Consistency: Remember, heuristics are just functions that take search states and return numbers that estimate the cost to a nearest goal. More effective heuristics will return values closer to the actual goal costs. To be *admissible*, the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal (and non-negative). To be *consistent*, it must additionally hold that if an action has cost *c*, then taking that action can only cause a drop in heuristic of at most *c*.

Remember that admissibility isn't enough to guarantee correctness in graph search – you need the stronger condition of consistency. However, admissible heuristics are usually also consistent, especially if they are derived from problem relaxations. Therefore it is usually easiest to start out by brainstorming admissible heuristics. Once you have an admissible heuristic that works well, you can check whether it is indeed consistent, too. The only way to

guarantee consistency is with a proof. Consistency can be verified for a heuristic by checking that for each node you expand, its child nodes are equal or lower in f-value. If this condition is violated for any node, then your heuristic is inconsistent. Moreover, if UCS (A* with the 0 heuristic) and A* ever return paths of different lengths, your heuristic is inconsistent. This stuff is tricky!

Non-Trivial Heuristics: The trivial heuristics are the ones that return zero everywhere (UCS) and the heuristic which computes the true completion cost. The former won't save you on the number of nodes expanded, while the latter will take too long to compute h-value. You want a heuristic which reduces total compute time, though for this assignment the autograder will only check node counts (aside from enforcing a reasonable time limit).

Hint. There were several examples of heuristics presented you during the lectures. You can take your inspiration from those.

Grading: Your heuristic must be a non-trivial non-negative consistent heuristic to receive any points. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll be graded:

Number of nodes expanded	Grade
more than 2000	0/5
at most 2000	2/5
at most 1600	3/5
at most 1200	5/5

Remember: If your heuristic is inconsistent, you will receive *no* credit, so be careful!

Question 6 (5 points): Eating All The Dots

Now we'll solve a hard search problem: eating all the Pacman food in as few steps as possible. For this, we'll need a new search problem definition which formalizes the food-clearing problem: `PacmanFoodSearchProblem` in `SearchProblem.java` (implemented for you). A solution is defined to be a path that collects all of the food in the Pacman world. If you have written your general search methods correctly, A* with a null heuristic (equivalent to uniform-cost search) should quickly find an optimal solution to `testSearch` with no code change on your part (total cost of 7).

```
java Pacman -l testSearch -f astar -p PacmanFoodSearchProblem
java Pacman -l tinySearch -f astar -p PacmanFoodSearchProblem
```

Note: Make sure to complete Question 3 before working on Question 6, because Question 6 builds upon your answer for Question 3.

Fill in the `value` method of `FoodHeuristic` in `SearchProblem.java` with a *consistent* heuristic for the `PacmanFoodSearchProblem`. Try your agent on the `trickySearch` board:

```
java Pacman -l trickySearch -f astar -p PacmanFoodSearchProblem -h FoodHeuristic
```

Our BFS finds the optimal solution after exploring over 16,000 nodes.

Any non-trivial non-negative consistent heuristic will receive 2 points. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll get additional points:

Number of nodes expanded	Grade
more than 15000	2/5
at most 15000	3/5
at most 12000	4/5
at most 9000	5/5

Remember: If your heuristic is inconsistent, you will receive *no* credit, so be careful! Can you solve `mediumSearch` in a short time? If so, we're either very, very impressed, or your heuristic is inconsistent.

Grading

The maximum number of points you can get is 30, but the total mark will be capped at 25. The good news is that even if you do not manage to come up with very accurate heuristics in Questions 5 and 6, you can still get (close to) the full mark.

Submission and Deadline

The deadline for submission is **9 December 2021 at 23:55** (via raptor folder). You need to upload the files `GraphSearch.java` and `SearchProblem.java` to your folder in `proj/comp5280/search/login` for COMP5280 students and in `proj/comp8250/search/login` for COMP8250 students. It is important that you do not modify the names of the classes and of the methods as your code will be marked automatically.

Feel free to contact me (by email or Teams) if you have any questions. I am happy to help you.

There will be 3 drop-in sessions (on November 24, December 1 and 8), where I will be present. Please do come if you have problems getting anything working or want any help with solving questions.

Note: Your code will be checked for plagiarism (using dedicated tools). Every identified case will result in a disciplinary case. You are expected to do everything on your own and not use anyone else's code, even if correctly referenced. So, please, work on the assignment independently. It will be a good coding exercise, but more importantly, a good mental exercise for understanding the search algorithms and how to formalise your problem as a search problem.