Project Title:
The Traveling Salesman Problem and Evolutionary Algorithms

New Mexico Supercomputing Challenge

Final Report
April 2, 2014

Team 82
Manzano High School

Team Members:

Ian Wesselkamper

Sponsoring Teacher:

Mr. Schum

Table of Contents

Introduction:

NP-complete problems are problems where the amount of time to solve the problem is defined by a non-polynomial equation. In other words, it increases extremely quickly with the size of the problem. Exact solutions to these problems become infeasible as the problem size increases due to large increases in computation time.

The traveling salesman problem (TSP) is a classic example of an NP-complete problem. In the traveling salesman problem, a salesman has many cities that he needs to visit. He wants to save time by traveling the shortest route possible. As more cities are added, the time it takes to exactly solve the problem increases in a non-polynomial fashion. This means that the time it takes to solve the problem increases exponentially. An exact solver for the problem, such as a brute force solver, takes longer at an ever-increasing rate, or non-polynomial time. A program that can solve the problem in polynomial time, albeit not exactly, is notable because the problem itself is NP-complete.

Evolutionary algorithms are excellent candidates for finding optimal but inexact solutions. Evolutionary algorithms apply the principles of evolution, random mutations using competition to perpetuate the best candidates, to finding an optimal solution. These algorithms are useful in solving problems which are otherwise difficult to solve. Evolutionary algorithms are a method to find an inexact solution (or sometimes the exact, but unproven, solution), while keeping computation time down.

Executive Summary:

In this project, I attempt to solve the Traveling Salesman Problem (TSP) using evolutionary algorithms, and compare this method to other methods like a brute force solver and a greedy solver. The Traveling Salesman Problem is NP-complete because more cities to travel to are added, the number of solutions that need to be compared or searched to increases in a non-polynomial fashion. The solution that an evolutionary algorithm uses ignores large numbers of poor solutions, and optimizes out solutions in a random, yet methodical way, like evolution.

First, I created problem generators. There were 3 generators – a line known-answer-test, a circle-and-spokes known answer test and a random generator.

Then I created a greedy solver. The created solver works very well on the line known-answer-test, but cannot always solve problems from the other 2 generators.

I also created a brute force solver that used recursion to crawl the solution tree. An improved brute force solver was written that pruned the tree to shorten the solution time. However, this was still solving the TSP in non-polynomial time. The brute force solver was used to generate known answers for random problems with small numbers of cities.

Then I generated an evolutionary algorithm solver. The evolutionary algorithm solver required some fine tuning to get the desired behavior. The most important improvements were around the types of mutations that it created. However, even then it would get stuck in local minimums when attempting to solve problems from the random generator. The evolutionary algorithm was able to get to its best solutions in polynomial time allowing solving problems with many, many more cities than would ever be possible with a brute force solver.

The optimal solver ended up being a hybrid solver. This was my most significant achievement on the project. I discovered that by seeding the first generation of parents with solutions from a greedy solver, the performance of the evolutionary algorithm was greatly enhanced. When following a greedy solver, the initial generation starts significantly closer to the real answer. Being closer to the start results in fewer generations to find the real answers.

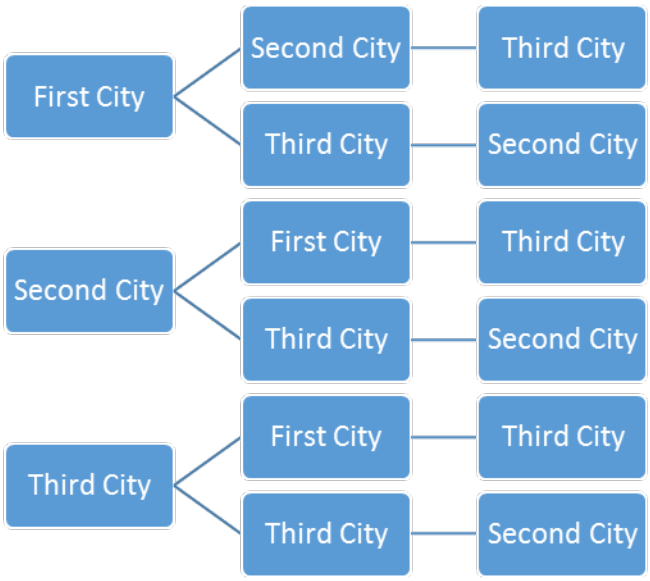
When comparing the solver, it is readily apparent how inefficient the brute force solver is. In one year of time on my PC, the brute force solver would solve at most a problem with 29 cities in 1 year. The pure evolutionary solver would solve 430 cities in 1 year. The hybrid

greed/evolutionary solver was several orders of magnitude faster than the pure evolutionary solver.

The Solvers:

Brute Force Solver

The first solver was the brute force solver. The brute force solver uses a recursive subroutine to search a tree of possible solutions. An example solution tree is shown in the figure below.



Example Solution Tree for 3 Cities

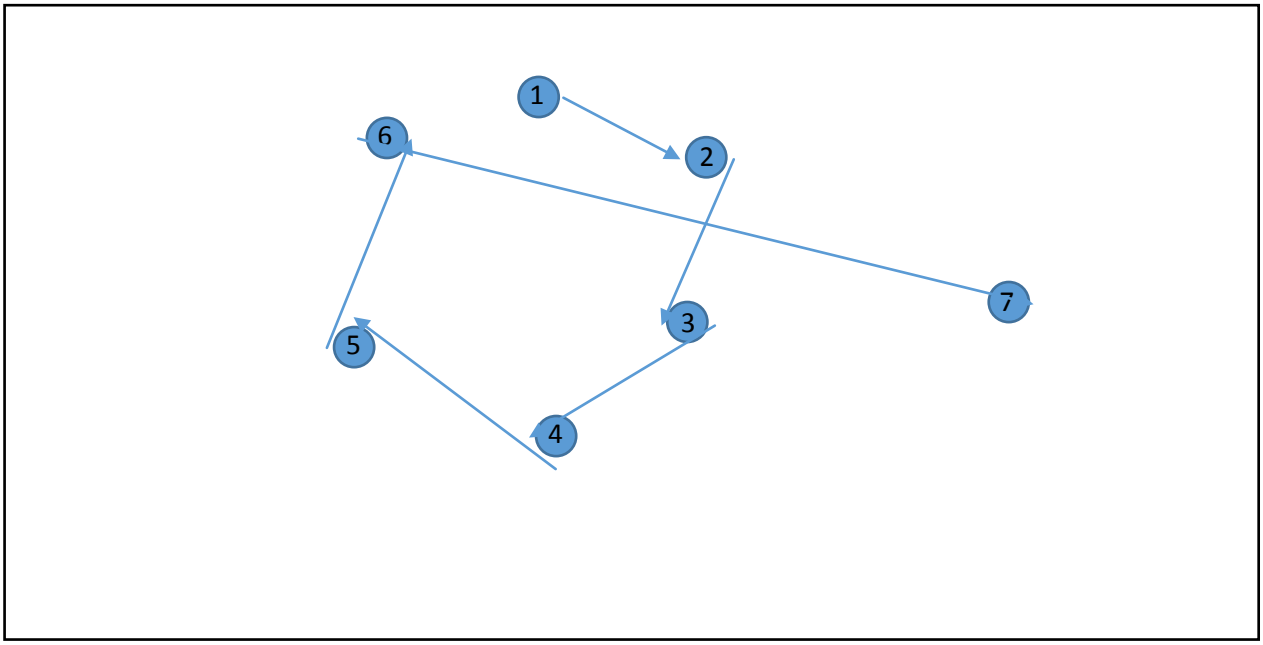
The pseudo-code for the recursive subroutine is as follows –

```
recursiveFunction (problem, solution_so_far):
    foreach city in (cities):
        if (city not already in solution_so_far) then
            append city to solution_so_far
            recursiveFunction(problem, solution_so_far)
```

An improved version of the brute force solver was created that abandoned the search on a given branch if the distance of the solution_so_far exceeded the optimal solution even before the all of the cities were added to the solution.

Greedy Solver

The second solver was a greedy solver. The greedy solver looks at only "n" possible solutions for a problem with "n" cities. It uses a different starting city for each possible solution. It builds each solution by creating a chain were the next city is the closest unchosen city to the current city in the chain. The greedy solver always works with the line known-answer-test. However, it is flawed and it easy to create problems that break the greedy solver. Below is an example of how the greedy solver would solve a solution starting at city 1 and always jumping to the next closest solution.

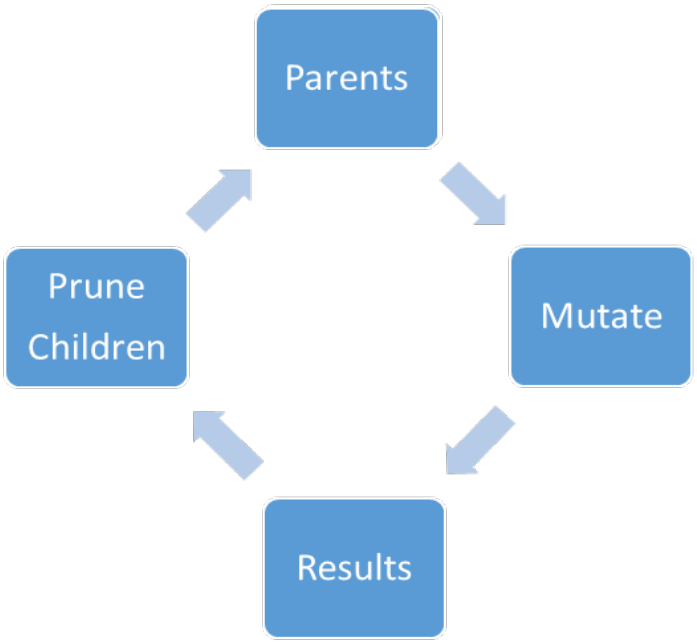


Example of Greedy Solver Starting at City 1

Notice that it gets pulled away from city 7 because it always chased the closest city. This can "trick" the simple algorithm into a solution that is not exact.

Evolutionary Algorithm Solver

The evolutionary solver was the focus of my project. The evolutionary solver consists of 4 steps: create parents, mutate, calculate results, prune children. These steps are shown in the figure below.



Steps in the Evolutionary Algorithm

The ways that the parents are mutated massively affects the outcome of the program. Much of my time was spent looking at where the evolutionary algorithm was getting stuck and what new mutations it needed to find better solutions. The first versions simply picked one individual city in the solution and moved it to a new place in the order of cities. This resulted in "optimal" solutions where whole subsets of cities that are in the wrong spot and moving cities one by one greatly decreases the likelihood that these subsets of almost right solutions get put in the right spot. As an example, an ideal solution could be (1, 2, 3, 4, 5) and this type of mutation would get stuck on (3, 4, 5, 1, 2). The solution to this problem was to move whole subsets of

cities around at random. The size of the subset to be moved is also random. This allows whole subsets to be moved into the correct space in only one operation.

The second problem was that some “optimal” solutions could not jump to the exact solution because the number of mutations required to reach that solution did not match the number of mutations the algorithm was making. This could either be 1 mutation was needed and the algorithm as making 3 mutations or it could be that 3 mutations were required but the algorithm was only doing 1 mutation and 1 mutation resulted in a really bad solution. The solution to this problem was to randomize the number of mutations as well.

A third problem was that some “optimal” solutions had subsets in the wrong order. For example, if (1, 2, 3, 4, 5) was the exact solution, the algorithm could become stuck on (3, 2, 1, 4, 5). The solution to this problem was to allow subsets to be reversed as well.

The first evolutionary algorithms that I created used completely random solutions for the 1st generation of parents. This solver was compared against the brute force solver using problems from the random generator for small numbers of cities. Large numbers of cities used the line known-answer-test, since the brute force solver could not solve problems with more than 10 or so cities in a reasonable amount of time.

Hybrid Greedy/Evolutionary Algorithm Solver

Towards the end of the project, I discovered that using a hybrid greedy/evolutionary solver produced exceptional results. This was my best achievement on the project. This algorithm would first run the greedy solver and save the solution for each possible starting city. These solutions would then be used as the first generation of parents in the evolutionary algorithm. This method significantly reduces the number of generations needed to solve a problem and also greatly improves the likelihood of finding the exact solution.

With the un-pruned brute force solver, 10 cities could be solved in 1 year on my PC. With a pruned brute force solver, 29 cities can be solved in 1 year. With the evolutionary solver,

430 cities can be solved in 1 year. Examining the number of permutations that each problem has really puts the efficiency of the algorithm into perspective. This is shown in the following table.

Algorithm	Maximum Cities	Permutations
Unpruned brute force	10	$10! = 3.6 \times 10^{6}$
Pruned brute force	29	$29! = 8.8 \times 10^{30}$
Evolutionary	430	$430! = 2.3 \times 10^{947}$

Number of Permutations for Maximum Cities in 1 Year of Compute Time

Verification of results:

There are many ways to verify or break solvers, and find problems in the solvers if they exist. I used three methods of breaking and testing my solvers. I wrote 3 problem generators. The main input to each generator was the total number of cities to use in the problem. The output of each generator was a 2D array of "n" x "n" size where "n" is the number of cities in the problem. This array was a lookup table for the distance between two cities. These problem generators were a Line generator, a Random generator, and a Circle and Spokes generator.

The line generator placed all of its cities equidistant apart on a straight line. The Line generator is a known answer test, for it only has 2 possibilities. The 2 possibilities being 1 to N in order, and the other possibility N to 1 in order.

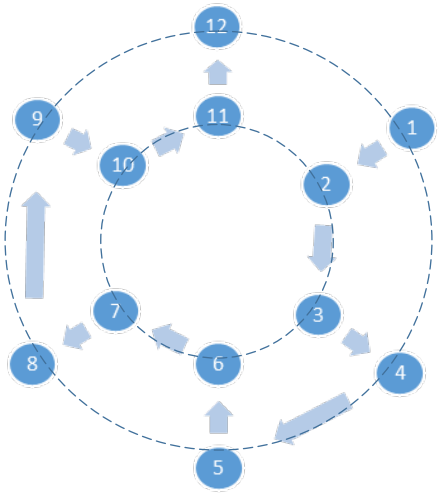
Line Problem with All of the Cities in an Ordered Straight Line

The line generator could be used in speed tests for the evolutionary solver and un-pruned brute force solver. The random nature of the evolutionary solver prevents it from seeing the contrived order of the problem. The unpruned brute force solver is an exhaustive search of every single possible solution so it too is immune to the contrived order of the problem. However the

line generator could not be tested for speed with a pruned brute force solver, because it would quickly find 1 to N within a short amount of time and then ignore an artificially large portion of the solution. The greedy solver also finds the contrived nature of the line generator problems easy to solve.

The next puzzle generator was random generator. There were 2 versions of this that were used during the course of the challenge. The first version of the random generator created random distances between cities. This could possibly result in cities that could not be represented on a 2D plane. This made visualizing the problem difficult. The second version created random x and y coordinates for each city and then the distance between the points using the distance formula and gave those distances to the solvers. This made visualizing the problem on a piece of paper easier. It also more closely represents a real world version of the TSP.

The third and final generator written for the project was Circle and Spokes, which was meant to stump the greedy solver. The generator created a circle of cities, then added 2 or more outliers that were further than the average distance between the cities on the center circle. When the greedy solver looks at this problem it will go around the circle first, then see that it has outliers to go to and reach those last. This results in a wrong answer. But the brute force solver will see all solutions and see that overall going around the circle, and stopping at the spokes as it passed would be an optimal solution.



Circle and Spokes Example with Solution Shown

Early on, I was able to get my brute force solver to correctly solve the problems with a small number of cities and could use that to compare against solutions the other solvers. This was important when using the random problem generator since it does not have known answers. This allowed me to find situations that the evolutionary algorithm solver cannot solve on its own.

At first I tested the evolutionary solver on problems from the line generator as that was an easy known answer test to use. Also the evolutionary solver doesn't see the line as a line, because it is random in how it solves the problem. This is useful because it doesn't require much extra code, compute time or file IO to find the answer because there are only 2 possible answers for every puzzle. I later on shifted the evolutionary solver problems from the random generator with known answers from the brute force solver. This showed me that it wasn't finding the accurate answers easily when given a wider set of possibilities. This is method of testing is where many of the improvements to the evolutionary algorithm came. It allowed for a large amount of testing on random puzzles up to 14 cities. More than 14 cities would take too long for the brute force solver to find the solution to use as a known answer.

By the end of the project, I had created the hybrid greedy/evolutionary solver that would run the greedy solver first and use the greedy solver solutions as the first generation of parents. This lead me to the discovery that was contrary to my initial predictions on the greedy solver. The greedy solver was able to solve many problems from the random problem generator (all under 14 cities because these are the only random problems with known answers).This led me to use the circle and spokes problem to force the greedy solver to fail, in which the evolutionary solver would find the correct answer in only a few more generations. With the circle and spokes problem I had the same limitation as random problems in that I don't have a method to find the exact answer for any generated circle and spokes problem.

Configuring the evolutionary solver:

As many people who have used evolutionary algorithms to solve their own problems should know, evolutionary algorithms can take a bit of tweaking and modification to make them work well enough for a specific problem. A “tuned” evolutionary algorithm is therefore often specific to a problem and what the problem requires. After tweaking the evolutionary solver a bit, I have figured out a few things to watch out for with an evolutionary solver for the Traveling Salesman Problem. Evolutionary algorithms center on a generation. The more ways a generation can change, adapt, and improve, the better the results will be. There are a few general things that people tweak most in the problems.

How many parents do you keep out of each generation? This effects a little bit of the variation in the generation. With the traveling salesman problem, the answers I am looking for are very straightforward and the best parent in one generation normally comes up as the best parent in the next generation too, so keeping the number of parents down can save time by not generating too many unused children.

How many children do you allow each parent to have? The answer is similar to the previous part. Because the problem has a very straightforward progression towards the correct answer. Ideally, there should be as few children as possible while still getting a steady improvement from one generation to the next. So this number also is better to be kept low.

One of the other tweaks that you do in evolutionary algorithms is in a whole segment of its own. How do you mutate the parent to get the child? This often asks questions like how you are mutating the parent to reach the result? How many times you mutate the parent in this way to reach your answer? And finally, how large is each mutation? In my program I simply have to reorder the cities of the solution. But how I did even that simple task changed throughout the project. I found that problems with local minimums were best solved by changing how the program mutates parents to get children.

At first, I started with only a single mutation type. I moved one city somewhere else in the order of cities in the solution. This resulted in a fairly good results for the line generator problems, but I also found that this could be slow when the program found a set of cities in an order but in the wrong spot. With the number of mutations not being high enough per generation,

and the fact that the odds of moving groups of cities anywhere in one generation being so low it had to be fixed. The simple solution was to instead of taking a single city at a time and moving it, I would move whole segments of the solution around. This resulted in a significant increase in getting the problem moved forward. A minor problem here was that also some segments were in a reverse order, which is problematic. The simple fix is to reverse the order of the group every now and then to get an improvement on that segment.

The last three things I did to the problem was to every generation add a new wildcard parent that did not come from the children. This was not as effective as expected. I introduced this after noticing that on a very rare occasion on smaller numbers of cities I would randomly guess the right city on the first try. This did not pay off as the number of cities in the problem grew.

I then set the number of mutations to make on the parent completely random, because every solution may require more or less mutations to change to what it needs improve. I finally prevented my program from allowing two parents in a generation to be the same, because if all the parents are all the same, my program is being redundant. I want to look at more solutions, not a bunch of the same.

Here is a list of the progression of mutations:

1) Move a single city to a new location
2) Move a subset of random length to a new location
3) Allow subsets to be reversed as they are moved
4) Add a wildcard parent each generation (this mutation was abandoned later)
5) Randomize the number of mutations
6) Remove duplicate parents

Results:

The number of permutations for a solution of “n” cities is n! (n factorial) which results in a non-polynomial time problem. I timed multiple solvers, which include the unpruned brute force algorithm, the pruned brute force algorithm, and the evolutionary solver. I did not time the greedy solver because I believed it was inferior and would too often find wrong answers to more complicated random generated cities sets. With my brute force solver times, I have demonstrated

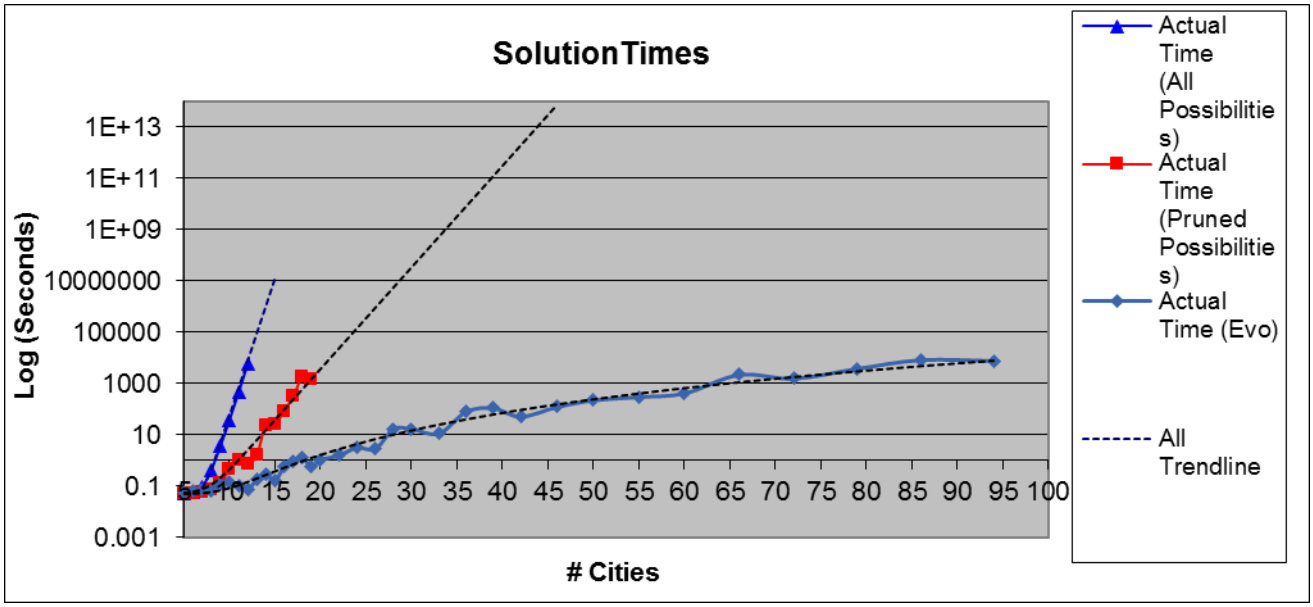
that the Traveling Salesman Problem is solvable in non-polynomial time. With my evolutionary solver I find a solution for the Traveling Salesman Problem demonstrating that the problem can often be “solved” in polynomial time. However, proving whether or not this “optimal” solution is the absolute best solution is quite another issue.

In order to find the results of my programs I need to time how long it takes them to run and find the correct answer. To do this I use a combination of the python subprocess module and the python timeit module. I time how long it takes for the subprocess call to my code to return. This allows me to set up a while loop to run through the a progressively larger number of cities. There is more discussed in appendix A. For the purposes of testing, I ensure that evolutionary solver both can, and does find the correct solution, by making the brute force solver save out the solution it finds. I feed this known correct answer to the evolutionary solver to check if it has found that known answer yet. If not, keep running. Originally I was only able to get my evolutionary solver to solve up to 13 cities at a time. This problem was mainly resulting from a well hidden bug in the code. In later testing, I still only tested up to 14 cities from the random generator because running the brute force solver to get the known answer would take too long for more than 14 cities. When the evolutionary solver solved more than 14 cities, I used problems from the line generator. The reason I can solve for problems from the line generator with the evolutionary solver and not have skewed time results is because the evolutionary solver is truly random, and often unpredictable. The same puzzle solved twice will take a different amount of time and could even find a different solution. This randomness means that a line does not show up differently to the evolutionary algorithm, just like any other shape of cities given to it. This is not true for the pruned brute force solver. It always looked at the first city compared to the second city, then the next city, and so on, given the same problem. This also means when doing the time tests I cannot set the pruned brute force solver, because I know that every time it will find the first line solution within a short amount of time, and then continually cut off large segments of the tree, resulting in an artificially shorter time. I didn’t bother timing the greedy solver because its initially predicted usability and quality of answer is significantly lower than both the brute force solver and the evolutionary algorithm solver. In later tests, when combined with an evolutionary solver, it proved to be more accurate than expected to be.

I know my solvers are reaching their solutions in non-polynomial time or polynomial time because of the shape of their lines on a graph. The equation of their trend lines also

demonstrate this. For example the pruned brute force equation has a trend line equal to $10^{0.3965x - 4.3843} + 0.05$ where x represents the number of cities. This has the x in the exponent making it a non-polynomial time solution. To the contrary, the evolutionary algorithm has a solution time trend line of $(x^{5.4697} / 8077602) + 0.05$ with x representing the number of cities. X is raised to the power of 5.4 making it a polynomial time solution.

Data and analysis:



The logarithmic chart shows solution times for an unpruned brute force, pruned brute force, evolutionary solver time, and predicted trend lines. The predicted trend lines were generated by hand after visually seeing which Excel trend line visually matched the actual line the best. The equation for the pruned brute force is $10^{0.3965x - 4.3843} + 0.05$. The equation for the evolutionary algorithm is $(x^{5.4697} / 8077602) + 0.05$. The graph above shows the difference between something solving in non-polynomial time, either the red or the blue lines, and something solved in polynomial time, the navy blue line. On a logarithmic chart, the equation 10^x is a straight line (i.e. a non-polynomial equation). It is also interesting to see how with the unpruned brute force solver (blue line) the times it takes is very consistent. It closely matches the trendline. The reason for this is that it always looks at every permutation. The pruned brute force solver and evolutionary solvers both fluctuate with the problem. What they prune is influenced by the random nature of the problem, or in the case of the evolutionary algorithm, its own random nature. Finally we have the evolutionary solver which can get the correct solution for most problems in a decent amount of time. The evolutionary solver is very clearly a random solver in that the randomness can make some puzzles take significantly longer than others even if they have the same number of cities.

A hybrid greedy/evolutionary algorithm was mentioned before in the paper, but couldn't accurately be timed here. The greedy algorithm can solve a line accurately in almost no time, and surprisingly could solve most random problems in almost no time due to the greedy solver outperforming its expectations. The greedy solver was initially expected to find a wrong answer for most the random sets of cities, and thus remained untested until very late in the project. This was a mistake as the greedy solver has proven itself more useful than expected, especially when used in a hybrid approach with the evolutionary algorithm. A circle and spokes problem is the best known answer test to demonstrate the short comings of the pure greedy solver. I therefore began using those problems as the benchmark for the hybrid solver. The problem with this is during the time of this research, a method of determining the known answer for every circle and spokes problem was not be found. Without a way to tell the evolutionary solvers that they found the correct answer, there wasn't a good found to tell them when to stop. Therefore, it is difficult to benchmark the solver.

Acknowledgements:

A special thanks to my dad, James Wesselkamper. This project was one that he added to the list of possible projects in the first place. He also helped with creating trend lines, and some debugging tips, especially early in the project. He also encouraged the use of a revision control system and set up a Perforce server to use. It saved me on more than once occasion.

A special thanks to my sponsoring teacher, Mr. Schum, who without I would have never found out about the supercomputing challenge and never would have entered.

A special thanks to my math teacher, Mr. Florence, whose math lessons helped with my project almost every day.

And finally, a special thanks to Jordan Medlock, a previous challenge winner, who gave me advice on what needed to be done to be successful.

Resources:

en.wikipedia.org/wiki/Evolutionary_algorithm

en.wikipedia.org/wiki/NP_complete

en.wikipedia.org/wiki/Travelling_salesman_problem

docs.python.org/2/

www.stackoverflow.com/

Comments about the code, and tools used:

First off, the choice of python was more for convenience than fast code. The team was originally more than one person and it was a common language with which we were the most comfortable. I knew that the results I got would still be proportional to someone else's even though mine runs in the slower python. The version control systems saved me so many times. It is an indispensable tool and one to never forget. Allowing you to bring back files from before you tried to do something that doesn't work is indispensable. Also notepad++ makes importing code into a document very easy with its exporter plugin built in. I did not include code from some of my scripted automated testers. These were quick scripts that would loop a desired number of times, and every time create a new set of cities n in size, count it and print the number to make sure it did it right, then run my python code using subprocess and timing it.

EVOLUTIOANRY ALGORYTHM SOLVER

```
import sys,pickle, pprint, random, signal, inspect, copy, os, traceback
from os import path as p
from operator import eq
import time

# Argument File Name Parsing
if len(sys.argv) < 2 or len(sys.argv) > 3:
            print("error, arg size invalid")
            print("asuming file.txt")
            inject = "file.txt"
else:
      inject = sys.argv[1]
# File Loading
output = open(p.split(p.abspath(inspect.getsourcefile(lambda _: None)))[0] +
"\\..\\Puzzles\\"+inject,"r")
c_cities = pickle.load(output)
output.close()

# c = constant, do not touch, at least mostly
# i = instance, for loops n stuff
# p = progress, for things you are working with in this area
# f = data used to output the final result

# tweaks
# how many children each one creates
# a large portion of these are/should be unused
c_parents  = 20
c_children = 10
p_ParentsList = []
lasttime = 0

# Switch to deside if its a line or a problem to find the solution from brute
force solvers false = line true = brute force verified
docompare = False
# code
# switch to output things
if docompare:
      output2 = open(p.split(p.abspath(inspect.getsourcefile(lambda _:
None)))[0] + "\\..\\Puzzles\\"+inject+"solution","r")
      f = output2.readline()
      output2.close()
      solution = int(float(f))

# generates the new parents for the beginning
def genNewParent():
      p_cities = []
      while len(p_cities) < len(c_cities):
            i_city = random.randint(0,len(c_cities)-1)
            if not (i_city in p_cities):
                  p_cities.append(i_city)
      return p_cities
```

```
def mutateParent(p_parent):
	if len(p_parent.getCities()) < p_parent.getChanges():
		print "not designed to run more changes than there are cities"
		exit(1)

	p_Child = []
	while len(p_Child) < p_parent.getChanges():
		# choose what city to mess up
		i_Change = chooseCity(p_Child, 0, len(p_parent.getCities())-1)
		# seperate segmetn from the point to mess up
		newGroup = removeGroup(p_parent, i_Change, 0,
len(p_parent.getCities())-1,0.7, 0.7)
		# reverse if needed
		if random.randint(0,10) < 8:
			newGroup.reverse()
		# insert into a new position
		placeGroup(p_parent, newGroup, 0, len(p_parent.getCities())-1)
	return p_parent

# This is a new method to do the chooseing city stage in muteate parent
def chooseCity(previouslyChosen,minSize,maxSize):
	while True:
		choice = random.randint(minSize,maxSize)
		if choice in previouslyChosen:
			continue
		else:
			previouslyChosen.append(choice)
			return choice

# Also for mutate parent, but removes parts instead
def removeGroup(p_parent, startPoint, minSize, maxSize,
chanceOfStartPointLater,chanceOfEndDistance):
	flip = False
	if random.random() > chanceOfStartPointLater:
		flip = True
		if startPoint-maxSize > 0:
			maxSize = startPoint
	else:
		if maxSize <= len(p_parent.getCities())-startPoint:
			maxSize = len(p_parent.getCities())-startPoint-1
	endDistance = random.randint(minSize,maxSize)
	if not flip and random.random() < chanceOfEndDistance:
		endDistance = 0

	if endDistance >= len(p_parent.getCities()):
		endDistance = len(p_parent.getCities())-1
	citySegment = []
	startPointCopy = startPoint
	while (flip == False and startPoint < endDistance) or (flip==True and
startPoint > endDistance):
		#print "======="
		#print "flip       = " + str(flip)
		#print "startPoint = " + str(startPoint)
		#print "endPoint   = " + str(endDistance)
		#print "lenCities  = " + str(len(p_parent.getCities()))

		if flip:
```

```
                  citySegment.append(p_parent.getCities().pop(startPoint))
                  startPoint -= 1
            else:

      citySegment.append(p_parent.getCities().pop(startPointCopy))
                  startPoint += 1
      return citySegment

# Also for mutate parent, but places the parts back
def placeGroup(p_parent, part, minPlace, maxPlace):
      location = random.randint(minPlace,maxPlace)
      for item in part:
            p_parent.getCities().insert(location,item)
            location += 1


# gets the distance from a list of cities
def getDistance(p_list):
      distance = 0
      city_index = 0
      while (city_index < len(p_list) -1):
            city_a = p_list[city_index]
            city_b = p_list[city_index+1]
            distance += c_cities[city_a][city_b]
            city_index += 1
      return distance

# this runs a generation, creates children, and kills them off
def generation(p_ParentsList):
      return optimizeGroup(createChildren(p_ParentsList))

def createChildren(p_ParentsList):
      # creates children
      p_ParentsListDupe = list(p_ParentsList)
      while len(p_ParentsListDupe) < (c_children+1)*len(p_ParentsList):
            for i_parent in p_ParentsList:
                  ParentChild = i_parent.clone(i_parent)
                  Child = mutateParent(ParentChild)
                  p_ParentsListDupe.append(Child)
      return p_ParentsListDupe

def minuteTime():
      numstring = str(time.time()/100)
      return int(float(numstring[:numstring.find('.')+3]))

def optimizeGroup(p_overallGrouping):
      # Parents Not yet integrated into the Parents list will be called
children
      global lasttime
      # goes through and sorts out if this is shorter
      p_optimizedParentsList = []
      for i_child in p_overallGrouping:
            keepThisSolution = False
            duplicate = False
            if len(p_optimizedParentsList) < c_parents:
                  i_child.mutateChanges()
                  p_optimizedParentsList.append(i_child)
```

```
            else:
                  lastLargest = -1
                  for item in p_optimizedParentsList:
                        if i_child.getDistance() < item.getDistance():
                              keepThisSolution = True
                        if  not False in map(eq, i_child.getCities(),
item.getCities()):
                              duplicate = True
                  if keepThisSolution and not duplicate:
                        marker = 0
                        while marker < c_parents:
                              if lastLargest == -1 or
p_optimizedParentsList[marker].getDistance() >
p_optimizedParentsList[lastLargest].getDistance() :
                                    lastLargest = marker
                              marker = marker+1
                        p_optimizedParentsList[lastLargest] = i_child
                        lasttime = minuteTime()
      return p_optimizedParentsList

#================= PARENT CLASS =================#
#================================================#
class parent:
      # vars
      maxChangesToChange = 3
      changesToMake = 3
      changesToMakeMin = 1
      groupingChangesMin, groupingChangesMax = 0, 0
      distance = 0
      cities = []
      def __init__(self):
            self.cities = genNewParent()
      def clone(self,p_parent):
            return copy.deepcopy(p_parent)
      def mutateChanges(self):
            while True:
                  x = random.randint(-3,2)
                  if self.changesToMake+x <= self.changesToMakeMin or
self.changesToMake+x >= len(self.cities):
                        pass
                  else:
                        self.changesToMake = self.changesToMake+x
                        break
      def setCities(self,ccities):
            self.cities=ccities
            self.updateDistance()
      def updateDistance(self):
            self.distance = getDistance(self.cities)
      def getCities(self):
            return self.cities
      def getDistance(self):
            self.updateDistance()
            return self.distance
      def getChanges(self):
            return self.changesToMake
      def setChanges(self, changes):
            self.changesToMake = changes
```

```
# ================= MAIN =================#
# ========================================#
def main():
      global p_ParentsList
      global parent
      global lasttime
      lastime = minuteTime()
      lineSolution = range(0,len(c_cities))
      reverseLineSolution = list(lineSolution)
      reverseLineSolution.reverse()
      iterations = 0
      while iterations <= 10:
            if p_ParentsList == []:
                  while len(p_ParentsList) < c_parents:
                        temp = parent()
                        p_ParentsList.append(temp)
                        del temp

            for i in p_ParentsList:
                  if i == None:
                        print "error i is none"
                        exit(1)
            # Add a wild card. Then run a generation
            #WildCard = parent()
            #p_ParentsList.append(WildCard)
            p_ParentsList = generation(p_ParentsList)
            if True: #print if true
                  i = 0
                  while i < len(p_ParentsList):
                        print str(i) + " " +
str(p_ParentsList[i].getCities()) + " " + str(p_ParentsList[i].getDistance())
+ " " + str(p_ParentsList[i].getChanges())
                        i+=1

                  print "-----------------------------------"
            # this code relates to finding a solution
            if docompare:
                  distances = []
                  for p_parent in p_ParentsList:
                        distances.append(p_parent.getDistance())
                  if solution in distances:
                        print distances
                        break
            else:
                  for p_parent in p_ParentsList:
                        if not False in map(eq, lineSolution,
p_parent.getCities()) or not False in map(eq, reverseLineSolution,
p_parent.getCities()):
                              signal_handler(None,None)
            if lastime+len(c_cities)<minuteTime():
                  print "i give up"
                  break
            iterations += 1
      signal_handler(None,None)
```

```
# ===ignore===
# This code is for when the program exits. Whether its its ctrl+c or by
timing out/solving the puzzle
def signal_handler(signal, frame):
      global output
      global p_ParentsList
      global parent
      print "\n"
      output3 = open(p.split(p.abspath(inspect.getsourcefile(lambda _:
None)))[0] + "\\..\\test\\"+inject,"w")
      for pparent in p_ParentsList:
            print str(pparent.getCities())+ " " + str(pparent.getDistance())
            for n in pparent.getCities():
                  output3.write(str(n))
                  output3.write("\t")
            output3.write("\n")
      print "==end?=="
      if docompare:
            output2 = open(p.split(p.abspath(inspect.getsourcefile(lambda _:
None)))[0] + "\\..\\Puzzles\\"+inject+"solution","r")
            x = output2.readline()
            y = output2.readline()
            print y + " " + x
      # output3 = open(p.split(p.abspath(inspect.getsourcefile(lambda _:
None)))[0] + "\\..\\Puzzles\\"+inject+"log","w")
      # pickle.dump(c_Log,output3)
      output3.close
      if not signal == None:
            print " " + 1
      exit(0)
signal.signal(signal.SIGINT, signal_handler)
main()
```

GREEDY SEEDED EVOLUTIOANRY ALGORYTHM SOLVER

```
import sys,pickle, pprint, random, signal, inspect, copy, os, traceback
from os import path as p
from operator import eq
import time

# Argument File Name Parsing
if len(sys.argv) < 2 or len(sys.argv) > 3:
            print("error, arg size invalid")
            print("asuming file.txt")
            inject = "file.txt"
else:
      inject = sys.argv[1]
# File Loading
output = open(p.split(p.abspath(inspect.getsourcefile(lambda _: None)))[0] +
"\\..\\Puzzles\\"+inject,"r")
c_cities = pickle.load(output)
output.close()

# c = constant, do not touch, at least mostly
# i = instance, for loops n stuff
# p = progress, for things you are working with in this area
# f = data used to output the final result

# tweaks
# how many children each one creates
# a large portion of these are/should be unused
c_parents  = 10
c_children = 10
p_ParentsList = []
lasttime = 0

# Switch to deside if its a line or a problem to find the solution from brute
force solvers false = line true = brute force verified
docompare = False
# code
# switch to output things
if docompare:
      output2 = open(p.split(p.abspath(inspect.getsourcefile(lambda _:
None)))[0] + "\\..\\Puzzles\\"+inject+"solution","r")
      f = output2.readline()
      output2.close()
      solution = int(float(f))

# generates the new parents for the beginning
def genNewParents():
      parentsList = []
      while len(parentsList) < len(c_cities):
            workingParent = []
            workingParent.append(len(parentsList))
            while len(workingParent) < len(c_cities):

      workingParent.append(findNextClosestUnusedCity(workingParent[-
1],workingParent))
                  #print "loop 1"
```

```
            parentsList.append(parent(workingParent))
            #print "loop 2"
      print "generated parents"
      return parentsList

def findNextClosestUnusedCity(currentCity,usedCities):
      i = 0
      target = -1
      while i < len(c_cities):
            if not i == currentCity and (target == -1 or
c_cities[currentCity][i] < c_cities[currentCity][target]) and (not i in
usedCities):
                  target = i
            i+=1
      return target

def mutateParent(p_parent):
      if len(p_parent.getCities()) < p_parent.getChanges():
            print "not designed to run more changes than there are cities"
            exit(1)

      p_Child = []
      while len(p_Child) < p_parent.getChanges():
            # choose what city to mess up
            i_Change = chooseCity(p_Child, 0, len(p_parent.getCities())-1)
            # seperate segmetn from the point to mess up
            newGroup = removeGroup(p_parent, i_Change, 0,
len(p_parent.getCities())-1,0.7, 0.7)
            # reverse if needed
            if random.randint(0,10) < 8:
                  newGroup.reverse()
            # insert into a new position
            placeGroup(p_parent, newGroup, 0, len(p_parent.getCities())-1)
      return p_parent

# This is a new method to do the chooseing city stage in muteate parent
def chooseCity(previouslyChosen,minSize,maxSize):
      while True:
            choice = random.randint(minSize,maxSize)
            if choice in previouslyChosen:
                  continue
            else:
                  previouslyChosen.append(choice)
                  return choice

# Also for mutate parent, but removes parts instead
def removeGroup(p_parent, startPoint, minSize, maxSize,
chanceOfStartPointLater,chanceOfEndDistance):
      flip = False
      if random.random() > chanceOfStartPointLater:
            flip = True
            if startPoint-maxSize > 0:
                  maxSize = startPoint
      else:
            if maxSize <= len(p_parent.getCities())-startPoint:
                  maxSize = len(p_parent.getCities())-startPoint-1
      endDistance = random.randint(minSize,maxSize)
```

```
	if not flip and random.random() < chanceOfEndDistance:
		endDistance = 0

	if endDistance >= len(p_parent.getCities()):
		endDistance = len(p_parent.getCities())-1
	citySegment = []
	startPointCopy = startPoint
	while (flip == False and startPoint < endDistance) or (flip==True and
startPoint > endDistance):
		#print "======="
		#print "flip       = " + str(flip)
		#print "startPoint = " + str(startPoint)
		#print "endPoint   = " + str(endDistance)
		#print "lenCities  = " + str(len(p_parent.getCities()))

		if flip:
			citySegment.append(p_parent.getCities().pop(startPoint))
			startPoint -= 1
		else:

	citySegment.append(p_parent.getCities().pop(startPointCopy))
			startPoint += 1
	return citySegment

# Also for mutate parent, but places the parts back
def placeGroup(p_parent, part, minPlace, maxPlace):
	location = random.randint(minPlace,maxPlace)
	for item in part:
		p_parent.getCities().insert(location,item)
		location += 1


# gets the distance from a list of cities
def getDistance(p_list):
	distance = 0
	city_index = 0
	while (city_index < len(p_list) -1):
		city_a = p_list[city_index]
		city_b = p_list[city_index+1]
		distance += c_cities[city_a][city_b]
		city_index += 1
	return distance

# this runs a generation, creates children, and kills them off
def generation(p_ParentsList):
	return optimizeGroup(createChildren(p_ParentsList))

def createChildren(p_ParentsList):
	# creates children
	p_ParentsListDupe = list(p_ParentsList)
	while len(p_ParentsListDupe) < (c_children+1)*len(p_ParentsList):
		for i_parent in p_ParentsList:
			ParentChild = i_parent.clone(i_parent)
			Child = mutateParent(ParentChild)
			p_ParentsListDupe.append(Child)
	return p_ParentsListDupe
```

```
def minuteTime():
	numstring = str(time.time()/100)
	return int(float(numstring[:numstring.find('.')+3]))

def optimizeGroup(p_overallGrouping):
	# Parents Not yet integrated into the Parents list will be called
children
	global lasttime
	# goes through and sorts out if this is shorter
	p_optimizedParentsList = []
	for i_child in p_overallGrouping:
		keepThisSolution = False
		duplicate = False
		if len(p_optimizedParentsList) < c_parents:
			i_child.mutateChanges()
			p_optimizedParentsList.append(i_child)
		else:
			lastLargest = -1
			for item in p_optimizedParentsList:
				if i_child.getDistance() < item.getDistance():
					keepThisSolution = True
				if  not False in map(eq, i_child.getCities(),
item.getCities()):
					duplicate = True
			if keepThisSolution and not duplicate:
				marker = 0
				while marker < c_parents:
					if lastLargest == -1 or
p_optimizedParentsList[marker].getDistance() >
p_optimizedParentsList[lastLargest].getDistance() :
						lastLargest = marker
					marker = marker+1
				p_optimizedParentsList[lastLargest] = i_child
				lasttime = minuteTime()
	return p_optimizedParentsList

#================= PARENT CLASS =================#
#================================================#
class parent:
	# vars
	maxChangesToChange = 3
	changesToMake = 3
	changesToMakeMin = 1
	groupingChangesMin, groupingChangesMax = 0, 0
	distance = 0
	cities = []
	def __init__(self, cityInput):
		self.cities = cityInput
	def clone(self,p_parent):
		return copy.deepcopy(p_parent)
	def mutateChanges(self):
		while True:
			x = random.randint(-3,2)
			if self.changesToMake+x <= self.changesToMakeMin or
self.changesToMake+x >= len(self.cities):
				pass
			else:
```

```
                        self.changesToMake = self.changesToMake+x
                        break
      def setCities(self,ccities):
            self.cities=ccities
            self.updateDistance()
      def updateDistance(self):
            self.distance = getDistance(self.cities)
      def getCities(self):
            return self.cities
      def getDistance(self):
            self.updateDistance()
            return self.distance
      def getChanges(self):
            return self.changesToMake
      def setChanges(self, changes):
            self.changesToMake = changes
# ================= MAIN =================#
# ========================================#
def main():
      global p_ParentsList
      global parent
      global lasttime
      lastime = minuteTime()
      lineSolution = range(0,len(c_cities))
      reverseLineSolution = list(lineSolution)
      reverseLineSolution.reverse()
      iterations = 0
      while iterations <= 10:
            if p_ParentsList == []:
                  p_ParentsList = genNewParents()
                  print "GENERATED NEW PARENTS"

            for i in p_ParentsList:
                  if i == None:
                        print "error i is none"
                        exit(1)
            # Add a wild card. Then run a generation
            #WildCard = parent()
            #p_ParentsList.append(WildCard)
            p_ParentsList = generation(p_ParentsList)
            if True: #print if true
                  i = 0
                  while i < len(p_ParentsList):
                        print str(i) + " " +
str(p_ParentsList[i].getCities()) + " " + str(p_ParentsList[i].getDistance())
+ " " + str(p_ParentsList[i].getChanges())
                        i+=1

                  print "-----------------------------------"
            # this code relates to finding a solution
            if docompare:
                  distances = []
                  for p_parent in p_ParentsList:
                        distances.append(p_parent.getDistance())
                  if solution in distances:
                        print distances
                        break
```

```
            else:
                  for p_parent in p_ParentsList:
                        if not False in map(eq, lineSolution,
p_parent.getCities()) or not False in map(eq, reverseLineSolution,
p_parent.getCities()):
                              signal_handler(None,None)
            if lastime+len(c_cities)<minuteTime():
                  print "i give up"
                  break
            iterations += 1
      signal_handler(None,None)
# ===ignore===
# This code is for when the program exits. Whether its its ctrl+c or by
timing out/solving the puzzle
def signal_handler(signal, frame):
      global output
      global p_ParentsList
      global parent
      print "\n"
      output3 = open(p.split(p.abspath(inspect.getsourcefile(lambda _:
None)))[0] + "\\..\\test\\"+inject,"w")
      for pparent in p_ParentsList:
            print str(pparent.getCities())+ " " + str(pparent.getDistance())
            for n in pparent.getCities():
                  output3.write(str(n))
                  output3.write("\t")
            output3.write("\n")
      print "==end?=="
      if docompare:
            output2 = open(p.split(p.abspath(inspect.getsourcefile(lambda _:
None)))[0] + "\\..\\Puzzles\\"+inject+"solution","r")
            x = output2.readline()
            y = output2.readline()
            print y + " " + x
      # output3 = open(p.split(p.abspath(inspect.getsourcefile(lambda _:
None)))[0] + "\\..\\Puzzles\\"+inject+"log","w")
      # pickle.dump(c_Log,output3)
      output3.close
      if not signal == None:
            print " " + 1
      exit(0)
signal.signal(signal.SIGINT, signal_handler)
main()
```

GREEDY SOLVER

```
import sys, pickle, pprint
#input into the program
if len(sys.argv) < 2 or len(sys.argv) > 3:
            print("error, arg size invalid")
            print("asuming file.txt")
            inject = "file.txt"
else:
      inject = sys.argv[1]
output = open("../Puzzles/"+inject,"r")
cities = pickle.load(output)
output.close()

#program itself
solutionset = []
distances = []
while len(solutionset) < len(cities):
      firstcity = len(solutionset)
      solutions = []

      solutions.append(firstcity)
      distances.append(0)
      while len(solutions) < len(cities)-1:
            closestcity = -1
            row = 0
            while row < len(cities):
                  #print str(lastcity) + " " + str(row) + " " +
str(lastshortest) + " " + str(cities[lastcity][row]) + " " +
str(cities[lastcity][lastshortest])
                  #print "LAST = " + str(solutions[-1]) + " CLOSEST = " +
str(closestcity) + " COMPARE TO " + str(row) + " WHICH IS " +
str(cities[solutions[-1]][row])
                  if ((closestcity == -1) or (cities[solutions[-
1]][closestcity] > cities[solutions[-1]][row])) and not (row in solutions):
                        closestcity = row
                  row += 1
            distances[-1] = distances[-1]+cities[solutions[-1]][closestcity]
            solutions.append(closestcity)
            #print "Solutions is now " + str(solutions)
      solutionset.append(solutions)
pprint.pprint(solutionset)
#print distances
least = -1
i = 0
while i < len(distances):
      if least == -1:
            least = i
      elif distances[i] < distances[least]:
            least = i
      i+=1
#print least
#print solutionset[least]
```

BRUTE FORCE SOLVER

```
import sys, pickle, pprint, inspect
from os import path as p
#input into the program
if len(sys.argv) < 2 or len(sys.argv) > 3:
            print("error, arg size invalid")
            print("asuming file.txt")
            inject = "file.txt"
else:
      inject = sys.argv[1]
output = open(p.split(p.abspath(inspect.getsourcefile(lambda _: None)))[0] +
"\\..\\Puzzles\\"+inject,"r")
c_cities = pickle.load(output)
output.close()
#startprogram yay
# c = constant, do not touch, at least mostly
# i = instance, for loops n stuff
# p = progress, for things you are working with in this area
# f = data used to output the final result
i_cityWidth = 0
f_distance = -1
f_cities = []


# p_used is progress for i am using it in progress in this recursefunction
def recurseFunction(p_thisCity, p_used, p_distance):
      p_used.append(p_thisCity)
      global f_distance
      global f_cities

      if len(p_used) >= len(c_cities):
            if f_distance == -1 or f_distance > p_distance:
                  f_distance = p_distance
                  f_cities = p_used
            return
      i_cityCheck = 0
      while i_cityCheck < len(c_cities):
            i_checkDistance = p_distance+c_cities[p_thisCity][i_cityCheck]
            if i_cityCheck in p_used:
                  i_cityCheck = i_cityCheck + 1
                  continue
            elif (not i_checkDistance >= f_distance) or (f_distance  == -1):
                  recurseFunction(i_cityCheck, list(p_used), i_checkDistance)
            i_cityCheck = i_cityCheck + 1

while i_cityWidth < len(c_cities):
      print i_cityWidth
      recurseFunction(i_cityWidth, [], 0)
      i_cityWidth = i_cityWidth+1
#print "success?"
print f_distance
print f_cities
```

```
output2 = open("../Puzzles/"+inject+"solution","w")
output2.write(str(f_distance)+"\n"+str(f_cities))
output2.close()
```

NORMAL GENERATOR FILE

```
"""This program is intended to generate the files that the solver will use to
find the shortest between several cities...
--features--
-options for number of cities
-options for shapes of cities(default of 2, line and non line)
-options for filename output
-options for outputing many files
-uses pickle to dump files to load later
work in progress file
"""
from pprint   import pprint
from optparse import OptionParser
from os import path as p
import re, random, pickle, sys, math, os, inspect

class gb():
      range = "1 1000"
      size = 7
      file = "file.txt"
      type = "random"

# generates cities in random places
def randomgenerate():
      size=options.size
      print("\nrandom Generation")
      seq = parserange()
      min, max = seq[0], seq[1]
      overall, row = 0, 0
      allcities,rowcities = [],[]
      print size
      while overall < size:
            rowcities=[]
            while row < size:
                  if overall == row:
                        rowcities.append(0)
                  elif overall>row:
                        rowcities.append(allcities[row][overall])
                  else:
                        rowcities.append(random.randrange(min,max))
                  row+=1
            allcities.append(rowcities)
            overall+=1
            row=0
      return allcities
def randomGenerateWithPoints():
      size = options.size
      group = []
      x = 0
      seq = parserange()
      min, max = seq[0], seq[1]
      while x < size:

      group.append(((random.randint(min,max)),(random.randint(min,max))))
```

```
            x+=1
      output = open( p.split(p.abspath(inspect.getsourcefile(lambda _:
None)))[0] + "\\..\\Puzzles\\points"+options.file,'w')
      pickle.dump(group,output)
      output.close()
      allcities = []
      for item in group:
            build = []
            for something in group:
                  distance = int(math.sqrt(math.pow((something[0]-
item[0]),2)+math.pow((something[1]-something[1]),2)))
                  build.append(distance)
            allcities.append(build)
      return allcities
      """
      overall, row = 0, 0
      allcities,rowcities = [],[]
      print size
      while overall < size:
            rowcities=[]
            while row < size:
                  if overall == row:
                        rowcities.append(0)
                  elif overall>row:
                        rowcities.append(allcities[row][overall])
                  else:
                        rowcities.append(random.randrange(min,max))
                  row+=1
            allcities.append(rowcities)
            overall+=1
            row=0
      return allcities
      print("\nrandom generation with pointed grid")
      """
# generates a straight line of cities, does nothing now
def linegenerate():
      size=options.size
      seq = parserange()
      min, max = seq[0],seq[1]
      overall, row = 0,0
      allcities, rowcities=[],[]
      #prepare segments
      difference= max-min
      segment = difference/size
      while overall < size:
            rowcities=[]
            while row < size:
                  cur=0
                  if overall == row:
                        rowcities.append(0)
                  elif overall>row:
                        rowcities.append(allcities[row][overall])
                  else:
                        rowcities.append(int(segment*math.fabs(overall-row)))
                  row+=1
            allcities.append(rowcities)
            overall+=1
```

```
            row=0
      return allcities

#generates the cities in circles/rings, intended to be hard to solve
def circlegenerate():
      print("WOOPS: Circle generation required too many changes, its in its
own project file.")
      exit(1)
def parserange():
      ranges = re.match("(\w+) (\w+)", options.range)
      return int(ranges.group(1)),int(ranges.group(2))

# ----options code----
parser = OptionParser()
parser.add_option("-s", "--size", dest="size", type="int",
                              help="the number of cities to have")
parser.add_option("-r", "--range", dest="range", type="string",
                              help="the range of distances for a city to
have, must have quotes")
parser.add_option("-f", "--file", dest="file", type="string",
                              help="the file to use, add .txt yourself")
parser.add_option("-t", "--type", dest="type", type="string",
                              help="the type of creation to use, possible
choices are line, and random for now\ndoes change usage of -r")
parser.add_option("-d", "--defaults", dest="defaults", type="string",
                              help="shows the default values possible for
this program, have ANYTHING in the space after example: \"-d 1\"")
(options, args) = parser.parse_args()

if not options.defaults is None:
      print("====DEFAULTS====")
      print("range: "+str(gb.range))
      print("size: "+str(gb.size))
      print("type: "+str(gb.type))
      print("file: "+str(gb.file))
      exit(0)

if options.range is None:
      options.range = gb.range
if options.size is None:
      options.size = gb.size
if options.type is None:
      options.type = gb.type
if options.file is None:
      options.file = gb.file
# and now we call the generation methods
# add new generation methods using a new if ==
if options.type.upper() == "RANDOM":
      cities = randomgenerate()
elif options.type.upper() == "LINE":
      cities = linegenerate()
elif options.type.upper() == "CIRCLE":
      cities = circlegenerate()
elif options.type.upper() == "POINT":
      cities = randomGenerateWithPoints()
elif options.type.upper() == "POINTS":
      cities = randomGenerateWithPoints()
```

```
# finally write files here
output = open(p.split(p.abspath(inspect.getsourcefile(lambda _: None)))[0] +
"\\..\\Puzzles\\"+options.file,'w')
pickle.dump(cities,output)
output.close()
print("done and success")
```

CIRCLE AND SPOKES GENERATOR FILE

```
"""This program is intended to generate the files that the solver will use to
find the shortest between several cities... this is also the circle generator
only
do to the different types of options this got its own file
work in progress file
"""
#Has a high chance of being merged with other code
from pprint   import pprint
from optparse import OptionParser
import re, random, pickle, sys, math
#try:
#     from globalvars import circlegeneratorglobal as gb
#except(ImportError):
#     print("wai u no have everything!")
#     print("\nglobalvars.py")
#     exit(1)

parser = OptionParser()
parser.add_option( "-i", "--ring1", dest="ring1", type="int",
                                 help="the number of cities in ring one")
parser.add_option( "-o", "--ring2", dest="ring2", type="int",
                                 help="the number of cities in ring two")
parser.add_option("-f", "--file", dest="file", type="string",
                                 help="the file to use, add .txt yourself")
parser.add_option("-d", "--defaults", dest="defaults", type="string",
                                 help="shows the default values possible for
this program, have ANYTHING in the space after example: \"-d 1\"")
(options, args) = parser.parse_args()
#if not options.defaults is None:
#     print("====DEFAULTS====")
#     print("cities ring one: "+str(gb.ring1))
#     print("cities ring two: "+str(gb.ring2))
#     print("file: "+str(gb.file))
#     exit(0)
if options.ring1 is None:
      options.ring1 = gb.ring1
if options.ring2 is None:
      options.ring2 = gb.ring2
if options.file is None:
      options.file = gb.file
#actually do things here because yes
cities = []
allcities = []
ring1scale=1000
ring2scale=2000
i=1
while i < options.ring1+1:
      x=ring1scale*math.sin((2*math.pi*i)/options.ring1)
      y=ring1scale*math.cos((2*math.pi*i)/options.ring1)
      cities.append((x,y))
      i+=1
i=1
while i < options.ring2+1:
```

```
      x=ring2scale*math.sin((2*math.pi*i)/options.ring2)
      y=ring2scale*math.cos((2*math.pi*i)/options.ring2)
      cities.append((x,y))
      i+=1
x = 0
y = 0
overall, row = 0,0
while overall < options.ring2+options.ring1:
      rowcities=[]
      while row < options.ring2+options.ring1:
            if overall == row:
                  rowcities.append(0)
            elif overall > row:
                  rowcities.append(allcities[row][overall])
            else:
                  rowcities.append(int(math.sqrt(math.pow(cities[row][0]-
cities[overall][0],2)+math.pow(cities[row][1]-cities[overall][1],2))))
            row+=1
      allcities.append(rowcities)
      overall+=1
      row=0
# finally write files here
output = open("../Puzzles/"+options.file,'w')
pickle.dump(allcities,output)
output.close()
print("done
```