CMSC 170 MACHINE PROBLEM 2 **Total**: **200 points** (20% of grade)
Constrained Problems **Deadline**: April 27, 2018 (Friday)

INTRODUCTION

The second Machine Problem is about *constraint satisfaction* and *constrained optimization* problems, and it will test your abilities and understanding of problem *modeling*, *constraints*, *backtracking*, *local search*, *simulated annealing, Tabu search*, and *genetic algorithms*. You will be working in **groups with 4 members**, and you are allowed to choose your own groupmates.

The project zip file contains all the codes needed, and an empty answer sheet. Write your answers to the questions below in the answer sheet. The code is tested on **Python 3.0** (important for reproducibility of experiments, especially items with random components).

PROBLEMS

1. **Plants Exhibit** *problem/plants.py*
   The task is to arrange five plants (rose, sunflower, tulip, corpse, vampire) in pots, subject to constraints

2. **Einstein's Puzzle** *problem/einstein.py*
   There are five houses. Each house has a different color. Each house owner has a different nationality, different pet, favorite beverage, and favorite social media. The task is to figure out the color, nationality, pet, beverage, and social media associated to each house, given some constraints.

3. **Magic Square** *problem/magic_square.py*
   A magic square of size N is an NxN grid filled with numbers from 1 to N^2, where the numbers in each row, each column, and in the forward and backward main diagonals, all add up to the same number.

   Example: Magic Square for N = 3

6	7	2
1	5	9
8	3	4

4. **Magic Series** *problem/magic_series.py*
   A magic series of size N is a series of numbers from index 0 to N, wherein at each index i, if the number is j, there must be j i's in the series. In the example below, at index 0, the number is 1 and there is one 0 in the series (at index 3). At index 1, the number is 2, and there are two 1s in the series (at index 0, 2).

   Example: Magic Series for N = 3

0	1	2	3	index
1	2	1	0	number

5. **Knapsack** *problem/knapsack.py*
   There is a knapsack with a limited weight capacity, and several items, each having their own weight and value. The task is to select a combination of the items whose total weight doesn't exceed the capacity of the knapsack. The optimization part is trying to maximize the total value of the items selected.

6. **Vertex Cover** *problem/vertex_cover.py*
   Given a graph with vertices and undirected edges, select a combination of vertices such that all edges are covered by at least one selected vertex. An edge is covered if at least one of its endpoints is included in the solution. The optimization part is trying to minimize the number of vertices used.

7. **Maxone** *problem/maxone.py*
   This is a toy problem whose goal is to maximize the number of 1s in a binary string of length N. The obvious optimal solution would be a binary string of N 1s.

PART 0. **CONSTRAINTS**

In this section, we will **add constraints** to a problem (task 1), and **implement constraints** for different problems (tasks 2-5). We will use **p0.py** to perform the tasks, which uses a *brute force solver*.

1. **Magic Square Problem**
   File to edit: problem/magic_square.py, *Function to edit*: problem
   (**5 pts)** Complete the problem definition for magic square by adding the constraints: You will be using the constraints *AllDifferent* and *ExactSum*. Use other problem definitions (plants.py, einstein.py, etc) as guides on how to add constraints to a problem. To test your code, run test1() on p0.py and check that the results for N = 3 and N = 4 are the same with check/p0.1.txt.
   Hint: Find out how to solve for the magic sum (the sum for each row, column, and main diagonals).

2. **Magic Square Constraint**
   File to edit: problem/constraints.py *Method to edit:* ExactSum.test
   (**5 pts)** Implement the constraint for magic square, which tests if the sum of the values assigned to the variables involved in the constraint, is equal to the target sum. To test your code, run test2() on p0.py and check that the results for N = 3 are the same with check/p0.2.txt.
   Question: Describe how the 8 solutions for magic square(N=3) are *similar*, and how they are *different*.

3. **Magic Series Constraint**
   File to edit: problem/constraints.py *Method to edit:* MagicSeries.test
   (**5 pts)** Implement the constraint for magic series, which tests that for each index, the number assigned to it is the number of times index appears in the series. To test your code, run test3() on p0.py and check that the results for N = 3 and N = 4 are the same with check/p0.3.txt
   Question: How many solutions are there for magic series(N=5)?
   Question: Find all solutions for magic series(N=6). How many iterations did it take?

4. **Knapsack Constraint**
   File to edit: problem/constraints.py *Method to edit:* KnapsackCapacity.test
   (**5 pts)** Implement the constraint for knapsack, which tests that the total weight of items included in the solution doesn't exceed the capacity. To test your code, run test4() on p0.py and check that the no. of solutions found for test_case = 0,1,2 are the same as the results in check/p0.4.txt
   Question: What is the total weight and total value of the optimal solution for knapsack(test_case=1)?
   Question: How many iterations and solutions were found for knapsack(test_case=3)?

5. **Vertex Cover Constraint**
   File to edit: problem/constraints.py *Method to edit:* VertexCover.test
   (**5 pts)** Implement the constraint for vertex cover, which tests that all the edges in the given graph are covered by at least one vertex included in the solution.To test your code, run test5() on p0.py and check that the no. of solutions found for test_case = 0,1,2 are the same as the results in check/p0.5.txt
   Question: How many solutions were found for vertex_cover(test_case=3,4,5)?
   Question: What are the vertices in the optimal solution for vertex_cover(test_case=3)?

6. **Brute Force: Combination vs Permutation**
   We are using a brute force solver in this section, which tests all possible combinations of values. However, for the Plants Exhibit and Magic Square problems, we don't need to check for all possible combinations. We only need to check permutations of values (different arrangements), since we are assured that all variables are assigned different values (*AllDifferent*). Run test6() on p0.py, to perform the experiment. *Note:* Don't run (magic_square, combination) anymore as it will take a very long time.

   Question: Fill in the table to compare the number of iterations and running time when using all possible combinations of values vs. using permutations of values for the plants and magic square problems. For (magic_square, combination), just compute the no. of iterations and estimate the running time.
   Question: Discuss the results above briefly.

PART 1. **BACKTRACKING**

In this section, we will experiment with **backtracking** (task 1), **filtering** (task 2), and implement a custom **variable ordering** function and **value ordering** function (task 3). We will use **p1.py** for this section.

1. **Brute Force vs Backtracking**

   Run the brute-force solver and basic backtracking solver (no filtering, no ordering) to solve different problems, and compare their performance in terms of number of iterations.

   Run: test1() on p1.py
   Problems: plants, magic square, magic series, knapsack, vertex cover
   Solvers: brute force, backtracking
   Solution Limit: 0 (find all), 1 (find one)

   Note: For backtracking, iteration count is increased every time a solution is tested for violations.

   Question: Fill in the table to compare the number of iterations when using brute force vs backtracking.
   Question. Discuss: (1) which situations did brute-force have more iterations than backtracking? (2) which situations did brute-force have fewer iterations than backtracking and why did this happen?
   Question: In solution_limit = 1, did any solver find the optimal solutions for knapsack and vertex cover? Explain why this happened.

2. **Filtering**

   Compare the performance of backtracking with filtering and without filtering. We will use forward checking to filter out bad values. Compare the number of iterations for different problems.

   Run: test2() on p1.py
   Problems: plants, einstein, magic square, magic series, knapsack, vertex cover
   Solvers: backtracking without filtering, backtracking with filtering
   Solution Limit: 0 (find all), 1 (find one)

   Note: You can reuse the results from task 1 (backtracking with no filtering).

   Question: Fill in the table to compare the number of iterations when using backtracking with and without filtering. Quantify the improvement caused by filtering.
   Question: Discuss the importance of filtering based on the results above.

3. **Variable and Value Ordering**

   File to edit: fn/bt.py *Functions to edit:* custom_variable_selector, custom_value_ordering

   (**10 pts**) Implement a variable ordering and value ordering function, to improve the performance of backtracking. Then, run the backtracking solver on different problems with filtering and ordering enabled. Compare the number of iterations when using only filtering vs using filtering + ordering.

   Run: test3() on p1.py
   Problems: plants, einstein, magic square, magic series, knapsack, vertex cover
   Solvers: backtracking with filtering, backtracking with filtering + ordering
   Solution Limit: 0 (find all), 1 (find one)

   Note: You can reuse the results from task 2 (backtracking with filtering).

   Question: Fill in the table to compare the number of iterations when using backtracking with filtering and with filtering + ordering. Quantify the improvement caused by ordering.
   Question: Discuss the importance of ordering based on the results above.
   Question: In solution_limit = 0, are there any problems which showed no improvement even after ordering was applied? Explain why this happened to these problems.
   Question: Compare the no. of iterations for brute force solver vs. backtracking solver with filtering and ordering. Discuss which solver is better.

PART 2. LOCAL SEARCH

In this section, we will implement **neighborhoods** (task 1) and **objective functions** (task 3) for local search, experiment which **neighborhoods** (task 2-3) and **hill climbing** variants (task 4) are appropriate for some problems, and test different tabu tenures for **tabu search** (task 5). For constraint satisfaction problems (plants, einstein, magic square, magic series), we will use *count_violations* as the objective function, with the goal of having score = 0 (no violations). We will use **p2.py** for this section.

1. **Neighborhood Implementation**

   File to edit: fn/ls.py *Functions to edit:* change_upto_two_values, swap_two_values

   (10 pts) Implement two neighborhoods: (1) *change up to two values*, where the neighbor solutions are created by selecting one or two variables and changing their values; and (2) *swap two values*, where the neighbor solutions are created by swapping the values of two variables. We will test your code on the plants problem. To test your code, run test1() on p2.py and check that the results are the same with check/p2.1.txt. Check that you have the same total neighbors and total legal neighbors.

2. **Neighborhoods**

   Find out which neighborhood is appropriate for the constraint satisfaction problems, by trying out 5 different neighborhoods for each problem.

   Run: test2() on p2.py

   Problems: plants, einstein, magic square, magic series

   Neighborhoods: change1, change2, swap2, min-conflict, max-min-conflict

   Question: Fill in the table to compare the total number of iterations of the different neighborhoods for each problem. Color the cell red if the solver didn't find a feasible solution (score > 0).

   Question: From the results above, which neighborhood is the best for these problems in general?

3. **Constrained Optimization Problems**

   File to edit: fn/objective.py *Functions to edit:* knapsack_objective, vertex_cover_objective

   (**10 pts**) Implement the objective functions for knapsack and vertex cover. To test your code, run test3() on p2.py for *change1* neighborhood, and check that the results are the same with check/p2.3.txt.

   Find out which neighborhood is appropriate for the optimization problems, by trying out 3 different neighborhoods for each problem, and using the implemented objective functions.

   Run: test3() on p2.py

   Problems: knapsack, vertex cover

   Neighborhoods: change1, change2, swap2

   Question: Fill in the table to compare the total number of iterations and best score of the different neighborhoods for each problem.

   Question: From the results above, which neighborhood is the best for knapsack and vertex cover, in general? Compare the best score obtained, and use the number of iterations as tie-breaker.

4. **Hill Climbing Variants**

   Compare the performance of 3 hill climbing variants, using different problems. Compare the number of iterations when using hill climbing (best improvement), hill walking (best no degradation), and random walking (random no degradation).

   Run: test4() on p2.py

   Problems: plants, einstein, magic square, magic series, knapsack, vertex cover

   - For CSPs, use *count_violations* as objective function
   - For CSPs, use *change2* as the neighborhood.
   - For knapsack & vertex cover, use the objective functions implemented in previous task.
   - For knapsack & vertex cover, use *swap2* as the neighborhood.

   Local Search: hill climbing, hill walking, random walking

   Question: Fill in the table to compare the number of iterations and best score of different hill climbing variants for each problem.

   Question: For the constraint satisfaction problems, which hill climbing variant worked best? Explain.

   Question: For the optimization problems, which hill climbing variant worked best? Explain.

   Question: From the results above, which was the worst hill climbing variant in general? Explain.

5. **Tabu Search**
   Compare the performance of different tabu tenures for tabu search, including tenure = 0 (no tabu, just plain local search), for different optimization problems.
   Run: test5() on p2.py
   Problems: knapsack(test_case=2,3), vertex cover(test_case=3,5)
   Configuration: swap2 neighborhood, knapsack / vertex cover objective, no degradation
   Tabu tenures: 0, 3, 5, 7
   Question: Fill in the table to compare the number of iterations and best score of different tabu tenures. Which was the best tabu tenure for the knapsack problems? For vertex cover?
   Question: Based on the results, discuss the importance of tabu search and choosing the right tenure.

PART 3. **ADVANCED LOCAL SEARCH**

In this section, we will implement and experiment with generic and custom **neighbor generators** (tasks 1-3), for **stochastic local search**. We will also experiment with various *temperature cooling schedules* for **simulated annealing** (task 4). We will use **p3.py** for this section.

1. **Neighbor Generators**
   File to edit: fn/ls.py *Functions:* change_upto_two_values_generator, swap_two_values_generator
   (10 pts) Implement two neighbor generators: (1) *change up to two values*, where the random neighbor solution is created by selecting one or two random variables and re-assigning them random values; and (2) *swap two values*, where the random neighbor solution is created by swapping the values of two random variables. We will test your code on the maxone problem. To test your code, run test1() on p3.py and check that the results are the same with check/p3.1.txt.

2. **Custom Neighbor Generators**
   File: fn/custom.py *Functions:* knapsack_neighbor_generator, vertex_cover_neighbor_generator
   (10 pts) Implement custom neighbor generators for knapsack and vertex cover. To test your code, run test2() on p3.py. You can check that the results of my implementation in check/p3.2.txt
   Note: your results doesn't have to be the same, as this depends on your implementation.

3. **Stochastic Local Search**
   Compare the performance of different neighbor generators for stochastic local search, in different optimization problems.
   Run: test3() on p3.py
   Problems: maxone, knapsack, vertex cover
   Neighbor generators: change1, change2, swap2, custom
   Question: Fill in the table to compare the iterations and best score of different neighbor generators.
   Question: For each problem, which neighbor generators performed best? Discuss the results.
   Question: Did the custom neighbor generators perform well in general? Explain why or why not.

4. **Simulated Annealing**
   Compare the performance of stochastic local search vs. simulated annealing, in different optimization problems. For simulated annealing, we will use different temperature cooling schedules by trying different values for alpha (multiplier for temperature, per iteration), such that $Temp_{new} = Temp_{old} * alpha$. For these experiments, we will be using test_case=99 (hard examples for these optimization problems). For knapsack, there are 50 item choices, while in vertex cover, there are 52 vertices and 150 edges.
   Run: test4() on p3.py
   Problems: knapsack, vertex cover
   Solvers: Stochastic Local Search, Simulated annealing (alpha = 0.5, 0.75, 0.95)
   Neighbor generators: change1, change2, swap2, custom
   Legal Neighbor: no degradation (hill walk), always improve (hill climb)
   Question: Fill in the table to compare the number of terations and best score of the different configs tested: stochastic local search (SLS) vs simulated annealing (SA), different alpha values for SA, neighbor generators, and hill walk vs hill climb.
   Question: Which alpha values worked best for knapsack and vertex cover, in general? Discuss.
   Question: Which configuration produced the best score for stochastic local search and for simulated annealing for knapsack and vertex cover?

PART 4. GENETIC ALGORITHMS

In this section, we will implement **fitness functions** for knapsack and vertex cover, to use the genetic algorithm solver. We will also experiment with *various configurations* of GA to find out the best score for the test_case=99 versions of knapsack and vertex cover. We will use **p4.py** for this section.

1. **Fitness Functions**

   File to edit: fn/fitness.py *Functions to edit:* knapsack_fitness, vertex_cover_fitness

   (**10 pts**) Implement the fitness functions for knapsack and vertex cover. To check if your code is correct, submit fitness.py to your instructor for checking (via Facebook).

2. **Genetic Algorithms**

   Test out different configurations of the genetic algorithm solver to find which configuration produces the best solution score.

   Run: main() on p4.py
   Problems: knapsack, vertex cover
   Population Model: generational, choose best
   Population Size: 20, 50
   Parent Selection: fitness proportionate, tournament
   Parent Similarity: 0.5, 0.9
   Crossover: one point, two point, uniform
   Crossover Probability: 0.6, 0.9
   Mutation: change1, change2, swap2
   Mutation Probability: 0.3, 0.5

   Question: Which configuration produced the best solution for knapsack and vertex cover?
   Question: Discuss why the best configurations (population model, selection, crossover, mutation, etc) for knapsack and vertex cover performed well.

SCORING & SUBMISSION

	Part 0	Part 1	Part 2	Part 3	Part 4	Total
Codes	25	10	20	20	10	**85**
Answers	20	30	30	20	15	**115**
					Total	**200**

1. Create a **PDF** of the answer sheet, and name it *mp2_lastname1_lastname2_lastname3_lastname4.pdf*.
2. Create a **zip file** containing the ff:
   a. **fn/bt.py** - contains implementations of custom variable and value ordering functions (for backtracking)
   b. **fn/custom.py** - contains code for custom neighbor generators for knapsack and vertex cover (for advanced local search)
   c. **fn/fitness.py** - contains implementations of fitness functions for knapsack and vertex cover (for genetic algorithms)
   d. **fn/ls.py** - contains implementations of neighborhood functions & generators (change2, swap2) (for local search)
   e. **fn/objective.py** - contains implementations of objective functions for knapsack and vertex cover (for local search)
   f. **problem/constraints.py** - contains implementations of constraints for magic square, magic series, knapsack, and vertex cover
   g. **problem/magic_square.py** - contains the completed problem definition of magic square, with the constraints added.
   h. **mp2_lname1_lname2_lname3_lname4.pdf** - answer sheet
3. Please follow this filename format for the zip file: *lastname1_lastname2_lastname3_lastname4.zip*
4. Email your output to jrdaradal@up.edu.ph on or before **April 27, 2018 (Friday), 11:59 PM**.
5. Please use this format as the subject: **[CMSC 170 - MP 2 - LName1 / LName2 / LName3 / LName4]**
6. Late submissions will receive a penalty of 5 points per late day (max = 15).
7. **Hard deadline**: Submissions after **April 30, 2018 (Monday)** will no longer be accepted.

Maximize your expected utility.