Shortest Total Path Length Spanning Tree via Wisdom of Artificial Crowds Algorithm

Madeline V. Hundley, Roman V. Yampolskiy

***Abstract*—This paper presents a hybrid genetic algorithm (GA) with Wisdom of Artificial Crowds (WoAC) approach to solving an NP-hard problem. This is a novel approach to solving Shortest total-Path-length Spanning Tree (SPST) problems. In our tests this approach achieved results up to 12% better than use of the genetic algorithm alone.**

***Index Terms*—genetic algorithm, shortest total path length spanning tree, wisdom of artificial crowds**

I. Introduction

THIS project addresses the NP-complete Shortest total-Path-length Spanning Tree (SPST) problem, proposed and discussed in [1-3]. For an edge-weighted, undirected graph, the objective of this problem is to find the spanning tree for which the total sum of total edge weights on the paths between every pair of vertices is minimized. NP-complete problems are those for whom no fast solution is known; as the size of the problem set grows the time any currently known algorithm requires to solve it increases incredibly quickly. SPST has been demonstrated as one such NP-complete problem [4, 5].

In this implementation of the problem we consider the distance between each vertex as the weight or cost of the edge. Specifically, we consider the Euclidean distances between vertices with Cartesian coordinates, calculated with the formula below. Therefore, the total distance of a path is the sum of the costs of each edge between each vertex in that path. The total-path-length of a spanning tree is the sum of the path length between every pair of vertices in the tree.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (1)$$

Genetic algorithms mimic the real-world phenomena whereby populations of biological species reproduce children based on the genes of the parents and natural selection steers each generation towards superior genes [6]. The genetic algorithm we incorporated into our hybrid approach initializes a random population of SPST solutions then reproduces successive generations of solutions. It analyzes the reproductive fitness of each member of a population, probabilistically selects stronger parents to reproduce, generates child solutions from the genes of the parent spanning trees, and sets up the next generation's population with the strongest members from the old population plus qualifying members from the child solutions.

The wisdom of crowds [7] describes the phenomenon where aggregated results from a population, or "crowd," approach the best answer to a problem, even though individual answers may have significant differences and deviations from the globally optimum solution. The Wisdom of Artificial Crowds makes use of the artificially generated populations from the genetic algorithm, treating these solutions as the "crowd" whose expertise can be consulted. One child per population is created by aggregating the edge choices of every SPST solution, probabilistically selecting a number of those edges to base the child on, then building the remainder of the spanning tree solution via locally greedy edge choices.

The program is written in Python 3.5, the graphics are generated with the module pygame, and the graphs are generated with the matplotlib module. The input provided to the program is a data file with the vertices' coordinates. The program must read in these coordinates, calculate the distance between each pair of vertices that are connected, generate a random initial population of spanning trees, then apply the genetic algorithm with wisdom of artificial crowds to generationally approach the best SPST for the graph provided.

The motivation behind developing this algorithm was to explore the interplay of the genetic algorithm component and wisdom of artificial crowds component. We also sought to explore whether wisdom of artificial crowds is a viable, successful variant to the genetic algorithm for the SPST problem as well as whether a genetic algorithm is a viable, successful source of good solutions for a wisdom of artificial crowds component.

II. Prior Work

The approach of a hybrid GA plus WoAC has been shown many times as a viable approach to solving many NP-complete problems [8-11], including the famous Travelling Salesperson problem [12]. Baseline genetic algorithms have been applied to solve spanning tree problems similar to the SPST problem [13] and other approximation algorithms have been applied to the SPST problem [14-16]. However, a hybrid GA + WoAC algorithm is, to our knowledge, a novel approach to the SPST problem.

III. Proposed Approach

To solve this SPST problem, the program uses a genetic algorithm with a wisdom of artificial crowds component. The program's first order of business is to read in the coordinates of each vertex from a file and calculate the Euclidean distances between, saving these in a dictionary for quick lookup. Its last order of business is to plot the best, average, and worst results from each generation of the algorithm and to graphically display the best spanning tree found, with the shortest total-path-length.

Overall, the genetic algorithm drives the development of a diverse, increasingly fit population every generation. The wisdom of the crowds aggregates the collective knowledge of these populations to discover better or at least comparable but diverse paths.

A. *SpanningTree Class*

This program is based on a *SpanningTree* class. Each instance of the class has an instance variable *sptree*, a set of the edges, stored as tuples, in the spanning tree. To initialize a *SpanningTree* the class should be provided a set of unique tuples, meaning no duplicates e.g. (1,2) and (2,1). Internally the *sptree* variable generates and stores both of these tuples, however, for easy lookups to the edges contained in this *SpanningTree* instance.

Upon initializing a new instance of a *SpanningTree*, the class function *calc_paths* calculates the path for every pair of vertices and stores it in the instance variable *paths*. It does this by using a recursive algorithm, the *recursive_expansion* function. Starting with the first vertex, the function "expands" this vertex, creating a "frontier" of the vertices that can next be reached given the edges constituting the spanning tree. For example, if the only edges in the spanning tree that contain the vertex '1' are the edges (1,2) and (1,3), expanding the vertex '1' would yield a frontier of '2' and '3'. It then arbitrarily chooses one of these vertices, adds it to the current path being traced, and with another call to *recursive_expansion* expands again on this new vertex. Important to cultivating the frontier is eliminating the vertices already in the current path so the algorithm does not try to expand "backwards." If the algorithm reaches a vertex whose frontier is empty, meaning the path has hit a dead end, that vertex is taken out of the path and the next option from the frontier at that level is selected and expanded. In this way, the recursion searches depth-first, backing up from dead-ends and expanding each vertex until the destination vertex has been found. The algorithm can stop searching upon first encountering the destination vertex because of an important feature of spanning trees: there can only be one valid path between any two vertices. Valid spanning trees by definition do not have cycles; if a tree contained a cycle this would offer more than one path from one vertex to another. Therefore, once the algorithm found a valid path it could stop searching as this was the only valid path. Finally, these paths are stored in the instance variable *paths*, a dictionary whose keys are pairs of vertices, stored as tuples, and whose values are lists containing each path. Only unique tuples, meaning no duplicates, are stored in this dictionary as lookups in one direction are only ever required.

Once all paths have been identified, the *SpanningTree* class function *calc_length* is called to sum the total lengths of these paths. The *calc_length* function simply needs to issue lookups to the dictionary of distances between every pair of vertices in the graph generated at the beginning of the program. The function looks up and sums the length of each edge in every path. The total-path-length of the *SpanningTree* is stored in its *length* instance variable.

Finally, the *SpanningTree* class has a *mutate* function, which will be discussed further in the context of the genetic algorithm, below.

B. *Genetic Algorithm*

To begin, the genetic algorithm sets up its initial population with the *get_init_pop* function by generating spanning trees randomly, calling the *get_rand_sptree* function. This function generates a valid spanning tree by tracking the vertices that have been added so far and randomly selecting an edge from one such vertex to one not yet added. This methodology prevents it from adding an edge "backwards" and introducing a cycle.

Each successive population in the genetic algorithm is generated by the *get_next_pop* function, which first produces one child based on the "wisdom of artificial crowds," discussed further in the next section. Then using calls to the *reproduce* function, it reproduces all remaining children required "genetically" from selected parents. *Reproduce* uses the *get_fitness* function to judge the fitness of each potential parent in the current population. Better spanning trees – those with shorter total-path-lengths – are judged to have a higher reproductive fitness and therefore have a greater probability of being selected as parents for the next generation. When two trees are selected to be parents, the *crossover* function generates two new children from those parents' "genes." Every time a child is produced there is a possibility that a mutation will occur in its "genes," handled by the *SpanningTree* class's built-in *mutate* function.

In the *get_fitness* function, the fitness ratios of every spanning tree in the population need to reflect how short their total-path-lengths are. The typical fitness ratio in a genetic algorithm tries to achieve selection of a maximum value; this will simply take each element's value out of the sum total of the population's values. If applied to the total-path-lengths this would "reward" the long paths rather than the short ones. For the SPST problem this function must instead seek to achieve selection of minimum values. Therefore, this algorithm's fitness ratio takes each tree's length and subtracts it from the highest value in the population and from 1, before dividing by the sum total of these adjusted costs. Finally, it scales so these probabilities all add up to 1, a requirement for a valid probability distribution. This method more heavily weights the shortest total-path-lengths. It also has the side effect of assigning the very worst spanning tree a fitness ratio of zero, which will cull it from potential selection in the next part of the algorithm. This is a perfectly acceptable, even desirable, side effect from this process.

The *crossover* function uses an experimentally-determined break point in each parent's "genes," discussed further in the next section, to determine the edges that will be passed on to the children. One child is first set up with a portion of edges taken from the "mom" parent tree and the second child is first set up with a portion of edges taken from the "dad" parent tree. Each of these children, currently just an incomplete set of edges, is then passed to the *complete_tree* function, along with the opposite parent from which it was first based on. The *complete_tree* function, as the name implies, completes the tree using only edges available in the parent provided. In this way, the first child is first built using "mom" then completed with "dad," and vice versa for the second child.

The *complete_tree* function is integral to both the genetic algorithm and the wisdom of artificial crowds component, whose use of it is discussed in further detail in the next section. This function first identifies all groups that exist in the edges provided. For example, if (1,2) and (2,3) are both edges in the set provided, the function identifies that {1,2,3} is a connected group of edges. Then vertices that are not yet connected, that have zero edges, are added as groups of one each by themselves. The function's next step is to choose and add edges which will connect these groups into a complete spanning tree. Starting at an arbitrary group called the *coregroup*, the function generates a list of valid edges it may choose from: all edges from a vertex in the current group to a vertex not in the group and only edges that are also present in the parent tree. It is important to point out that only edges from an added vertex to an unadded vertex are those allowed. As with the process for finding the paths in a tree or for building a random tree, selecting edges in this way ensures the tree is never grown "backwards" i.e. that it is never allowed to introduce a cycle. After identifying the valid edges, the function then greedily selects the shortest one, adding that edge to the tree. It must also identify the group that vertex belongs to and incorporate all that group's vertices into the *coregroup*. This process is repeated until all groups are incorporated, growing the *coregroup* across the graph until it includes all vertices and the child spanning tree is complete.

There are a couple important items to note regarding the *complete_tree* function. First, the arbitrary selection of the initial *coregroup* does not affect the final spanning tree. It does not matter in what order the groups are added to the *coregroup* because of the greedy nature of the edge selection and the fact any vertex not currently in the *coregroup* is allowed as the next addition. For instance, if vertex '1' is in one group, vertex '2' is in another, and their edge is the shortest available to join those two groups, it will always be the one selected whether vertex '2's group is incorporated first, somewhere in the middle, or last. Second, the greedy decision to add the shortest edge available from the parent was based on the assumption that these shortest edges available will contribute to an overall shorter total-path-length for the spanning tree. There are some instances where including a longer edge instead leads to overall better paths. However, these locally optimal choices demonstrated through testing to often be globally optimal as well. In addition, the fact the child is based in portion on a parent and not wholly on local greedy decisions helps to compensate and works to retain knowledge of which long routes are favorable.

After *crossover* has completed both kids, they are each checked for mutations using the *SpanningTree* class's built-in *mutate* function. If the test against the probability of mutation occurring hits, the function randomly selects a path that involves two edges (three vertices), discards the first edge and adds a replacement connecting the first edge straight to the third. This approach allows for a test of whether going straight from the first to the third vertex, rather than through the second vertex, offers better paths for the tree in terms of minimizing total-path-length. Because the selection of which two-edge path to mutate is random, it is equally likely to affect "central" vertices with many connections and "leaf" vertices with just one. This is an advantageous approach because either situation has the potential to benefit from such a mutation, so any path should be eligible for selection. Finally, this approach is guaranteed never to introduce a cycle, maintaining the integrity of the mutating spanning tree.

The *get_next_pop* function has a predetermined number of children it is required to generate. Children that are duplicates of trees that are in the old population or children already generated are not permitted, however. This is a useful feature to ensure that genetic diversity is maintained in the population. If all viable parents have been used to reproduce but there aren't enough new children yet, random trees are generated to be used as children. This option was very rarely needed, if ever, but is an important feature nonetheless. In addition, the crossover rate is set very high, but approximately 5% of the time two randomly-generated children are returned from reproduction. Again, this infuses some genetic diversity in to the population. After the required number of children have been generated, the *get_next_pop* function replaces up to a set number of the worst solutions in the old population with children, but only if the children are superior to those being discarded. In addition to purposefully driving successively better results each generation, it also mimics the real-world phenomenon where weak members of an animal species do not survive infancy. In this way, it is an improvement both to the function and the spirit of this genetic algorithm.

C. Wisdom of Artificial Crowds

In every population, the "Wisdom of Artificial Crowds" is applied to generate one child. Every occurrence of each edge in the population is tallied and its percentage of the total edges is calculated. These percentages serve as the probabilities the *get_crowd_wisdom* function uses when selecting edges the child will contain. The more often a particular edge occurs throughout the population, the higher the probability it will be selected to be part of the child. Edges that never occur in the population have a zero probability of being selected for inclusion in the child, a desirable effect in the algorithm.

Some wisdom-of-the-crowd implementations use only a portion of a population to study, choosing to consult only answers it deems as "experts." In our approach, we purposefully chose to use the entire population and consider every spanning tree an "expert" worth consulting. First, this is because we wanted a comprehensive view of the population's aggregate knowledge, not a narrow one. Second, it is because every spanning tree in that population has been carefully built and selected from previous populations, meaning it will contain knowledge worth considering. Because successive populations do not allow for bad spanning trees i.e. those worse than their predecessors to ever be included, every tree in the "crowd" is good enough to lend its input.

The child is initialized with these selected edges and then *complete_tree* is called to fill out the rest of the edges. The parent passed to *complete_tree* in this case is a dummy *SpanningTree* initialized with every vertex pair possible for the graph. In this way, when *complete_tree* is greedily selecting the best edges to join the disparate groups of edges within the child spanning tree, every possible connection is available as an option.

There is an important logic check in *complete_tree* that does not come into play during its use by the genetic algorithm but plays a key role during its use by the wisdom-of-artificial-crowds component. This check is part of the initial process of sorting the child's edges into their groups. If a group containing only two vertices, therefore representing an edge, is detected to have not just one but both of its vertices already in a group, it is discarded. By definition if two vertices are already in a group together, an edge between them would introduce a cycle. Such situations may occur because *get_crowd_wisdom* selects edges only based on probabilities. This means it may select edges that, if all included, create a cycle e.g. (1,2), (2,3), (1,3). This check in *complete_tree* vets the edges of the child it has been provided. For example, the function will begin by building a group from (1,2) and (2,3) of {1,2,3}, then detect that (1,3) is invalid and drop it. This situation does not occur during the genetic algorithm because the child passed to *complete_tree* in that instance is based on a parent tree which is guaranteed to be valid (unless it was somehow otherwise corrupted). Therefore, if the parent does not contain any cycles then the subset of the parent used to build the child is also guaranteed to be cycle-free.

The greedy decisions of *complete_tree* drive the development of optimal spanning trees. The inclusion of every path possible in the parent in the case of the wisdom-of-artificial-crowd child helps contribute both to optimal spanning trees and a diverse population because it may include a new edge not seen anywhere else in the population but that turns out to be highly optimal.

IV. Experimental results

A. Data

The data used was generated by the Concorde TSP Solver [17]. While designed to solve Travelling Salesperson Problems, it is also suitable for generating datasets for a variety of similar problems. Using the Windows GUI, we could randomly generate graphs of any number of vertices, which are saved in a standardized TSP format listing the vertices numbered 1 to n with corresponding coordinates. The program reads in this file line by line and extracts the coordinates. Below is an example for 10 vertices:

```
NAME: concorde10
TYPE: TSP
COMMENT: Generated by CCutil_writetsplib
COMMENT: Write called for by Concorde GUI
DIMENSION: 10
EDGE_WEIGHT_TYPE: EUC_2D
NODE_COORD_SECTION
1 22.549020 89.029536
2 23.039216 81.434599
3 30.392157 79.324895
4 40.277778 80.379747
5 38.071895 60.759494
6 23.774510 59.704641
7 25.245098 67.721519
8 30.065359 66.244726
9 36.029412 70.886076
10 49.264706 71.940928
```

B. Results

The genetic algorithm alone yielded the results shown in Table 1, all from 30 generations per execution of the algorithm. Figures 1 and 2 show example outputs of these results. Figures 1A and 2A graph the worst, average, and best solutions per generation in red, blue, and green, respectively. Figures 1B and 2B display the resulting best spanning tree found at the completion of the algorithm.

Vertices	**Population**	**Runtime (s)**	**Best Length**
30	50	37.56215	33174.15536
30	50	42.87745	33454.12696
30	50	38.18318	37393.45269
50	50	177.57016	115716.17169
50	50	235.58247	119350.27524
50	50	188.93781	125145.69157
77	77	1212.46735	277132.34661
77	77	1380.45296	286723.39976
77	77	1442.40950	271129.73645

Table 1: Results from Genetic Algorithm

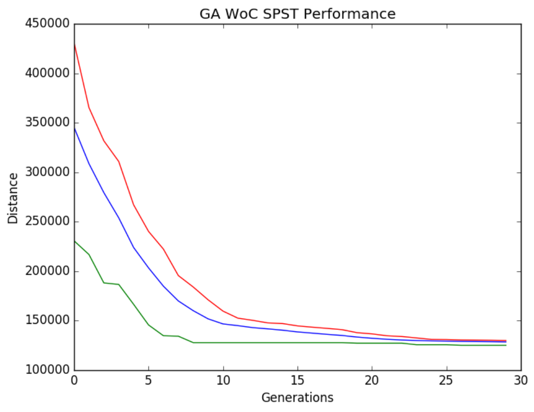

Figure 1A: Worst, Average, and Best Solutions per Generation

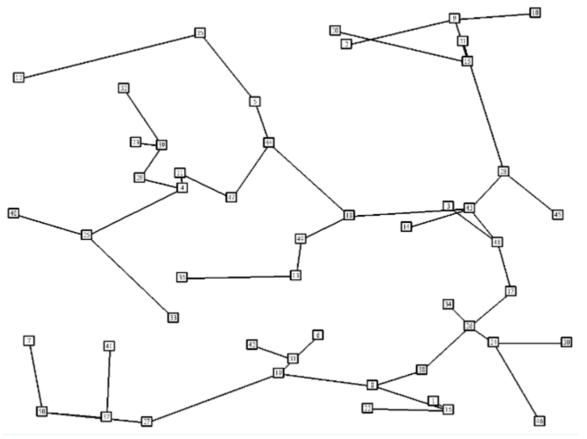

Figure 1B: Spanning Tree of Total-Path-Length 125145.69157

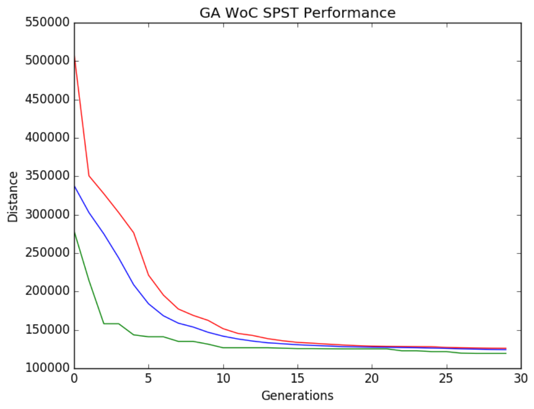

Figure 2A: Worst, Average, and Best Solutions per Generation

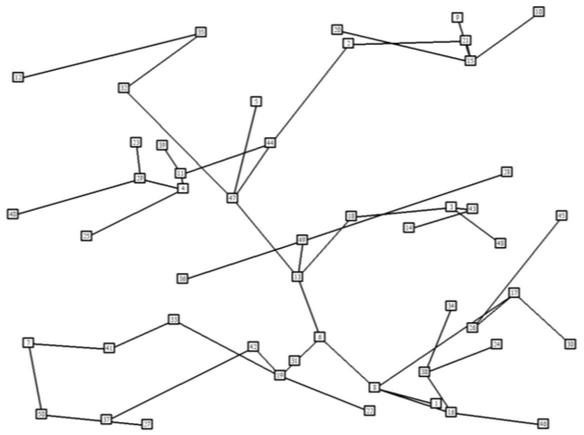

Figure 2B: Spanning Tree of Total-Path-Length 277132.34661

The full hybrid algorithm yielded the results in Table 2, also from 30 generations per execution of the algorithm. Figures 3 and 4 show example output from the hybrid algorithm. Figures 3A and 4A graph the worst, average, and best solutions per generation in red, blue, and green, respectively. Figures 3B and 4B display the resulting best spanning tree found at the completion of the algorithm.

Vertices	Population	Runtime (s)	Best Length
30	50	37.52615	32912.40819
30	50	39.67027	33800.07715
30	50	35.05000	34155.32438
50	50	215.91535	103152.67340
50	50	268.56736	106292.14450
50	50	226.07593	107272.21349
77	77	1723.61959	253756.81567
77	77	1423.14640	257199.90750
77	77	1375.86069	262635.91412

Table 2: Results from Hybrid Algorithm

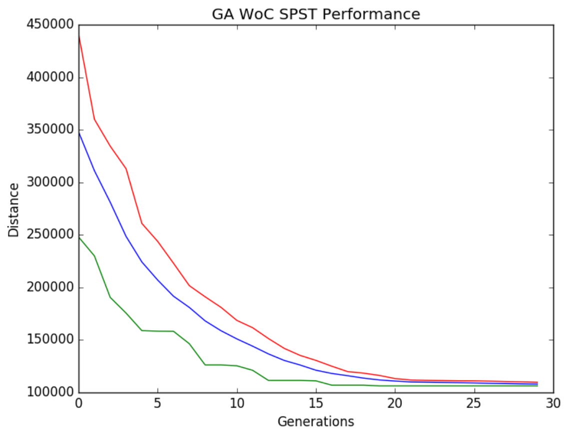

Figure 3A: Worst, Average, and Best Solutions per Generation

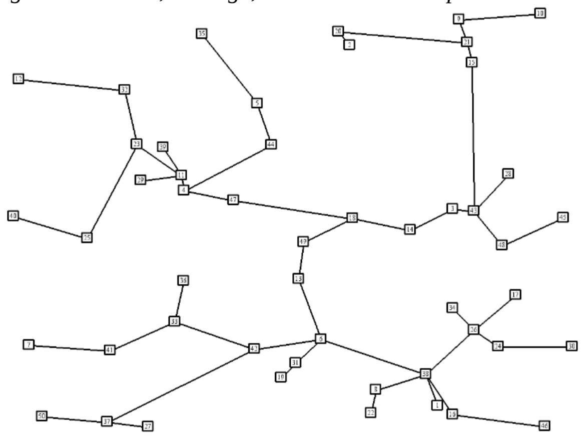

Figure 3B: Spanning Tree of Total-Path-Length 106292.14450

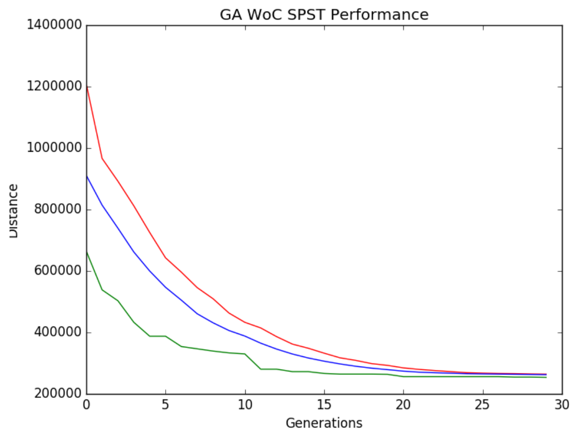

Figure 4A: Worst, Average, and Best Solutions per Generation

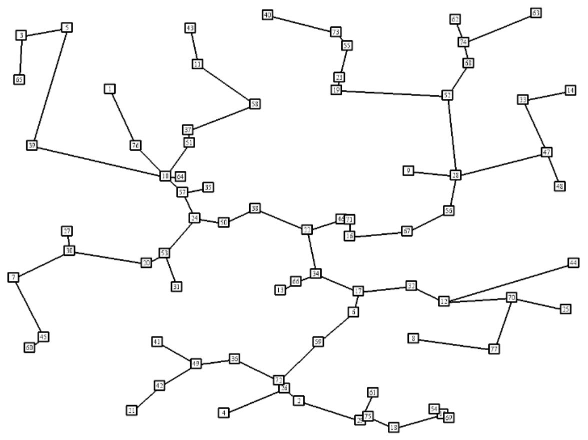

Figure 4B: Spanning Tree of Total-Path-Length 253756.81567

For 30 vertices and a population of 50, the hybrid algorithm required 5.38% less time and produced results of an average 3.03% shorter final total-path-length over the genetic algorithm alone. For 50 vertices and a population of 50, the hybrid algorithm required 18.02% more time and produced results an average 12.07% shorter than the genetic algorithm alone. For 77 vertices and a population of 77, the hybrid algorithm required 12.08% more time to execute and produced results an average 7.35% shorter than the hybrid algorithm alone.

This hybrid approach was capable of finding good solutions to larger datasets, as shown here in Table 3, though with long runtimes. Figures 5 and 6 show example outputs of these results. Figures 5A and 6A graph the worst, average, and best solutions per generation in red, blue, and green, respectively. Figures 5B and 6B display the resulting best spanning tree found at the completion of the algorithm.

Ver-tices	Popu-lation	Gene-rations	Runtime (s)	Best Length
80	80	30	1976.99008	271001.70025
80	40	30	853.11680	289119.11059
100	100	20	3228.44966	507786.11792
100	20	20	702.63019	663055.61606
222	100	10	38360.50510	3912193.54795
222	20	10	8892.24261	5096472.87880

Table 3: Results from Hybrid Algorithm on Large Datasets

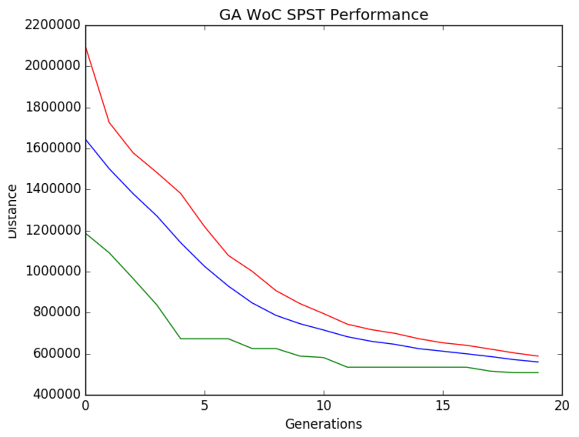

Figure 5A: Worst, Average, and Best Solutions per Generation

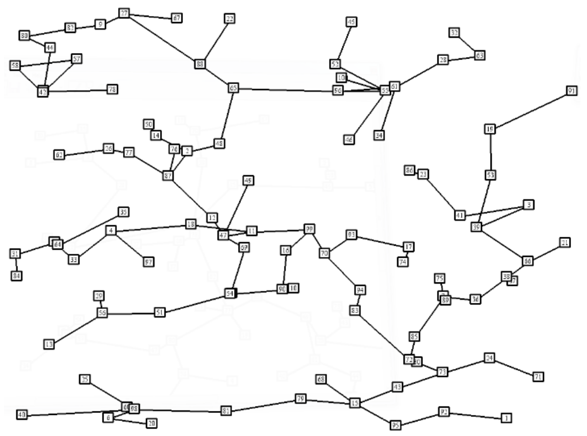

Figure 5B: Spanning Tree of Total-Path-Length 507786.11792

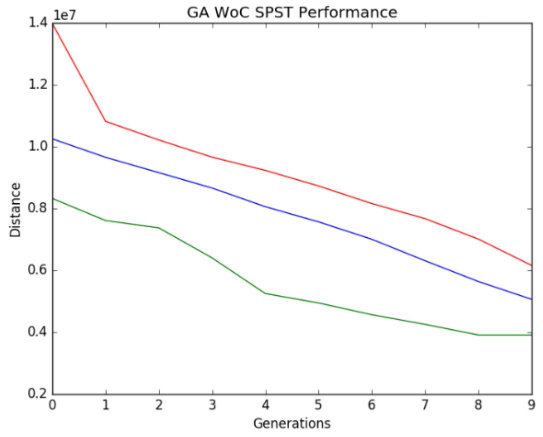

Figure 6A: Worst, Average, and Best Solutions per Generation

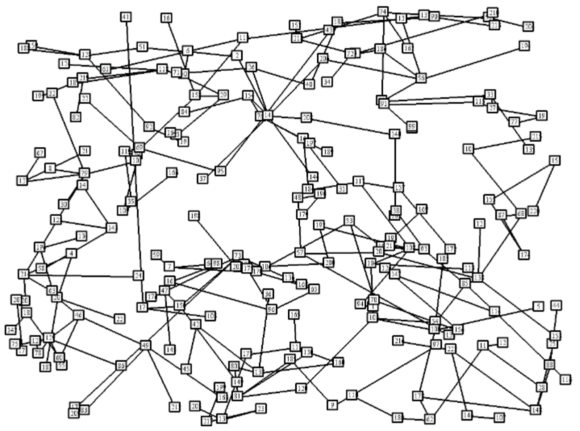

Figure 6B: Spanning Tree of Total-Path-Length 3912193.54795

V. Conclusions

This combination of genetic algorithm with wisdom of artificial crowds is complete, meaning it will always find an answer. It is not guaranteed to be optimal, however this algorithm is still successful and is capable of not only finding some result but with good implementation can optimize to find a better, if not best, result.

The relationship of the genetic algorithm to the crowd-wisdom component proved to be advantageous. As the genetic algorithm developed artificial results to inform the wisdom of the crowds, the resultant child would in turn drive the diversity and fitness of the genetic algorithm's population. Therefore, the hybrid algorithm as a whole developed good results.

Finally, it is also important to highlight that while not optimal, this algorithm is relatively fast. Some algorithms are incapable of finding any answer for datasets of the size tested here. Overall, as a problem-solving technique, this combined algorithm has many advantages in that it is relatively efficient, complete, and finds a good answer to the Shortest Total-Path-Length Problem for large datasets.

References

[1] T. C. Hu, "Optimum communication spanning trees," *SIAM Journal on Computing*, vol. 3, no. 3, pp. 188-195, 1974.

[2] D. S. Johnson, J. K. Lenstra, and A. Kan, "The complexity of the network design problem," *Networks*, vol. 8, no. 4, pp. 279-285, 1978.

[3] P. M. Pardalos, D. W. Hearn, and W. W. Hager, *Network optimization*. Springer Science & Business Media, 2012.

[4] M. R. Gary and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-completeness," ed: WH Freeman and Company, New York, 1979.

[5] K. Mehlhorn, "Selected Topics in Algorithms 2009 NP-completeness of Minimum Fundamental Cycle Basis Problem," 2009.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press Cambridge, 2001.

[7] J. Surowiecki, *The wisdom of crowds*. Anchor, 2005.

[8] D. Bonomo, A. P. Lauf, and R. Yampolskiy, "A crossword puzzle generator using genetic algorithms with Wisdom of Artificial Crowds," in *Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games (CGAMES), 2015*, 2015, pp. 44-49: IEEE.

[9] L. H. Ashby and R. V. Yampolskiy, "Genetic algorithm and Wisdom of Artificial Crowds algorithm applied to Light up," in *2011 16th International Conference on Computer Games (CGAMES)*, 2011.

[10] J. Redding, J. Schreiver, C. Shrum, A. Lauf, and R. Yampolskiy, "Solving NP-hard number matrix games with Wisdom of Artificial Crowds," in *Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games (CGAMES), 2015*, 2015, pp. 38-43: IEEE.

[11] A. C. Port and R. V. Yampolskiy, "Using a GA and Wisdom of Artificial Crowds to solve solitaire battleship puzzles," in *Computer Games (CGAMES), 2012 17th International Conference on*, 2012, pp. 25-29: IEEE.

[12] R. V. Yampolskiy and A. El-Barkouky, "Wisdom of artificial crowds algorithm for solving NP-hard

problems," *International Journal of Bio-inspired computation*, vol. 3, no. 6, pp. 358-369, 2011.

[13] J. Fletcher, T. Fernando, H. Iu, M. Reynolds, and S. Fani, "A case study on optimizing an electrical distribution network using a genetic algorithm," in *2015 IEEE 24th International Symposium on Industrial Electronics (ISIE)*, 2015, pp. 20-25: IEEE.

[14] B. Y. Wu, K. M. Chao, and C. Y. Tang, "Approximation algorithms for the shortest total path length spanning tree problem," *Discrete applied mathematics*, vol. 105, no. 1, pp. 273-289, 2000.

[15] R. T. Wong, "Worst-case analysis of network design problem heuristics," *SIAM Journal on Algebraic Discrete Methods*, vol. 1, no. 1, pp. 51-63, 1980.

[16] B. Y. Wu, G. Lancia, V. Bafna, K.-M. Chao, R. Ravi, and C. Y. Tang, "A polynomial-time approximation scheme for minimum routing cost spanning trees," *SIAM Journal on Computing*, vol. 29, no. 3, pp. 761-778, 2000.

[17] W. Cook, "Concorde TSP solver," *See:* http://www. *tsp. gatech. edu/concorde. html*, 2005.