TSP

CS 470 University of Alabama

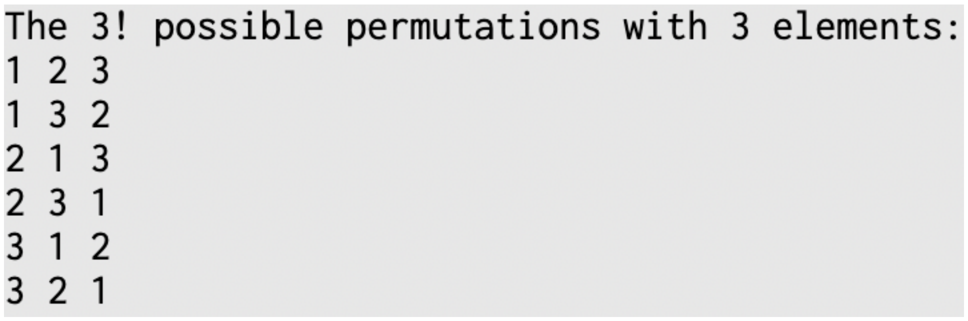
Jacob Santos

02-23-24

The Traveling Salesman Problem (TSP) is one of the most popular Computer Science problems defined as an NP Hard problem where NP means non-deterministic polynomial time. And the reason as to why this problem is considered as such is because the runtime for any algorithm to solve this problem is going to be much higher than polynomial. To give a short introduction to the problem, think of a set of cities that have distances defined between all other cities within that set. Your job is to find the optimal path to visit all these cities and eventually return to the city that you started in. At first glance, you might think it's simple, and for small sets of cities, you'd be correct. But what about a set of, say 100 cities? Now what about 1,000 cities? Even with lots of advancements in mathematics and technology, the algorithm to solve the TSP in polynomial time is still unknown. In fact, it's such a difficult algorithm to figure out, there's a 1-million-dollar reward for whoever could find such an algorithm. Thus, only the brute force algorithm can find the true answer with a runtime of O(N!) and all the other current algorithms serve as only approximations to the solution. And in the remainder of this paper, we will be going over the brute force and two other noteworthy algorithms for the TSP and my thought process behind constructing them all.

Brute Force Algorithm

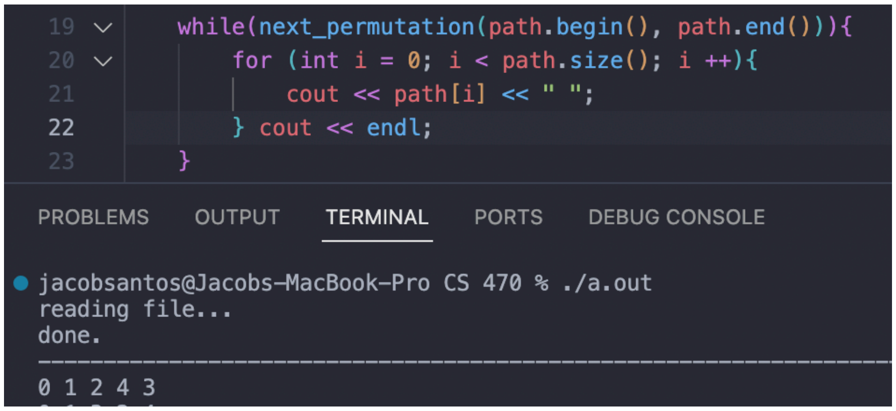
When planning how I should go about constructing the brute Force Algorithm, the point was to find an algorithm that exhausts all possible routes to then find the best one, but at a much higher cost than any other algorithm. So, I knew it would go something like this: for every node in the set of "cities", compute the distance for every possible order of n cities that it was connected to and keep track of the shortest distance of them all. And to do so, I at first thought of doing it recursively where I could then try all possible combinations. But as I continued to research how to find all combinations in a vector in C++, I came across a built-in function called "next_permutation", where for a vector with n nodes, it finds the n! possible permutations (combinations) within the vector. For example, if there's a vector with 3 different cities, the next permutation would go through all 3! permutations which can be seen in **figure 1.**



```
The 3! possible permutations with 3 elements:
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Figure 1 – next_permutation for 3 elements

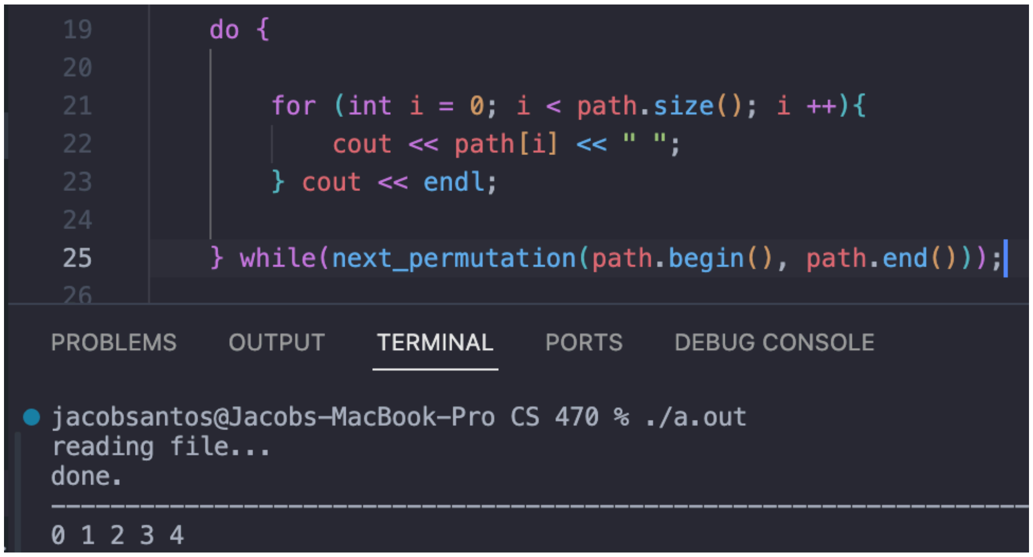
After finding out about this built in function, I decided to just use it instead of going with the recursive approach as it seemed like a much easier way of doing things. When I first used it, I just threw it in a while loop and printed out each permutation to make sure I was using the function correctly, and thankfully I did so because I ran into my first problem which is shown in **figure 2**. The problem being that when it was in a while loop, it was skipping the first permutation since the function was being called in the argument of the while-loop before I could access it. Because of this problem, it printed out "0 1 2 4 3", completely skipping "0 1 2 3 4". So instead of using a regular while-loop, I put it inside of a do while-loop; this allowed me to access the first permutation before, which can then be seen in **figure 3**.



```
while(next_permutation(path.begin(), path.end())){
    for (int i = 0; i < path.size(); i ++){
        cout << path[i] << " ";
    } cout << endl;
}
```

```
jacobsantos@Jacobs-MacBook-Pro CS 470 % ./a.out
reading file...
done.
--------------------------------------------------------------------------------
0 1 2 4 3
```

Figure 2 – next_permutation in regular while-loop



```
    do {

        for (int i = 0; i < path.size(); i ++){
            cout << path[i] << " ";
        } cout << endl;

    } while(next_permutation(path.begin(), path.end()));
```

```
jacobsantos@Jacobs-MacBook-Pro CS 470 % ./a.out
reading file...
done.
--------------------------------------------------------------------
0 1 2 3 4
```

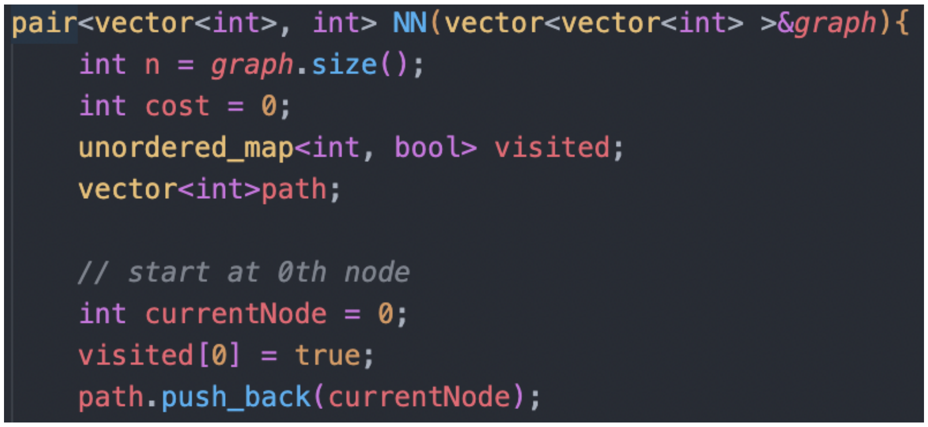
After throwing it in this do while-loop, the rest of the algorithm was super simple as I replaced the print statement with a line to compute the current cost and another to update the path as well.

Figure 3 – next_permutation in do while-loop

Again, it is super important to note that although this algorithm is super simple to construct, the run time is O(n!), meaning it is very slow. For example, it will run through 120 permutations for a set of 5 cities and adding 5 more cities to the set for a total of 10 will increase the number of permutations all the way to 3,628,800 different permutations. Which is why this algorithm is only feasible for about up to 13-15 cities.

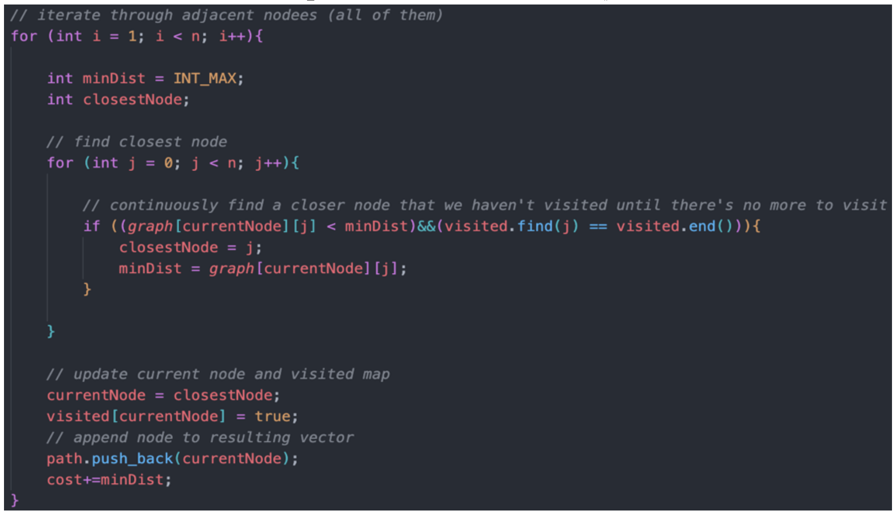
Nearest Neighbor Algorithm

The next algorithm is called the nearest neighbor algorithm which works like this: start at a node and traverse the set based on which ever city is the closest and eventually map back to the starting node. For example, if you have a set of say 5 cities and start at city 1, then go to which ever city is closest out of the 4 other cities and do that until you've traversed through all cities. At first glance, I knew it would consist of two for loops since you must compare each city's cost from each city ultimately making it an O(N^2) algorithm. I also knew I would need a way to keep track of cities that I already visited, so the algorithm wouldn't mistakenly go back to a city it's already accounted for, so I decided to use an unordered map for that as look up times are only O(1). Lastly, I would need a separate vector to store the best path that nearest neighbor could find and integer variables to keep track of the cost and current node that we are on. **Figure 4** shows the beginning of the function where I declare all these needed variables. Note, in this case, I decided to just start at the 0th node since we were instructed to do so for this project. Additionally, in **Figure 5** it shows the main part of the algorithm where there's a nested for loop to compare distances and traverse on to the closest node. For every nested loop, I had to set minDist to INT_MAX that way there was a guaranteed way for the first node it traverses to, to be the temporary closest node until a closer one is found later via the if statement in the nested loop. The second argument in the if statement is ensuring that the current node that is being checked hasn't been visited already. When you try and find something in an unordered_map that doesn't exist, it will equate to the visited.end(), which is what we're looking for. And finally we add the closest node into the visited map and append it to the path vector.



```
pair<vector<int>, int> NN(vector<vector<int> >&graph){
    int n = graph.size();
    int cost = 0;
    unordered_map<int, bool> visited;
    vector<int>path;

    // start at 0th node
    int currentNode = 0;
    visited[0] = true;
    path.push_back(currentNode);
```

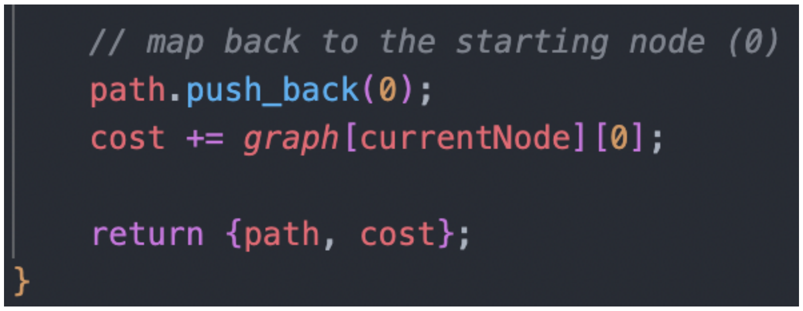
Figure 4 – beginning of Nearest Neighbor



```
// iterate through adjacent nodees (all of them)
for (int i = 1; i < n; i++){

    int minDist = INT_MAX;
    int closestNode;

    // find closest node
    for (int j = 0; j < n; j++){

        // continuously find a closer node that we haven't visited until there's no more to visit
        if ((graph[currentNode][j] < minDist)&&(visited.find(j) == visited.end())){
            closestNode = j;
            minDist = graph[currentNode][j];
        }

    }

    // update current node and visited map
    currentNode = closestNode;
    visited[currentNode] = true;
    // append node to resulting vector
    path.push_back(currentNode);
    cost+=minDist;
}
```

Figure 5 – main part of Nearest Neighbor

After the main part, all that needed to be done was just append the 0th node to path again, compute the cost from the current node back to the 0th node, and return the path and the cost that was computed which can be seen in **Figure 6** below. This algorithm overall is much quicker than the Brute Force method as it's time complexity is only O(N^2) versus the O(N!) that Brute Force offers. The only downside is that most times, especially for larger graphs, Nearest Neighbor only computes an approximation of the actual answer. Hence as to why the Nearest Neighbor Algorithm is called an approximation algorithm.

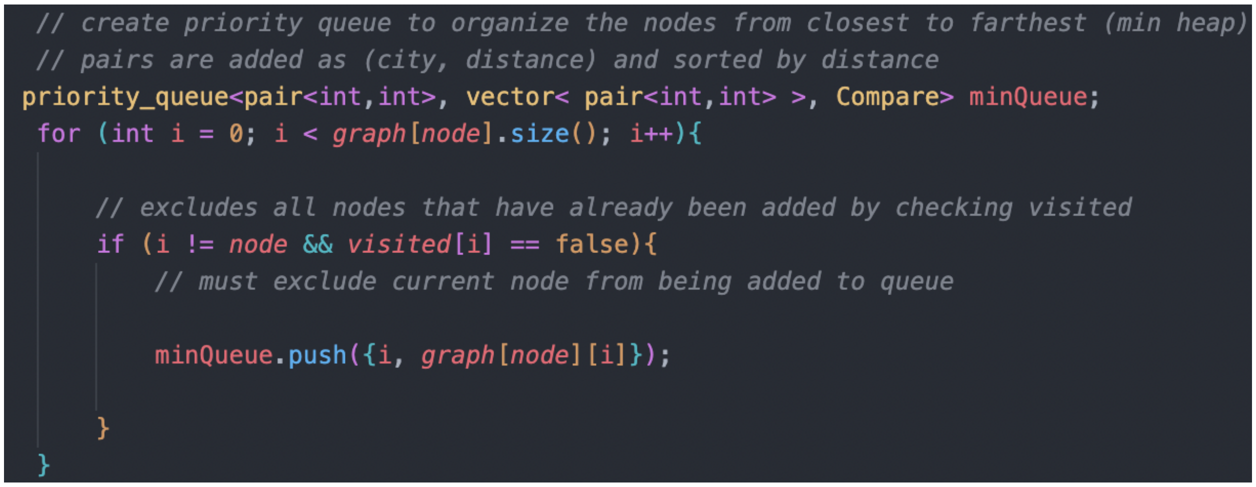


```
    // map back to the starting node (0)
    path.push_back(0);
    cost += graph[currentNode][0];

    return {path, cost};
}
```

Figure 6 – map back to 0th node

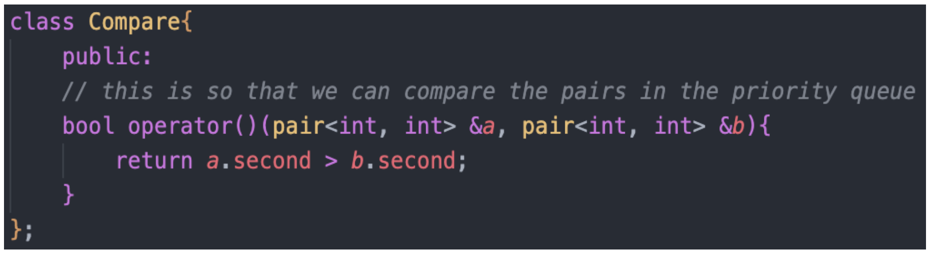
My Algorithm

When it came time to make my own algorithm, I decided to do something similar to nearest neighbor, but use a depth first search approach and repeat the process for every node as the starting node and keep track of the smallest distance. I also wanted to make my algorithm work so that if it ever computes a higher cost than the previous cost that was computed, the algorithm would just break and return the previous cost and path to make the time complexity a little bit better. Additionally, instead of just comparing each node to find the closest city, I wanted to store them into a minQueue so I could just pop the top of the queue to get the smallest city which can all be seen in **Figure 7**. Furthermore, I needed to store pairs into the minQueue so that the node and it's distance could be stored together. In order to do that, I then needed to create a separate class, which is shown in **Figure 8**, that had a function to compare the pairs and their distances to maintain the minQueue structure. Although this seemed like a good idea at first, it ended up just making my algorithm way slower that I intended so it didn't end up making the final cut of my algorithm. Nonetheless, I thought it was still important to discuss as it did still work the same as comparing every other node to find the closest one.



```
  // create priority queue to organize the nodes from closest to farthest (min heap)
  // pairs are added as (city, distance) and sorted by distance
 priority_queue<pair<int,int>, vector< pair<int,int> >, Compare> minQueue;
  for (int i = 0; i < graph[node].size(); i++){

      // excludes all nodes that have already been added by checking visited
      if (i != node && visited[i] == false){
          // must exclude current node from being added to queue

          minQueue.push({i, graph[node][i]});

      }
  }
```

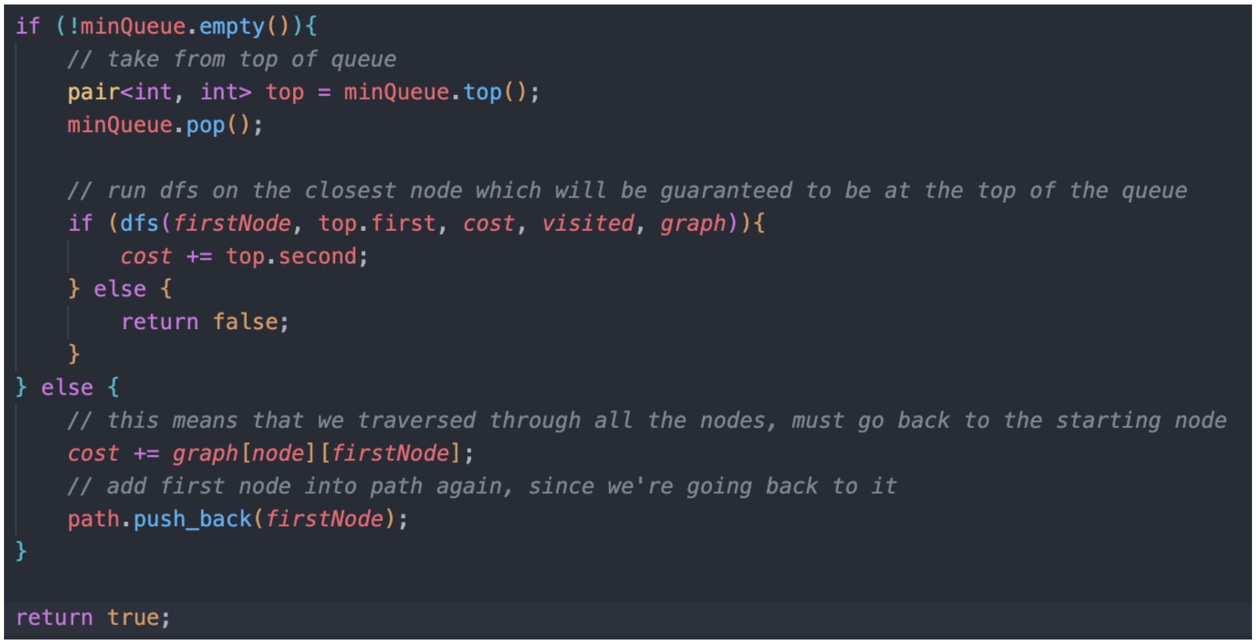
Figure 7 – Storing into minQueue



```
class Compare{
    public:
    // this is so that we can compare the pairs in the priority queue
    bool operator()(pair<int, int> &a, pair<int, int> &b){
        return a.second > b.second;
    }
};
```

Figure 8 – Using Compare Class for minQueue

After adding into the queue, I then ran dfs recursively after popping the top of the queue but if the queue was empty, then that means we've traversed through all the nodes, and I could just add the path from the current node back to the starting node like I did in the Nearest Neighbor. Which can all be seen in **Figure 9**. However, after successfully creating this algorithm, I shortly found out that it was much slower than Nearest Neighbor and it would only find a slightly better solution that Nearest Neighbor. And since the cost to reward of using dfs was not worth it, I decided to then scrap the whole dfs approach entirely and instead modify Nearest Neighbor itself and incorporate another function called 2-optimization that I found while doing more research.



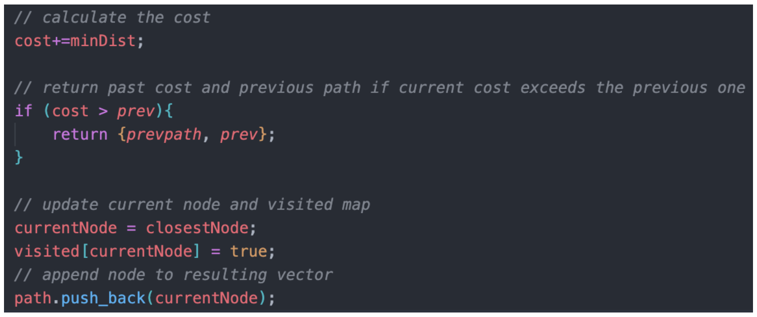
```
if (!minQueue.empty()){
    // take from top of queue
    pair<int, int> top = minQueue.top();
    minQueue.pop();

    // run dfs on the closest node which will be guaranteed to be at the top of the queue
    if (dfs(firstNode, top.first, cost, visited, graph)){
        cost += top.second;
    } else {
        return false;
    }
} else {
    // this means that we traversed through all the nodes, must go back to the starting node
    cost += graph[node][firstNode];
    // add first node into path again, since we're going back to it
    path.push_back(firstNode);
}

return true;
```

Figure 9 – last step in my old Algorithm

The way I modified the Nearest neighbor algorithm was by first adding more variables into the argument to do two more things:

1. Start at every other node instead of just 0
2. Return the previous minCost and path in the event that the current cost is already higher than the previous minCost

So looking back at figures 4 & 6, instead of starting at and mapping back to the 0th node, my modification starts and maps back to the ith node, where i is the current node that we are performing nearest neighbor on. Additionally, **Figure 10** shows the step where I incorporate an if statement to check if the current cost is already higher than the previous cost. Which in the best case, improves time complexity by a lot.



```
// calculate the cost
cost+=minDist;

// return past cost and previous path if current cost exceeds the previous one
if (cost > prev){
    return {prevpath, prev};
}

// update current node and visited map
currentNode = closestNode;
visited[currentNode] = true;
// append node to resulting vector
path.push_back(currentNode);
```

Figure 10 – checking prevCost

Along with these changes to the Nearest Neighbor, I then added a separate function that was mentioned briefly called 2-optimization or 2-opt for short terms. The way this function works is that it goes through a path that was computed by Nearest Neighbor and untangles any paths that are crossed to compute a shorter path. For example, in **Figure 11**, you can see how at first, two paths are crossed, then after 2-opt, it untangles them resulting in a shorter path. Since Nearest Neighbor is an approximation algorithm, there's bound to be some tangles within a path it computes and after testing it, 2-opt decreases the path cost significantly. Which is why in the for loop where I call nearest neighbor and 2-opt again, I needed to have two separate variables to keep track of the minimum nearest neighbor output and then the output of that minimum after 2-opt. Otherwise, nearest neighbor would never find a better path than the first optimized path since I altered it to where it breaks whenever the current cost becomes greater than the previous cost. Ultimately, resulting in a bad approximation. In **figure 12**, you can see the implementation of these variables and how I had to make a nested if statement for this to work.

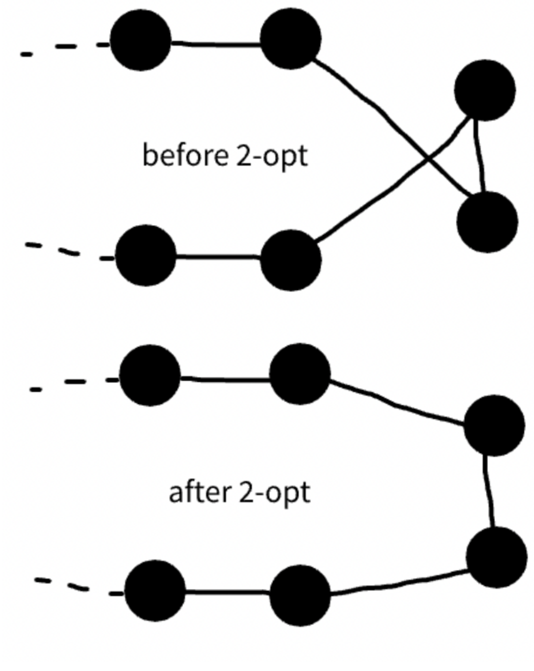
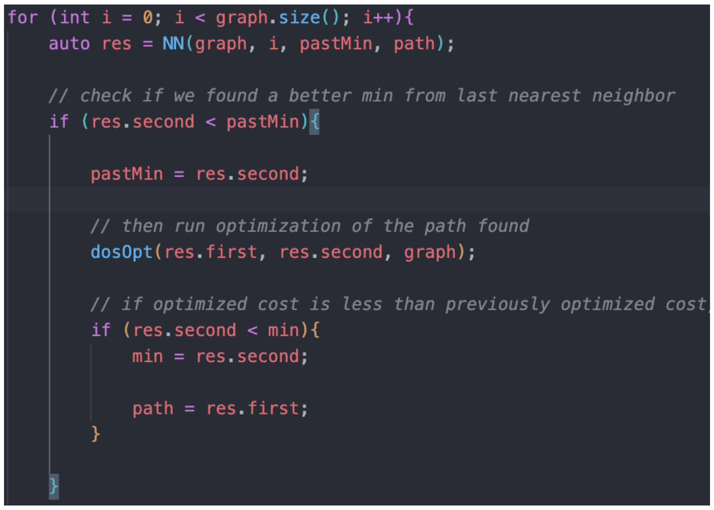


Figure 11 – 2-opt visualization



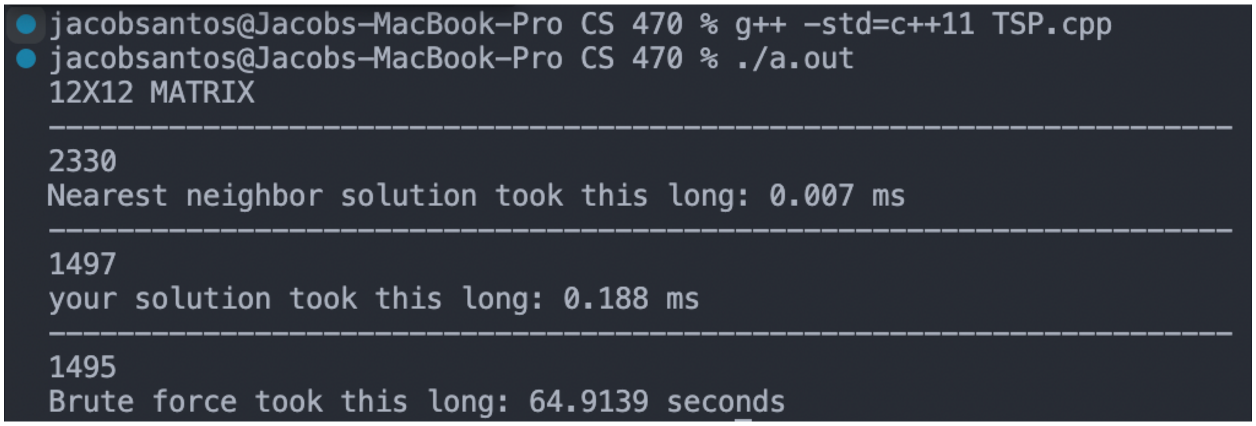
```
for (int i = 0; i < graph.size(); i++){
    auto res = NN(graph, i, pastMin, path);

    // check if we found a better min from last nearest neighbor
    if (res.second < pastMin){

        pastMin = res.second;

        // then run optimization of the path found
        dosOpt(res.first, res.second, graph);

        // if optimized cost is less than previously optimized cost
        if (res.second < min){
            min = res.second;

            path = res.first;
        }

    }
```

Figure 12 – usage of 2 different min variables

Overall, my algorithm after it was built had an average run time of O(N^3) but in the best cases, the Nearest Neighbor algorithm returns much earlier and doesn't even need to be 2-optimized since it's raw path already exceeded the previous min path. The highest I was able to run this algorithm within 30 minutes was a 10,000 node graph and after that it would take over an hour, and probably even multiple hours for a 20,000 node graph. Nonetheless, I think this was a pretty good algorithm since it gives a pretty good approximation for a fairly large set of cities.

Comparison of The Three Algorithms

I put all these functions into the same file and used a header file that C++ offers to keep track of the time for each algorithm. Considering that Brute Force can't really go higher than 15 nodes, I tested all three algorithms in a graph of 12 nodes to see which would finish that fastest and which would get the best answer. **Figure 13** shows the following results.



```
● jacobsantos@Jacobs-MacBook-Pro CS 470 % g++ -std=c++11 TSP.cpp
● jacobsantos@Jacobs-MacBook-Pro CS 470 % ./a.out
  12X12 MATRIX
  ----------------------------------------------------------------------
  2330
  Nearest neighbor solution took this long: 0.007 ms
  ----------------------------------------------------------------------
  1497
  your solution took this long: 0.188 ms
  ----------------------------------------------------------------------
  1495
  Brute force took this long: 64.9139 seconds
```

Figure 13 – comparisons of the three algorithms

As shown above, the nearest neighbor algorithm took the fastest to compute, coming in at 0.007 milliseconds, whereas my algorithm finished in 0.188 milliseconds. However, my algorithm's solution produced a far better answer of 1497 which was only 2 away from the brute force answer. The only thing is, it took the Brute Force algorithm a whole minute to finish computing.

Conclusion:

To conclude, the TSP is not an easy problem to solve at all as only the brute force algorithm is the only way to find the true answer but at a very expensive cost. And throughout the three algorithms discussed, it's obvious that the most accurate of them all is the Brute Force. However, the other two algorithms are the only ones that are suitable for solving larger graphs as the Brute Force takes a whole minute to solve only a 12-node graph. And considering the time complexity grows exponentially, imagine how long it would take for it to solve a 100-node graph. In contrast, the nearest neighbor takes the shortest amount to solve but also provides the worst answer for larger graphs deeming it to be an approximation algorithm. My own algorithm kind of takes the best of both worlds as it provides a generally good approximation but at a slower computing time than nearest neighbor and faster computing time than the brute force algorithm. Overall, I thought it was interesting to try and solve the TSP in polynomial time as it hasn't been done before and to be honest, I am not sure if it's even possible. And although my algorithm servers as only an approximation, I'm still proud of it and hopefully in the future I'll be able to come back and make an even better one.

References

Brute Force:

- Next_permutation: https://cplusplus.com/reference/algorithm/next_permutation/

Nearest Neighbor:

- Algorithm Explanation: https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

My Algorithm:

- 2-opt: https://en.wikipedia.org/wiki/2-opt
- Storing pairs in minQueue**: https://www.geeksforgeeks.org/priority-queue-of-pairs-in-c-with-ordering-by-first-and-second-element/**