5-1-2021

Comparing a New Algorithm for the Traveling Salesman Problem to Previous Deterministic and Stochastic Algorithms

Edward Friesema

COMPARING A NEW ALGORITHM FOR THE TRAVELING SALESMAN PROBLEM TO PREVIOUS DETERMINISTIC AND STOCHASTIC ALGORITHMS

By

Edward Friesema

Bachelor of Science - Electrical Engineering
Pennsylvania State University
1998

Master of Engineering - Electrical and Computer Engineering
University of California, San Diego
2001

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
May 2021

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

May 14, 2021

This thesis prepared by

Edward Friesema

entitled

Comparing a New Algorithm for the Traveling Salesman Problem to Previous Deterministic and Stochastic Algorithms

is approved in partial fulfillment of the requirements for the degree of

Master of Science – Computer Science
Department of Computer Science

Evangelos Yfantis, Ph.D.
Examination Committee Chair

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Interim Dean

Andreas Stefik, Ph.D.
Examination Committee Member

Hal Berghel, Ph.D.
Examination Committee Member

William Culbreth, Ph.D.
Graduate College Faculty Representative

Abstract

The challenges of drone navigation have driven many advances in the development of autonomous systems. Unmanned Autonomous Vehicles(UAVs) operate in a rapidly changing flight space and have to balance a complex set of constraints and objectives. Many of these objectives can be represented in variations of the classic Traveling Salesman Problem. Numerous approximate solutions to TSP have been proposed over the years, but these approaches have difficulty when adding new constraints that require rapid recalculation of the solution. Either they are fast but do not provide solutions that are close to the optimum, or they provide excellent solutions but they take a large amount of computational resources to arrive at a solution. We are proposing a new algorithm that will be able to provide a very competitive solution to TSP compared to other probabilistic and deterministic approaches. We will demonstrate that this new algorithm is robust and efficient enough to be effective within the strict computational constraints of a typical UAV avionics system.

Keywords : Traveling Salesman Problem, genetic algorithm, divide-and-conquer, UAV navigation, Prim's Algorithm,

Acknowledgments

I would like to thank my sisters and parents for their unwavering support and encouragement. Also I would like to thank my advisor Dr. Yfantis without his mentorship this research would not have been possible. The code for this application is available at my GitHub page (http://github.com/efriesema) for any interested parties to explore and test.

Table of Contents

List of Tables

List of Figures

Introduction

In this paper, we introduce a new algorithm for the long term navigation of unmanned autonomous vehicles(UAVs). Over a century ago. the Wright brothers made heavier-than-air flight a reality and the world changed drastically but the skies are still dangerous. Now UAVs are utilized in many applications that would prove too dangerous or too challenging for a human pilot. The most fundamental challenge of an avionics systems is the limitation of fuel. Energy must be spent to keep a machine in the air. The problem of aerial reconnaissance involves covering an area from enough angles to get a complete view regardless of buildings, hills, or other obstructions. The Traveling Salesman Problem(TSP) can represent the challenge of the avionics system to find the most fuel-efficient path that covers all of the space needed. In addition, there is the constraint that flying upwards is much more expensive than flying downwards. So our optimization algorithm must take into account how to create a path that starts at the highest level and gradually descend.

The Traveling Salesman Problem has obvious applicability to the current problem where the cost function consists of the Euclidean distance between points with an added penalty for each time the path ascends. Fuel considerations make any ascending path a much higher cost than a path of the same distance which is level or descending. Genetic algorithms(GA) and Simulated Annealing(SA) Algorithms both provide us with a probabilistic methods which are both very efficient and easy to implement in a parallel processing environment. Prim's algorithm and our method both take a deterministic approach where Prim's is the classic greedy algorithm and our algorithm uses the divide-and-conquer approach. We will outline each method and compare results on multiple point sets. Our results indicate that our new algorithm presents a set of results which compare favorably to other deterministic algorithms and to probabilistic approaches both in speed and in quality of the achieved solution.

This paper is part of a larger project to design a UAV that can maintain altitude for long duration to provide surveillance of large areas of national forest land in order to protect them from fires, diseases, and to provide detailed 3-D mapping of the forest canopy. In emergency situations like forest fires, the rapid destruction of cellular, radio and traditional phone line infrastructure can create immense challenges in coordinating and planning civilian emergency response. Long-duration UAV's are uniquely qualified to support firefighting efforts on several

levels. The rapidly changing and highly dangerous nature of a large-scale fire make it critical that the system can plot an optimal course rapidly and efficiently with its limited onboard computing resources.

1) Do deterministic algorithms that approximate(but do not always achieve) the optimal solution presents the best combination of an usable solution that is found in an efficient amount of computation.
2) Do the probabilistic methods obtain solutions that are close enough to the optimal to provide useful solutions in real world applications given their computation speed and ability to handle cost function of a more complicated nature.

Background

Traveling Salesman and NP-hard problems

For decades, the Traveling Salesman Problem(TSP) has been a classical example of an NP-hard problem. The usefulness of being able to find the shortest path that touches all points in a set is so apparent that computer scientists have spent prodigious energy in trying to find innovative solutions to the TSP. The TSP is a classic example of the NP-hard class with a huge solution space that increases as (N-1)!/2. For even a small value of N = 10 number of possible directional paths is 181,440. Its obvious applicability to networking and logistics problems has driven many computer scientists to attempt to find an optimal solution in less than the time required for the prohibitively expensive brute force method. In the 70's, many attempts using dynamic programming and a version, which approximates a solution using Prim's algorithm. To setup a baseline and to find what the truly optimal solution is for comparison we implemented a brute force version of the algorithm which attempts every possible path. The process grows by roughly the factorial of n so in practice for this project we capped testing our sets at 14 points. The classic difficulty combined with a very clear and well-defined formulation has made TSP a classic benchmark for testing the power of optimization algorithms[4].

As the title of this paper indicates we will be comparing our algorithm with one deterministic algorithm and two probabilistic algorithms. For clarity's sake we would like to clearly define these three terms. When we call an algorithm **deterministic,** we mean that at each point in the algorithm the next step is entirely determined by its current state and any inputs to the algorithm. There is no random chance involved in the algorithm deciding on its next step. By contrast we

define an algorithm as **probabilistic**, using the Cambridge English Dictionary's definition, "Based on or adapted to a theory of probability; subject to or involving chance variation." Finally, when we call an algorithm **stochastic**, we mean that the selection process of the algorithm can be well-defined by a random probability distribution.. Often probabilistic and stochastic are used interchangeably but there is a distinction. Prim's algorithm and our new algorithm are both deterministic algorithms, even though there is a random element in Prim's when it comes to choosing which node will serve as the root node. We will cover the implications of this more in the results section. Simulated annealing and genetic algorithms are both probabilistic and stochastic algorithms. They are probabilistic in that the next steps and the mutation procedure from one generation to the next are both determined by chance. They are stochastic in the sense that for SA the probability of keeping a non-improving solution is determined by an exponential distribution with the key parameter, *T*. While in our genetic algorithm, the random selection of the next generation and the start and end of the ordered crossover mutation are both determined by uniform random variables.

The Traveling Salesman Problem is mathematically described where we are given a complete undirected graph *G=(V,E)* that has a nonnegative integer cost *c(u,v)* associated with each edge $(u,v)\in E$ and we must find a Hamiltonian cycle(a tour) of *G* with minimum cost. Other costs than the squared distance can be used and additional penalty terms and rewards can be added which allow for solutions that answer problems even outside of the boundaries of traditional TSP. There are several TSP variations such as the Bottleneck and Smuggler problems which offer additional constraints on the classical TSP that can strongly affect the optimal solution.

Simulated Annealing

Probabilistic methods are nothing new in the field of computation. Since the 1950s at Princeton's Institute of Advanced Study John Von Neumann and Stanislaw Ulam, working on complex problems ranging from weather prediction to hydrogen fusion in a thermonuclear device, saw the potential for computers to find solutions to analytically intractable problems by testing a series of solutions generated by some stochastic probability distribution. Careful selection of the probabilities and the evaluation of the success of the method yielded surprising results. Von Neumann dubbed them Monte Carlo methods after the famed European casino and they have been an powerful part of the computer scientist's toolkit ever since.[10,11]

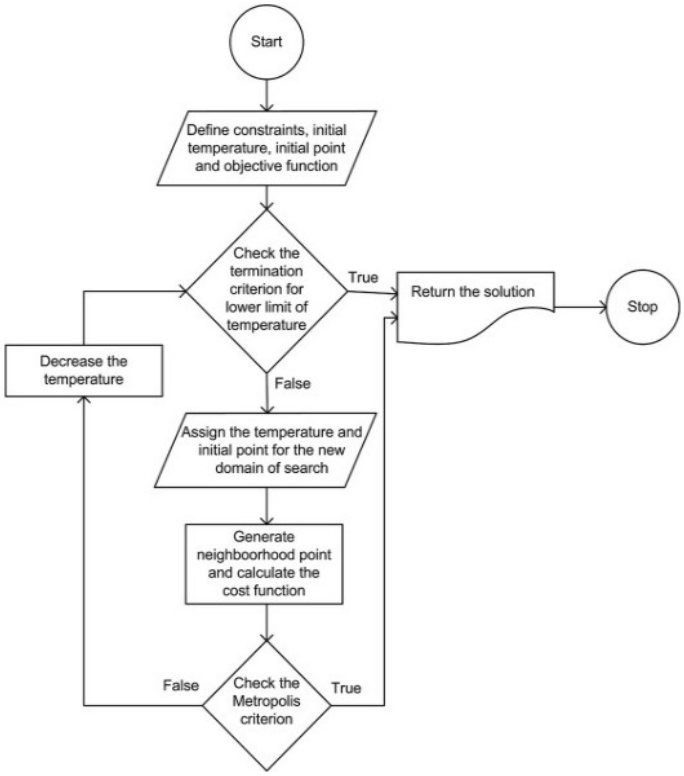


Figure 1: Simulated Annealing Flow Diagram

Simulated annealing uses a principle borrowed from statistical mechanics and thermodynamics to allow it efficiently search a large solution space with possibly many local minima and still find an optimal solution. It makes an analogy to the controlled thermal cooling of a highly heated metal. Initially, as the T parameter is high the solution is randomized and if it produces a superior solution then take it, but if it finds a solutions that is worse there is still a chance that it will accept this solution proportional to e^{-T}. While at first glance this seems counter-intuitive, it allows the algorithm to very efficiently search the possible state space. It avoids the trap of the more straightforward greedy approach by allowing for the possibility of solutions which may not immediately improve the current solution to still be explored. If there might be superior solutions outside of the local minima, then simulated annealing will check for that possibility. As the solution improves and the chance of finding better solutions decreases, the T parameter descends(i.e. decreases by 5%). At each step fewer worse solutions are accepted and the algorithm converges upon a global minimum. The algorithm works by making an analogy between a cooling metal finding the lowest possible energy state. For our problem the cost function of the path distance is substituted for energy equation in a classical physics example. In both cases nature and our program are searching for a configuration which minimizes the cost function. Figure 1[8] provides a flow chart of the algorithm.

While simulated annealing and genetic algorithms both use stochastic methods to efficiently explore the solution space, there are two key differences between our simulated annealing approach and the genetic algorithm approach that we explore next. First, the simulated annealing method does not select from a population highest fitness scores. In simulated annealing there is only one potential path that is constantly being mutated. If it improves then the new mutation is

kept. If it has a lower fitness score then it still might be kept to generate future solutions. The probability of this depends on the current value of the temperature parameter, T.

Second, the string is varied by selecting between two different types of mutation with a probability of 50% each. The two options were produced by using either a Reverse mutation, where a random sub-string of the path was reversed in order, or a Transport mutation, where one sub-string was moved to some other part of the path. The two methods are illustrated in Figures 2 and 3.

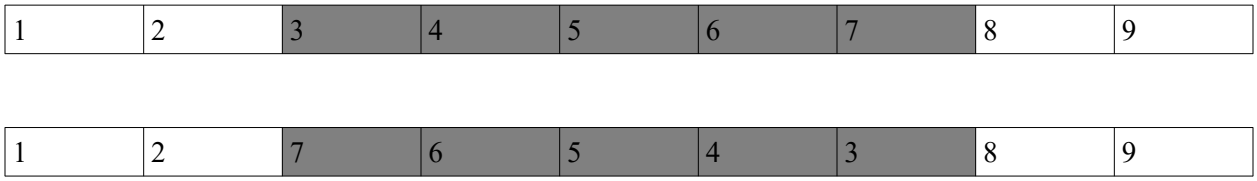


Figure 2: Reverse Mutation

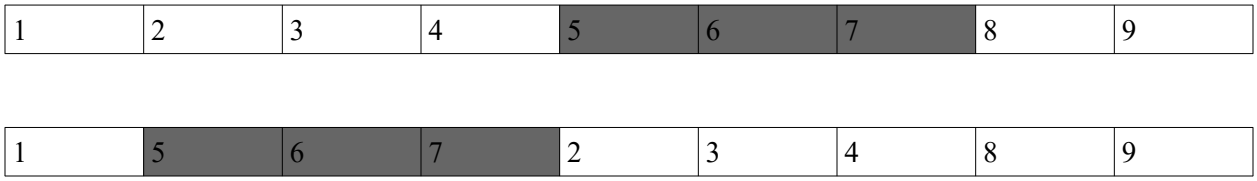


Figure 3: Transport Mutation

Either method was selected with a 50%. probability. It criteria for determining when to decrease the temperature is known as the **Metropolis criteria,** named after Nick Metropolis, a founder of the simulated annealing method [11], where it required a set number steps , in our case 10*n where it

created no improvements. In our implementation the mutation rate would then increase two times after a set number of attempts to try to find the truly optimal solution. This gave our algorithm one last chance to check that it had truly centered on an optimal solution. An important point to consider which arises with genetic algorithms is that the mutation cannot duplicate and nodes in the solution or else it will not qualify as a valid TSP solution.

Genetic Algorithms

The basic genetic algorithm takes advantage of the simple evolutionary concept that the best solutions have elements that if combined together would have a high probability of producing even more efficient solutions. To keep with commonly used terms in computer science a few terms will be defined for future use. The **population** consists of a set of strings which represent a set of possible solutions to the problem. A **generation** is the population at one point in time, for our algorithm each point in time produces a new generation that is populated from the fittest member of the last generation. A **chromosome,** or **individual**, is a single string that represents one member of the population. A **gene,** is any single element of a chromosome**.** The fitness of any individual chromosome for the solution is determined by a **cost function, c,** which is a function which is a mathematical expression which takes a chromosome and assigns it a value. Optimization methods are then used to try to maximize or minimze that value.
. Whether the problem is looking for a low score(minimization) or a high score(maximization) does not make any practical difference in how we approach the problem.

The mutation takes advantage of the fact that the top performers from the previous generation can be selected to create the next generation. The simplicity of this approach belies how efficiently it covers the huge solution space of TSP. The interested reader can review Goldberg's text[3] for a mathematical derivation of the proposition that for a population size of N a GA covers the solution space at a rate of $O(N^{3)}$. So for a relatively small amount of memory a set of

parallel processors can easily apply the algorithm to find a top-percentile solution in a smaller fraction of the time than that required for more traditional TSP algorithms. Figure 2 shows the essential steps of the algorithm.

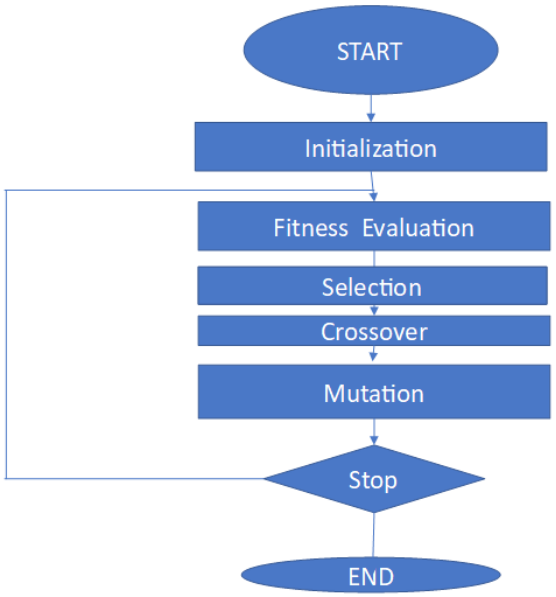


Figure 4: Canonical GAs

The selection process for the next generation can be based on the roulette method The first step is we sort the generation is what percentage of the fittest members of the population were to be used to create the next generation. for our algorithm we selected the top 25%. their probability of selection was based on the value of each chromosome cost function as a percentage of all of the top quarter, the elite of the population. Then a uniform random variable from [0,1] was used to select the members of the next population. so the chance that a chromosome is selected for the next generation is weighted by is fitness compared to all of the other elite members of its generation. This presented a challenge when designing our genetic algorithm. It was said earlier

that there was no practical difference between maximization or minimization version's of the problem. But one difference for us occurred when we were calculating the roulette table for selecting the next generation. The initial plan is to divide each member of the select population's score by the sum of all the select population's scores, then assign a stretch of the value from [0,1] to each member. A uniform random variable selects each member of the next generation and the chromosomes with the highest scores have the highest probability to get selected for the next round.

But in a minimization problem we want the opposite result to occur. We want the chromosomes with the lowest score to have the highest probability to be selected for the next generation. So our process is as follows:

1. As in the maximization case, we first take the score of each individual and divide it by the sum of the population's scores
2. Now we invert each value. 20% becomes 5, 50% becomes 2.
3. Now we take an average of the new values.

The following table illustrates the process for a population of four.

Original Score	Weighted score	Inverse	Final Weighted Score
10	0.10	10	0.48
30	0.30	3.33	0.16
40	0.40	2.5	0.12
20	0.20	5	0.24

Table 1: Minimization Roulette Calculation Example

As can be seen the new calculation gives an appropriate weight to the more desirable lower scores by giving them a higher probability.

One consideration for genetic algorithms when applied to TSP is that the more traditional crossover mutation and randomization methods for generating a population will not work for TSP because a viable solution must only have one instance of each node in the path. Therefore a different mutation scheme called ordered crossover is used instead. The ordered crossover mutation requires that the randomly selected sub-string from the first parent be inserted into the second parent. Then any elements that have been replaced in the child are then moved out and replace the sub-string elements in their new location in the child. Figure 5 illustrates the process.

PARENTS:

1	2	5	6	9	3	7	8	4

5	6	4	2	8	9	1	3	7

CHILDREN:

7	2	5	6	8	9	1	3	4

5	6	4	2	9	3	7	8	1

Figure 5: Ordered Crossover Mutation

Prim's Approximation Method

Prim's algorithm has been known since 1953 although it was later discovered that a Czech theorist named Jancek had discovered the method and published as early as 1937. The basic approximation method is simple and is outlined in Cormen [1.] In there the author lays out a detailed proof that, at best, this can approximate a solutions to TSP but it does give some reasons why we might be able to expect a good solution, if not optimal.

Pseudo-code for Prim's Algorithm:

1) Select a node r to be the root node of the MST
2) Perform Prim's to create MST
    1. for each node u in G,V
        a) u.key = infinity
        b) u.pi = NULL
    2. r.key =0
    3. Q = G,V
    4. while Q is not empty
        a) u = EXTRACT_MIN(Q)
        b) for each node v adjacent to u
            1. if v is in Q and w(u,v) < v.key
                i. v.pi =u
                ii. v.key = w(u,v)
3) Perform a preorder traversal of the completed MST for TSP solution.

It requires us to build a Minimum Spanning Tree(MST) of the graph, assuming all paths connecting the nodes are possible and bidirectional. The key values of each node are its

distances to the parent node in the MST. An initial point is chosen at random to be the root of the MST. The pi values are a pointer to the nodes parent in the MST. Prim's algorithm is used to build the MST and then a preorder traversal produces the solution path. Even though this method is deterministic technically it does have one random element: the selection of the root node. In our experience, if the root node selected happens to be close to one of the endpoints of the truly optimal path than Prim's approximate method produces a very satisfactory, if not necessarily optimum, solution. But if the root node is somewhere in the middle of the path some large divergences in the final path score can occur. Of course as the number of points grow the chance that you will randomly choose a point near the endpoint decreases substantially. And the resultant path can be many times longer than the optimum. But in Prim's Favor as it not even attempting to search the space of possible paths but instead uses a priority queue which can be sorted and read out in $n*logn$ time the speed of this algorithm is orders of magnitude faster than even the stochastic simulated annealing and genetic algorithms we have attempted. Unfortunately one limitation Prim's has is that it cannot be improved by parallel implementation at any point. While this may not seem like a large detraction given the speed in which it find s a solution it can still be considered a disadvantage when dealing with problems of a large value of n. We will cover these issues and trade-offs in more detail in the results section.

The New Algorithm

Our new algorithm uses a deterministic approach that finds the points that our nearest to each other and combining them into longer and longer sub-paths. It is similar to some cellular automata methods ,in that the closest points can be viewed as seeds that grow sub-paths that eventually merge. While it would take careful work to prevent race conditions it is clear that this algorithm could be multi-threaded with a thread created for a set number of sub-path seeds. Whether this would prove worth the extra overhead of keeping track of each thread and when they would need to be merged would have to be tested with large benchmark data sets.. In practice this algorithm works very quickly in roughly $O(n^3)$ time. Unlike Prim's algorithm, the new algorithm does not require initial random selection of starting point and therefore its results do not vary from one run to the next. It will always produce the same result and that result proves to be very close to the optimal as we will demonstrate in the results section. The next section will outline the algorithm steps in detail.

The new algorithm uses a minimum priority queue to create a sorted list of each points nearest neighbor. As nearest neighbors are merged into sub-paths the sub-paths begin to grow eventually merging with new points or one another. Our merge step takes into account the possibility that the new point or path might make for a shorter overall path if merged internally into the new sub paths instead of linking the endpoints its overall a simple idea with precedents in some cellular automata and divide-and-conquer algorithms where we solve the problem of which sub-path most efficiently connects a small group of nodes and then merging those sub-paths together. For purposes of our algorithm when use the term "free node" we mean and node which is either unconnected to any sub-path or is at the endpoint of a sub-path. No internal sub-path points are eligible for connection in our algorithm because then we would create a path that does not meet the restrictions of the TSP. For clarity several definitions need to be established.

Definition 1 - A **sub-path** is defined as path connecting two or more points of the network.

Definition 2 - The **distance between a point and a sub-path** is defined as the minimum squared distance between that point and either endpoint of the sub-path.

Definition 3 - The **distance between any two sub-paths** is defined as the minimum distance between any two endpoints of the sub-paths

Data Structure and Space Requirements

The input for this algorithm set of N points in a 2D plane. The algorithm produces as an out put a path that touches all N point once for a a path that covers the minimum amount of distance.

Data structures

Distance matrix – 2D integer matrix containing all the squared distances between points. N^2 amount of space is required.

Nearest Points list – a sorted Dictionary of N key value pairs,. The key being each free node - value being Euclidean distances

SubPaths – list of string indicating sub-paths that the algorithm is creating ordered by distance from shortest to longest. The algorithm terminates when this list has one element and that element is a path of length N

Free points list – a list of all points that are either not connected to any path or are the endpoints of valid sub-paths. This is used to determine which p points need to still be tracked in the nearest points list and which points can no longer be considered eligible neighbors because they are internal to some other sub-path.

New Deterministic Divide-and-Conquer TSP Algorithm

A list of the algorithms step with big Oh- estimates of each step.

1) Create a 2D matrix to record the Euclidean distance between all points $O(N^2)$

2) Find each point's closest neighboring point and add to NP list – $O(N^2)$

3) Sort the NearestPoints list in ascending order - O(NlogN)

4) Until the sub-paths list only contains one path of length N - O(N-1)

a) Remove a point the top of NearestPoints list O(1)

i) If it does not match the endpoints of any current sub-paths, add it to the sub-paths list as a new 2-point sub-path

ii) If it matches endpoints with one other sub-path then merge that endpoint with the current sub-path according to the mergePoint algorithms. O(p)

iii) If it matches endpoints with 2 distinct sub-paths then perform mergePaths algorithm. O(p)

b) Add newly created path from step 4a to subPaths list. - O(1)

c) Remove any sub-paths that were contained in the new sub-path from the subPaths list -O(1)

d) Check for any NearestPoints that close the loop on the newly created sub-path and remove them.- O(p)

e) Check for any NearestPoints that have an endpoint in the internal points of the new path and remove them -O(q)

f) If any of the remaining sub-path endpoints have had their nearestPoint removed from NearestPoints list, find their nearest neighbor from among the remaining sub-paths endpoints and add it to NearestPoints list.- O(p)

g)Re-sort Nearest Points list in ascending order from shortest distance to greatest. - O(qlogq)

The key to the efficiency of this algorithm is the mergePoint and mergePaths functions which respectively merge either a point or a smaller sub-path with the sub-path in question

<u>MergePoint subalgorithm</u>

1) Check if any point on the sub-path is closer to the point than at the endpoints

a) If it is determine which of the two adjacent points to the new closest point is closer to the point.

b) If the endpoint distance plus the distance of the two internal points is greater than the distance between the point and the internal points.

i) then merge the point internally by connecting it to the two closest points.

ii) else connect the point to the sub-path at the closest endpoint.

2) If it is not then connect the two paths at the closest endpoints.

MergePaths subalgorithm

1) Check if any point on the longer sub-path is closer to the shorter sub-path than at the endpoints

    a) If it is determine which of the two adjacent points to the new closest point is closer to the other endpoint of the sub-paths.

    b) If the endpoint distance plus the distance of the two internal points is greater than the distance between the shorter sub-path endpoints and the internal points.

    then merge the sub-path within the larger sub-paths

    else merge the two sub-paths at their endpoint

2) If it is not, simply merge the two paths at the endpoint.

Big-Oh Time Complexity Analysis

The complexity estimates provided were based on examining the necessary steps most of the computation involved can be implemented with unsigned integers providing faster performance in almost any RISC or x86 processor compared to a similar algorithm using floating-point values. This fact was why we chose distance-squared instead of straight Euclidean distance as our cost measure so that we could avoid the costly square root function and necessary floating point arithmetic . For problems with very large values of N there is the potential to further improve the algorithm by using multiprocessors to each handle the task of merging new points to an existing sub-path until it to is finally merged into a greater path.

In the following analysis p is the length of the current subpath the algorithm is processing. And q are the remaining free points which are still unattached to any subpath. By definition both of these points must be less than N. Also as p tends to increase increase then q will be decreasing meaning that their combined computational load will tend to a steady-state. For simplicity we

have simply assumed that since we can guarantee N will always form an upper-bound on these values then we can always treat O(p) and O(q) as O(N).

$$O(f(N,p,q)) = O(N^2) + O(N^2) + O(N\log N) + O(N-1)*[O(N) * O(p) + O(1) + O(1) + O(p) + O(q) + O(p) + O(q\log q)]$$

Reduces to :

$$O(f(N,p,q)) = 2O(N^2) + O(N\log N) + O(N^2)*O(p)$$

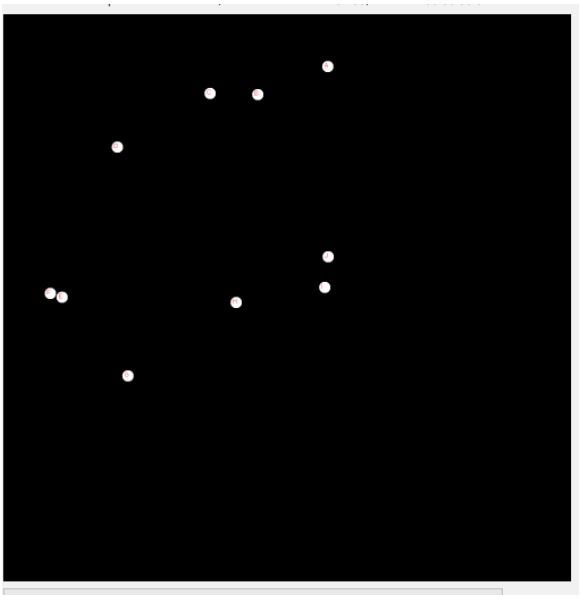
since always $p < N$ but it is approaching N for the entire algorithm.

$$O(f(N,p,q)) = \mathbf{O(N^3)}$$

While $O(N^3)$ is not insignificant it is much better than the (N-1)! search space of TSP would offer for any greedy brute force approach.

Step-by-step Walk-through with a 10-point Data Set

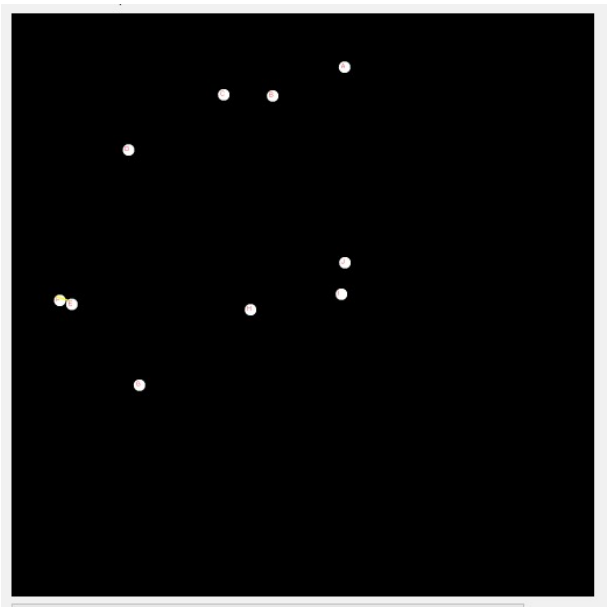
We have spent a lot of time outlining the algorithm steps, but a worked through example might provide a much clearer picture of how it works.



```
Nearest Points
EF     1061
IJ     6642
BC     15885
AB     39701
HI     56356
GE     73540
JH     73690
CD     80189
FG     89549
DB     156962
```

Figure 6: Initial State

Step 1 - Create Subpath EF(note:points are so close path is not visible in image)



Subpaths:

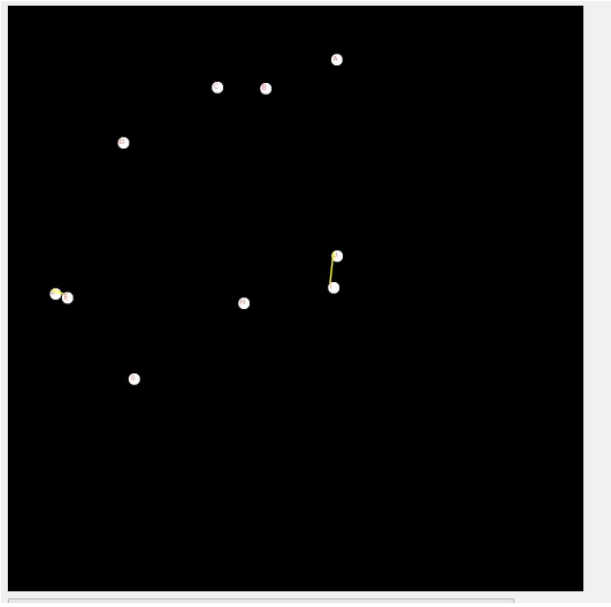
EF	1061

Nearest Points:

IJ	6642
BC	15885
AB	39701
HI	56356
EG	73540
GE	73540
JH	73690
CD	80189
FG	89549
DB	156962

Figure 7: Step 1

Step 2 - Create sub-path IJ



Subpaths:

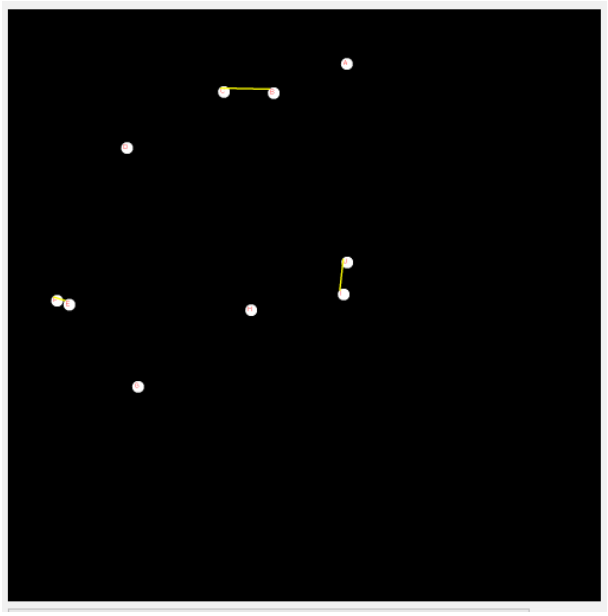
EF	1061
IJ	6642

Nearest Points:

BC	15885
AB	39701
IH	56356
HI	56356
EG	73540
GE	73540
JH	73690
CD	80189
FG	89549
DB	156962

Figure 8: Step 2

Step 3 – Create sub-path BC

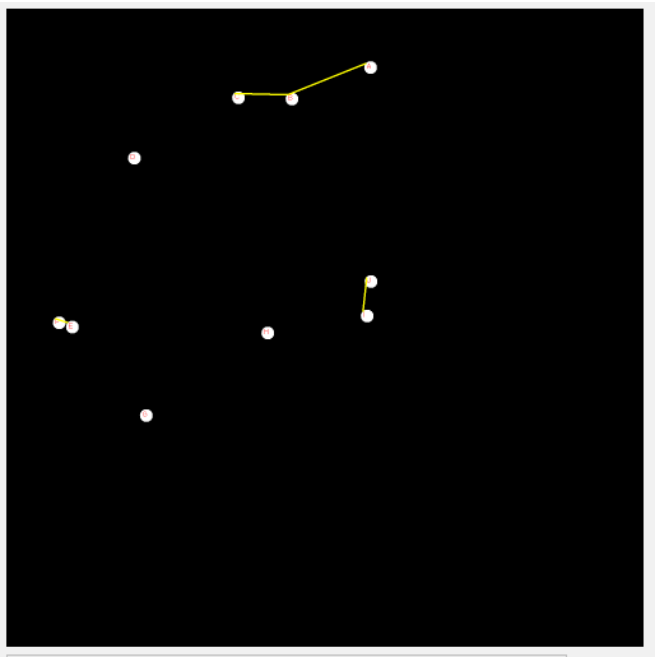


```
Subpaths:
EF      1061
IJ      6642
BC      15885

Nearest Points:
BA      39701
AB      39701
IH      56356
HI      56356
EG      73540
GE      73540
JH      73690
CD      80189
FG      89549
DB      156962
```

Figure 9: Step 3

Step 4 – Merge point A to subpath BC

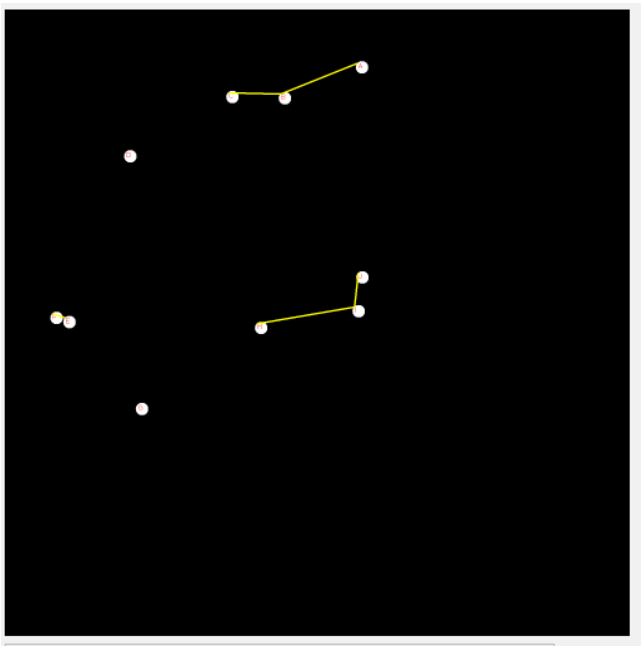


```
Subpaths:
EF      1061
IJ      6642
ABC     55586

Nearest Points:
IH      56356
HI      56356
EG      73540
GE      73540
JH      73690
DC      80189
CD      80189
FG      89549
AJ      253010
```

Figure 10: Step 4

Step 5 – Merge point H with subpath IJ

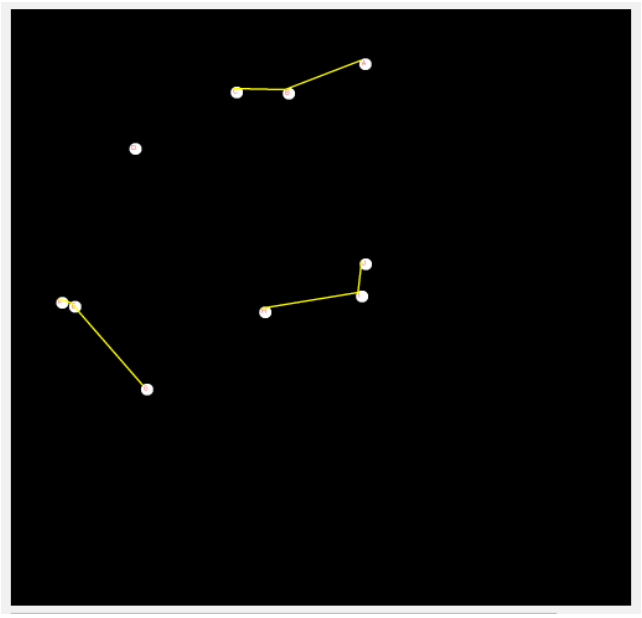


```
Subpaths:
EF      1061
HIJ     62998
ABC     55586

Nearest Points:
EG      73540
GE      73540
DC      80189
CD      80189
FG      89549
HG      119432
JA      253010
AJ      253010
```

Figure 11 : Step 5

Step 6- Merge point G with EF

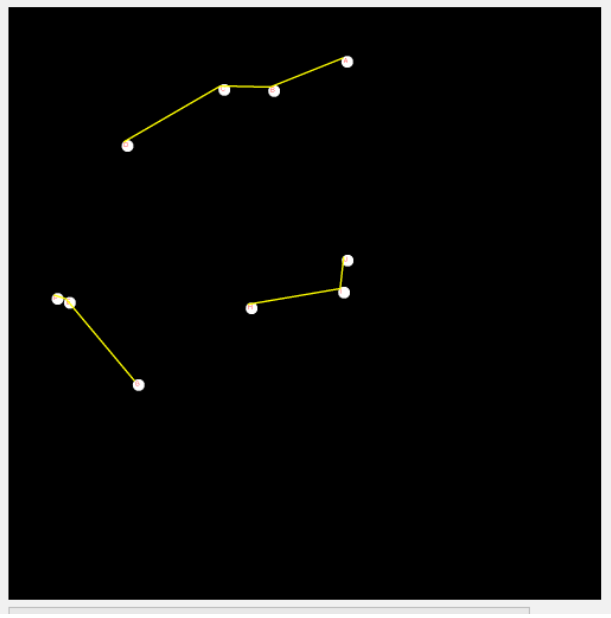


```
Subpaths:
GEF     74601
HIJ     62998
ABC     55586

Nearest Points:
DC      80189
CD      80189
GH      119432
HG      119432
FD      181098
JA      253010
AJ      253010
```

Figure 12: Step 6

Step 7- Merge point D with ABC

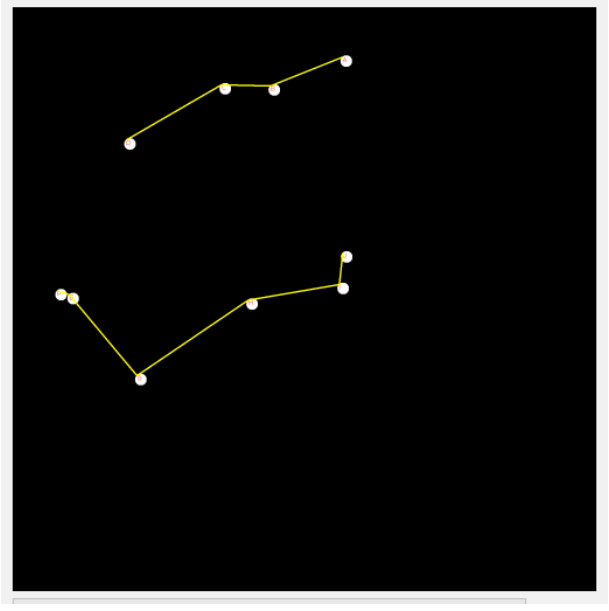


```
Subpaths:
GEF     74601
HIJ     62998
ABCD    135775

Nearest Points:
GH      119432
HG      119432
DF      181098
FD      181098
JA      253010
AJ      253010
```

Figure 13: Step 7

Step 8 – Merge GEF and HIJ

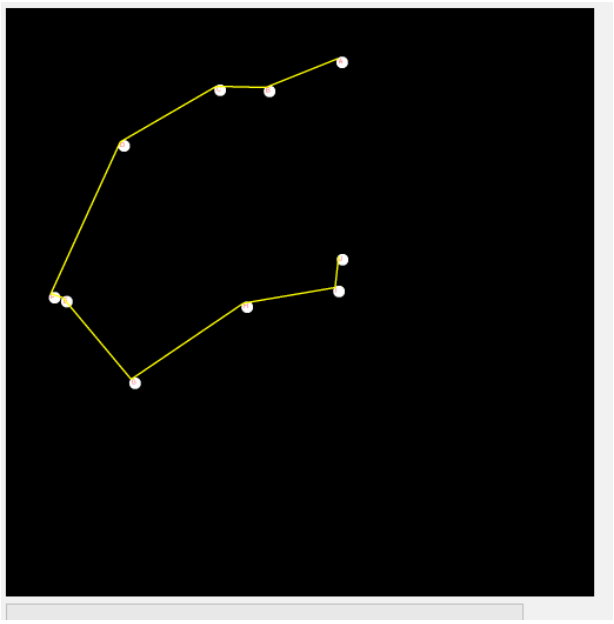


```
Subpaths:
FEGHIJ 257031
ABCD   135775

Nearest Points:
DF      181098
FD      181098
JA      253010
AJ      253010
```

Figure 14: Step 8

Step 9 – Merge FEGHIJ and ABCD



```
Subpaths:
ABCDFEGHIJ    573904

Algorithm has completed!
```

Figure 15: Step 9

mergePoint Example

Since it is an important part of how the algorithm can still produce high-quality paths without a complicated list of rules. When merging points and paths with any sub-path , if the point or path to be merged is closer to the sub-path at some point other than the endpoint we must check to see if we can create a shorter path by merging it within the path. The following two figures show a situation where it merging the point within the path produces a better path score than simply attaching the point to the nearest endpoint.

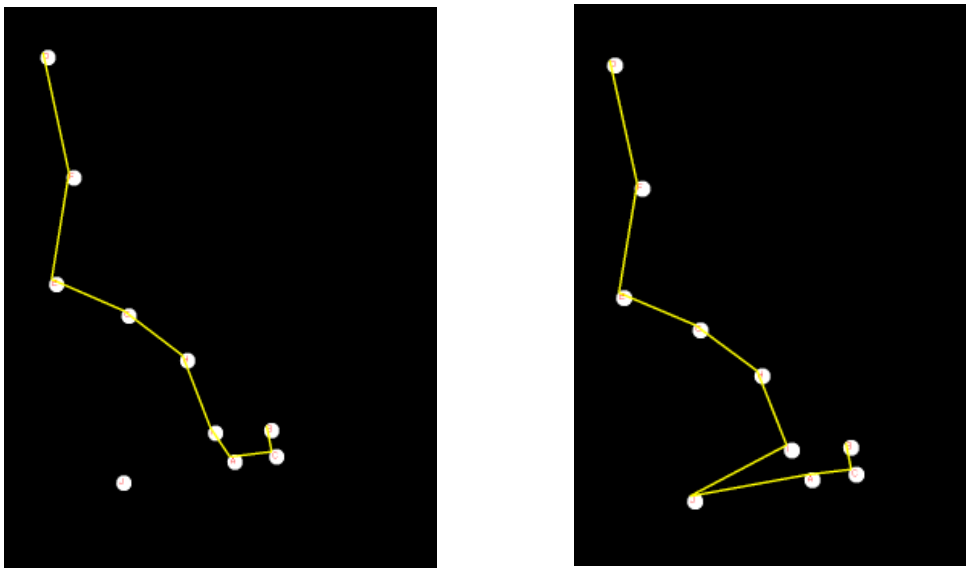


Figure 16: MergePoint example

Results

The purpose of our methodology is to test all of the algorithms versus two main criteria: Speed and efficiency. We judge speed in terms measured time to compute an answer. In the methods where it is applicable we will also examine what percentage of the total possible paths have been checked. The second metric is how long it takes before an algorithm converges to the best possible in terms of the cost. For simplicity we divided the given cost by the optimal to give us our difference factor which indicates how far from the optimal the solution is

The next three tables illustrate the comparison running times for 3 different ten point data sets. For ten point data sets the total number of possible paths equals 118,440. The three following tables indicate each.

Method	Type	Paths checked	% of total	Final path	Final distance	Time(s)	Scale factor
Brute Force	Deterministic	181440	1000	ABCDFEGHIJ	573904	3.08	1
Simulated Annealing	Probabilistic	83281	4.59	ABCDFEGHIJ	573904	0.05	1
Genetic Algorithm	Probabilistic	45.9	2.67	ABCDFEGHIJ	573904	0.03	1
Prim's Algorithm	Deterministic	NA	NA	FABCDEGHIJ	1467959	0.006	2.56
New Algorithm	Deterministic	NA	NA	ABCDFEGHIJ	573904	0.023	1

Table 2: First 10-point Data Set

Method	Type	Paths checked	% of total	Final path	Final distance	Time(s)	Scale factor
Brute Force	Deterministic	181440	100	CBEJDIFAHG	285582	2.89	1
Simulated Annealing	Probabilistic	62955	34.7	CBEJDIFAHG	285582	0.03	1
Genetic	Probabilistic	4920	2.7	CBEJDIFAHG	285582	0.01	1

Algorithm							
Prim's Algorithm	Deterministic	NA	NA	EAHGDIFJBC	1070834	0.001	3.74
New Algorithm	Deterministic	NA	NA	CBJEDIFAHG	311526	0.001	1.09

Table 3: Second 10-point Data Set

Method	Type	Paths checked	% of total	Final path	Final distance	Time(s)	Scale factor
Brute Force	Deterministic	181440	100	DFEGHIBCAJ	234363	3.4	1
Simulated Annealing	Probabilistic	82885	45.68	DFEGHIBCAJ	234363	0.05	1
Genetic Algorithm	Probabilistic	7920	4.36	DFEGHIBCAJ	234363	0.03	1
Prim's Algorithm	Deterministic	NA	NA	DAIHGEFJCB	1282971	0.001	5.4
New Algorithm	Deterministic	NA	NA	DFEGHIJACB	263663	0.001	1.12

Table 4 : Third 10-point Data Set

The next two tables are for 12-point data sets and the total paths in these sets is 19,958,400.

Method	Type	Paths checked	% of total	Final path	Final distance	Time(s)Scale factor	Scale factor
Brute Force	Deterministic	19958400	100	DEIFCBGHKAJL	804657	187	1
Simulated Annealing	Probabilistic	83166	4.16	DEIFCBGHKAJL	804657	0.06	1
Genetic Algorithm	Probabilistic	7296	0.03	DEIFCBGHKAJL	804657	0.03	1
Prim's Algorithm	Deterministic	NA	NA	LKAJHGEIFBCD	2189412	0.001	2.72
New Algorithm	Deterministic	NA	NA	DEIFBCGHKAJL	829245	0.002	1.03

Table 5: First 12-point Data Set

Method	Type	Paths checked	% of total	Final path	Final distance	Time(s)	Scale factor
Brute Force	Deterministic	19958400	100	KILAFJBGCDHE	803843	156	1
Simulated Annealing	Probabilistic	136511	0.68	KILAFJBGCDHE	803843	0.08	1
Genetic Algorithm	Probabilistic	12192	0.06	KILAFJBGCDHE	803843	0.05	1
Prim's Algorithm	Deterministic	NA	NA	GAFLIKBJCDHE	1809260	0.001	2.25
New Algorithm	Deterministic	NA	NA	KILAFJBGCDHE	803843	0.001	1

Table 6: Second 12-point Data Set

And the final set of points is for 14 points and a total of 3.113.510,400.

Method	Type	Paths checked	% of total	Final path	Final distance	Time(s)	Scale factor
Brute Force	Deterministic	3113510400	100	HGJBKNCAIEFLDM	6997131	221	1
Simulated Annealing	Probabilistic	157190	5E-05	HGJBKNCAIEFLDM	6997131	0.08	1
Genetic Algorithm	Probabilistic	10360	3.00E-06	DMLFEJHGBKINAC	845113	0.04	1.21
Prims Algorithm	Deterministic	NA	NA	MANKBJGHIEFLCD	2632599	0.001	3.76
New Algorithm	Deterministic	NA	NA	HGJCANKBIEFLDM	1068023	0.002	1.52

Table 7 : 14-point Data Set

Immediately some points become evident from looking at the data. The probabilistic algorithms find the optimal path in all of the cases listed. in my experience working with the program I can say that sometimes both the simulated annealing and the genetic algorithm failed to find the absolute optimal solutions but the both performed very well and generally the genetic algorithm

found the answer in less time and with fewer paths checked than the simulated annealing. The deterministic algorithms run much faster than the genetic algorithms but they vary much more in performance, Prim's especially. I wrote these programs all with C# and Microsoft Web Forms for ease of use but I did not do anything close to a rigorous optimization. Therefore these time figures should not be taken as absolute measures but relative performance metrics taken with the same program on the same machine.

New Algorithm versus Prim's Approximation Algorithm

When comparing the algorithms it's important to recognize that by their nature each type has its own strengths and weaknesses. Also that no algorithm will provide an absolutely optimal solution in a polynomial order of time. We are looking for what provides the best combination of fast execution and a quality solution. In many modern problems with the prevalence and low cost of multi-core processors and boards with thousands of GPU cores, the importance of leveraging the opportunities for parallel attack on a problem cannot be overlooked. As mentioned earlier in this paper, the power of probabilistic methods has been known since before the dawn of computers, Poincare and Kolmogorov speculated and Von Neumann demonstrated, there is incredible power available when you use chance to help find a solution. In our case, the deterministic algorithms are both faster than their probabilistic counterparts by at least an order of magnitude.

While Prim's algorithm has at least one element of chance that has a strong impact on the eventual solution : which node will serve as the root node for the minimum spanning tree that the algorithm will build? That node is selected by chance and generally the quality of the final solution is dependent upon this nodes selection. As a rule of the thumb the closer the root node is to one of the endpoints of the true optimum solution, than the better solution Prim's algorithm produces. To demonstrate on we'll take a data set and show what happens when you select different points as the root node.

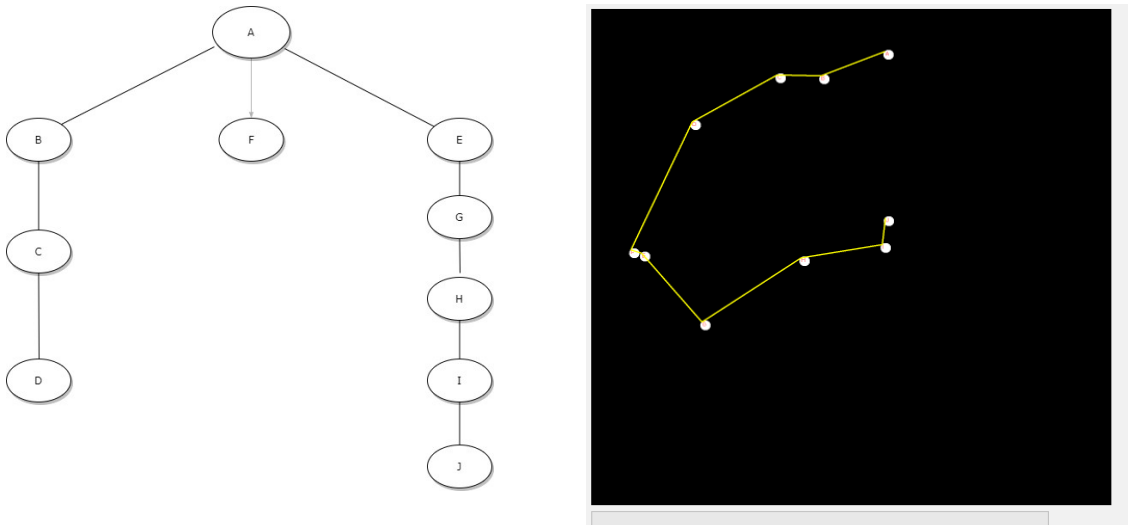


Figure 17: MST from Node 'A' and Resultant Path

Now we will attempt the same dataset but starting from node E.

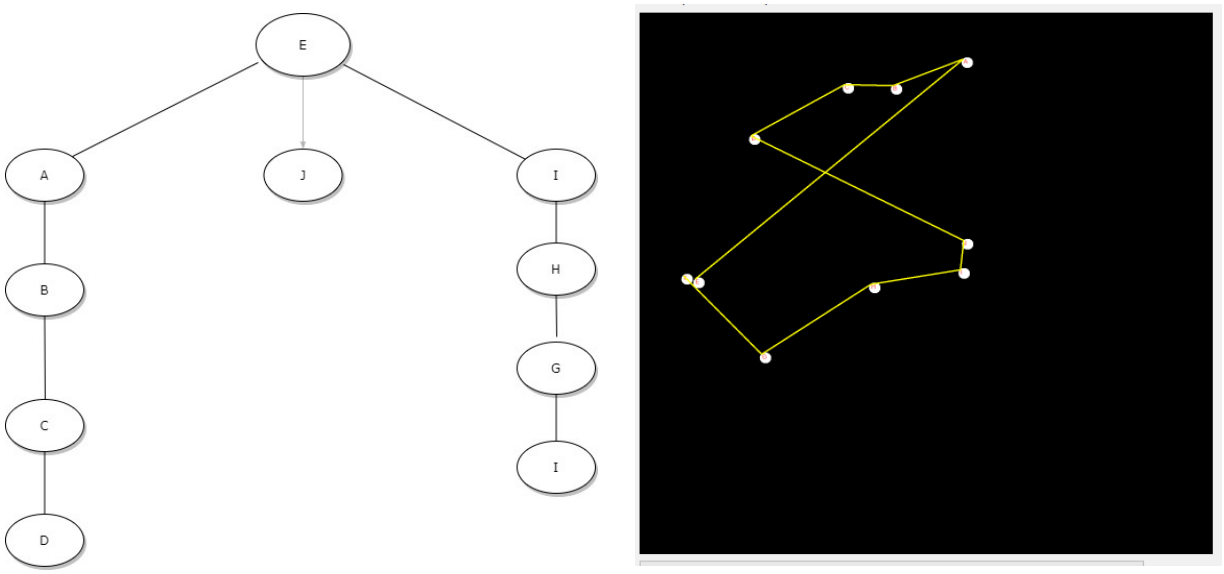


Figure 18: MST from Node 'E' and Resultant Path

The differences between the optimum path length and the path length in the second diagram is $\frac{1667007}{573904} = 2.904$. So even with the same points and the same algorithm the

starting point choice can make a large difference in results. Of course in almost all cases it is not feasible to determine which point will be and endpoint to an optimum solution a priori. Intuition might indicate that as the height of the MST approaches N you would be getting closer to the optimal solution. While that might be true in many cases, it is interesting to note that the height of these two trees is 6 and 5 respectively. Yet one represents and optimal path and one represents one of the worst possible solutions that the algorithm could produce. It should also be considered that there are some applications where the starting point has been decided in advance. in those cases Prim's design bug has become a beneficial feature.

New Algorithm versus Probabilistic Algorithms

One good parameter for evaluating how successfully a new algorithm converges on a solution is to watch the function of cost over algorithm steps. Since with simulated annealing we only have one path being constantly permuted while genetic algorithms have a population of paths per generation. The simulated annealing best cost(purple) while the genetic algorithms have both the best cost (blue) and average cost(green) plotted. We will show one graph versus a 10-point, 12-point and 14-point data set.

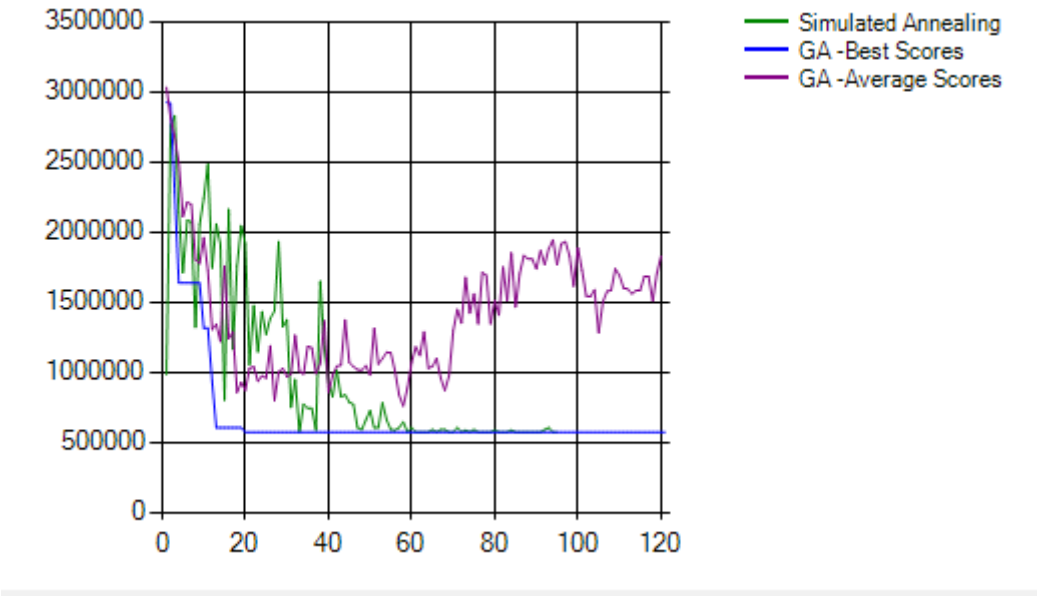


Figure 19: Genetic Algorithm vs. Simulated Annealing- 10 points

In figure 18 the average score for the genetic algorithm average score of the population noticeably get worse towards the end. That is the result of the fact that our algorithm increased the mutation rate as the system got closer to convergence. This was an attempt to insure that the genetic algorithm had truly explored the solution space and had not just been trapped in the nearest local minimum. the larger the value of n, then the solution space increases by $n!$. Therefore the more likely that the algorithm might get caught before it finds the global minimum.

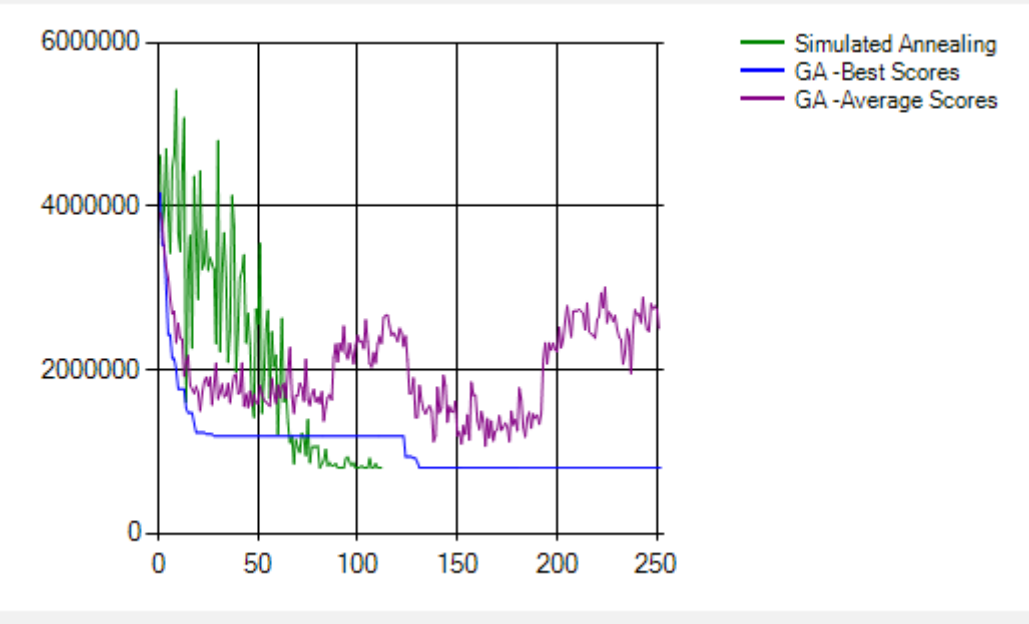


Figure 20: Genetic Algorithm vs. Simulated Annealing- 12 points

In Figure 19 notice at about generation 120 the best score drops. This is the reason why the mutation increased. In this case the graph shows that after it had settled on one solutions it eventually still found a better choice that improved the algorithm. This is an important engineering trade-off consideration. O On the one hand you can see that for a large part of the algorithm tail no better solution is found. It could be argued that that extra time is wasted and a smaller convergence criteria would make for a faster algorithm. But sometimes that extra searching does a provide a better a solution. Unfortunately there is no strict mathematical relation that guarantees if you spend x amount of time checking convergence that you will get a

solution in the top 10 or top 5 percentile of possible solutions. It is still very much and art of trial and error.

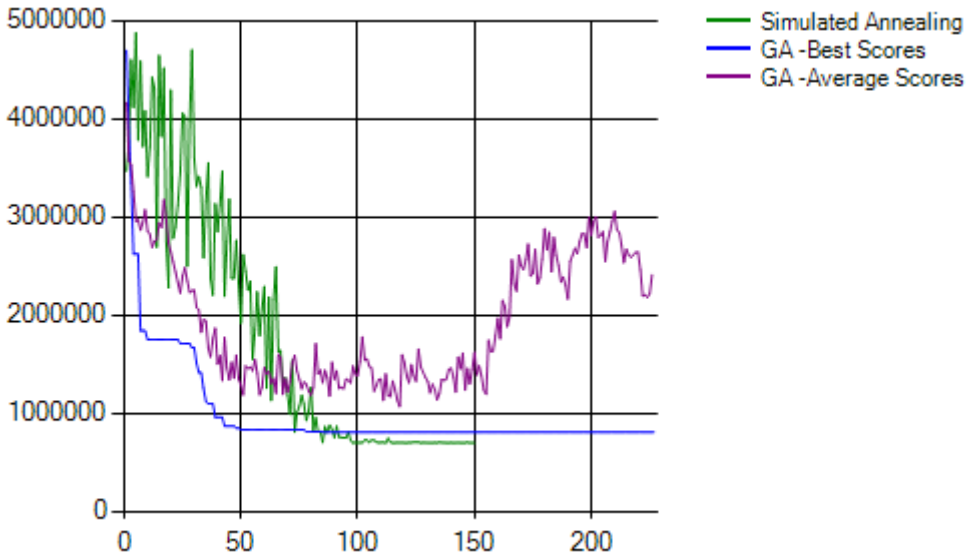


Figure 21: Genetic Algorithm vs. Simulated Annealing- 14 points

The final figure shows it for 14 points. We can note that the genetic algorithm never approaches the minimum value that the simulate annealing algorithm achieved. By the 50th generation it had already found the best solution it was going to find. The simulated annealing method took longer to converge on its final solution but it found a better one in the end. The simulated annealing method also had a shorter convergence criteria so as in this example it finished on the 150th step whereas the genetic algorithm took a little longer. Does this mean that the simulated annealing is superior to genetic algorithms in efficiency and speed. This set of tests is not nearly thorough enough to come to a definitive conclusion on that point, but more research definitely seems warranted.

Future Research Directions

Looking at the complete results the new algorithm comes out with an impressive performance for our limited test set. It was faster by several orders of magnitude than any of the probabilistic

algorithms. Prim's algorithm, the other deterministic algorithm we tested did not often give nearly as optimal a solution. The combination of speed and a near-optimal solution makes this new algorithm an enticing option for handling the UAV navigation in our aviation project. But like anything there is always room for improvement and we will review some of those options here.

The first main improvement is a matter of simplification. When the steps in our algorithm as it stands are compared to the pseudocode presented for Prim's algorithm it is clear that Prim's algorithm is a cleaner and more direct programming structure. In our first attempt to implement and test everything lots of elements were added that were originally intended to make the system more efficient but a review of the code and the actual flow diagrams will almost certainly reveal some dead code areas and branches where coding decisions can be made more efficiently. This program definitely stands to be improved by a careful analysis of its mergePoint and mergePath procedures. There will also eventually be interest in seeing if the algorithm can be improved to handle the case of three dimensional flying space and incorporate the constraint against paths where the UAV is forced to expend extra energy in the ascent.

A second thought that was considered was if the divide-and-conquer approach demonstrated by this algorithm would make it amenable to a parallelization attempt. Initially there was some thinking that each new sub-path thread could be assigned to a processor which would be in charge of handling the merging of new points. But currently each thread would also have to atomically edit the NearestPoints list. The need to lock the NearestPoints every time a thread merges a new point makes any performance gains from multi-threading this application doubtful.

TSP is a classic NP-hard problem and because of its obvious utility lots of attempted solutions have been attempted[4]. There are a lot of challenges to attempting to truly compare different algorithms efficiency on a problem that by definition has a huge solution space. but the promising initial results of this algorithm indicate that it would be worthwhile setting up tests that would give a better idea of where this algorithm stands in the range of possible solutions to TSP.

Conclusion

This paper introduced a new algorithm for approximating a solution to the Traveling Salesman . Our interest in finding a cost-effective solution to this problem is part of a larger project to build a long-duration UAV for monitoring forest conditions including fires and disease epidemics. The forest environment, especially during a fire, is a dangerous and constantly changing environment. The ability to provide an efficient an powerful avionics program which can handle the space and make quick and correct navigation decisions is critical to the mission's success. We compared two probabilistic algorithm, simulated annealing and genetic algorithms, with two deterministic algorithms, the new algorithm and Prim's approximate. The deterministic were much faster than the probabilistic algorithms although the probabilistic algorithms showed a remarkably efficient job in covering the solution space. in cases where more varied performance criteria are need the probabilistic algorithms offer an intriguing blend of speed and adaptability. For the more straightforward problem of our UAV navigation speed is key. both Prim's and the new algorithm are orders of magnitude faster than the probabilistic approaches. Between Prim's and the new algorithm Prim's performs much better when its randomly chosen root node is near and optimum path's endpoint. Unfortunately there is no a priori way to know where that endpoint will be and the odds of randomly guessing correctly goes down as N increases. In the general case the new algorithm offers a powerful blend of speed and providing a near-optimal solution. It would be interesting to see how it performs against some of the other premier algorithms in a rigorous test benchmark. The search for better algorithms to tackle NP-hard problems continues, but these solutions offer some powerful options for any engineer who needs to solve the Traveling Salesman Problem.

Appendix A: New Algorithm Code

```
public void TS_NEW()
{
    nearestPoints = new Dictionary<string, double>();
    subPaths = new Dictionary<string,double>();
    double minPathCost;
    char [] minPathChar = new char[2];
    string minPath;
    Stopwatch stopWatch = new Stopwatch();
    char temp;
    char[] revPathChar = new char[2];
    string revPath;
    string bestPath = "";

    if (checkBox1.Checked)
    {
        stopWatch.Start();
    }
    Console.WriteLine("Starting YFANTIS Algorithm");
    freePoints = "";
    subPaths.Clear();
    nearestPoints.Clear();
    //1)Calculate Distance MAtrix
    maxCost = CalculateDistanceMatrix();   //set minPathCost to the maximum cost
                                               in the matrix to initialize
    //2) Create String of FreePoints
    //3) create an orderedList of the shortest path between rows
    for (int i=0; i<n; i++)
    {
        freePoints = String.Concat(freePoints, (char)(65 + i));
        minPathChar[0] = (char)(65 + i);
        minPathCost = maxCost;
        for(int j =0; j<n; j++)
        {

            temp = (char)(65 + j);
            revPathChar[0] = temp;
            revPathChar[1] = minPathChar[0];
            revPath = new string(revPathChar);
            //Console.WriteLine(String.Format("Checking if nearestPoints contains
             revPath:({0})", revPath));
            //check if reverse of path is already in list
            if (!nearestPoints.ContainsKey(revPath))
            {
                if (i!= j && distanceMatrix[i,j]< minPathCost)
                {
                    minPathCost = distanceMatrix[i, j];
```

```
                    minPathChar[1] = temp;
                }
            }
        }


        minPath = new string(minPathChar);
        Console.WriteLine(String.Format("Adding path:{0}, distance:{1}", minPath,
             minPathCost));
        nearestPoints.Add(minPath, minPathCost);
    }
    Console.WriteLine("Free Points: " + freePoints);
    //PrintDistanceMatrix();
    //Sort nearestPoints in ascending order
    var sortedPaths = from entry in nearestPoints orderby entry.Value ascending
      select entry;
    nearestPoints = sortedPaths.ToDictionary(x=>x.Key, x=>x.Value);
    Console.WriteLine("Nearest Points:");
    PrintPaths(nearestPoints);

    if (checkBox1.Checked)
    {
        int steps = 0;
        do
        {
            steps++;
            TSP_NEW_STEP(ref bestPath);
            Console.WriteLine("Step #" + steps.ToString());
            Console.WriteLine("Printing subPaths");
            PrintPaths(subPaths);
        } while (!(subPaths.Count == 1 && subPaths.First().Key.Length == n));

        //translate best path to paintPath
        StringToIntArray(bestPath, ref paintPath);
        bestScore = FindPathDistance(bestPath);
        //print out times
        stopWatch.Stop();
        string results1 = String.Format("New Batch:Best path ={0}, Best distance
             ={1}", bestPath, bestScore);
        Console.WriteLine(results1);
        StringToIntArray(bestPath, ref paintPath);
        TimeSpan ts = stopWatch.Elapsed;
        string elapsedTime = String.Format("{0:00}:{1:00}:{2:00}.{3:00000}",
        ts.Hours, ts.Minutes, ts.Seconds,
        ts.Milliseconds / 10);
        string results2 = "RunTime " + elapsedTime;
        Console.WriteLine(results2);
        label2.Text = results1 + ", " + results2;
    }
}
```

```
public void TSP_NEW_STEP( ref string bestPath)
{
    string internalPoints = "";
    List<string> removedPaths = new List<string>();

    //Selects the next subpath to add into subPaths
    int matches = 0;
    int cIndex;
    string nearestPoint;
    string replacePath = "";
    string newPath;
    double minPathCost;
    double temp;
    char otherPoint;
    List<string> matchedStrings = new List<string>();

    if (nearestPoints.Count == 0)
    {
        MessageBox.Show(String.Format("Error: nearestPoints Dictionary is
             empty!"));
        return;
    }
    Console.WriteLine("***********Starting Yfantis Step *****************");
    nearestPoint = nearestPoints.First().Key;
   //Console.WriteLine("Free Points: " + freePoints);
    foreach (string path in subPaths.Keys)
    {
        //Checks how many times the next nearestPoint entry connects to one of
             the subpaths
        if (MatchString(path, nearestPoint))
        {
            matches++;
            matchedStrings.Add(path);
            //Console.WriteLine("Adding " + path + " to matchedStrings");
        }
        if (matches >= 2) break;

    }
    //if the first entry has no endpoint matches with current subpaths ---makes a
      new 2-point subpath
    if (matches == 0)
    {
        // add 2-point subpath  entry to subpaths and delete entry from
             nearestPoints dictionary
        subPaths.Add(nearestPoint, nearestPoints.First().Value);
        //Console.WriteLine("match =0:Adding " + nearestPoint + " distance : " +
             nearestPoints.First().Value);
        nearestPoints.Remove(nearestPoint);
        newPath = "";
    }
```

```
else if (matches == 1)
{

    //else if the top entry does match the endpoints with only one subpath
    ///  then merge point into new subpath  and delete old subpath
    subPaths.Remove(matchedStrings[0]);
    //Console.WriteLine("Merging " + matchedStrings[0] + " and " +
        nearestPoint);
    newPath = MergePoint(matchedStrings[0], nearestPoint);
    subPaths.Add(newPath, FindPathDistance(newPath));
    //Console.WriteLine("MergePoint:match =1 adding " + newPath + "
        distance : " + FindPathDistance(newPath));
    nearestPoints.Remove(nearestPoint);


}
else
///  else it matches endpoints with two distinct subpaths(merging a subpath
        with a subpath)
{
    subPaths.Remove(matchedStrings[0]);
    subPaths.Remove(matchedStrings[1]);
    newPath = MergePaths(matchedStrings[0], matchedStrings[1], nearestPoint);
    //Console.WriteLine("MergePaths:match =2 adding " + newPath + "
        distance : " + FindPathDistance(newPath));
    subPaths.Add(newPath, FindPathDistance(newPath));
    nearestPoints.Remove(nearestPoint);
}

if (newPath.Length > 2)
{
    //Find internal points of newPath
    internalPoints = newPath.Substring(1, newPath.Length - 2);
    foreach (char c in internalPoints)
    {
        //Remove internal points of newPath from FreePoints string
        if (freePoints.Contains(c))
        {
            cIndex = freePoints.IndexOf(c);
            freePoints =freePoints.Remove(cIndex, 1);
        }

        //check nearestPoints for any paths containing internal point of new
         path
        foreach (var key in nearestPoints.Keys)
        {
            if (key.Contains(c))
            {
        //Mark for removal any nearest point with internalPoints as
         endPoints
```

```
                removedPaths.Add(key);
                //Console.WriteLine("Removing " + key + " From nearestPoints
                      because it contains an internal point.");
                //Console.WriteLine("Removed Paths Count = " +
                      removedPaths.Count);
            }
        }
    }
    // remove from nearestPoints anything marked for removal
    foreach(var path in removedPaths)
    {
        //Console.WriteLine("Removing " + path + " from nearestPoints.");
        nearestPoints.Remove(path);
    }
    removedPaths.Clear();
    //Mark for removal any nearestPoint that closes the loop of newPath
    foreach (string key in nearestPoints.Keys)
    {
        if (newPath.Contains(key.First()) && newPath.Contains(key.Last()))
        {
            removedPaths.Add(key);
            //Console.WriteLine(key + " closes " + newPath + " so it was
                removed.");
            //Console.WriteLine("Removed Paths Count = " +
                removedPaths.Count);
        }
    }
    // remove from nearestPoints anything marked for removal
    foreach(var path in removedPaths)
    {
        Console.WriteLine("Removing " + path + " from nearestPoints.");
        nearestPoints.Remove(path);
    }
}
//Console.WriteLine("New Freepoints : " + freePoints);
Console.WriteLine("Printing subPaths:");
PrintPaths(subPaths);

//TODO:  add new nearest point for any freePoints
if (!(subPaths.Count == 1 && subPaths.First().Key.Length == n))
{
    foreach (char c in freePoints)
    {
        Console.WriteLine("Checking if we need to find a replace path
         starting with " + c);
        if (IsNotInNearestPoints(c))
        {
            otherPoint = '1';
            //find c's other subpath endpoint
            foreach (string key in subPaths.Keys)
```

```
                {
                    if (key.Contains(c))
                    {
                        if (c.Equals(key.First()))
                        {
                            otherPoint = key.Last();
                        }
                        else
                        {
                            otherPoint = key.First();
                        }
                    }
                }
                //find new nearest point starts with c and ends any freepoint
                   other than its other endpoint
                int i = (int)(c) - 65;
                minPathCost = maxCost;
                foreach (char fp in freePoints)
                {
                    if (!fp.Equals(otherPoint) && !fp.Equals(c))
                    {
                        int j = (int)(fp) - 65;
                        temp = distanceMatrix[i, j];
                        if (minPathCost >= temp)
                        {
                            minPathCost = temp;
                            replacePath = String.Concat(c, fp);
                        }

                    }
                }
                nearestPoints.Add(replacePath, minPathCost);
                Console.WriteLine("Adding " + replacePath + "(" + minPathCost +
                   ") to NearestPoints");
            }
        }
        //Re-sort  NearestPoints list
        var sortedPaths = from entry in nearestPoints orderby entry.Value
             ascending select entry;
        nearestPoints = sortedPaths.ToDictionary(x => x.Key, x => x.Value);
        Console.WriteLine("Nearest Points list:");
        PrintPaths(nearestPoints);
    }
    else
    {
        bestPath = subPaths.First().Key;
        Console.WriteLine("Success: algorithm has found a solution : " + bestPath
             + ", distance=" + subPaths.First().Value.ToString());
    }
}
```

Appendix B: Genetic Algorithm Code

```
public void TSP_GA()
 {
     bool hasConverged = false;
     int pathsChecked;     //tally of total number of paths checked
     int threshold;          // number of generation that must not improve best
                                  score for convergence
     int dniCount;           //tacks the number of generations where the score did
                                  not improve
     int populationCount;
     int eliteCount;
     int temp;
     int start, end;
     int totalPaths;
     double totalScores;
     double eliteTotal;
     double minScore;
     double avgScore;
     List<string> population = new List<String>();
     string[] children;
     string bestPath;
     double[] nextGenScores;         //score of each child in the population
     double[] roulette;              //The Distribution of top boundaries of
                                  probability values of a certain population
     double mutationRate = 0.1;
     String parent1 = "";
     String child1 = "";
     String child2 = "";
     List<double> bestScores = new List<double>();
     List<double> avgScores = new List<double>();
     Stopwatch stopWatch = new Stopwatch();

     //****************Beign GA algorithm
     Console.WriteLine("Starting GA");
     stopWatch.Start();
     //initialize population
     populationCount = 4*n;
     eliteCount = populationCount / 4;

     threshold = 10*n;
     children = new string[populationCount];
     nextGenScores = new double[populationCount];
     roulette = new double[eliteCount];
     dniCount = 0;


     CalculateDistanceMatrix();
     //create initial sample population
     for (int i =0; i<populationCount; i++)
```

```
{
    parent1 = GenerateRandomPath(n, rand);
    while (population.Contains(parent1))
    {
        parent1 = GenerateRandomPath(populationCount, rand);
    }
    population.Add(parent1);
}
  // set minScore to the first path score to initialize
minScore = FindPathDistance(population[0]);
bestPath = population[0];
//Console.WriteLine(population[0] + ":Initial score set to " +
  minScore.ToString());
bestScores.Add(minScore);


while (!hasConverged)
{

    totalScores = 0;
    //Step 1 crossover and mutate all members of a population
    for (int i = 0; i < populationCount; i += 2) {
        minScore = bestScores.Last();
        start = rand.Next(n);
        temp = rand.Next(n);
        while (temp == start)
        {
            temp = rand.Next(n);
        }
        end = temp;
        Utils.OrderedCrossover(population[i], population[i + 1], start, end,
                                            ref child1, ref child2);
        //run SwapMutation
        children[i] = Utils.SwapMutation(child1, mutationRate, rand);
        children[i+1] = Utils.SwapMutation(child2, mutationRate, rand);
        //step2 - check the performance of new members and find their score
        nextGenScores[i] = FindPathDistance(children[i]);
        nextGenScores[i + 1] = FindPathDistance(children[i + 1]);
        //   record best, average score, and sum of scores
        totalScores += nextGenScores[i] + nextGenScores[i + 1];
        //find min
        if (nextGenScores[i] < minScore)
        {
            minScore = nextGenScores[i];
            bestPath = children[i];
        }
        if (nextGenScores[i + 1] < minScore)
        {
            minScore = nextGenScores[i + 1];
            bestPath = children[i + 1];
```

```
    }
  }
  Array.Sort(nextGenScores,children);
  /*Console.WriteLine("Sorted children by scores:");
  for (int i=0; i< children.Length; i++)
  {
      Console.WriteLine(String.Format("rank:{0}, path:{1}, score:{2}", i +
       1, children[i], nextGenScores[i]));
  }*/
  //Add average Score
  avgScore = totalScores / populationCount;
  avgScores.Add(avgScore);
  bestScore = bestScores.Last();
  if(bestScore > minScore)
  {
      dniCount=0;
      bestScores.Add(minScore);
  }else
  {
      dniCount++;
      bestScores.Add(bestScore);
  }


  Console.WriteLine(String.Format("Gen : {6}\nTotal Score = {0}, Avg Score
= {1}, bestScore(minScore) = {2}({3}), bestPath = {4} ,did not improve =
{5}", totalScores, avgScore, bestScores.Last(),minScore, bestPath,
dniCount,bestScores.Count));
  eliteTotal = 0;
  for(int i=0; i <eliteCount; i++)
  {
      eliteTotal += nextGenScores[i];
  }
  //Build roulette table
  //Console.WriteLine("Roulette Table");
  for(int i =0; i<eliteCount; i++)
  {
      //Console.WriteLine(String.Format("i({0} == last gen :{1}, next gen:
       {2}, Score:{3}",i,population[i],children[i],nextGenScores[i] ));
      if (i == 0) { roulette[i] = nextGenScores[i] / eliteTotal; }
      else { roulette[i] = roulette[i-1] + nextGenScores[i] /eliteTotal; }
      //Console.WriteLine(String.Format("i:{0}, score:{1},
       roulette[{0}]={2}", i, nextGenScores[i] / totalScores,
       roulette[i]));
  }
  // if best score has not improved in 'threshold' # of generations
       hasConverged = true
  if(dniCount > threshold)
  {
      hasConverged = true;
```

```
                    Console.WriteLine(String.Format("Score Converged!!\nBest path :{0},
                     best Score {1}, generations:{2}", bestPath,bestScores.Last(),
                     bestScores.Count));
                } else
                {
                    // else  step 3 select new population based on roulette
                    for(int i =0; i< populationCount; i++)
                    {
                        population[i] = SelectPath(children, roulette, rand);
                    }
                }
                if (dniCount < threshold / 2) { mutationRate = .1; }
                else { mutationRate = 0.2; }
            }

            //************End GA Algorithm

            pathsChecked = bestScores.Count * populationCount;
            totalPaths = Utils.Factorial(n-1)/2;
            Console.WriteLine(String.Format("Final:  best Path ={0}, best score = {1}",
bestPath, bestScore));
            PlotCostData(bestScores, chart1, "GA -Best Scores");
            PlotCostData(avgScores, chart1, "GA -Average Scores");
            label3.Text = String.Format("GA Cost vs. temp step- best path = {0}, best
score = {1}\nPaths checked ({2}/{3})={4:F3}%", bestPath, bestScore, pathsChecked,
totalPaths, (double)pathsChecked * 100 / totalPaths);

            //print out times
            stopWatch.Stop();
            string results1 = String.Format("GeneticAlgorithm :Best path ={0}, Best
distance ={1:F02}", bestPath, bestScore);
            Console.WriteLine(results1);
            StringToIntArray(bestPath, ref paintPath);
            TimeSpan ts = stopWatch.Elapsed;
            string elapsedTime = String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
            ts.Hours, ts.Minutes, ts.Seconds,
            ts.Milliseconds / 10);
            string results2 = "RunTime " + elapsedTime;
            Console.WriteLine(results2);
            label2.Text = results1 + ", " + results2;
        }
```

Appendix C: Simulated Annealing Code

```
public void TSP_SA()
{
    bool hasConverged = false;
    int pathsChecked;     //tally of total number of paths checked in this
    int counter;
    int improveCount;
    int epochCounter = 0;
    double startT;       // SA start temperature
    double T;
    double Tfactor = 0.95;
    double deltaT;
    int epochN;
    int threshold;
    double costDifference = 0;
    double coin;
    string newPath;
    double tempParameter;
    string bestPath;

    Stopwatch stopWatch = new Stopwatch();
    List<double> scores = new List<double>();
    double newScore;
    //build basePath String
    StringBuilder basePath = new StringBuilder("", n);
    char c;

    Console.WriteLine("Start SA algorithm");
    stopWatch.Start();
    CalculateDistanceMatrix();
    Console.WriteLine("Calculated Distance MAtrix");
    //PrintDistanceMAtrix();
    for (int i = 0; i < n; i++)
    {
        c = (char)(65 + i);
        basePath.Append(c);
    }
    //initialize path temperature, and theshld values
    bestPath = basePath.ToString();
    int start;
    int end;
    int temp;
    startT = FindPathDistance(bestPath);
    bestScore = startT;
    T = startT;
    deltaT = startT * Tfactor;
    epochN = 100 * n;
```

```
threshold = 10 * n;
improveCount = 0;
counter = 0;
pathsChecked = 0;

///////////////  Begin Simualted Annealing algorithm

//While the system has not converged
while (!hasConverged)
{
    //Select two different random points in the sequence
    start = rand.Next(n);
    temp = rand.Next(n);
    while (temp == start)
    {
        temp = rand.Next(n);
    }
    end = temp;
    //Based on Rand(0-1) choose to reverse or Transport the string
    coin = rand.NextDouble();
    if ( coin > 0.5)
    {
        newPath = Utils.Reverse(bestPath, start, end);
    } else
    {
        newPath = Utils.Transport(bestPath, start, end, rand);
    }
    //Calculate the cost difference between the previous bestScore and the
         new string
    newScore = FindPathDistance(newPath);
    costDifference = newScore - bestScore;
    tempParameter = Math.Exp(-(costDifference / T));
    //if the cost diffference is negative keep the new string, increment
         improved, and go to the next step
    coin = rand.NextDouble();
    if ( costDifference < 0)
    {
        improveCount++;
        bestPath = newPath;
        bestScore = newScore;
    } else if ( tempParameter> coin)
    //if the cost difference is positive if e^- (costDifference)/T) >  random
         number from 0-1
    {
        bestPath = newPath;
        bestScore = newScore;
    }
    //else discard the new string and go to the next step
    //increment counter and n
    counter++;
```

```
                pathsChecked++;
                //end of loop check condition
                //if counter > epochN && improved ==0
                if (counter >epochN && improveCount ==0)
                {
                    hasConverged =true;
                    scores.Add(bestScore);
                    epochCounter++;
                } else if (counter> epochN ||improveCount > threshold)
                {
                    /*double Ttest = costDifference / startT;
                    if ( Ttest<= 0.1 && Ttest>0.05)
                    {
                        Tfactor = .95;
                    } else if ( Ttest <=0.05 && Ttest > 0.025) {
                        Tfactor = 0.975;
                    } else if (Ttest <= 0.025)
                    {
                        Tfactor = 0.99;
                    }*/
            T *= Tfactor;
                    counter =0;
                    improveCount= 0;
                    scores.Add(bestScore);
                    epochCounter++;
                    //Console.WriteLine(String.Format("New epoch: best Path ={0}, best
score = {1}, T ={2}", bestPath, bestScore,T));
                }

            }
```

Appendix D: Prim's Approximate Algorithm Code

```
public void TSP_PRIMS_APPRX()
{
    char root;
    String preTraversal;
    Random rand = new Random((int)DateTime.Now.Ticks);
    Stopwatch stopWatch = new Stopwatch();
    int temp;
    double distance;
    Node u,p;
    MST = new List<Node>();
    Q = new List<Node>();

    stopWatch.Start();
    //1)Calculate Distance MAtrix
    maxCost = CalculateDistanceMatrix();
    //PrintDistanceMatrix();

    //Select a random Vertex to be the root
    temp = rand.Next(n);
    //temp = 0;
    root = (char)(temp + 65);
    //root = 'A';
    MST.Add(new Node { Name = root , Parent = '0', Children = "" });
    Console.WriteLine("Selecting '" + root + " as root node of MST.");
    //Find MST using Prims
    //populate Q with all other non-root nodes and their Value is their distance
      from root
    //sort Q by its nodes' Value property in ascending order
    var sortedQ = from node in Q orderby node.Value ascending select node;
    Q = sortedQ.ToList();
    //Console.WriteLine("Q List");
    for (int i = 0; i < n; i++)
    {
        if ( i != (int)root-65)
        {
            char c = (char)(i + 65);
            Q.Add(new Node { Name = c, Parent = root,Children = "", Value =
             distanceMatrix[temp, i] });
            //Console.WriteLine(String.Format("Name:{0}, Parent:{1}. Value {2}",
             Q.Last().Name, Q.Last().Parent, Q.Last().Value));
        }

    }


    //while Q.Count>0
    while (Q.Count > 0)
    {
```

```
    //take closest point(u), add to MST,
    u = Q.First();
    MST.Add(u);
    //remove from Q
    Q.RemoveAt(0);
    //add it's Name to it's parents children
    p=MST.Find(x => x.Name.Equals(u.Parent));
    p.Children = p.Children + u.Name.ToString();
    //search all remaining points in Q -v, to see if v's distance to u is
         less than their current Value
    foreach ( Node v in Q )
    {
        distance = distanceMatrix[(int)(u.Name) - 65, (int)(v.Name) - 65];
        if (v.Value > distance)
        {
            //if so update v's Parent and Value
            v.Value = distance;
            v.Parent = u.Name;
        }
    }
    //Console.WriteLine("Printing MST: step #"+Q.Count);
    //foreach (var node in MST)
    //{
    //    Console.WriteLine(String.Format("Name:{0}, Parent:{1}. CHILDREN
         {2}", node.Name, node.Parent, node.Children));
    //}
    //sort Q by its nodes' Value property in ascending order
    sortedQ = from node in Q orderby node.Value ascending select node;
    Q = sortedQ.ToList();
    //Console.WriteLine("Printing Q:");
    //foreach (var node in Q)
    //{
    //    Console.WriteLine(String.Format("Name:{0}, Parent:{1}. Value {2}",
         node.Name, node.Parent, node.Value));
    //}
    //PrintNode(MST.First(), MST);
}

//Find Preorder Traversal of MST  as bestPAth
preTraversal = PreOrder(MST.First(),MST);
bestScore = FindPathDistance(preTraversal);

stopWatch.Stop();
string results1 = String.Format("Prims Aproximate:Best path ={0}, Best
  distance ={1}", preTraversal, bestScore);
Console.WriteLine(results1);
StringToIntArray(preTraversal, ref paintPath);
TimeSpan ts = stopWatch.Elapsed;
string elapsedTime = String.Format("{0:00}:{1:00}:{2:00}.{3:00000}",
```

```
        ts.Hours, ts.Minutes, ts.Seconds,
        ts.Milliseconds / 10);
        string results2 = "RunTime " + elapsedTime;
        Console.WriteLine(results2);
        label2.Text = results1 + ", " + results2;

    }
```

References

[1] T.H.Cormen, C.E. Leiserson, R.L.Rivest, and C. Stein. Introduction to Algorithms, 3rd ed. Cambridge, MA:MIT Press, 2009.

[2] D. Wang, D. Lin. ”Monte Carlo Simplification Model for Traveling Salesman” Appl. Math. Inf. Sci. 9, No. 2, 721-727 (2015).

[3] David E. Goldberg. *Genetic Algorithms for Search, Optimization and Machine Learning* Addison Wesley Co. 2008.

[4]Nagham Azmi AL-Madi and Ahamad Tajudin Khader ,”The Traveling Salesman Problem as a Benchmark Test for a Social-Based Genetic Algorithm” , Journal of Computer Science 4 (10): 871-876, 2008 ISSN 1549-3636

[5]C. Papadimitriou and M. Yannakakis, "Optimization, approximation, and complexity classes," in Proc. 20th Annual ACM Symp. on Theory of Computing, New York, NY: ACM Press, 1988, pp. 229-234.

[6]A. Rao, C. Papadimitriou, S. Shenker, and I. Stoica, "Geographic routing without location information," in Proc. 9th Annual Intl. Conf. on Mobile Computing and Networking (MOBICOM '03), New York, NY: The Association for Computing Machinery, Inc., 2003, pp. 96-108.

[7] C. Papadimitriou and S. Vempala, "On the approximability of the traveling salesman problem," Combinatorica, vol. 26, no. 1, pp. 101-120, Feb. 2006.

[8] Yavuz Eren, İbrahim B. Küçükdemiral, İlker Üstoğlu, ed. Ozan Erdinç , *Optimization in Renewable Energy Systems:Recent Perspectives,* 2017 Butterworth-Heineman p.27-74

[9]Černý, V. (1985). "Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm".*Journal of Optimization Theory and Applications*. 45: 41–51. doi:10.1007/BF00940812. S2CID 122729427.

[10]Nicolas Metropolis."The Beginning of the Monte Carlo Method", *Los Alamos Science*, No. 15, Page 125.

[11]N. Metropolis; A.W. Rosenbluth; M.N. Rosenbluth; A.H. Teller & E. Teller (1953). "Equation of State Calculations by Fast Computing Machines". *Journal of Chemical Physics*. 21 (6): 1087–1092. Bibcode:1953JChPh..21.1087M.

[12]Abbbb, Abbb & Shariat, Afshin & Babaei, Mohsen. (2012). "An Efficient Crossover Operator for the Traveling Salesman Problem" . Int. J. Optim. Civil Eng. 2. 607-619.

[13] Ahmed, Zakir. (2014). "The Ordered Clustered Travelling Salesman Problem: A Hybrid Genetic Algorithm." *The Scientific World Journal.* 2014. 258207. 10.1155/2014/258207.

[14] Shafer, C.A., *Data Structures & Algorithm Analysis in C++*. 3rd ed. 2011 Dover Publishing.

Curriculum Vitae

Edward Arturo Friesema Github page: github.com/efriesema
Software Engineer

roninfilms01@gmail.com

Experience: **UNLV RebelSAT-1 Student Satellite Team**
Communications Team Lead(November 2020-present)

- Coordinated with SatNOGS and designed ground station system based on SDR, YAGI antennae, and COSMOS open-source ground station software.

UNLV College of Education, Teaching and Learning Department
Software Engineering GAship(January 2020-present)

- Created and tested a full-stack Python Flask web application for recording and displaying UNLV student-athletes' workout data for analysis by coaches and training staff.
- Conducted teacher workshop for FIRST Robotics competition teaching C algorithm design..

Software Engineering Intern
Jet Propulsion Laboratory(June 2018-August 2018)

- Imported and updated hardware 84 C&DH elements for the Team X systems engineering tools.
- Developed 3 Python and 2 Javascript plugins to add new 5 interface features and options to the original design workflow.
- Presented a software proposal for Marshall Space Flight Center, "Genetic Algorithms for Improved Orbital Calculations."

Languages C/C++, Python 3, C# ASP.NET MVC 5, Verilog, Javascript, HTML5, CSS SQL

Awards 2019 Golden Key National Honors Society Inductee
2018 Nevada Space Grant Fellowship
2017 Herox Cubesat Challenge Engineering Prize - "TweetSat: Communications in a Crisis"- https://www.herox.com/cubesat-challenge/community

Education **University of Nevada-Las Vegas**(June 2017-present)
M.S. focusing on machine learning and AI - 4.0 GPA –
Thesis: "Comparing a New Algorithm for the Traveling Salesman Problem to Previous Deterministic and Stochastic Algorithms"
Tau Beta Pi Chapter Vice President 2018

University of California at San Diego (June 2001)
M.Eng – Electrical and Computer Engineering 3.3 GPA - Optics Specialization
Thesis: "20Hz linewidth millimeter-wave generation by optical sideband filtering" Proceedings of the SPIE,2000

Pennsylvania State University(June 1997**)**
B.S. in Electrical Engineering 3.5 GPA,
Initiated into Tau Beta Pi National Engineering Honors Fraternity