Solving the Travelling Salesman Problem with a Simulated Rover

Jacob Clarke McRae

Queen's University

CISC 499

The Problem

The travelling salesman problem (TSP) asks the following question: “Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?” This is an NP-hard optimization problem that was first formulated in 1930 and is one of the most intensively studied problems in optimization. The decision version of this problem, where given a length L the task is to decide whether it is possible to find any tour of the cities shorter than L, belongs to the class of NP-complete problems. The travelling salesman problem is not only a useful problem for research and algorithm development, but also has many real world applications. Solutions to this problem are used in planning, logistics, DNA sequencing, and the manufacturing of microchips.

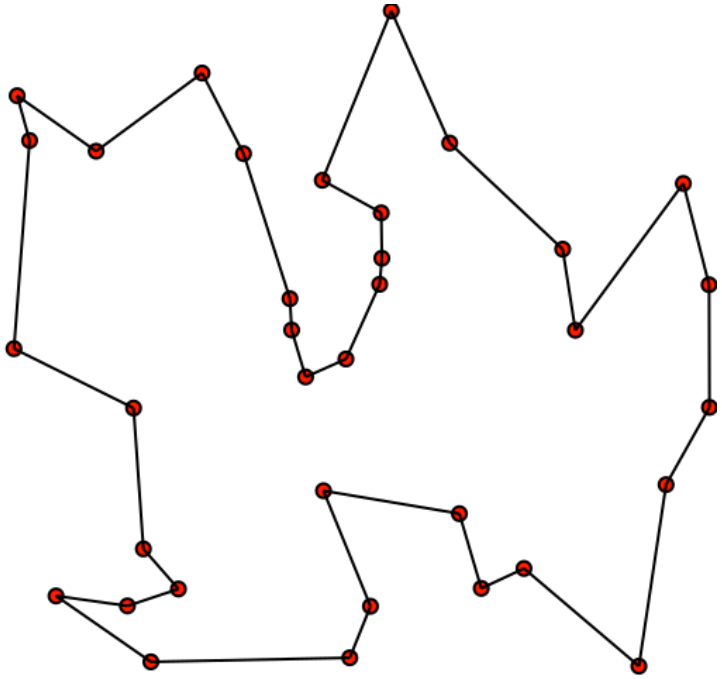

Figure 1: Example of a possible travelling salesman solution connecting all cities

https://en.wikipedia.org/wiki/Travelling_salesman_problem#/media/File:GLPK_solution_of_a_travelling_salesman_problem.svg

Although I set out with the goal of solving the travelling salesman problem, a great deal of my project was spent figuring out how to automate the driving of a virtual rover. In order to have a virtual rover be able to solve the travelling salesman problem, ideally I would have an algorithm determine which “city” the rover should visit next, and then be able to automatically send the rover to that city using a simple command that specifies the coordinates for that city. Combining the algorithm of finding the next unvisited city to visit with the ability to send the virtual rover to that city would complete my goal of solving the travelling salesman problem with a virtual rover.

Based on the above, I was able to split up my problem into two main components, the automation of driving the rover to a specified location, and the implementation of an algorithm to solve the travelling salesman problem. I started out this project attempting to tackle the first problem, automating the driving of the rover.

The problem of automating the driving of the rover based on a specified set of coordinates is a different problem entirely from the problem of solving the travelling salesman problem, and was more complicated than I initially thought. The rover I am working with is a model based on the mars Curiosity rover, and it resides in a Unity simulation environment. It has a set of 6 wheels and I am able to control 3 of these wheels at a time (either the 3 left wheels or the three right wheels). My task was to write a piece of code that would give the necessary commands to the rover so that it drives itself to the specified coordinates.

My Approach

Automating the Driving of the Rover

I began tackling this problem by planning out what I wanted to do, and how I wanted to go about doing it. The task itself is fairly simple: drive a rover from its current location to a given location in the simulation. The simulation does not provide me with very much information, but provides me enough to make this possible. I can receive the x, y, and z coordinates of the rover, as well as the direction the rover is facing. The x, y, and z coordinates are given as floating point numbers representing the position relative to the x, y, and z axes. These axes are broken down in unity the following way:

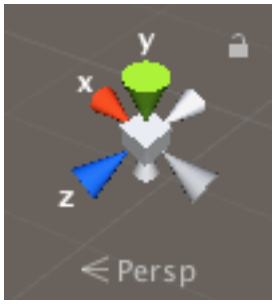

Figure 2: Unity's x, y, and z axes

As you can see above, the x and z axes represent length and width, while the y-axis represents height. Because of the nature of the rover and the flat plane that it operates on, I do not need to use the y coordinate of the rover at all. The rover will always be on top of a level plane, and so the rover will always have the exact same y coordinate regardless of where it is on the plane. I do however need to use the x and z coordinates of the rover often so that I know where on the plane the rover lies. These coordinates can give me information other than the location of the rover, such as the distance between the rover and some other x, z point (using the Pythagorean theorem). I can also receive the direction the rover is facing as a floating-point number between 0 and 360. This number represents the clockwise angle (in degrees) between the direction the rover is facing and the positive end of the z-axis. Below is a simplified version of the unity environment

showing the x and z axes, along with the degrees of direction for the rover relative to the two axes. Lastly, I have access to a list of points or “cities” to visit in order to complete the travelling salesman problem. These points are represented as objects of a class *Point* that I made which has two double precision floating point attributes: x, representing the x coordinate of the city and z representing the z coordinate of the city.

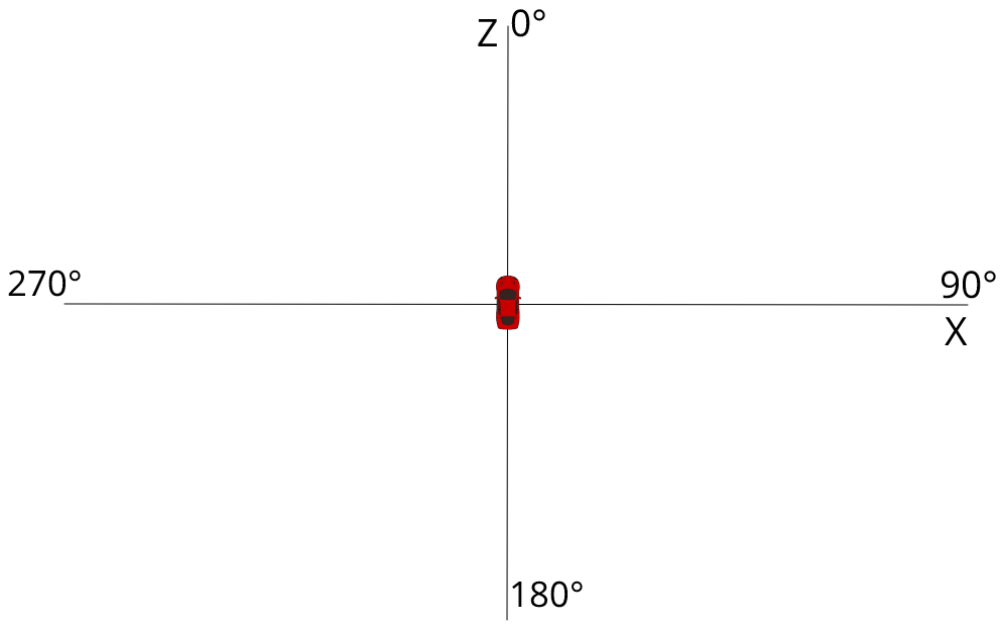

Figure 3: Simplified diagram of the rover in the unity environment, showing x and z axes along with degrees to illustrate which angle corresponds to which direction the rover is facing

Now that I knew what I had to work with, my task was to figure out how to make the rover drive from its current position to the new position it needed to reach. Two possible solutions for this came to mind. I could either immediately start driving and turn towards the goal city while the rover was moving, or I could simply rotate towards the goal city, and drive in a straight line towards that city. After comparing several situations including the one shown in the diagram below, I decided that the most effective solution to this would be to rotate the rover on spot, and then proceed to drive the rover in a straight line to its destination. My thought process was that it would be best to reduce driving distance (potentially

reducing other things like fuel usage and time driving). On top of these benefits, the travelling salesman problem is almost always solved using straight lines between cities as this minimizes distance travelled (which is the goal of the problem) so doing the same with the rover in my project seemed fitting.

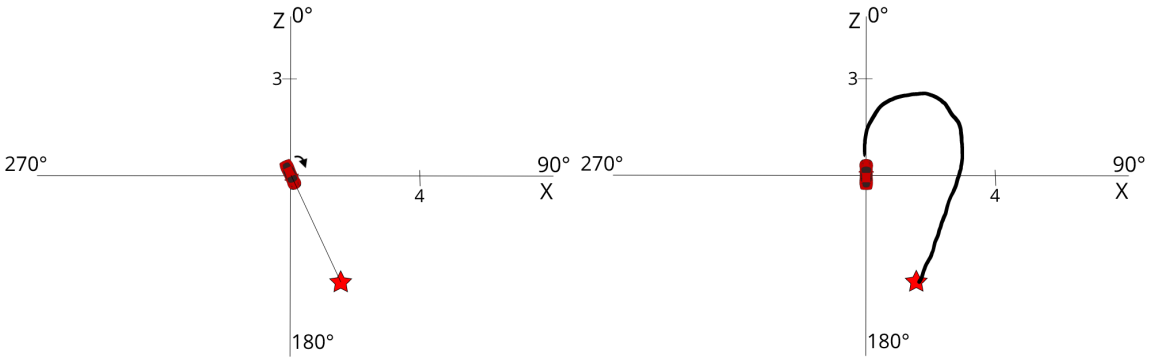

Figure 4: Comparison between two possible approaches to have a rover reach a destination

Once this choice was decided, I needed to start thinking about how to rotate the rover the correct angle so that it is facing the destination city. I thought back to what pieces of information I have available. I have the x and z coordinate for both the rover and the cities, as well as the direction that the rover is facing. I realized that a right angle triangle can be made between the rover and any x, z point with the two perpendicular sides of the triangle being the difference in x and z values as shown in the diagram below.

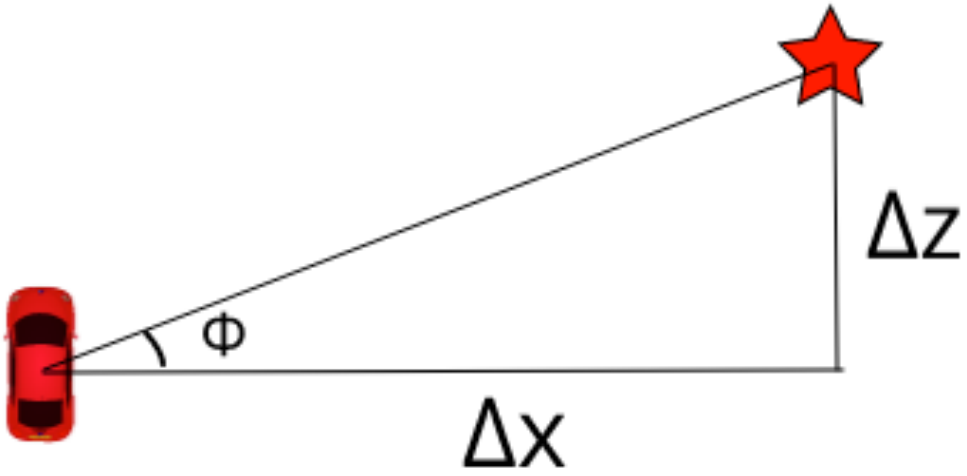

Figure 5: Diagram showing how a right angle triangle can be made between the rover and a city

Using the trigonometric ratios, we know that sin(ϕ) is equal to opposite side / hypotenuse of the triangle. With the above diagram, we have $\sin(\phi) = \Delta z / \sqrt{(\Delta z^2 + \Delta x^2)}$. Now we can reorder the equation to solve for the angle ϕ like this: $\phi = \sin^{-1}(\Delta z / \sqrt{(\Delta z^2 + \Delta x^2)})$. We now have the angle ϕ, which is helpful, but not exactly what we are looking for. We want the angle between the direction the rover is facing and the goal city (θ), which in the above case would be equal to 90 – ϕ. We need to subtract the angle from 90 because we have found the angle between the city and the x-axis rather than the angle between the city and the z-axis. We want to find the angle between the city and the z-axis because the z-axis represents 0 degrees of rotation for the rover (refer to Figure 3) and once we the angle relative to 0 degrees of rotation for the rover, all we need to do is subtract the rotation angle of the rover from that angle and we will have the correct angle we are looking for: the angle between the direction the rover is facing and the goal city.

Now, in the above example, the direction the rover is facing is represented by 0 degrees, so the angle between the rover and the city (we will represent this angle by θ), could be represented by the equation θ = 90 – ϕ. If however, the rover wasn't faced straight forwards, we would also have to adjust for this by subtracting the angle of the rover from the equation so that it becomes the following: θ = 90 – ϕ – compass. We are very close to having a general equation to find the angle wherever the rover and the city are located.

The last thing needed to do is to generalize the equation so that it still holds true regardless of where the city is relative to the rover. All our work so far has been using the example from figure 5 where the city is to the right and above the rover (when the city is in the first quadrant of the plane relative to the rover). There are still 3 quadrants of the plane that we have not found the correct equation for. In order to generalize the equation so that it covers all four quadrants, we only need to change the highlighted part of the equation: θ = 90 – ϕ – compass. This part of the equation came from trying to take the angle ϕ and show that angle as the angle between the city and the positive end

of the z-axis instead of the angle between the city and the x-axis. In the other quadrants, the angle ϕ will always still be the angle between the city and the x-axis, but the method of converting that angle to be relative to the positive end of the z-axis will change depending on which quadrant the city is in, since different quadrants lie different degrees away from the positive end of the z-axis. Here is a table showing the necessary adjustments depending on the quadrant:

Quadrant	Equation
I (as shown in figure 5) up and right	θ = 90 – ϕ – compass
II up and left	θ = 270 + ϕ – compass
III down and left	θ = 180 + ϕ – compass
IV down and right	θ = 90 + ϕ – compass

Below is a diagram that shows the complete four equations depending on the quadrant the city is located in. Looking at the degree markings located at each end of both axes can give you a better idea of where these adjustments are coming from when you remember that the angle ϕ is always the angle relative to the x-axis.

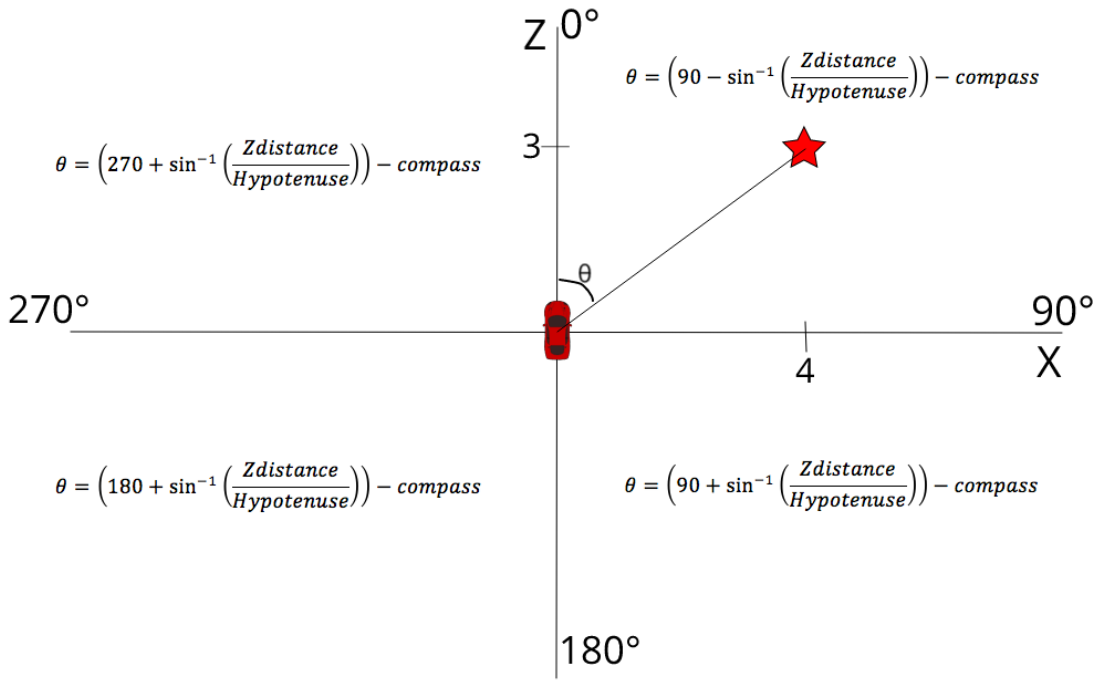

Figure 6: Complete equations for all four quadrants shown on a plane

We are now able to find the angle between the rover and any city in degrees. Our next task is to be able to rotate the rover by any given angle, so that we can give take our calculated angle and have the rover in the simulation rotate and point towards the next city it needs to visit. The command to send the rover to make it rotate clockwise on spot is:

```
setLRPower(n,-n)
```

where n is the speed that the wheels are set to. The first parameter represents the speed for the left set of wheels and the second parameter represents the speed for the right set of wheels. These speeds need to have the same magnitude but different signs in order to correctly rotate on spot.

Using the above command, the angle of rotation is determined by the delay between the time when the speed for the wheels is set to n, to the time that the speed of the wheels is set to 0. After some experimentation, I decided to use the speed 500 for a rotation. So the angle of rotation for the following code would be completely dependent on the time m in milliseconds that the program sleeps for in between starting the rotation and setting the speed of the wheels back to 0.

```
setLRPower(500,-500)
Thread.sleep(m)
setLRPower(0,0)
```

My first thought when trying to figure out how to determine the exact number of milliseconds to sleep for to rotate a specific angle was simple: figure out how many milliseconds the delay needs to be to rotate 180 degrees (let's call that delay d), and then in order to rotate an angle θ, I could just set the time to sleep to m = θ/180 * d. This approach would work perfectly if the wheels of the rover instantly jumped to a speed of 500 and -500 when the first command was sent, and instantly stopped rotating when the second command was sent. The simulation is more realistic than this though, it takes time for the wheels to reach the speed they are set to. The acceleration that is present in this simulation leads to this problem not scaling linearly, which means my simple idea of setting m to a fraction of the milliseconds it takes to rotate 180 does not work accurately.

This makes things slightly more complicated, but is definitely not an unsolvable problem. If some function is not linear, than there should still be some equation that can map angle desired to milliseconds delay. Curve fitting can be used to approximate this equation. There are many different curve-fitting models, but after some research, I learned that the 4 Parameter Logistic (4PL) curve is the most recommended for fitting a standard curve with an upper bound. This is the equation that is used for a 4PL curve:

$$y = d + \frac{a - d}{1 + \left(\frac{x}{c}\right)^b}$$

At this point, all I need to do is to find adequate a, b, c, and d values that can approximate the relationship between milliseconds of delay and angle of rotation. After some more research I found a curve-fitting tool online that does exactly that (https://mycurvefit.com/). This tool allows you to enter a set of data points (minimum of 4) and it will give you values for a, b, c, and d that approximates a curve that fits the given data points. I entered 11 data points, each having an angle in degrees as well as the associated milliseconds that my program slept for to obtain that angle. I did this by outputting the angle the rover was facing before and after a rotation where I would sleep for m milliseconds. In code this was simple and looked something like this:

```
before = getCompass()
setLRPower(500,-500)
Thread.sleep(m)
setLRPower(0,0)
after = getCompass()
print(after – before)
```

As I said before, I tried 11 different m values and entered those along with the angle that was rotated as data points into the curving fitting program. This was the resulting equation I was given by the program:

$$y = 44689.9 + \left(\frac{38.8 - 44689.9}{1 + \left(\frac{x}{203.8}\right)^{2.1}}\right)$$

This equation was very simple to integrate into my code as a java method. After doing this, I was able to determine necessary delay time in milliseconds with a quick method call having the angle of rotation as a parameter. Because I already figured out how to calculate the angle of rotation, I now had everything I needed in order to rotate the rover a set amount of degrees. The following simplified code is able to rotate the rover any specified number of degrees:

```
// Rotates Rover Left by "degreesLeft" degrees
private static void rotateLeft(double degreesLeft){
     int rotateTime = timeToRotate(degreesLeft);

     // Start Rotating Rover Left
     rover.setLRPower(-500,500);
     // Sleep Until Rover is facing goal
     Thread.sleep(rotateTime);
     // Stop Rotating Rover
     rover.setLRPower(0,0);
}
```

Where “timeToRotate” is a method that implements my curve fit equation.

Now that the rotation has been figured out, all I needed to do to bring the rover to the goal city was to drive it in a straight line until it got there. Doing this was much simpler than rotating the rover. Driving works very similarly to rotating, you give power to the rover’s wheels, sleep for a set amount of time, and then stop the rover. The code needed to do this is very similar to the code for rotating the rover with the only change being the speeds the wheels are set to:

```
setLRPower(100,100)
Thread.sleep(m)
setLRPower(0,0)
```

I decided to use a speed of 100 for driving in a straight line as it provided a more consistent drive between sets of points. This is because acceleration is very slow in the simulation and so setting high speeds can have the rover seem to travel much faster between further separated points while not improving overall rover speed by a noticeable amount (since so much of the time the rover is moving is taken up by acceleration and not the speed the rover gets set to). I wanted to have the rover’s speed seem to be about the same between all cities.

This acceleration also led to the same problem that it did for rotating: the relationship between distance travelled and delay time was not linear. I used the exact same solution to this problem that I did for rotating the rover, I used the curve-fitting tool on a set of data points I found experimentally (with the same process I used for rotating) in order to find the following 4PL function:

$$y = 167025.8 + \left(\frac{-54.3 - 167025.8}{1 + \left(\frac{x}{333.3}\right)^{1.2}}\right)$$

After implementing this equation into my code, I was able to convert travel distance to milliseconds of delay time. Once I was able to do this, I was able to write a method that would drive the rover any specified amount of units forward, just like my method for rotating the rover:

```
// Drive forward for "distance" units
private static void driveFor(double distance){
     int driveTime = timeToDrive(distance);

     // Start Rover
     rover.setLRPower(100,100);
     // Sleep until rover has reached destination
     Thread.sleep(driveTime);
     // Stop Rover
     rover.setLRPower(0,0);
}
```

Because of the approximations done by the curve fitting where I only provided about a dozen data points, the angle rotated and distance travelled can be off by a small margin, but after some tests I found that the rover would rotate within about one degree of the desired angle and would drive within about one unit of the desired distance. This accuracy was more than enough for my simulation, especially since these inaccuracies never add up because of the fact that I request and use the exact current x and y coordinate as well as direction angle of the rover after every visited city (rather than the angle and coordinates that the rover should be in based on my calculations). If I wanted even higher accuracy than I was getting, I could have simply entered many more data points into the curve fitter and received a more accurate approximation equation.

I now have everything I need to complete my first task of automating the driving of the rover. Combining the functionality I have outlined above, I can make one function that takes an x and a z coordinate and automatically drives the rover to that coordinate without the need of any manual interaction. We will call this function “driveTo” which takes two parameters, an x and a z coordinate.

Solving the Travelling Salesman Problem

My next task is to actually implement a travelling salesman algorithm in order to solve the problem. All the algorithm has to do is determine the next city the rover needs to visit, and by using my driving function, the rover will drive to all cities and then return to its starting city, completing a tour and finding a solution to the travelling salesman problem. As I mentioned earlier, I have access to a list of Point objects, each object specifying the x and z coordinate that make up that point.

There are several different algorithms that can be used to solve this problem. Some of these algorithms find an approximate solution to the problem while some find the exact or optimal solution to the problem. Approximate solutions take less time to compute but often find longer routes while exact solutions take longer to compute but find the shortest possible route. I decided to implement the nearest neighbour (NN) algorithm, a greedy algorithm which simply chooses the nearest unvisited city to travel to next. This will find an approximate solution to the problem, not necessarily the optimal solution.

All I need to do to implement this algorithm is create a function that returns the nearest unvisited point to the rover, and removes that point from the list of unvisited points. Because I have a list of unvisited points already, I can create a simple function to find which of these points is closest to the rover. Here is some pseudo code that shows how I wrote a function to do this. We will refer to this function as “greedyNextPoint” which takes one parameter, the current location of the rover. The list of unvisited points is stored as a global variable so there is no need to pass that list as a parameter.

```
for every point in list of unvisited points:
    find the distance between current point and the rover
    if that point is closer than the nearest stored point:
        store that point as the new nearest stored point

remove the nearest stored point from the list of unvisited
points

return the nearest stored point
```

The calculation to find the distance between a point and the rover can be done easily by using the Pythagorean theorem:

$$distance = \sqrt{((point.x - rover.x)^2 + (point.z - rover.z)^2)}$$

With this function implemented and working correctly, I can have the rover solve the travelling salesman problem with a simple loop. One thing I have to make sure of is that the rover starts at one city. It does not matter which city the rover starts at because if you look at a completed tour of cities (see figure 1), any city can be seen as the first and last city. Depending on the configuration of the cities, the rover may not start on top of a city when the simulation is started, so to ensure it does we need to add a line that picks the nearest city and chooses that city as the start and end point of the rover's tour. We then need to drive to that start city before and after the main loop, which will drive to every other city in the list. Here is some pseudo code that shows how to have the rover drive one tour, visiting every city and returning to its starting city:

```
current = new Point(rover.x, rover.z)
startCity = greedyNextPoint(current)
driveTo(startCity)

while list of unvisited points is not empty:
    destinationPoint = greedyNextPoint(current)
    driveTo(destinationPoint.x, destinationPoint.z)
    current = destinationPoint

driveTo(startCity)
```

At this point I have successfully managed to have the rover automatically solve any instance of the travelling salesman problem, using the NN algorithm.

Results

Comparing Nearest Neighbour Algorithm with Optimal Solution

Now that I can use the nearest neighbour algorithm with the rover, I wanted to see how this algorithm compares to the optimal solution, on average. I needed to figure out a way to quickly compare these two ways of solving the problem, and the fact that the rover moves slowly and takes multiple minutes to solve each instance of the problem meant that running the complete simulation wasn't very feasible for obtaining results.

Instead of running the simulation, I could simply use my "greedyNextPoint" function to figure out the next point the rover would travel to, find the distance between the rover and that point, then add that distance to some running sum of total distance the rover would cover using this algorithm. This is very simple and will return the total distance for a tour practically instantly. Here is some pseudo code showing how I am able to do this.

```
sumDistance = 0
intialPosition = new Point(rover.x, rover.z)
startPoint = greedyNextPoint(intialPosition)
current = startPoint

while list of unvisited points is not empty:
    destinationPoint = greedyNextPoint(current)
    sumDistance += distanceBetweenPoints(current, destinationPoint)
    current = destinationPoint

sumDistance += distanceBetweenPoints(current, startPoint)

return sumDistance
```

The above code uses a function "distanceBetweenPoints" which simply calculates the distance between points the same way as in the "greedyNextPoint" function, using the Pythagorean theorem, which I showed in more detail above. Now I can enter any list of cities into the list of unvisited points, run the above code and receive the total length the nearest neighbour algorithm would cover when solving the problem with that list of cities.

I decided to create 5 randomly generated sets of points, each set containing 20 different points. I did this by randomly choosing two values for each point, one for the x coordinate and one for the z coordinate. To ensure these sets of data are sets that could really be used in my simulation, I had to set upper and lower limits for the x and z values. When making points to test my simulation earlier, I figured that it was best to keep points with minimum x and z values as -35 and maximum x and z values as +35. This way none of the points would be too far from the rover or off of the plane that the rover lies on. This allows these data sets to be used as real cities for my simulation if I ever wanted to actually watch the rover's solutions. I also decided to keep the starting and ending point of the rover the same between each set of data. I did this because I found that a simple starting point located right in front of the rover reduced set up time where the rover had to drive to the initial city (even though this wouldn't matter when using my code to calculate the total distance rather than running the complete simulation). Here are the 5 randomly generated sets of data.

Set 1:
(0,6),(15,-34),(-12,28),(31,-20),(-19,-2),(0,30),(1,0),(-31,-4),(33,-30),(18,23),(-25,2),(6,-11),(-5,29),(34,28),(-15,5),(-8,32),(34,32),(-12,-32),(-11,-20),(24,-30)

Set 2:
(0,6),(15,21),(-29,20),(-15,31),(-13,-26),(1,16),(-21,22),(-25,-26),(23,-20),(-27,14),(-9,-11),(3,1),(2,2),(22,-29),(17,-9),(23,25),(-13,22),(-17,-25),(-14,-20),(33,-21)

Set 3:
(0,6),(18,5),(-27,1),(28,5),(24,-34),(-4,-26),(6,-1),(-1,1),(3,28),(-24,-16),(-18,9),(32,-6),(-5,6),(28,-2),(-18,-3),(-2,21),(-25,-29),(17,6),(-9,-2),(-5,-23)

Set 4:
(0,6),(-1,3),(-4,-17),(21,-13),(5,-7),(-14,-32),(-13,-24),(-12,-4),(-9,-28),(-27,5),(28,-16),(13,-4),(-14,6),(-19,21),(0,-9),(-6,15),(14,5),(32,24),(16,9),(-25,29)

Set 5:
(0,6),(2,18),(19,18),(32,-35),(1,30),(8,25),(8,20),(32,4),(14,9),(-30,-33),(28,-1),(-13,-17),(7,-6),(20,34),(7,6),(23,6),(26,-2),(-19,28),(-33,-26),(-26,6)

I entered each of these sets as the list of unvisited cities, and then used my code to tell me the distance of the greedy solutions to these five instances of the problem. My next task was to compare these distances to the optimal solutions for these sets of twenty points. Wolfram MathWorld provides a command FindShortestTour[g] that takes a graph or set of points and finds the optimal solution to the travelling salesman problem using that graph or set of points. This command can be used in the Wolfram Alpha computational knowledge engine which is available online at https://www.wolframalpha.com/. I used this to calculate the shortest possible distance for each set of point and then compared the results as shown below.

Data Set (n = 20)	Greedy Distance Travelled	Optimal Distance Travelled
Set 1	299.8	280.0
Set 2	291.0	263.7
Set 3	293.0	249.2
Set 4	309.6	256.5
Set 5	388.1	312.8
Average	316.3	272.4

From these five sets of data, each having 20 points, we can see that that the average distance travelled using the greedy algorithm was 316.3 units, while the average distance travelled for the shortest possible route was 272.4 units. The nearest neighbour algorithm on average, took a route that was 16.1% longer than the shortest possible route. The worst run for the greedy algorithm was on data set 5 where it took a route that was 24.1% slower than the shortest route. The best run for the greedy algorithm was on data set 1 where the greedy algorithm took a route that was only 7.1% longer than the optimal route.

These results are actually better than I expected. According to David S. Johnson et al., researchers who wrote a paper in 1997 about optimizing the travelling salesman problem, the nearest neighbour algorithm yields a path 25% longer than the optimal path on average. In addition to this, Gregory Gutina et al. showed in a 2002 study, that there exists many specially arranged city distributions that lead to the nearest neighbour algorithm producing the worst possible route. I believe the reason I obtained slightly better results was that my test cases were restricted to having only 20 points randomly distributed within an area of 4900 units. Using more points or a bigger area for the problem would lead to the algorithm performing even worse on average, and would produce a result closer to the 25% longer path on average. Using a higher number of data points and data sets would increase the sample size and lead to more accurate results.

Overall, I concluded that if minimizing distance travelled was the top priority, it is best to avoid any approximation algorithms, especially the nearest neighbour algorithm as they will lead to longer routes. If this is the case, only optimal solutions to the problem should be used. There are a number of algorithms that can be used to find optimal solutions including branch and bound algorithms and dynamic algorithms. The best complexity for one of these exact algorithms discovered so far is $O(n^2 2^n)$.

If, however, finding the absolute shortest path is not necessary, and reducing computational time is of greater priority, then there is no need to use an exact algorithm to find the optimal solution to this problem. Using the nearest neighbour algorithm or other approximation algorithms is just fine, especially when working with a smaller environment with fewer cities like I was working with. The nearest neighbour algorithm runs in $O(n^2)$ time which is a great deal less computationally intensive than exact algorithms for problems with a large number of cities. If this seems more appealing than finding the exact solution every time, it might be worth exploring some different approximation algorithms than the nearest neighbour solution. They will almost certainly have worse complexities, but may lead to a noticeable improvement on average tour distance.

Future Work

Had I had more time to work on this project, or encountered fewer hurdles along the way, there are several things I would have liked to have done. I think it would have been very interesting to compare the results of several different approximation algorithms in addition to the nearest neighbour algorithm. Some algorithms that I became interested in during this problem were: Christofides algorithm, genetic algorithms, and ant colony optimization algorithms. Ideally it would have been nice to be able to compare these algorithms to the optimal solution for a number of different sets of cities, and then compare how they performed relative to each other while also looking at the complexity of the algorithms. It would have been interesting to see if there was one approximation algorithm that seemed to have the best combination of performance and complexity, so I could find one algorithm to recommend if an individual is looking to use an approximation algorithm to solve this problem.

Another thing that I would have found interesting would be to compare the algorithms based on more than one metric. For instance I could compare not only the distance of the tours that each algorithm comes up with, but also the time it took the rover to complete these tours. The time the rover takes would probably be closely proportional to the distance of the tour, but things like the angle the rover has to rotate to point at the next destination may cause some differences between time and distance. Another more complex metric could be fuel used per solution. The rate of fuel used could depend on the speed of the rover and whether the rover was driving or rotating. I think each of these metrics would be closely related, but still very interesting to see subtle differences between them.

References

ELISA Analysis – Free ELISA Software, ELISA Curve Fitting, ELISA Data Analysis Software, 4PL Curve Fitting. (n.d.). Retrieved March 15, 2018, from http://elisaanalysis.com/knowledge-base/elisa-software-4-parameter-logistic-4pl-nonlinear-regression/

Four Parameter Logistic Regression. (n.d.). Retrieved March 15, 2018, from https://www.myassays.com/four-parameter-logistic-regression.html

Gutina, G., Yeob, A., & Zverovich, A. (2002). Traveling salesman should not be greedy: Domination analysis of greedy-type heuristics for the TSP. *Discrete Applied Mathematics*. Retrieved March 7, 2018.

Johnson, D. S., & McGeoch, L. A. (1997). The Traveling Salesman Problem: A Case Study in Local Optimization. *Local Search in Combinatorial Optimization,* 215-310. Retrieved March 21, 2018, from https://www.cs.ubc.ca/~hutter/previous-earg/EmpAlgReadingGroup/TSP-JohMcg97.pdf.

Traveling Salesman Problem. (n.d.). Retrieved March 21, 2018, from http://mathworld.wolfram.com/TravelingSalesmanProblem.html

Web Applications Used for Data Processing:

https://mycurvefit.com/

http://www.wolframalpha.com/