**Project 1: Hamiltonian Paths**

Richard Douglas

M.S. Computer Science

Speed School of Engineering

University of Louisville, USA

[rcdoug@louisville.edu](mailto:rcdoug@louisville.edu)

1. **Introduction** 
   1. Goals:

As a first step, we want to implement various solutions to calculating hamiltonian circuits on a given set of nodes. This is in essence, the TSP problem. We want to keep track of the runtimes and best path cost for each of these algorithms to compare them later. From this data we can observe the runtime efficiency of various algorithms for an NP-Complete problem.

* 1. Definitions:

**Node:** This problem involves graphs. I will be using the terms node and city interchangeably synonymously with the term vertex from graph theory; vertices are an element of a graph. Possible details of a node are its location, what nodes it is connected to (in this case all other nodes, making this a fully connected graph), and the cost of ‘traveling’ from one node to another.

**TSP (Traveling Sales Person):** A famous combinatorial optimization problem. it is described in the goals. Given a set of cities (nodes), what is the shortest possible path to touch all of the nodes, then return to the starting city? How can we find the best path? *Can*  we find the best path in a reasonable time? How can we even be certain we have found the best path? What solutions exist for finding *good* paths in a reasonable amount of time?

**Hamiltonian Circuit:** Given a set of vertices in a graph, a hamiltonian circuit is a path that touches all of the vertices in the graph once. It is also referred to as a hamiltonian cycle.

1. **Approach** 
   1. Concept
   2. Implementation
      1. Python - I chose python because of the ease with which one can manipulate data using this language. The codes is easy to read and maintain, and many well maintained libraries exist for solving the problems that are germain to the assignment, but not the core learning objectives of the assignment.
      2. Reusability - I actually created a few different classes for this, so that I can import those into a driver script and run that. below I describe the three python files I used and what they do.
         1. TSP\_parser.py - this is the bulk of the code. It contains a class from which I can instantiate objects that make the code portable.

**Instance variables** (there are actually others but these are the most important)

**Node dictionary:** this starts as an empty dictionary and is populated with text from a tsp file keys will be all of the nodes’ names, and its values will be the (x,y) coordinates of that node, and a dictionary that will hold the distances to all of the other nodes.

**Cycle cost list:** dictionary to store the information computed by the various algorithms upon running them - algorithm run, runtime of algorithm, number of nodes run on, best calculated cycle cost (smallest distance traveled in order to make the cycle).

**Instance Functions**

**Create nodes:** a function to

algorithms, a function for calculating the costs for each node to all others,

**Calculate Costs:** to save compute on iterations, and to ease the implementation of algorithms, I computed the distances between each node and all nodes, and assigned that to the node dictionary described above. Rember, memory access is generally less expensive than compute.

**Sort Costs:** to simplify the implementation of the greedy algorithm, I sort the dictionary for each node that stores the costs to other nodes.

**Algorithms -** These all append a dictionary with the appropriate values to the cycle cost list upon completion.

**Brute force:** I believe this is the only algorithm we were required to implement. I’ll admit that I used the permutation iterator from the python itertools library. It creates an iterator object, and on each next call to that iterator object, it returns the following permutation of the input iterable. It also accepts an optional argument ‘r’, which is the desired length of the permutations you would like returned on next. It would be nearly impossible to run this algorithm without the use of an iterator object like this. I used a for loop to call the iterator until completion, a neat feature of pythonic for loops on iterator objects. in that loop I calculate the cost of traversing the permutation, then add the distance from the last node back to the starting node. I compare this calculated cost to a stored best cost and update it if less than.

**Brute force restart**: very similar, but the calculation of the current permutation cost breaks if the cost ever exceeds the current best. Although still very bad, it’s an improvement.

**Random:** randomly select a node which has not yet been visited, and move to that node. Add the cost to a running total until all nodes have been visited.

**Random restart:** same as random but with a condition similar to brute force restart where the calculation will stop if the current ever exceeds the best. This algorithm runs 20,000 times and returns the best cost. Amazingly, after the factorial increase in operations on brute force solutions, this is still WAY faster.

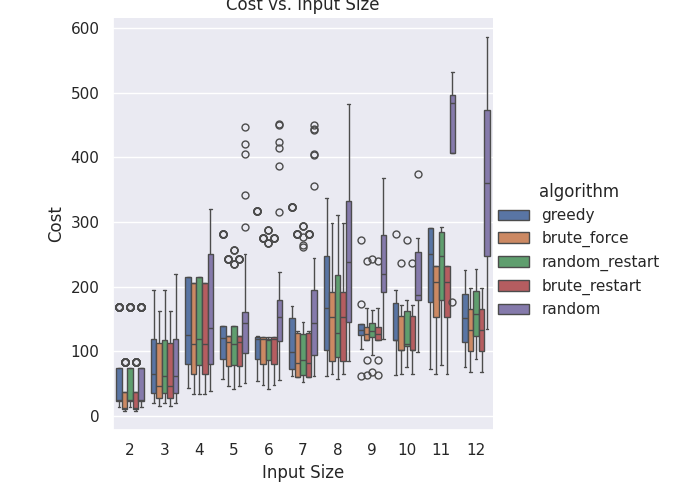
**Greedy:** Traverse the nodes by always moving to the closest node to the current node. This approach provides fairly decent results, actually, and is fast. However, it does not return consistently good results and as the input size, n nodes, increases, so does the unreliable nature of this heuristic.

* + - 1. Plotting.py - I won't go into excruciating details but I used matplotlib, pandas, and seaborn to visualize the data collected in the logs.
      2. tsp\_driver.py - again, I will refrain from providing overwhelming detail here but the code is available for reference. This script instantiates an object of the tsp\_parser, and iterates through all of the tsp files in the current working directory, applying the aforementioned algorithms to the nodes therein, and logging the results in csv files. I want to add that for each file, the algorithms are performed on a node size up to a desired value, or the maximum amount found in the tsp file, whichever is less. The script then calls the functions in the plotting class to create the plots below.

1. **Results** (How well did the algorithm perform?)

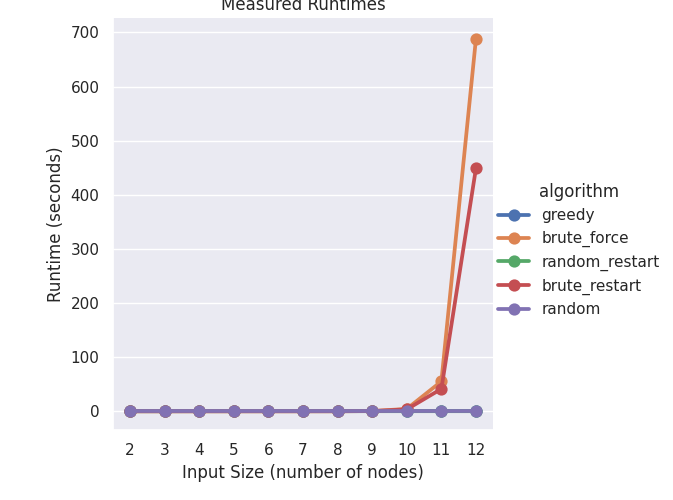
In a general sense, I achieved the goal of this project.

* 1. **Data** (Describe the data you used.) :
     1. TSP Files: I used the .tsp files provided in the .zip file Professor Yampolskiy provided. The format is shown below:
     2. Logging results: the visual representation of my results were created with the logs I saved in .csv format from the results of running the algorithms I wrote on the .tsp files. The following are written to the logs in each row:
        1. Algorithm: which algorithm I used
        2. Input size: on the iteration for
        3. Cost: the best cost calculated by the algorithm implemented
        4. Runtime: The measured runtime calculated by the difference between a timestamp taken immediately before the algorithm starts, and one taken immediately after it completes (end - start).
        5. File: the TSP file used in making the calculation. Although it isn’t germain to this report, it was useful for debugging.
  2. **Results** (Numerical results and any figures or tables.) :
     1. Figure 1: Calculated best path results



This is a categorical box plot of the best costs calculated. It is worth noting outliers here, and how to read this plot. the upper and lower limits of the colored boxes denote the 75th and 25 percentile limits of the results, the lines on either side of that mark the upper and lower outlier bounds, calculated by 1.5 \* the upper and lower outlier bounds. Outlier results falling outside of that bound are the circles. **Most noteworthy** is the horizontal line in the colored portion - the median value; this is chart gives a general sense of how algorithms performed in finding “good” solutions to the given TSP problems, though perhaps there are better means of visualizing the data, but I am out of time, and will have to work on that in future projects. It is especially of note that the brute force solution and the brute force restart solution provide the (same) optimal results. I was also surprised by how well the random restart solution performed! Mind you, it ran 20,000 iterations each time.

* + 1. Figure 2: Runtime results



This chart is much easier to read. Clearly any advantage in providing an optimal path are far outweighed by the compute cost and time cost of running such algorithms. Perfection is the thief og “good enough, it would seem.

1. **Discussion**

I have already discussed the results… But I have some future considerations. First of all, I really tried to put effort towards making the code reusable for future projects, so I hope that helps. In discussions with other students, I learned that my brute force solution it faster… remember that memory access is less expensive than computation. I also though of some improvements I could make to the implementations of the simple algorithms used in this project, but won’t go into detail there, since they are modest improvements and the algorithms are already fairly weak. In reading about tsp problems, I did notice that some researchers have used machine learning to provide solutions. I am hopeful we will discuss some of that in this class… Although I know UofL offers other classes on that subject. Lastly, I’m looking forward to implementing a GUI to display the path being traced by algorithms.

1. **References**

on hamiltonian circuits (though I already knew this:

<https://en.wikipedia.org/wiki/Hamiltonian_path#:~:text=A%20Hamiltonian%20cycle%2C%20Hamiltonian%20circuit,is%20called%20a%20Hamiltonian%20graph>.

python documentation:

<https://seaborn.pydata.org/generated/seaborn.catplot.html>

<https://docs.python.org/3/library/sys.html>

<https://docs.python.org/3/library/os.html>

<https://docs.python.org/3/library/datetime.html>

<https://docs.python.org/3/library/gc.html>

<https://docs.python.org/3/library/csv.html>

<https://matplotlib.org/stable/index.html>

<https://pandas.pydata.org/docs/index.html>

<https://docs.python.org/3/library/math.html>

<https://docs.python.org/3/library/itertools.html#itertools.permutations>

<https://docs.python.org/3/library/random.html>