Traveling Salesman Problem

Final Report

Andre Cutuli, 80361441
Linh Trinh, 49172220
Jiaxin Li, 19683688

Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is to discover the cheapest and optimal route to visit all cities at least once and return to the beginning point given a collection of cities and the cost of travel between each pair of them. Every single city has a direct path leading them to all the other existing cities within the graph. The TSP is one of the computational mathematics problems that has received the greatest attention despite its apparent simplicity. The direct (i.e. Euclidean) distance between two cities is the definition of "cost" for this project.

Restrictions

During the construction of the two algorithms, Stochastic Local Search (SLS) and the Branch and Bound Backtracking DFS, we had to implement several restrictions on our code. The first was the time limit involved. Both algorithms have an established time limit of 10 minutes to run the program with the provided file containing the matrix (and the size). This is due to runtime constraints because depending on the algorithm used or the provided matrix is massive, the program could easily take hours to compute all possibilities to determine the most efficient path. This could lead to early forced termination of the program without it ever providing its current best results discovered so far in the matrix. The second restriction is the user's input is restricted to a text file created by the genTSP.py file provided by Kask which generates a file containing the matrix size and matrix. This aspect is to avoid the hassle of manually inputting the matrix which would be increasingly tedious as the matrix increases in size.

Stochastic Local Search with 2-opt

Data Structure

In this project, the array list is achieved using the Tkinter python package to input the generated matrix from genTSP.py (or benchmarking problems) into distance_matrix.

How the algorithm works

For solving this problem and making it stochastic, 2-opt must begin with a random route for each iteration. It can then produce nearby solutions and launch the optimization process from there. So first, we generate a list of all city identifiers (referred to as citie_names) and then iteratively choose a city at random and add it to our solution. Since each city may only be precisely visited once, we remove its identification from the list of city identifiers after adding it to our solution.

This approach applies 2-Opt to each result (as one iteration) after repeatedly running the stochastic local search algorithm. We are investigating the neighborhood of numerous different starting solutions by repeatedly doing this technique. By doing this, we dramatically increase the size of our search area and increase our chances of discovering local or even global optima.

2-opt Implementation

2-opt is a simple Stochastic Local Search method for solving the Travelling Salesman Problem. These algorithms begin with an initial solution and iteratively search for chances for improvement in the vicinity of that solution. As long as it is practical, any type of solution can be used as the first step. The 2-opt algorithm functions as follows: remove 2 arcs from the route, and join them together, its primary goal is to take an existing route that crosses itself and reorganize it so that it does not, and get the new journey distance. The existing route is updated if this alteration has resulted in a shorter overall trip distance. The program keeps improving the route and repeats the stages. This cycle is repeated until no further advancements can be made or until a predetermined number of iterations have been made.

Example: consider the following route: Amsterdam — Brussels — Paris — Berlin — Copenhagen — Helsinki — London — Amsterdam. One arch could be Brussel-Paris, and

another could be Copenhagen-Helsinki. 2-Opt alternate the joints in these arches, i.e. the route now runs from Brussel-Copenhagen and Paris-Helsinki.

- Old route: Amsterdam — **Brussels — Paris** — Berlin — Copenhagen — Helsinki — London — Amsterdam
- New route: Amsterdam — **Brussels — Copenhagen** — Berlin — Paris — Helsinki — London — Amsterdam

Below is the mechanism of how 2-opt is used to swap manipulates a given route. Here v1 and v2 are the vertices that you wish to swap when searching through the route.

```
procedure 2optSwap(route, v1, v2)
{
   1. take route[0] to route[v1] and add them in order to new_route
   2. take route[v1+1] to route[v2] and add them in reverse order to
      new_route
   3. take route[v2+1] to route[end] and add them in order to new_route
return new_route;
}
```

Describe properties

Alongside the 2-opt algorithm as the core to improve the route for each iteration, several functions are also implemented. After generating a list of city names and distance matrix, they are passed to RouteFinder Class. Under each iteration, a random initial route is established, then passed to Solver Class for 2-opt performance.

Solver Class has 4 main functions:

- `def calculate_path_dist(distance_matrix, path):` a method that calculates the total distance between the first city in the given path to the last city in the path

- `def exhaustive_search(self):` a method used to update the best distance after each iteration
- `def two_opt(cities,improvement_threshold = 0.0001):` a method that includes a 2-opt algorithm to find the best route with a given new route in each iteration. The improvement threshold is set low to make sure the path found is the shortest path it can be for each randomly generated path.
- `def swap(path, swap_first, swap_last):` a method to swap the location of cities whenever a better route is determined.

After Solver is run, it returns the current best distance and current best route found for that specific initial route, RouteFinder Class would update the best distance and best route so far whenever better results are returned, or in another word, a shorter path. Elapsed times are printed at the same time to help keep track of each iteration and to see how long results are returned for each matrix size. After 1000 iterations, the algorithm returns the best distance along with the best path found (printed out under city names). There are two restrictions for our SLS-2-opt algorithm. The first one is iterations, the algorithm would run up to 1000 times and report the best one found so far. Second is the 10 minutes limit. It would break if elapsed time exceeds 600000 msec and return the best result found so far.

Result

The 2-opt algorithm above is used to check with the 100x100 matrix (equals 100 cities to visit) with iterations equal to 1000.

```
route_finder = RouteFinder(cities, cities_names, iterations=1000)
best_distance, best_route = route_finder.solve()
print(best_distance)
print(best_route)
```

As a result, it takes 95097 msec ~ 1.6 minutes to report the optimal route distance equals 2832.5 and also the best path to obtain that distance.

```
In [30]: route_finder = RouteFinder(cities, cities_names, iterations=1000)
         best_distance, best_route = route_finder.solve()
         print(best_distance)
         print(best_route)
```

```
93855 msec
93921 msec
93991 msec
94091 msec
94158 msec
94288 msec
94347 msec
94416 msec
94482 msec
94645 msec
94796 msec
94862 msec
94931 msec
95097 msec
2834.47
[0, 10, 35, 50, 62, 89, 29, 84, 23, 5, 74, 87, 52, 4, 85, 92, 2, 39, 93, 75, 98, 8, 97, 49, 22, 66, 34, 82, 19, 37, 8
1, 20, 71, 3, 44, 86, 17, 12, 56, 53, 45, 55, 73, 33, 78, 95, 21, 68, 18, 69, 1, 90, 11, 59, 28, 24, 79, 54, 15, 40,
13, 61, 43, 42, 36, 57, 31, 48, 26, 64, 77, 41, 94, 83, 25, 67, 51, 14, 99, 70, 91, 6, 63, 9, 58, 76, 30, 60, 7, 72,
88, 32, 38, 46, 65, 16, 47, 27, 80, 96]
```

Observation and Issues

Note that the 2-opt algorithm doesn't guarantee the global optimum similar to other heuristic search algorithms, it gives you the best result for each iteration and then continuously compared with new distance found to achieve the best one found. This algorithm is set to break after 1000 iterations or exceeding 10 minutes, whichever comes first. So, the results can vary in each iteration. By doing the process repeatedly, using high numbers of iterations, the chance of finding the global minimum is also increased and is achievable. The issue remains that you can’t know if this result each time is optimal (except comparing it with the actual cost), and it takes a long time to finish the matrix at a bigger size. Starting from the matrix of size 600, it goes from 1 minute to around 10 minutes to get the problem solved.

The cost of creating a new route and determining its length can be extremely high; it is typically displayed as O(n)O(n), where n is the number of vertices in the route. This can be transformed into an operation with display style O(1). One suggestion is that create a variable to compare ahead whether the swap could help decrease cost or not, 2-opt operation is then

performed. Since a 2-opt operation involves removing 2 edges and adding 2 different edges, we can subtract and add the distances of only those edges.

```
cal_length = - dist(route[v1], route[v1+1]) - dist(route[v2], route[v2+1]) + dist(route[v1+1], route[v2+1]) + dist(route[v1], route[v2])
```

If cal_length is negative, the new distance following the exchange would be shorter. Once it is established that cal_length is negative, a 2-opt exchange is then carried out. This helps us avoid doing a lot of computation.

Additionally, changing the input from the array list to the NumPy array may also help fasten runtime. The algorithm would take less time to compute the distance after swapping or accessing the matrix or array as the NumPy array parses through the matrix more quickly than the lists.

Branch and Bound Backtracking Depth First Search

How the Algorithm Works

In this project, we chose arrays as our data structure to keep track of necessary information for the algorithm. The Branch and Bound Backtracking DFS algorithm functions similarly to the DFS algorithm. The difference is that there is a lower and lower bound that we utilize to prune certain branches. So, the algorithm begins in the function BNB_DFS() checking the base cases before proceeding to initialize the parameters to input for the BNB_Tracking() function.

In the BNB_Tracking function, there are 4 segments with their function. First off, it is checking if the allotted time allowed has been exceeded. If so, it halts the program and returns the best results it found so far. Second, it checks if the current route reached max depth meaning

all cities have been visited once. It determines if the current path should be stored and updated by comparing it with the current best path. Otherwise, nothing will occur and returns.

The third segment is the branch and bound DFS. It irritates through the existing unexplored cities and prunes branches if the lower bound exceeds the upper bound (in this case, the best distance). The lower bound is composed of 3 elements to calculate the estimated distance to the goal at any point in the path traveled. The first component is the current distance traveled from the initial state to the most recently visited state. Then, there's the heuristic function that comprises the second and third components in its computation that we add together. It obtains the path cost between the most recent state to the next unvisited state that is being observed in the current traversal of the algorithm. In addition, we look for the smallest minimum path in any of the unvisited cities in the matrix. Then we take the minimum value and multiply it by how many remaining unvisited cities there are including the return path to the initial state. Thus, the lower bound formula in simplified terms is *lower_bound* = current distance + matrix[visited city][unvisited city] + (minvalue*remaining cities).

This aspect ensures that the algorithm would be admissible as the estimated distance will be less than the optimal path cost since it is on the basis that the remaining paths will have the same cost. Once all three components are achieved, they are added together to finalize the lower-bound computation. So, if the lower bound is determined to be less than the upper bound, it stores the results and calls on the BNB_Tracking function, and repeats the process in the next depth until the condition that the lower bound is greater than the upper bound is no longer satisfied.

Lastly, the final segment in BNB_Tracking handles the resetting array information to its previous state. This is necessary as the algorithm uses few arrays to store information about the

current route as it brute forces through 1 possible route at a time. So, if a route fails to satisfy the upper bound condition or needs to explore other potential routes after reaching maximum depth, the unwanted data from the previous route needs to be removed while retaining the necessary data to explore a new branching route. Thus, the BNB_tracking algorithm will continue to loop until it explores all possible options and returns the solution for the best path, distance, and time elapsed.

Functions

It first opens the file explorer and asks the user for a file input containing the size and matrix to run the algorithm on. Then it proceeds to call on the function BNB_DFS.

- BNB_DFS(self): It checks if all the necessary base cases are satisfied before initializing all the necessary variables and arrays to pass onto the main function in the BNB_DFS Class, BNB_Tracker. The time is recorded to check if the algorithm exceeds the time limit. The initial state for the algorithm is 0.
- BNB_Tracker(self, visited, cur_dist, cur_path, depth): This function has several jobs. It checks the timer to determine whether it continues or stops the function. Then check if it's at the lowest depth and determine if the best distance and path should be updated. Then moves into starting the iteration for the current depth. It first calculates the lower bound by calling on the heuristic function and 2 other variables before it determines if it is less than the upper bound. If less than bound, it updates the variables and calls on itself with new parameters. If the lower bound is greater than the upper bound, it backtracks the data and resets it in order to explore a new branching route.
    - visited: an array that keeps track of if a city is visited or not with a True or False
    - cur_dist: total distance cost of all visited cities so far

    - cur_path: an array that tracks the order of cities visited from initial state at index 0
    - depth: current level in the BNB_Tracker function.
- heuristic(matrix, visited): It searches for the smallest path cost out of all the unvisited cities in the matrix. Once it finishes parsing the matrix for the minimum value, it returns the value times the number of remaining unvisited cities left including the return path to initial state.

Heuristic Changes

The overall structure of the BNB DFS algorithm stayed the same compared to the previous design. However, the lower bound underwent multiple development changes regarding the heuristic. Particularly, the segment in my draft design "computes the average of the distance from node *j* to the rest of the unexplored nodes." The design heuristic's main flaw was that it was estimating the distance two steps ahead of where the algorithm was currently positioned. Consequently, the pruning of paths was unbeneficial as the algorithm could effectively prune when near the maximum depth and not an earlier depth.

As a result, the first idea was scrapped and replaced with a new heuristic that computed the average cost for each unvisited city and summed up all the costs. This caused issues with the algorithm runtime and the admissibility of the algorithm. The computation was parsing through the matrix and checking if the location in the matrix was unvisited in both cities, so the meticulousness of the function was unintentionally creating a penalty on runtime length as it is called upon in every iteration. Regarding the admissibility issue, it was due to the averaging of the cost for each row which would influence the overall perception of any city's path costs. This is prevalent if a city had multiple routes that had a minimal path cost, but the average is offset by an outlier. If this was the issue for multiple cities in the matrix, then the estimated distance cost

to the goal would be easily greater than the optimal distance cost making the algorithm inadmissible.

So, our group settled for a more simplistic function which is searching for the minimum path cost in any of the unvisited cities and then multiplying by the amount of remaining unvisited cities accounting for returning to the starting city. The requirement for checking if the path cost is between both unvisited cities was removed as it would require further parsing through the matrix which would impact the algorithm's runtime. This means that the minimum path cost may not be a path option on the current path iteration; however, this allows the algorithm to be considered admissible.

Observations and Issues

When testing the algorithm, we started by testing with a size 1 matrix to test the base cases then worked our way up by increasing the matrix size until it would hit the time limit of 10 minutes. From size 2 to 10, it took seconds to compute a result, but it changed when we decided to test size 15 where the time elapsed changed drastically. It ended up taking roughly less than 3 minutes on average to output a result. Beyond the size 17 matrices, the algorithm started to time out past 10 minutes. This doesn't manage to even complete the size 25 matrices for the competition which is the smallest size to run on. It makes sense because as the matrix size gets larger, the number of iterations the algorithm needs to traverse to reach the maximum depth becomes larger. This includes all potential combinations of paths because the deeper the depth the wider the fanouts become, requiring constant backtracking and heuristic calculations to determine if the branch can be pruned or not.

As it stands, as long as the algorithm can run to completion, the output should be the optimal path. Otherwise in the situation where the algorithm times out, the result may not be the optimal path majority of the time.

Ideas for Improvements

The way I could improve the algorithm is to adjust the heuristic function calculation so it is not underestimating its estimated distance to the goal. For example, if the smallest path cost found is 0.0001, but the second smallest path cost is 2. Then say there are 9 remaining cities to visit (including a return to start), then we have .0009 and 18 for estimated path cost. If the 0.0009 value is used, then it is likely that it would proceed into the next depth whereas the 18 might be greater than the upper bound. The underestimation causes the algorithm to prune fewer branches in the long run, but we would eventually find the most optimal path if the time limit restriction was removed and let it run until completion. Even so, that is not the case. Thus, one idea to improve the pruning is to parse the unvisited cities in the matrix to find an arbitrary amount of the smallest values to compute an average and multiple by remaining cities. This would potentially reduce the heuristic's underestimation problem and prune more frequently.

Beyond the heuristic issue, the consideration of opting to utilize NumPy arrays from the Numpy library to access the matrix rather than the python lists would benefit the runtime. The NumPy array parses through the matrix faster than the lists, so it would cut down on the time spent computing the heuristic or accessing the matrix or array to check. The question remains how much more significant is the time difference using the NumPy array than the list for the algorithm. If it doesn't improve the performance to the extent that we could run a size 18 or 25 matrices within 10, then questioning whether its implementation is worthwhile would be up for debate later.

Appendix

Each benchmarking problems would be run using SLS-2-opt with 1000 iterations. The algorithm would stop whenever it reaches 1000 iterations or exceeds 10 minutes, whichever comes first.

SLS	Runtime	Tour Length
Size 25	849 msec	1348.8
Size 50	3924 msec	2128.02
Size 100	0.28 min	4179.76
Size 200	0.66 min	17725.96
Size 300	2.8 mins	11675.45
Size 400	5.2 mins	14059.78
Size 600	~ 10 mins	23538.81
Size 800	~ 10 mins	69378.38
Size 1000	~ 10 mins	30795.96

References

GeeksforGeeks. “Traveling Salesman Problem Using Branch and Bound.” *Geeks for Geeks*, 31 Oct. 2022, www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2.

Wikipedia contributors. “2-opt.” *Wikipedia*, 30 Sept. 2022, en.wikipedia.org/wiki/2-opt.

Venhuis, Mikko. “The Basics of Search Algorithms Explained With Intuitive Visualizations.” *Towards Data Science*, 12 Feb. 2019, https://towardsdatascience.com/around-the-world-in-90-414-kilometers-ce84c03b8552.