Assignment 1

Due Date: Thursday, October 4, 2018 at 11:59pm

Instructions:

- Submit the assignment in the Dropbox labeled Assignment 1 Submissions in the Assignment 1 folder on LEARN.
- No late assignment will be accepted.
- This assignment is to be done individually.
- For any programming question, you may use the language of your choice. We highly recommend using Python if you are familiar with it.

Lead TA: Atrisha Sarkar (atrisha.sarkar@uwaterloo.ca). Atrisha's office hours will be posted on the course website.

Question 1 (30 points) (Search on the Missionaries and Cannibals Problem)

The "missionaries and cannibals" problem is a variant of the river crossing problem discussed in class. Three missionaries and three cannibals are on one side of the river, along with a boat that can hold one or two people. Find a way to get everyone to the other side of the river without ever leaving a group of missionaries outnumbered by the cannibals on either side of the river. The boat must have at least one person in it to cross the river.

(a) Formulate this problem as a search problem. Be sure to describe the definition of the states, the start state, the goal state, the successor function, and the costs in detail.

(b) Draw the complete search graph, which contains all of the states and all of the arcs based on the successor function.

(c) Draw the search tree of Breadth-First Search until you have expanded 6 nodes.

(d) Draw the search tree of Depth-First Search until you have expanded 6 nodes.

(e) Propose an admissible heuristic function h and describe it in detail. Be sure to explain why it is admissible. Your heuristic will be judged by how good it is.

(f) Describe an optimal solution to this problem and the cost of this optimal solution. You do not need to show how you obtained the solution.

Please **submit** a writeup containing the answers to all the six questions above.

Question 2 (50 points) (A* search on the Traveling Salesperson Problem)

You will implement a search algorithm to solve the Traveling Salesperson Problem (TSP).

In a TSP, a salesperson must start from a city, visit every one of n cities exactly once and return to the starting city. The goal is to find the tour with the lowest total distance. Assume that the distance of traveling from one city to another is the Euclidean distance between the two cities. For example, given two cities with coordinates (x_1, y_1), and (x_2, y_2), the distance between the two cities is the square root of ((x_1 – x_2)^2 + (y_1 – y_2)^2).

We have provided you with many instances of the TSP problem. You can find them in the file tsp_problems.zip in the Assignment 1 folder on LEARN.

The format of each problem instance file is as follows:

```
<number of cities>
<city id> <X> <Y>
<city id> <X> <Y>
…         …   …
<city id> <X> <Y>
```

The first line has the number of cities. Each subsequent line contains the letter and the location of one city. On each line, there is a capital letter <city id>, followed by the <X> coordinate and the <Y> coordinate of the city. The X and Y coordinates are integers.

Write a program that uses the A* search algorithm to solve the TSP problem. Let the cost to move from one city to another be the Euclidean distance between the two cities.

Use the following conventions in your program formulation and your implementation.

- Assume that A is always the starting city.
- Generate the successors of any state in alphabetical order.

Complete the following tasks and write a short report to describe the results.

(a) Formulate the TSP problem as a search problem. Include the definition of a state, the start state, the goal state, and the successor function.

(b) Choose a good admissible heuristic function h and describe h in detail in your report. Be careful with choosing the heuristic function. For some poor choices of the h function, A* search may not terminate in a reasonable amount of time.

See further instructions on the next page.

(c) Implement and run A* search with your heuristic function h on the provided TSP problems.

If your program spends more than 5 minutes on a 10-city TSP problem, something is going horribly wrong. Terminate your run, rethink your heuristic function and debug your implementation.

Otherwise, if you program can solve a 10-city TSP problem quickly, then it is like that your heuristic and implementation are fine.

For each number of cities (from 1 to 16), determine and plot the average number of nodes that A* search generates for each number of cities. We have provided you with 10 TSP problem instances for each number of cities (from 1 to 16).

(d) Based on the plots from part (c), extrapolate the number of nodes A* search would generate for a 36-city TSP problem. (You may want to use a logarithmic scale on the y-axis.)

Give an estimate of how long the A* search with your heuristic function would take to solve a 36-city TSP problem instance.

(You can try running your implementation on the 36-city problem instance, but we do not require your implementation to solve this problem instance.)

(e) Repeat part (c) with the heuristic function h(n) = 0 for all n.

For each number of cities (from 1 to 16), determine and plot the average number of nodes that A* search generates.

If the search takes more than 5 minutes (or you run out of memory) on any of the runs, then you can stop early and record that it did not terminate.

(f) Based on the plots from part (e), give an estimate of how long the A* search with the heuristic function h(n) = 0 for all n would take to solve a 36-city TSP problem instance.

(g) Discuss the difference in the performance of A* search when using the two different heuristic functions. 1-2 paragraphs should suffice.

See a list of items that you need to submit on the following page.

Please **submit** the following:

1. Your problem representation.
2. The description of your heuristic function.
3. A well document copy of your code.
4. A plot of the average number of nodes generated for each number of cities for your heuristic function.
5. A discussion of how you expect A* search to perform on a 36-city problem instance for your heuristic function.
6. The plot of the average number of nodes generated for each number of cities for the heuristic function h(n) = 0.
7. A discussion of how you expect A* search to perform on a 36-city problem instance for the heuristic function h(n) = 0.
8. A discussion on the difference in performance when using the two different heuristic functions.

Question 3 (50 points) (CSP and backtracking search on Sudoku)

In the question you are asked to implement a CSP algorithm for solving Sudoku puzzles. Sudoku is a simple game of deduction, usually played on a 9x9 grid. In the goal state, each number from 1 to 9 appears exactly once in each row and column on the grid. Additionally, the grid is subdivided into 9 3x3 non-overlapping sub-grids, and each number must appear exactly once in each sub-grid. A Sudoku puzzle usually starts with some numbers filled in, so that there is a single unique solution. An example of such a puzzle is given below:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

We have provided you with many instances of the Sudoku problem. You can find them in the file sudoku_problems.zip on the course website. In a particular sub-directory, all puzzles in a particular sub-directory have the same number of initial values (corresponding to the name of the sub-directory).

Each file contains 9 rows of exactly 9 numbers. Numbers within a line are separated by whitespace. The value 0 is used to indicate a blank space in the grid. For example, the first line of the example puzzle above would be written:

```
5 3 0 0 7 0 0 0 0
```

See instructions on the next page.

a) Write a formal description of Sudoku as a CSP, giving a list of variables, their domains, and the constraints between them.

b) Implement three version of a CSP solver for Sudoku. For each version above, have your solver count the total number of variable assignments it makes (including when it backtracks).

    Version A : Standard backtracking search
    Version B : Standard backtracking search + forward checking
    Version C : Standard backtracking search + forward checking + heuristics (most restricted variable, most constraining variable (for tie breaking) and least constraining value).

c) Run each of the 3 versions of your solver on each problem instance, counting the number of variable assignments made on each instance. To avoid spending a (very) long time collecting data, your solver should 'give up' if it needs to take more than 10,000 steps.

d) For each version of your solver, create a plot of your findings. The x-axis has the number of initial values, and the y-axis has the average number of variable assignments for each initial value.

e) You can produce three different plots (one for each version of your solver) or include the three plots in a single graph. Make sure that everything is well labelled on your plots.

f) Describe your findings and provide an explanation for what you observe. 1-3 paragraphs should suffice.

Please **submit** the following.

1. Your CSP representation.
2. A well-documented copy of your code.
3. The plot of the average number of variable assignments against the number of initial values for the provided Sudoku problems.
4. A discussion of your findings and an explanation for why your program behaves as it does.