SGN-44006: Artificial Intelligence **(Due: 17/01/20)**

Exercise set 1

Instructor: Tapio Elomaa
Course Assistant: Aashish Sah

Problem 1: Pen & Paper (6 pts points)

Consider a following problem presented in figure 1.

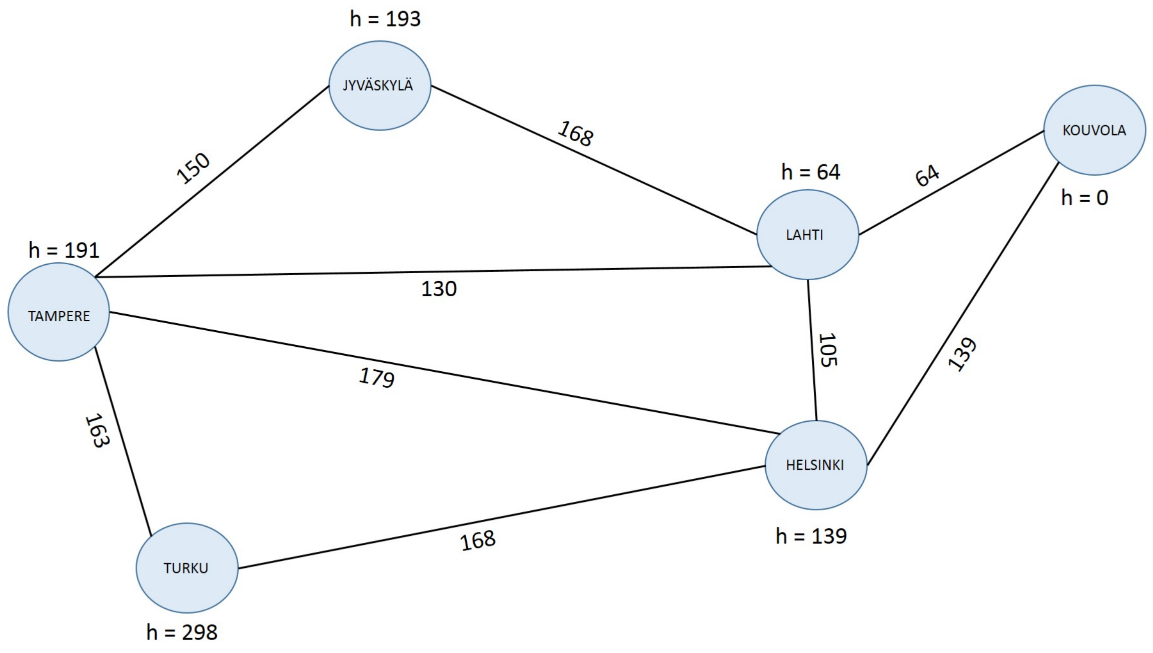

Figure 1: Route graph for Tampere-Kouvola

(a) Sketch the state space and search space graph for the route in Figure 1.

For each of the following graph search strategies, find out the order in which the states are expanded and also path returned by graph search. The start and goal states are Tampere and Kouvola respectively.

(b) Depth-first search.

(c) Breadth-first search.

Problem 2: Programming (6 pts points)

We have adopted the programming task from UC Berkeley Pacman project. Since they were teated as a longer project work and not weekly exercises, we will omit some of the implementations and only focus on some that are relevant to our course structure. However, if one wishes to do them all then they can find the original files from the above link. The codes are compatible **only** with python versions 3.xx and are available on the Moodle page.

There are handful of files in the directory but you need to edit only one file:

search.py - Where all of your search algorithms will reside.

You may want to look into these files:

pacman.py - The main file that runs Pacman games. This file describes a Pacman GameState type, which you use in this project.

game.py - The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.

util.py - Useful data structures for implementing search algorithms.

searchAgents.py - Where all of your search-based agents will reside.

Welcome to the Pacman

Download the python template for pacman from the moodle page. You should be able to play a game of Pacman by typing the following at the command line:

```
>> python3 pacman.py
```

Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain.

The simplest agent in **searchAgents.py** is called the **GoWestAgent**, which always goes West (a trivial reflex agent). This agent can occasionally win:

```
>> python3 pacman.py --layout testMaze --pacman GoWestAgent
```

But, things get ugly for this agent when turning is required:

```
>> python3 pacman.py --layout tinyMaze --pacman GoWestAgent
```

If Pacman gets stuck, you can exit the game by typing **CTRL-c** into your terminal.

Soon, your agent will solve not only tinyMaze, but any maze you want.

Note that pacman.py supports a number of options that can each be expressed in a long way (e.g., –layout) or a short way (e.g., -l). You can see the list of all options and their default values via:

```
>> python3 pacman.py -h
```

Also, all of the commands that appear in this project also appear in commands.txt, for easy copying and pasting. In UNIX/Mac OS X, you can even run all these commands in order with bash commands.txt

Implement following search algorithms to find a fixed food dot.

(a) Depth first search

In **searchAgents.py**, you'll find a fully implemented **SearchAgent**, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan are not implemented – that's your job. As you work through the following questions, you might find it useful to refer to the object glossary (the second to last tab in the navigation bar above).

First, test that the SearchAgent is working correctly by running:

```
>> python3 pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

The command above tells the **SearchAgent** to use **tinyMazeSearch** as its search algorithm, which is implemented in **search.py**. Pacman should navigate the maze successfully.

Now it's time to write full-fledged generic search functions to help Pacman plan routes! Pseudocode for the search algorithms you'll write can be found in the lecture slides. Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) which gets to that state.

Important note: All of your search functions need to return a 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).

Important note: Make sure to use the **Stack**, **Queue** and **PriorityQueue** data structures provided to you in **util.py**!

Hint: Each algorithm is very similar. Algorithms for DFS, BFS, UCS, and A* differ only in the details of how the fringe 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 queuing strategy. (Your implementation need not be of this form to receive full credit).

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

Your code should quickly find a solution for:

```
>> python3 pacman.py -l tinyMaze -p SearchAgent
>> python3 pacman.py -l mediumMaze -p SearchAgent
>> python3 pacman.py -l bigMaze -z .5 -p SearchAgent
```

The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration). Is the exploration order what you would have expected? Does Pacman actually go to all the explored squares on his way to the goal?

Hint: If you use a 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 **getSuccessors**; 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.

(b) Breadth first search

Implement the breadth-first search (BFS) algorithm in the **breadthFirstSearch** function in **search.py**. Again, write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for *depth-first search.*

```
>> python3 pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
>> python3 pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

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

Hint: If Pacman moves too slowly for you, try the option –frameTime 0.

Note: Don't get scared by the amount of text. Please read the text carefully, *the devil is in the details*, before jumping to the coding part.

Those who have not used python or are not familiar with running python through command prompt should come to the Monday programming help session. We will go through the basics first before delving into the exercises.

Remember all the help will be provided to those who ask for it!