Due Date: Wednesday, October 4, 2017 at 11:59pm

Submission Instructions: Assignments are to be submitted through LEARN, in the Dropbox labelled *Assignment 1 Submissions* in the Assignment 1 folder. Late assignments will be accepted up until Friday, October 6 at 11:59pm. Please read the course policy on assignments submitted after the official due date. *No assignment will be accepted, for any reason, after 11:59pm on October 6, 2017.*

Lead TA: Milad Khaki (milad.khaki@uwaterloo.ca) Office hours on Fridays 4:00pm-5:30pm in DC 3515.

Announcements: The following exercises are to be done individually. For the programming questions you may use the language of your choice.

Question 1 (50 points)

In this question you are asked to implement a search algorithm for solving the Traveling Salesperson Problem (TSP). In a TSP a salesperson must visit n cities, visiting every city exactly once and returning to the city they started from. The goal is to find the tour with the lowest cost. Assume that the cost of traveling from one city to the next is the Euclidean distance between the two cities. That is, given two cities with coordinates (x_1, y_1) and (x_2, y_2), the cost of traveling between the two is

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

The data can be found in the file *TSPproblem.zip* which is located in the Assignment 1 folder. The data format of the problem instances is as follows;

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

The < city id > is a capital letter. The identifiers < X > and < Y > represent the x and y coordinates (integers) of the city. Each city is on its own line and the number of cities is on the first line by itself.
Example:

```
3
A  17  55
B  24  38
C  91  57
```

Write a program that uses the A* algorithm to solve the Traveling Salesperson Problem (TSP). Let the cost to move between cities be the Euclidean distance. In the code, try to separate the domain independent part (search algorithm) from the domain specific part (TSP related functions).

a. Give a representation of the problem suitable for A* search. That is, describe states, their representation, the initial state, the goal state, the operators, and their cost.

Use the following conventions;

- Always start from node A. (You might as well since every tour has to go through A; thus A never needs to be backtracked.)
- Generate the successors of any given node in alphabetical order.
- Count nodes as they are generated as successors.

Generate and use a *good admissible* h function, and describe it in detail in the write-up. Think carefully about how to construct the h function since some choices of h functions may be too poor to allow you to execute the search in practice. If your program spends more than 5 minutes on a problem instance of 10 cities, something is going horribly wrong: terminate the run and rethink your design. (On the other hand, don't panic if you can solve the 10-city instances quickly. That likely means your heuristic and implementation is fine.)

b. Describe your heuristic function.

c. Run A* search with your heuristic function on the provided random test instances. Note that there are 10 instances for each number of cities. Determine how many nodes, on average, A* generates for each number of cities, and plot them. Extrapolate from these results roughly how many nodes A* search would generate for a 36 city problem. To do this, you may want to use a logarithmic scale on the y-axis. How long would such a search take? (If you want, you can try to run your implementation on the 36-city problem, but we do not require that your implementation successfully solves this particular instance).

d. Repeat the same experiment described in part c. using the heuristic function $h(n) = 0$ for all n. 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.

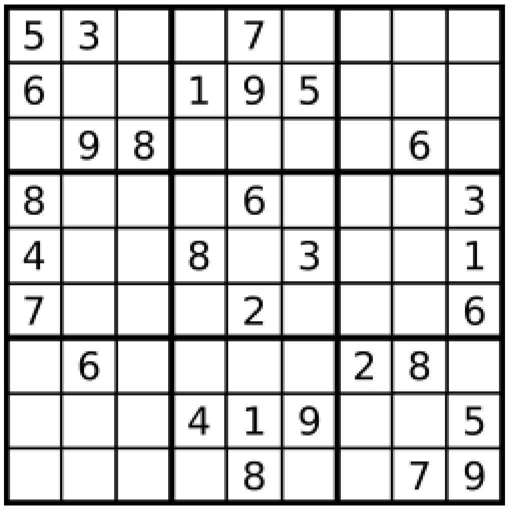
Submit the following

- A well documented copy of your code.
- Your problem representation.
- The description of your heuristic function.

- The plot of the average number of nodes generated for each number of cities for both your heuristic function and the heuristic function $h(n) = 0$. You can do this using two different plots, or by plotting everything on the same graph. Just make sure that everything is well labelled.
- A discussion on the difference in performance when using the two different heuristic functions.
- A discussion of how you expect A* search to perform on a 36-city problem instance for both heuristic functions.

Question 2 [50 pts]

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 subgrids, and each number must appear exactly once in each subgrid. 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

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

b. Implement a CSP solver for Sudoku. In particular your solver should have three versions:

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).

Have your solver (for each version) count the total number of variable assignments it makes (including when it backtracks).

c. You can find a set of Sudoku problems with different numbers of initial values in the SudokuProblem.zip file included in the A1 folder on Learn. The puzzles are organized so that 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:

$$5\ 3\ 0\ 0\ 7\ 0\ 0\ 0\ 0$$

For each version of your solver (Version A, Version B and Version C), run it 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**.

For each version of your solver, create a plot of your findings, where on the x-axis you have the number of initial values, and the y axis is the *average* number of variable assignments for each initial value. You can choose to either produce three different plots (one for each version of your solver) or include the three plots in a single graph. Just make sure that everything is well labelled.

Describe your findings and provide an explanation for what you observe.

Submit the following

- A well documented copy of your code
- A written description of your problem representation
- The plot of the average number of variable assignments against the number of initial values the sample Sudoku problems
- A discussion of your findings and an explanation for why your program behaves as it does.