CS 325 Project 5 Report (TSP) - Group 11

Michael Jones, Suzanne Thrasher, and Jackie Lamb

Detailed Description of the Ideas behind our Algorithm:

We first looked into several exact solution methods. To better appreciate the problem we were dealing with we first created a brute force algorithm that would derive the exact answer. For less than 10 cities this worked brilliantly. It was shortly after that, that the true complexity of the problem settled in. We knew that we would need to handle more cities, so we first attempted using branch and bound. It was at this point that we realized that we were not looking to derive an exact solution to the problem. A heuristic method was going to be the only way to derive a solution to larger test cases.[1]

Our goal was to find an approach that would generate as many possible instances within the allotted time frame. Nearest Neighbor was an easy algorithm to start with, but the paths created were all over the place based on the random starting point. Our greedy implementation of it is still used in the current algorithm. Modifying nearest neighbor did not allow us to find a precise enough solution and it required building nearly the full path before we found out it was not the shortest so far.

Next, we built a distance table for our algorithms to use. We wanted our program to only calculate the distances once, so they could be quickly accessed without recalculation. Of course it is at the cost of the speed to generate the table. However based off of how our algorithm works only half of the table needs to be created. We create a full table, but since the distance from city 1 - 2 is the same as 2 - 1 they can both be assigned at the same time.

We settled on the opt methods after watching this series of lectures on how the algorithm worked. [2] We used nearest neighbor to determine our starting path because it was an easy to implement algorithm. We wanted to spend our time on the refinement of the path not generating an initial lowest path.

The switch process of the opt functions are fairly straight forward. Where the real challenge comes in is deciding how many switches are needed and what elements you should be allowed to switch. The pairwise_exchange function was the first that we implemented. For small instances, such as 200 cities the speed is very good, but speed slowed on larger input. A straightforward 2-opt does give us the best results because every possibility within that path is tested but because of this, speed really suffers.

In reviewing how we could narrow this field down we discovered localized searches. This is what the quadrant function does. The original list of cities is split into 4 quadrants based off of the average x and y value. That neighborhood value is appended to the end of each city and can now be compared when reviewing a switch. We chose four neighborhoods just because it was an easy first step to implement.

With these neighborhoods we could now run a localized 2-opt where the only cities that were considered to be switched were those in the same neighborhood. The speed of the algorithm really picked up with this technique, however the accuracy of the results given did suffer because even if the next closest neighbor might yield an improvement in the length of the path, if it is outside the given neighborhood it is not considered for a switch.

With this is mind we now attempted a 3-opt solution, switching three cities at a time instead of two so long as they are in the same neighborhood. This gets decently fast results for even the largest cases with more time involved setting the matrixes and neighborhoods, than actually paring down the result. Further improvement was found here by also calling one of the 2-opt functions listed above from 3-opt for smaller test cases.

Pseudocode:

1. Parse file into (city number, x, y) format into a list.

---

[1] http://en.wikipedia.org/wiki/Travelling_salesman_problem

[2] https://www.youtube.com/watch?v=-cLsEHP0qt0

2. Create 4 neighborhoods based off of locations of cities.
Take the average value of x and y coordinates and make that x and y access and append fourth element to each city that will denote neighborhood.
3. Create a 2d list as distance_matrix. Where distance_matrix[1][2] would hold the distance between cities 2 and 3. If city is comparing to itself save “inf”.
4. Call find_nearest in an infinite while loop sending the city listing and distance matrix.
5. find_nearest is our driver function for the remainder of the calculations.
    1. Call find_path_nearest which is our nearest neighbor function and if lowest set to lowest
    2. Save temp copies of the list and call threeopt
6. threeopt().
    while no_improve > 0:
        Set a path to a “fresh” path or the path calculated in a prior iteration, set no_improve = 0
        in three for loops, loop over every element and consider switching of cities.
            if all points are in the same neighborhood setup swap.
                Set a path to a “fresh” path or the path calculated in a prior iteration
                switch the i, j and k elements places in the path
                set the curr_distance to the above path current path
                if curr_distance < low_distance:
                    set low_distance to curr_distance, set lowest_path to current path,
                    increment no_improve to see if we can improve on it
                if ith or jth iteration of the path is greater than lowest distance break out of ith or jth
        set the current low path found to run again and see if we can improve it.
    When no more improvements can be made while loop stops and if our path is less than the global overall path. Set global lowest_distance and lowest_path to the current one found. Also create a new copy for use in the next called functions
    if the path is less than 275 cities run this path through pairwise_exchange to see if we can improve
    elif: if path is less than 5000 cities run through localized_pairwise_exchange to see if we can improve
    if greater than 5000 cities we are done.

    7. pairwise_exchange() - for small city instances pair switch without regard to neighborhood.
    8. localized_pairwise_exchange() - for larger instances localized pairwise switches pairs only within its assigned neighborhoods just like threeopt but with two city switches.

Possible Improvements:
1. Dynamically creating “quadrants” base off of input size. So our 15,000 city list may create 20 neighborhoods or 30 neighborhoods based off of what percentage of the cities you want in each neighborhood. The more neighborhoods used the faster the algorithm will be.
2. Possibly “overlapping” sub-quadrants to solve the problem of close cities not getting included just because they are on the other side of the “street”
3. Use Christofides algorithm to create the original path. This would give us a more accurate starting point.

Further sourcing in code