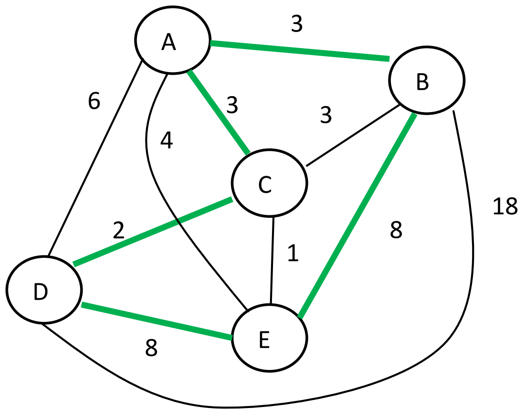
PROJECT 4: GRAPHS

OVERVIEW

This assignment focuses on the representation of computational problems as graph structures and the use of graph algorithms to solve them. It also focuses on heuristic search methods as a means to solve non-tractable or NP hard problems in a reasonable period of time at the risk of finding sub-optimal solutions.

The computational problem for this project is to find a route from a city to all other cities given a network of roads and cities that connect them so that a traveler visits each city at least once and returns to the origin at a minimal cost of travel. If each city in the travel plans may only be visited exactly once, the travel problem becomes the famous Traveling Salesman Problem (TSP). The problem can be modeled as an undirected, weighted graph in which the vertices represent the cities, edges represent the paths between the cities, and weights associated with the edges represent the cost of travel between the cities. The edges in the graph are undirected allowing travel to occur in both directions at the cost associated with the edges. The TSP is typically applied to a fully connected graph. The figure below shows a fully-connected graph of a network and a possible route marked in green.



The problem of finding an optimal solution to the TSP problem has real-world applications. For example, a package-delivery service may want to find a route to deliver packages in a city that reduces the overall time of delivery of each package and the cost of fuel.

BRUTE-FORCE SOLUTION

Finding the solution to a TSP is non-tractable or NP-hard because it requires an algorithm to evaluate all possible routes for its cost to choose the least-cost route. Evaluating all possible routes for a network with *n* cities requires the evaluation of a total of *(n-1)!* permutations of cities, each describing the order by which cities are visited. Note that the starting city remains fixed while all the others may be visited in any order. As *n* grows, the number of options, also known as the search space, becomes exponentially large. For the network of cities shown in the graph above, a total of 4! or 24 routes would have to be evaluated. Even for 15 cities, the number of routes to evaluate grows to 87,178,291,200.

Approximate and Heuristic Solutions

Instead of solving the problem by brute-force, many other algorithms exist that will find a sub-optimal solution in a shorter period of time eliminating the need to evaluate all permutations. We can classify the algorithms that compute a sub-optimal solution as Greedy, approximate algorithms or Heuristic Search algorithms.

Greedy, Approximate Algorithms

By allowing cities to be visited more than once in the original TSP problem, algorithms can be applied that dramatically reduce the number of options to evaluate. For example, a Minimal Spanning Tree (MST) selects paths between cities that connect all of the cities in a network at a minimal cost. However, to traverse the minimal spanning tree and visit all the cities as required by the TSP problem the algorithm will need to travel in both directions over some of the edges in the MST increasing the total cost to complete a round trip. Thus, the round trip cost may be dramatically larger than an optimal TSP solution.

Heuristic Search Algorithms

A different class of algorithms, known as Heuristic Search, apply a set of rules to compute a solution. One such algorithm is the Genetic algorithm that applies the idea of mutation and cross-over to an initial population to generate offsprings. By selecting offsprings that produce a lower cost by chance and combining them with lower-cost solutions from the original population, the algorithm chooses a genetically more fit population to generate more offsprings in the next step, and so forth. Solutions that have a higher cost will “die” and disappear from the population. A short outline of the algorithm is given below:

1. Choose the initial population of tours (generate a set of permutations of the orderings of the cities)
2. Evaluate the fitness of each individual tour.
3. Repeat until termination criteria is met:
    a. Select top-ranking individuals or elites to reproduce.
    b. Create a new generation through crossover, mutation or both (genetic operations) and give birth to offsprings.
    c. Evaluate the individual fitness of each offspring.
    d. Replace worst ranked solutions of the current population with offsprings.

The mutation operation makes a minor change in a solution such as the swap of two, three, or four cities in a permutation. The cross-over operation takes two solutions from a population and splices the end of one ordering to the end of another. However, the operation may result in duplicate cities in the sequence representing an incorrect permutation of cities. Hence, the algorithm may perform post-processing to fix errors introduced into cross-over generated offsprings. For example, given the cities A, B, C, D, E, F. The figure below shows two permutations and two new permutations when the original two permutations are spliced into two halves after the third city in the sequence and the resulting halves are crossed-over.

Both original permutations with splicing and cross-over indicated:	New offsprings after cross-over:
A B C F E D	A B C D A C
F B E D A C	F B E F E D

The two new offsprings would have to be “repaired” to generate permutations of the known cities. Elites are the few best solutions in your population.

A possible termination criteria may be when:

1. an elite meets a minimum criteria in terms of the round-trip cost or
2. a fixed number of generations have been produced or
3. a population emerges that no longer changes significantly in terms of the cost after cross-over and mutation is applied.

THE PROGRAM

Write three C programs that implement a Brute-Force approach, an MST approach, and a Genetic approach to the TSP problem. The programs must accept an input file that describes a network of cities and the cost of travel between them to compute a solution. To execute each program at the command line it must called as follows:

```
tspBruteForce <input> <numCities>

tspMST <input> <numCities>

tspGenetic <input> <numCities> <numGen> <numTours> <percent>
```

The parameter <input> refers to an input file that describes the problem the program must solve. The parameter `<numCities>` describes the number of cities to process from the input file, the parameter `<numTours>` generates the number of tours in the initial population, the parameter `<numGen>` describes the maximum number of generations to run the algorithm, the parameter `<percent>` describes the percentage of mutatation and cross-over operations at each stage.

When each program runs it must chose one of the cities as a starting point, run its algorithm to solve the problem, and print to the screen the following information: the route, the cost of the route, and the elapsed time to solve it. As an output for the route simply print the sequence of cities in the optimal or sub-optimal route. The program that implements the MST may need to print out some cities multiple times as the algorithm traverses the MST, backtrack on some of the connections, to compute a complete route.

INPUT DATA

Each program will utilize an input file that contains the weights of the edges between cities. For simplicity, assume that all cities are connected to all others - in other words, the cities and the paths between them form a complete graph. From the data in the file compile a list of `<numCities>` cities that will be used together with the weight of the edges from the file to generate a single, fully-connected graph to run the experiment.

PROJECT OUTCOMES

- To use brute-force techniques for solving a computation problem.
- To use heuristic and approximate techniques to solve a computational problem.
- To experiment and choose techniques that solve a computational problem best within a given time period.
- To use a Makefile for compiling source code into an executable.

EXPERIMENTATION

Experiment with various values for the input parameters to see which provides the best results for a given number of generations. You will compare performance of the two approaches for different numbers of cities. Start at 10 cities and go up by 1 from there until you see a runtime of greater than 5 minutes on the brute force solution. You will not have to go far to get there. Create a table that compares the time results from the approximation compared to the brute force from 10 cities up until the brute force method requires more than 5 minutes. Show what percentage of the optimal solution (eg: 120% of optimal) your approximate solutions via the Genetic and MST algorithms provide at each run. Write a report that briefly describes each approach and summarizes your results from the experiment and the conclusions you derived from the results.

DEMONSTRATION & DELIVERABLES

Submit your complete solution as a single zip file (only standard zip files will be accepted) containing A) source code for each program, B) a single Makefile to compile the program, and C) a README file (if applicable) to the corresponding Dropbox in *eLearning*. In addition, submit the source code and the Makefile as individual files to your own directory on the shared drive under `/shared/dropbox/treichherzer/cop5990`. Be sure to follow the Project Submission Instructions and Code Documentation and Structuring Guidelines posted by your instructor in *eLearning* under Course Materials to avoid unnecessary point deductions on your project grade.

Finally, be sure that your code is well organized. Functions should be refactored so that they can be easily reviewed and maintained. This means that any large function (functions with a body of about 20 or more lines of code) must be broken down into several smaller functions. You should refactor code such as the body of a loop or an if-else-statement when it exceeds a large number of lines of code or when the statement contains additional nested if-else or loop statements!

DUE DATE

The project is due on the date shown in the syllabus and *eLearning* calendar. Late submissions will not be accepted and the grader will not accept any emailed solutions. The syllabus contains details in regards to late submissions.

IMPORTANT NOTES:

1. Projects will be graded on whether they correctly solve the problem and whether they adhere to good programming practices.

2. Projects must be submitted by the time specified on the due date. Projects submitted after that time will get a grade of zero.

3. Please review UWF's academic conduct policy that was described in the syllabus. Note that viewing another student's solution, whether in whole or in part, is considered academic dishonesty. Also note that submitting code obtained through the Internet or other sources, whether in whole or in part, is considered academic dishonesty. All programs submitted will be reviewed for evidence of academic dishonesty, and all violations will be handled accordingly.

GRADING

This project is worth 100 points in total. The rubric used by the grader is included below. Keep in mind that there will be a substantial deduction if your code does not compile.

Project Submission	**Perfect**	**Deficient**		
eLearning	5 points individual files have been uploaded	0 points files are missing		
shared drive	5 points individual files have been uploaded	0 points files are missing		
Compilation	**Perfect**	**Good**	**Attempted**	**Deficient**
Makefile	5 points make file works; includes clean rule	3 points missing clean rule	2 points missing rules; doesn't compile project	0 points make file is missing
compilation	10 points no errors	7 points some warnings	3 points some errors	0 points many errors
Documentation & Program Structure	**Perfect**	**Good**	**Attempted**	**Deficient**
documentation & program structure	5 points follows documentation and code structure guidelines	3 points follows mostly documentation and code structure guidelines; minor deviations	2 points some documentation and/or code structure lacks consistency	0 points missing or insufficient documentation and/or code structure is poor; review sample code and guidelines
Graphs and the TSP Problem	**Perfect**	**Good**	**Attempted**	**Deficient**
constructs a graph in each program	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing
implements the Brute Force approach to solve the TSP problem	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing
implements the MST approach to solve the TSP problem	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing
implements the Genetic approach to solve the TSP problem	20 points correct, completed	14 points minor errors	8 points incomplete	0 points missing
complete report with discussion of approach, results from experiment, and conclusions	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing

I will evaluate your solution as attempted or insufficient if your code does not compile. This means, if you submit your solution according to my instructions, document and structure your code properly, provide a Makefile but the submitted code does not compile or crashes immediately you can expect at most 52 out of 100 points. So be sure your code compiles and executes.