Shara Orcutt
Vinh Dong
Sam Chew

Travelling Salesperson
Group Project

March 16, 2018
CS325 ProjGrp14

Project Description

The purpose of this project is to complete a Travelling Salesman Problem by researching applicable algorithms, selecting an implementation method, and then implement and evaluate that algorithm against both known, and unknown, minimum distances for given tours.

The following were givens for the implementation:

- Each city (c) has an x- and y-coordinate
- DIstance between two cities = SQRT((x2 - x1)^2 + (y2 - y1))
- Must research a minimum of three algorithms
- Accept problem instances on the command line
- Name the output file as the input file's name with .tour appended (for example input tsp_example_1.txt will output tsp_example_1.txt.tour)
- Compile/Execute correctly and without debugging on flip2.engr.oreognstate.edu according to specifications and any documentation you provide.
- Record results for the three example input files, with no time limit
- Record results for the competition inputs (three min and no time limit).

Summary of Branch & Bound Algorithms

The Branch & Bound Algorithms attempt to identify different TSP tours while evaluating the lowest possible tour value. By checking the lowest possible tour value, the B&B Algorithms prioritize tours of minimum bounded values and may avoid evaluating certain tours entirely. The result of the B&B Algorithms will yield an optimal solution, but may be susceptible to long worst case running times.

Algorithm Description

Every B&B Algorithms is characterized by three aspects: Branching Strategy, Bounding Strategy, and Node Selection Strategy. The Branching Strategy guides the creation of a decision tree, which identifies the potential tour permutations and represents the relative changes between evaluation cycles. The Bounding Strategy is a method of calculating a lower bound value of a potential tour, which is used to discriminate between the tours during the evaluation cycle. The Node Selection Strategy identifies the next viable node of the decision tree for evaluation and subsequently the next potential tour for consideration. The three aspects of B&B Algorithms work in conjunction to avoid

subtours and early termination, to 'prune' the evaluation tree to avoid unnecessary calculations, and to focus evaluation on tour value minima to optimize the results.

For the Rudimentary B&B Algorithms variation, the Branching Method selects all of the valid potential edges from a given node as unique tour evaluation branches. A potential edge is valid if it is not previously used in the current tour and if it does not connect to a repeat node until the tour is complete. The evaluation of valid edges represents the cost permutations of the routes in and out of a given city. The Rudimentary Bounding Strategy is the sum total of the previously selected edges to a given node and the smallest potential edges extending from the given node. This sum establishes the Lower Bound of any potential tour value that may result from the given node, which will only increase as additional edges are added. The Rudimentary Node Selection selects the the minimum value from the set of decision tree leaves as the next node for branching and evaluation. Overall, the Rudimentary B&B Algorithm incrementally assembles possible tours in the decision tree similar to a Breadth First Search algorithm.

For the Hungarian B&B Algorithm variation, the Node Selection Strategy still selects the next node/tour from a decision tree for evaluation according to the minimum Lower Bound of the tour value. However, the Hungarian variation uses an assignment algorithm as a relaxed version of the TSP problem, which uses 'reduced' edge weight matrices to create potential complete tours for evaluation. It is possible, but unlikely, that the assignment algorithm may successfully identify a valid tour with the optimal lowest bound. However, more frequently, an assignment tour will not be valid (or contains a set of subtours) such that a new potential tour must be generated and evaluated. The new tours are generated each time with an additional constraint condition, which would block the previously invalid subtours. Hence, the Bounding Strategy establishes the Lower Bound value by summing the edges of the potential tour of the assignment algorithm. Additionally, the Branching Strategy creates branches of the decision tree by identifying the inclusion or exclusion of the lowest edge that could break previous subtours. Overall, the Hungarian B&B Algorithm considers entire tours, while incrementally introducing constraints until a valid tour of minimum value is found.

Process/Pseudocode

1. Build a symmetric matrix of city to city distances from the input file (using infinity for city to self values)
2. For each row, select a minimum value
3. Reduce the row elements by the respective row minimum value
4. For each column, select a minimum value
5. Reduce the column elements by the respective column minimum value
6. Initiate the 'assignment procedure'
    6.1. For each row with a single viable '0' value, store the matrix position assignment (invalidate remaining '0' values from the column of a row assigned '0')

6.2. For each column with a single viable '0' value, store the matrix position assignment (invalidate remaining '0' values from the row of a column assigned '0')

7. If the quantity of matrix position assignments is less than the quantity of cities, then initiate the 'ticking procedure'
   7.1. Record each row without an assigned '0' value position
   7.2. For each row recorded in 8.1, record each column of a '0' value element.
   7.3. For each of the recorded columns of 8.2, if there is an assigned '0' value position, record each row of the assigned '0' value position
   7.4. Only using recorded rows from 8.1 & 8.3 and while ignoring recorded columns from 8.2, find the minimum value of available elements, which will be referred to as theta
8. Recalculate the matrix according to the following calculations:
   8.1. If a matrix element position is within the recorded rows from 8.1 & 8.3 and not the columns from 8.2, then subtract theta
   8.2. If a matrix element position is within the columns from 8.2 and not from 8.1 & 8.3, then add the theta
   8.3. Remaining matrix elements will remain unchanged
9. Initiate the assignment procedure of 6
10. If there are any rows without a '0' value assignment due to multiple '0' value options, commit an arbitrary assignment and repeat assignment procedure with remaining '0' value elements
11. The result should yield an assignment list according to the matrix position indices
12. If the assignment list is a feasible TSP solution, then return with optimal solution
13. If the assignment is not feasible (due to subtours), then create a subtour matrix to calculate a weight to add to the Lower Bound of the current assignment tour.
14. Define branches in the decision tree to a) force inclusion of an edge of the smallest subtour (change the matrix position value to infinity) and b) force exclusion of of an edge of the smallest subtour.
15. Repeat assignment procedure and evaluate resulting lower bound until a feasible solution is found

Running Time

Considering that the Branch & Bound Algorithm relies upon a short-cutting technique, the worst case running time is technically exponential and do not guarantee polynomial time. However, in practice, performance is often better than the theoretical worst case because successful branch elimination will occur.

Summary of Genetic Algorithm

The Genetic Algorithm is an iterative process to find the optimal solution to TSP. The initial city is set up and each distance between each city is evaluated. The distance between all the cities would then be found and added up to find the total distance of the tour. The tour would then be designated the parent tour and a subset of its cities are transferred to a new tour that becomes a child of the parent. The process is called a crossover and can pull from another parent to fill in the rest of the subset for the child. There are other methods of doing the crossover process, which can yield better results. Another constraint would be to add mutation, where a random position in the child tour would swap with another random position to increase variability. The tour of each "generation" or iteration is then kept in a population class. We can also keep the best tour from the previous generation and put it into the new "generation", which is a method known as elitism. We would then start the process over again, now with the child becoming the new parent. After 200 iterations or "generations: are done, we then find the iteration with the shortest distance and get the ordered list of cities from that tour and write it to a .txt.tour file.

The genetic algorithm can be broken down to the following steps or pseudo code:

1. [Start] Create a city class and read the city id, x and y location into a city object.
   Create a vector to hold the list of cities.
   The class should have getter functions for city: getvalue, getx and gety
   Function distance(city, city){ sqrt((getx-city.getx)^2+ (gety-city.gety)^2)}
2. [City List] Create a CityList class to hold the vector of cities.
   Have addCity function to add a City to object
   Have get function to get city at specific index
   Have getLIst function to return city as a vector
3. [population] Create a class population by shuffling the order of the cities.
   Using the Class population we can initialize a 50 set of randomly ordered city to pick from
   Create a class Tour, and assign the Vector of the cities to this tour.
   Create a getter function to get city and position within the tour
   Create a function to set city and position within the tour
   Create a function to find the total distance of the tour, totalD = sum(distance between all cities)
   Create a function to evaluate the fitness of the tour, where fitness = 1/totalD
4. [Crossover]
   In the population class, create a evolve function to select 2 parent tours from the population.
   We can use a tournament function to randomly choose a small set of tours with the best fitness.

After the 1st generation we can also pick the tour with the highest fitness from the previous generation., We can copy a subset of cities from the parent1 to make the new child tour.
The fill in the rest of the new child tour with another parent tour. (Make sure the same city is not selected and all cities are accounted for)

5. [Mutation] Use random function to reorder the new child tour by swapping cities at 2 random positions. Then add the child to a new tour and start a new population of 50 “improved” ordered cities
6. [Loop] Go to step 4, end when the specified number of iterations (generations) have been met.
7. Find the shortest distance in the population and write the order of the tour cities to the output file.

Summary of Greedy Based Algorithms

Algorithm Description
A greedy algorithm picks the optimal choice for each decision in a candidate set and hopefully returns an optimal overall solution. Applying a greedy algorithm to the Traveling Salesman Problem (TSP) would pick the shortest distance to the nearest unvisited city, for each city in a dataset, until all cities have been visited. Throughout our course work we have seen instances when a greedy algorithm application resulted in an optimal solution, or a good approximation. We have also seen instances when it did not result in an optimal solution.

Several greedy algorithm applications were examined to determine if any would be good candidates for the TSP, where the objective of the TSP is to visit each city in the shortest distance travelled. Greedy algorithms were selected for research because (1) they are easy to understand, (2) we have gained experience in implementing and evaluating several types of greedy algorithms and it would be interesting to see it run against a more complex solution set, (3) it is easy to implement, and (4) they may be able to approximate a good enough solution in a reasonable time. But after doing further research, it was decided that, while an application of a greedy algorithm could play a part in determining a TSP tour with lowest overall distance, that a greedy algorithm by itself would not be a good candidate algorithm for this project. Below is s general layout of a greedy algorithm, followed by discussion of three greedy candidates that were researched.

Greedy algorithms consist of the following:

- A candidate set (list of cities)
- A function that checks to see if a candidate is feasible
- A selection function that chooses the best feasible candidate

- An objective function that returns the value of a solution
- The final solution

NEAREST-NEIGHBOR

The nearest neighbor algorithm is a pure greedy algorithm that starts at city c_i and evaluates the distance to all the other cities in the candidate set. The nearest neighbor is the unvisited city with the lowest distance from c_i. This step is repeated until all of the cities in the candidate set have been visited. Based on web searches this algorithm is fast and efficient for approximating small candidate sets. However, the resulting tour could also represent a worst case. On average, research suggests that a greedy algorithm could generate a tour that is within 1.25 of the shortest possible path. This is within the 1.25 overall tolerance that was given as a requirement for this project. Since the candidate set that will be provided is not known to converge to a tour of 1.25 optimal, it would be a significant risk to utilize this type of algorithm.

Pseudocode

```
greedyTSP(candidateSet[ ])
        solutionTour = [SIZE = candidateSet.size]
        thisCity = candidateSet[0]
        solutionTour[0] = thisCity
        sort the candidateSet by distance from thisCity
        For i = 1 to candidateSet.size
                nextCity = candateSet[i]
                If nextCity.visited = false
                        solutionTour[i] = nextCity
                        sort the remaining candidateSet by distance from nextCity
        return solutionTour
```

KRUSKALS

Kruskal's is another form of greedy algorithm that results in a minimum spanning tree (MST). As with the nearest neighbor algorithm, this greedy algorithm also sorts by distance, but this one sorts the all of the edges of a graph individually, in ascending order. In TSP, this algorithm would select the smallest of all distances to any unvisited city, for each city. Then it would pick the next smallest distance between two cities, where at least one of those cities is unvisited. This cycle would continue until all vertices in the graph are connected through a single MST. As described above, this would not be a good candidate for approximating a salesman's route, as it would require some vertices to be visited twice. It is, however, easy to see how this algorithm might be modified to constrain the MST creation to better represent a single visit route. One untested/unverified way that popped into my head might be to track the number of remaining edges and once the limiting number of edges remains, then apply another algorithm to join the smaller MSTs with an optimized sequence of edges remaining. This algorithm could be a good candidate for determining a seed tour for a more complex

randomized algorithm. There were documented results of using a Kruskal's algorithm to generate a tour within 2 times the optimal path. However, the uncertainty of the inputs, as well as the documented results being worse than the nearest neighbor, caused this algorithm to be eliminated from contention. Below is a pseudocode example of a Kruskal's algorithm for the TSP.

Pseudocode

```
kruskalTSP(candidateSet[ ])
        solutionTour = [ ]
        sortedSet = [ ]
        sort the candidate set by distance between vertices, in ascending order and add
        them to the sortedSet
        For i = 1 to sortedSet
                For each city in a distance calculation  //max will be two cities
                        If city is unvisited
                                add city to array
                        else
                                If city is un-connected  //meaning that there isn't a
                                                         connected path to that city already
                                                         via a shorter edge
                                        add city to array
        return solutionTour
```

TOLERANCE BASED GREEDY

One of the modifications to a weighted edge greedy algorithms is the concept of the tolerance based greedy algorithm. The idea is has been examined for several Branch and Bound algorithms, but not as much for the greedy approach. While the upfront work for a tolerance based algorithm is more intensive, the experimental results confirm that tolerance based decisions perform better than the weight based decisions for the same class of algorithm (greedy, branch and bound, etc).

The concept of tolerances is applied by expanding/contracting paths over edge weights and coming up with an upper and lower tolerance for each edge. While that may sound complicated, it can be visualized by taking a set of vertices and performing either a scan down the paths of that vertice (think Breadth First Search or a Depth First Search), and then applying an upper and lower bounds on that path as an optimal solution. Then picking the next node with desired tolerance range, and running the scan (expansion/contraction of a path) again.

For this TSP, the complexity of this problem would be in trying to create a tolerance function that could take the weighted edges of the connected graph in the candidate set, and convert that to a decision based on the tolerances. Because there is not much research yet on this idea for greedy algorithms, attempts were made to try to create

functions to convert distances to a weighted tolerance, that would be general enough for any candidate set that might be thrown at us, as well as improve the predictive tour to less than the 1.25 from a greedy algorithm. Estimates from this paper were that this idea could improve the greedy approach by 10 times. If that is true, and using the average greedy result of 1.25, this idea could result in a tour that was potentially up to 1.025% of the optimal path.

Most of the algorithms used for this study were generated based on the type of TSP graph that our candidate set will generate. Since this is currently an unknown, and since this concept is not well documented or vetted outside the experiments referenced, it was determined that this is not a good candidate for this TSP project.

Summary of Simulated Annealing

Algorithm Description
Simulated annealing is a seeded algorithm that can be used for approximating an optimal path in the TSP. The concept of simulated annealing was born out of annealing in the metallurgical field, where temperature (heating and cooling) was controlled in such a way as to maintain shape and structure. As the temperature cooled, the structure becomes fixed. The idea, which has now been applied successfully to computer science, is that when the temperature is high, two cities are swapped and the new distances evaluated. The algorithm has a higher probability of picking solutions that are worse. But as the temperature decreases that probability decreases until the optimal path becomes fixes. This is achieved by the following for a candidate set of cities:

1. Select a random tour, or start with a pre-calculated tour (maybe a good option for a greedy calculated one), and select initial high temperature.
2. Loop through the algorithm until cooled or until solution found.
3. Pick two cities on that initial tour at random and swap them. Note: this could also include the first city and its neighbor.
4. Recalculate the tour distances with the swapped cities
5. If swap-distance is shorter, then keep the resulting solution
6. If swap-distance is longer, then keep it with a calculated probability (cost/temp) and determine which is the better longer tour based on that probability. When temp is there is a greater likelihood of picking worse solution.
7. Lower the temperature and repeat steps 2 thru 6 with this new tour.

One concern of using simulated annealing is that the tour solution can diverge quickly under specific conditions in the random set. This may occur when the temperature is not set high enough to allow the algorithm to explore the entire graph. But, it could also occur under a given set of cities. One way to offset this is to not let it diverge too far, or

focus into one area of the graph for too long. When the conditions set to control this are triggered (i.e. X degrees of cooling and/or acceptance probability modifications), the algorithm can be reset and begin again with the best-known path so far. With this in place, the algorithm may run slower than any of the above greedy algorithms (under certain inputs), but it is able to find an optimal solution. This algorithm has the potential to have high resource usage and memory demand. However, there are several ways this could be optimized further. As an example, one of the ways to improve performance would be to pre-calculate all of the distances between cities in advance. Overall, this algorithm would be easy to implement in Python, and it would generate a solution much closer to an optimal solution than any of the previously mentioned greedy algorithms, and is a good candidate for the TSP.

Pseudocode

```
simAnnealTSP(tourINIT, Temp)                    //NOTE:  could also take a random tour
        thisTour = {NULL}
        nextTour = tourINIT
        thisDist = tourINITDistance
        while thisTemp > 1
                        Do random swap of two cities on nextTour
                        nextDist = calculateTourDistance(nextTour)

                        if nextDist < thisDist
                                pickIt = doProbability(nextDist, thisDist, thisTemp)
                                If pickIt
                                        thisDist = nextDist
                                        thisTour = nextTour
                        thisTemp = tempFunction (thisTour)
                        checkDivergence(thisTour)
        return thisTour
```

Selected Algorithm

Genetic Algorithm

The genetic algorithm and the simulated annealing algorithms were similar enough in theory, with the initial tour selection and swapping cities to try to minimize the tour distance, that it came down to the predicted pros and cons of each. In general, the simulated annealing algorithm was a concern in that it could get caught up in divergent tours and/or result in several tour resets. Because of this, the genetic algorithm was selected. Prior to development, online examples of genetic algorithm implementations and different coding languages were looked over. This resulted in the decision to use

C++ for the coding language and to utilize arrays for storing the tours and manipulating the populations.

Algorithm Development

Five different classes of objects were generated using the source/header structure in C++. The classes were:

- tspCity: contains the constructor, as well as all of the get and set methods for creating and accessing tspCity objects.
- cl: is similar to a tour in that it is a vector of cities, but has very specific functionality that helps to manage the lists of cities within a population.
- tour: contains the tour constructors for creating a tour of cities, as well as the methods for getting and manipulating the tour object.
- population: contains the constructors for creating a population of tours, get and set methods for a population, and a method for determining the fittest.
- GA: is the genetic algorithm object and contains all of the specialized methods for performing a genetic evolution on a population of tours. These specialized methods include evolving the population, mutating the population, crossing over tour values to generate child populations from an input set of parent populations, and also the tournament selection logic that generates the parent populations for evolution, mutation, and crossover.

In addition to the above objects, a source file was created with helper functions for loading and displaying the tours. A main function was used to load the input files, call the genetic algorithm and population methods, and output the results.

Debugging & Optimization Processes

After getting the Genetic Algorithm code compiled, we moved on to debugging. Utilizing arrays was supposed to simplify the process of manipulating the population of tours, but became quite cumbersome when managing the pointers. It was unanimously decided to switch implementation to vectors over arrays, two days prior to the deadline. The array code was reformatted to replace the arrays, and array functions with vector logic and functions. This simplified the rest of the debugging process.

During testing we tried to optimize the processes by implementing selective evolution, modifying the mutation and crossover tour management and randomization, and even tried to implement sorted parent:child tours. Ultimately, there were too many optimizations to get them all working. Some of the optimizations we would have like to implement included reducing the iterative vector calls and instead managing a sub-set of fittest parent tours that evolved into candidate fittest child tours. Another optimization could have been to reduce the number of SQRT function calls. The $distance^2$ could have been compared during iterations, rather than the $SQRT(distance^2)$, with only the final getDistance and getFittest calculations using the SQRT function. The evolution offset and elite logic had several options for optimization that we didn't have time to

chase down. Ultimately, the end result is a genetic algorithm that works pretty good for smaller city sets than for the larger city sets found in tsp_example_3.txt and in some of the competition cases. Below are the observed test results.

Test Results

Input File	Calculated Min Distance	Execution Time (min)	Expected Distance	Comp. Ratio
tsp_example_1.txt	125,798	1:27	108,159	1.163
tsp_example_2.txt	4,897	25:32	2,579	1.899
tsp_example_3.txt	127,025,280	9:36	1,573,084	80.75

Competition Results

Test Case	Calculated Min Distance	Iterations (evolving generations)	Execution Time (min)
1	5787	5,000	0:47
2	10,128	10,000	2:58
3	21,612	10,000	7:29
4	49,971	10,000	14:54
5	112,894	10,000	30:08
6	381,578	2,000	11:55
7	1,683, 723	1,000	15:33

Team Contribution Summary

Overall, each team member contributed several hours of independent work and also many hours on voice conferences discussing design, implementation, debugging, testing, and optimizing. Each member participated in each of the following:

Date	Task	Participants
2/23	Initial introductions	chews, dongv, orcutts
3/2	Picked an algorithm to research	chews, dongv, orcutts
3/9	Submit summary of research	Chews - branch and bound, Dongv - genetic algorithm, Orcutts - greedy and simulated annealing
3/10	Team Voice Conference: Discussed all research and decided on an algorithm	Chews - city and input, Dongv - population and GA, Orcutts - tour and tourManager
3/11	Worked independently Team Voice Conference: Discussed vector vs array implementation	chews, dongv, orcutts
3/13	Integration of all code Team Voice Conference: walked through debugging full program	chews, dongv, orcutts
3/14	Submitted individual contributions Attempted adjustments for successful array implementation. Team Voice Conference: code walk thru, new direction (vectors)	Chews - integration Dongv - GA, Orcutts - tour and population
3/15	Team Voice Conference: compiling and debugging new vector code	chews, dongv, orcutts
3/16	Team Voice Conference: Enhancements and code improvement research, debugging, optimizations, final write-up and competition testing / analysis	chews, dongv, orcutts

Additionally, here are some of the individual contributions

- chews: initiated the documentation and the group chats, as well as lead and hosted most of the debugging/voice chats and setup chat tools
- dongv: completed the bulk of the genetic algorithm research while looking for ways to optimize and make code improvements., as well as created a shared project on github.
- orcutts: pulled together the final documentation and code submittal

References

Branch & Bound Algorithms

1. https://www.youtube.com/watch?v=nN4K8xA8ShM
2. https://www.geeksforgeeks.org/branch-bound-set-5-traveling-salesman-problem/
3. http://lcm.csa.iisc.ernet.in/dsa/node187.html#fig:tsp5

Genetic Algorithms

4. http://www.technical-recipes.com/2012/genetic-algorithms-applied-to-travelling-saleman-problems-in-c/
   http://www.theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5
5. https://arxiv.org/pdf/1402.4699.pdf
6. https://techtorials.me/coding-projects/the-traveling-salesman-problem-genetic-algorithm-in-c-and-cuda/
7. https://www.lalena.com/AI/TSP/
8. https://github.com/marcoscastro/tsp_genetic/blob/master/src/tsp.cpp
9. https://github.com/Adam-Flammino/TravelingSalesmanGeneticAlgorithim/blob/master/TravelingSalesmanGeneticAlgorithim.cpp

Greedy Algorithms

10. Cormen, T. Introduction To Algorithms, third edition.
11. CS325 Lecture notes
12. https://en.wikipedia.org/wiki/Greedy_algorithm
13. https://github.com/tibbsm/CS325_Group_Project/blob/master/Project%20Report%20(LaTeX)/Nearest_Neighbor.pdf
14. https://en.wikipedia.org/wiki/Travelling_salesman_problem
15. https://www.programiz.com/dsa/kruskal-algorithm

Tolerance Based Greedy Algorithm

16. http://www.cs.rhul.ac.uk/~gutin/paperstsp/rrgreedy.pdf
17. https://www.hse.ru/data/2010/12/11/1223077546/EJOR%2008.pdf

Simulated Annealing

18. http://codecapsule.com/2010/04/06/simulated-annealing-traveling-salesman/
19. https://en.wikipedia.org/wiki/Simulated_annealing
20. https://www.mathworks.com/matlabcentral/fileexchange/9612-traveling-salesman-problem--tsp--using-simulated-annealing?s_tid=gn_loc_drop
21. http://www.theprojectspot.com/tutorial-post/simulated-annealing-algorithm-for-beginners/6