**CECS 545 Project 5 report**

**Ryan Schron**

**October 28, 2019**

**Introduction**

The Traveling Salesman Problem (TSP) is an NP-Complete problem in which we are asked to find the lowest cost Hamiltonian cycle in a graph. We’ve approached solving this problem using a brute force solution, search heuristics and greedy methods, but as the problem scales these methods become untenable due to the complexity of the problem. With this in mind, we willingly trade off finding the optimal solution for getting close, in a reasonable amount of time for larger graphs.

Such tradeoffs were made in project 4 when we approached solving these problems using genetic algorithms. These algorithms are advantageous in that they do not get suck at local maxima when trying to optimize the cost of the cycle and do a good job of providing a strong solution in a reasonable amount of time.

This project attempts to improve on a genetic algorithm by creating a hybrid algorithm that implements a wisdom of crowd’s heuristic. We will run multiple instances of our genetic algorithm (as it is a stochastic algorithm) and attempt to aggregate the results to construct a more optimal path.

**Running the Program**

This program was developed using IPython due to its ability to display visualizations inline with the code. It is recommended that this program be run in Jupyter Notebooks. Once loaded, you may navigate to cell>run all to run the notebook. The input file may be changed by editing the line:

nodes = read\_tsp("Random22.tsp")

in cell 3 by passing to the function read\_tsp() the file path of the desired input.

**Code Description**

*Genetic Algorithm*

The code is initialized by reading the desired input file into memory as a list of nodes. It is then passed to the function run\_ga\_tsp() 15 times to run multiple instances of the genetic algorithm. This algorithm matches the control condition from project 4, as it was the best performer. To summarize for this project, it is run with the following parameters:

|  |  |
| --- | --- |
| Parameter | Value |
| Generations | 200 |
| Mutation Rate | 1% |
| Population Size | 200 |
| Breeding Pool Size | 10 |
| Ranking Heuristic | Elite Ranking |
| Cross Over Method | Single Point Crossover |
| Mutation Method | Random Swaps |

Of these 15 instances, the top 10 were selected for aggregation in building a new solution.

*Computing Cost*

The first step in aggregating the results is to generate a collect cost value for optimization. This is done by first finding the proportion of candidates that found each possible node connection to be part of an optimal path, and store these values in a NxN matrix in which N is the number of paths in the problem. That is, if 3 of 10 candidates found the path to include a connection between nodes 1 and 2, then cost\_matrix[1][2] would equal 0.3.

Next each value is transformed into a path cost by applying a nonlinear, monotonic function. In this case, we use the inverse of the incomplete beta function:

https://latex.codecogs.com/gif.latex?%5Clarge%20c_%7Bij%7D%20%3D%20I_%7Ba_%7Bij%7D%7D%5E%7B-1%7D%28b_%7B1%7D%2C%20b_%7B2%7D%29

In the case of this experiment, two matrices were defined. Beta1 is the result of using the transform where b1=b2=1, and Beta3 where b1=b2=3. This was done to test the differences in application as a result of the transforms hyperparameters.

*Aggregation*

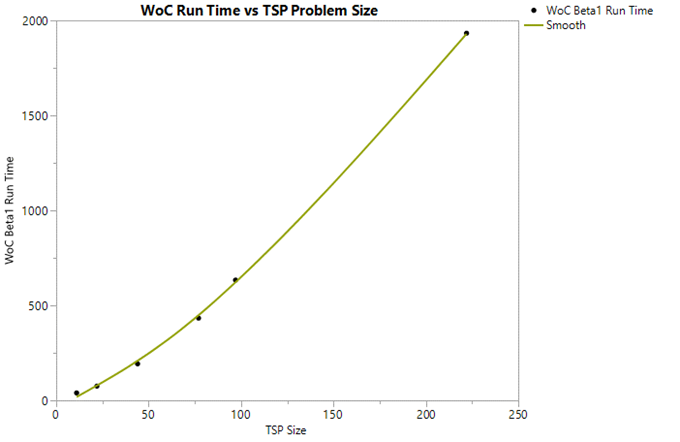
After computing the cost matrices, the problem must be solved again using these values in place of Euclidean distance, as is the case with the genetic algorithm. Because the cost matrices are also a representation of the graph, this is itself another TSP problem and can be solved using varying methods, creating a hybrid algorithm. In this case, we used a greedy heuristic because it is deterministic, and provides an elegant way to combine methods between projects in the course. This method tests, for each node, the optimal path based on the computed cost parameter, and returns the best found path.

**Code Performance**

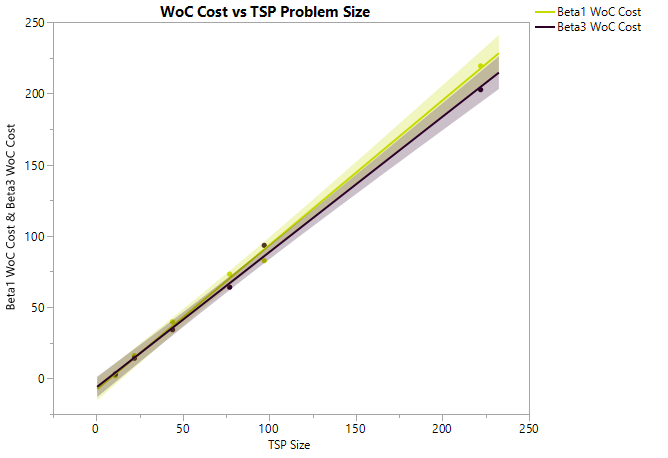
All sample graphs were run (up to n=222) with both the Beta1 and Beta3 cost results. The results are summarized in Figure 1. Figure 2 shows how the Wisdom of Crowds solution scales in complexity in relation to problem size. Figure 3 shows how the Beta1 and Beta3 cost matrices scale in optimal cost as the problems get larger, and that they act in the same way. Figure 4 examines the correlation between the wisdom of crowds and genetic algorithms solution, showing a near perfect correlation between the two. Figure 5 compares the complexity of the hybrid algorithm to using just the genetic algorithm, showing they don’t begin diverge until larger graph sizes. Finally, Figure 6 compares the cost functions, showing that there is no significant difference between the two.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Graph Size | GA Run Time | Beta1 GA Cost | Beta3 GA Cost | Beta1 WoC Cost | Beta3 WoC Cost | WoC Beta1 Run Time | WoC Beta3 Run Time |
| 11 | 39.72 | 1.909 | 3.101 | 1.909 | 3.101 | 39.725 | 39.725 |
| 22 | 75.18 | 14.09 | 12.51 | 16.09 | 14.278 | 75.33 | 75.33 |
| 44 | 192.6 | 39.0909 | 33.144 | 39.772 | 34.209 | 192.893 | 192.828 |
| 77 | 431.49 | 73.103 | 63.408 | 73.246 | 64.058 | 433.23 | 433.19 |
| 97 | 628.92 | 93.484 | 82.299 | 82.649 | 93.402 | 633.811 | 634.003 |
| 222 | 1837.935 | 219.4 | 200.752 | 219.27 | 202.63 | 1932.963 | 1933.042 |

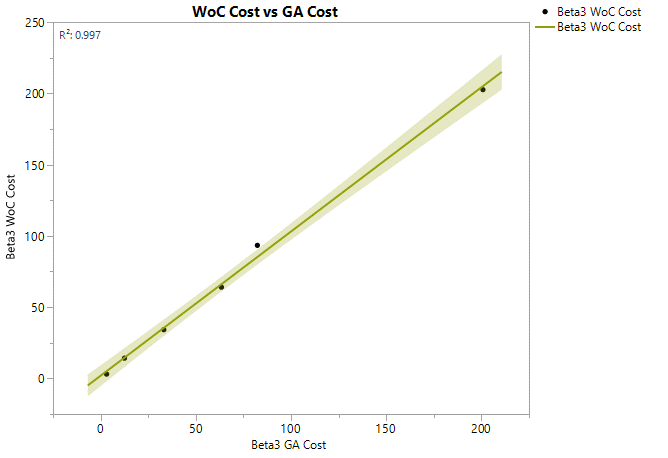
**Figure 1**



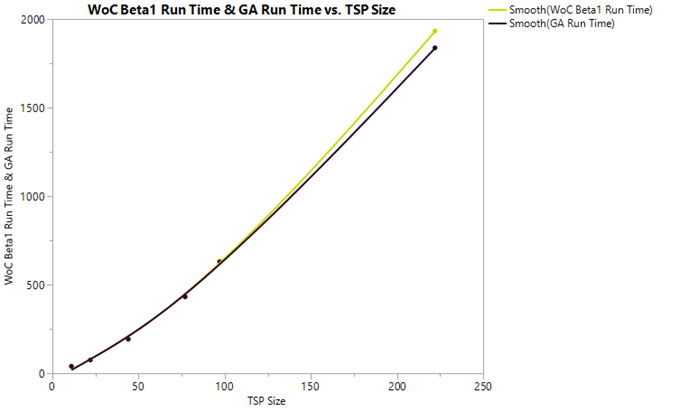
**Figure 2**



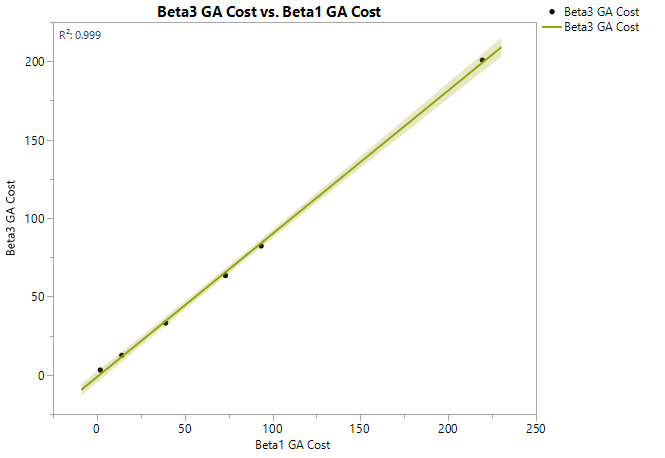
**Figure 3**



**Figure 4**



**Figure 5**



**Figure 6**

**Conclusions**

Unfortunately, in this experiment there was no difference in performance between the hybrid algorithm implemented a Wisdom of Crowds heuristic and using just the genetic algorithm. This could be due to the chosen greedy aggregation method, and it is likely the case other methods of aggregation could’ve improved upon the proposed solution. This was observed, in part, because multiple instances of the genetic algorithm were in high agreement with one another, as seen in the results of running the 11 node graph:

(351.04587994184993, [8, 11, 2, 9, 5, 4, 6, 1, 7, 3, 10]

(351.04587994184993, [4, 5, 9, 2, 11, 8, 10, 3, 7, 1, 6]

(351.04587994184993, [4, 5, 9, 2, 11, 8, 10, 3, 7, 1, 6]

(351.04587994184993, [8, 11, 2, 9, 5, 4, 6, 1, 7, 3, 10]

(351.04587994184993, [4, 5, 9, 2, 11, 8, 10, 3, 7, 1, 6]

(351.04587994185, [2, 11, 8, 10, 3, 7, 1, 6, 4, 5, 9]

(351.04587994185, [11, 2, 9, 5, 4, 6, 1, 7, 3, 10, 8]

(354.8731614418321, [4, 5, 9, 2, 11, 3, 7, 1, 10, 8, 6]

(354.8731614418321, [4, 5, 9, 2, 11, 3, 7, 1, 10, 8, 6]

(354.8731614418321, [5, 9, 2, 11, 3, 7, 1, 10, 8, 6, 4]

(354.8731614418321, [4, 5, 9, 2, 11, 3, 7, 1, 10, 8, 6],

(357.5376312445959, [2, 11, 10, 3, 7, 1, 6, 8, 4, 5, 9]

(364.9911097162351, [4, 6, 8, 11, 10, 1, 7, 3, 2, 9, 5]

(365.395007125745, [6, 4, 5, 9, 2, 11, 7, 1, 3, 10, 8]

(366.05039079159434, [1, 7, 3, 10, 4, 5, 9, 2, 11, 8, 6]

For the above 10 results, the first number is the path length, and second the identified path. The runs seemed to generally converged at an optimal path length.

To compare the methods, the optimal paths for the 11 node graphs were:

Genetic Algorithm: [8, 11, 2, 9, 5, 4, 6, 1, 7, 3, 10]

Hybrid Algorithm (Beta1): [4, 5, 9, 2, 11, 8, 10, 3, 7, 1, 6, 4]

Hybrid Algorithm (Beta3): [6, 1, 7, 3, 10, 8, 11, 2, 9, 5, 4, 6]

While the paths were different, the connections remained largely unchained with random chance determining different starting nodes.

Additionally, varying the hyperparameters of the cost function seemed to make no difference in the results. We also observed that the hybrid algorithm did not add any noticeable complexity overhead for smaller sized graphs. While the wisdom of crowds heuristic has potential to make improvements, it did not in this instance likely due to a combination of the genetic algorithms parameters and the chosen method of aggregation.

**Reference**

[1] Yi, S., Steyvers, M., Lee, M. *Wisdom of Crowds in Traveling Salesman Problems.*

[2] Yompolsiky, R. V., EL-Barkouky, A. (2010). *Wisdom of Artificial Crowds Algorithm for Solving NP-Hard Problems*