Branch and Bound, Back-tracking, Depth-first Search
&
Local Search

Group #29

Micah Fadrigo, Student ID #: 42923836
Cecilia Nguyen, Student ID #: 44328584
Ni Ki Chong, Student ID #: 64322012

1 Branch-and-Bound, Back-tracking, Depth-First Search

Overview

The **problem** we are trying to solve is: given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? Since we are essentially making a full circle, it does not matter what node is designated as the “origin city” or “start node”. For simplicity purposes, when reporting results, we always start from node 0, the first node given in the distance matrix.

The state-space consists of all paths, starting from the start location and ending at some other location, without necessarily going through all the locations. This algorithm utilizes **backtracking (depth-first search)** to reach a complete solution. It simultaneously utilizes **branch-and-bound** to prune partial solutions if they are worse than our known (or current) best solution. In this algorithm, a partial solution is only pursued if its total estimated cost is less than the cost associated with the current best solution. The total estimated cost of a path through a **node n** is calculated via the **sum** of the known cost of the path to reach **n** and the heuristic evaluation at **n**.

Our heuristic is based on our knowledge of the **cheapest edge in the graph** (excluding self-loops). Our heuristic is **admissible** because given the minimum edge cost in the graph **m**, if there are **x** remaining nodes left to visit on a particular path, then the minimum additional cost to visit the rest of the remaining nodes for that path is (**m**)*(**x**).

The Branch and Bound algorithm utilizes a **stack** *to* contain the lists that represent partial solutions. When a partial solution is popped from the stack, its total estimated cost is evaluated and depending on whether or not that cost is *less than* our “current-best” minimum cost, that partial solution is either pursued further or reaches a dead end (respectively). In the case that a partial solution is pursued further, the algorithm considers all “next” nodes from the last real decision point (last node of the path), and pushes the new possible partial solutions (previous partial solution + a “next” node) to the stack. When the stack is empty, the function will return the best path that is found.

Experiment & Results

Our algorithm seems to work in reasonable time to find the optimal path, however, it becomes inefficient with a larger graph. For example, for a graph showing paths with 5 or 10 cities in total, it requires only less than a second to find the optimal path, as shown below.

```
----------------------------------------------------------------------
Ending cost: 9.967799857258797
Ending path:
0 → 2 → 4 → 3 → 1 → 0
----------------------------------------------------------------------
Total time taken for 5 cities: 0.000499725341796875 seconds
----------------------------------------------------------------------
Solution with path cost 9.967799857258797
```

```
----------------------------------------------------------------------
Ending cost: 2.673500020056963
Ending path:
0 → 4 → 9 → 2 → 7 → 3 → 5 → 1 → 6 → 8 → 0
----------------------------------------------------------------------
Total time taken for 10 cities: 0.46094298362731934 seconds
----------------------------------------------------------------------
Solution with path cost 2.673500020056963
```

However, for a larger number of cities, the runtime becomes exponential. For example, a number of 50 cities or 100 cities would take more than two hours running. This might be because the BnB DFS algorithm isn't using the most efficient heuristic. Our heuristic cost, or distance from a node n to the "goal" node (in this case, the origin city) is calculated by multiplying the cheapest edge in the distance matrix by the number of unvisited nodes (remaining nodes to visit in order to have a complete path). This heuristic is more useful and prunes a little more aggressively than a heuristic = 0. An algorithm with such a trivial heuristic would simply backtrack to every last real-decision point and choose the next available node so long as it is less than the current upper bound.

Explanation of Results & Further Improvements

Given different graphs of different sizes to traverse, there might be different time complexities. There is slower performance if a distance graph contains more edges; for example, if an optimal path is encountered later in the distance graph. If our heuristic is sub-optimal and nearly all paths in the beginning of the algorithm are similar, nearly all different permutations of the path may need to be explored for the most part.

The algorithm could be optimized for the worst-case scenario by improving a better heuristic. An admissible heuristic that is able to get a more accurate estimate of how close a partial solution is to the goal state would result in pruning higher in the search tree, and thus cutting off large portions of the search tree. Therefore, it would reduce total runtime massively and cause the algorithm to find the shortest path much faster.

We may be able to improve our algorithm by reducing the costs of the entire matrix in general. The reduction of the distance matrix by performing row and column reduction separately (subtracting values in rows and columns by minimum value), will result in our row or column to contain at least one 0. By applying this matrix reduction, instead of having to update our current cost with to compute the lower bound, the lower bound of through each cities left for a possible path would be zero, saving additional calculations. Then, a comparison between the cost of the partial path solution and the current best full solution, where the cost of the partial path is greater, can help prune non-viable paths.

Analysis & Conclusion

The time complexity of a depth-first search branch and bound with our heuristic of calculating our h(n), the estimated cost from the current node to the goal, depends on when the depth-first search encounters the optimal path. The time complexity would be the order in which our algorithm encountered the optimal solution, the worst time space complexity would be O(2^n), exponential time complexity, as we would be utilizing depth-first search, in which we may encounter the worst path cost, in an order from worst-cost to best-cost. This would require traversing too many branches to obtain the optimal path cost. Oppositely, if we encountered the optimal solution in the beginning of the algorithm, the algorithm would prune the other paths more aggressively.

With the branch and bound algorithm, time complexity is strongly dependent on the quality of the admissible heuristic. However, the order of encountering the optimal solution also strongly impacts the runtime.

2 Stochastic Local Search

Overview

Simulated Annealing begins with a high frequency randomization as the temperature is high and gradually slows the randomization process as the temperature decreases. The “driver” of our implementation of local search revolves around the usage of the Metropolis acceptance criterion. In doing so, the annealing temperature and the metropolis probability are large contributing factors in the process of path swaps when solving for the optimal path in the Travelling Salesman problem. Gradually, the annealing process slows and the number of accepted path swaps will reach to 0, that is when the algorithm ends and the local best path is returned.

Experiment & Results

For the purpose of experimentation, a few factors were tuned in order to identify the impact on the results in solving the Travelling Salesman problem. In this section, we present the results of each experiment, namely changing the annealing temperature, the number of max paths allowed, and the number of successful changes permitted at every iteration.

In recognition of the randomness that Local Search has, the experiment was run 10 times for each test and results were averaged in order to determine shortest distance obtained with the tuned parameter. Given the scope of the project and the computational limitation, the problem was modeled with 10 cities.

The following figures are the top 5 results of each parameter change, and the resulting distance.

Max changes	result
0.6	2.792199978977440
0.2	2.889950006082650
0.8	2.972919995710250
0.5	2.9739699997007800
0.7	2.984499991312620

Fig 2.1

Temperature	result
0.7	2.7575200024992200
0.3	2.827299973368650
0.8	2.9026600036770100
0.6	2.9039200112223600
0.4	2.9091099958866800

Fig 2.2

Max path	result
200	2.708979990705850
100	2.8870899852365300
150	3.01507000438869
50	3.1843699865043200

Fig 2.3

A similar approach in measuring the different combinations of tuned parameters, with its results as below.

Temperature, Max changes, Max paths	result
0.6, 0.5, 100	2.6394199926406100
0.7, 0.6, 200	2.6536899898201200
0.8, 0.6, 200	2.6607500065118100
0.7, 0.2, 200	2.6637700028717500
0.5, 0.3, 200	2.664500005915760

Fig 2.4

Analysis

Given the size and limitation of this project, analysis and experimentation has been narrowed to 10 cities for the Travelling Salesman problem, and the following analysis applies to only 10 cities.

Maximum number of paths evaluated

For this variable, we consider the maximum degree of which the number of path swaps are being evaluated at every iteration; Given 10 cities and a maximum path value of 100, at each temperature, the maximum number of iterations tested will be 1000 (number of cities x maximum path value).

Based on fig 2.3, it appears that the higher the maximum number of iterations, the better the algorithm performs. For this variable, the lowest distance cost found was 2.71, at max path value of 200. This aligns with the nature of Local Search, whereby the randomness will ultimately lead to an optimal solution given that the algorithm runs as long as it can. Here, we see that the difference between each proposed max path value is fairly significant (compared to the other parameter results), which indicates that in order for this algorithm to produce a desirable result this variable would ideally need to be maxed out.

Temperature

For this variable, we consider the annealing schedule at which the algorithm follows. This variable dictates the rate at which the algorithm performs; Depending on the temperature, its rate of annealing can determine whether the “optimal path” is found quicker or slower. This variable determines the temperature at which the algorithm begins with, and at every iteration this variable is exponentially decaying by a factor of 0.9.

Unlike the max path variable, the temperature value behaves slightly differently, whereby the ideal starting temperature schedule is 0.7. The average (10 samples) distance cost found at this temperature was 2.76. At the remaining temperatures, the algorithm appears to hover around an average distance cost of 2.9.

Maximum accepted path changes

This variable refers to the number of allowed swaps at each temperature during the annealing process. This value is represented by fractions, as opposed to whole numbers in order to take into account the varying number of cities in different TSP variations, i.e. for max changes at 0.5 and number of cities as 10, and number of max paths at 100, this value would be 500 $((number\ of\ cities\ *\ number\ of\ max\ paths)\ *\ max\ changes)$.

Based on the results collected, the maximum accepted path changes is optimally at 0.6 of the number of paths tried at every iteration. This yields a minimum cost distance of 2.79 for 10 cities.

Maximum paths, Maximum accepted path changes, Temperature

Additionally, distinct combinations of each variable were also tested in order to find the ideal combination of parameters. We find that at an annealing temperature of 0.6, number of accepted path changes at 0.5 and number of paths tried at every iteration at 100 times the number of cities, we get the lowest possible distance of 2.64.

It is worthy to note that upon re-running the process, the results yielded an entirely different set of combinations that resulted in the lowest distance cost, as seen below.

Temperature, Max changes, Max paths	result
0.1, 0.2, 200	2.5818599984049800
0.4, 0.3, 200	2.611009992659090
0.9, 0.4, 100	2.619549998641010
0.4, 0.8, 150	2.6296399895101800
0.5, 0.2, 200	2.632729995623230

Conclusion

While the results may indicate certain relationships between the proposed variables, it is inconclusive. Given the nature of local search, randomness is a huge factor in the results, and it is not guaranteed that the algorithm will produce the most optimal result.

Despite its variability, local search is certainly capable of producing a near-optimal result. For instance:

```
Starting cost: 7.900000091642141
Starting path:
1 → 3 → 8 → 9 → 2 → 0 → 4 → 6 → 5 → 7
----------------------------------------------------------------
Ending cost: 2.704399984329939
Ending path:
9 → 5 → 4 → 0 → 8 → 7 → 1 → 2 → 6 → 3
----------------------------------------------------------------
Total iterations: 56
Total time taken for 10 cities: 0.925349235534668 seconds
----------------------------------------------------------------
Solution with path cost 2.704399984329939 at temperature 0.0008216782349860233
```

Through randomly swapping the connections between cities, the algorithm produces a significantly more optimal path.

For the purpose of solving the traveling salesman problem, local search may not be a viable algorithm if the requirement is to consistently find the most optimal solution. However, in practice, there are other factors that would come into consideration in determining the best algorithm to solve this problem, i.e. computational efficiency. Simulated Annealing (Bertsimas, n.d.) begins the algorithm by performing a high number of random swaps but will quickly reach a stable state where a near-optimal solution will be found. In other words, local search is able to reach a near-optimal solution fairly quickly compared to other algorithms, but the trade-off is that the most optimal solution is not guaranteed.

Further improvements

Despite the randomness of local search, there are certain factors that can be controlled in order to improve the performance of the algorithm, specifically Simulated Annealing.
The annealing schedule (temperature) plays a role in the rate at which the algorithm finds a solution. Moving forward, it might be worthwhile to explore different variations of annealing temperature and different rates at which it decays. For the current solution presented, we start at an annealing temperature of 0.6 and reduce it by 0.9 times after each iteration.

BnB and SLS

In terms of comparisons between the two algorithms, branch-and-bound dfs might be a more efficient algorithm, as local stochastic search starts with a random start and might not reach the most optimal path, settling for the “local” optimal path after some iterations. With branch-and-bound, we are guaranteed to find the best solution throughout the entire graph. Both algorithms can be implemented to solve the traveling salesman problem.

References

Stochastic Local Search

Bertsimas, D., Tsitsiklis, J. (n.d.). Simulated Annealing. Retrieved November 20, 2022, from http://web.mit.edu/people/jnt/Papers/J045-93-ber-anneal.pdf

Geng, X., Chen, Z., Yang, W., Shi, D., & Zhao, K. (2011). Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search. Applied Soft Computing, 11(4), 3680–3689. https://doi.org/10.1016/j.asoc.2011.01.039

Vitali, T., Mele, U.J., Gambardella, L.M., Montemanni, R. (2022). Machine Learning Constructives and Local Searches for the Travelling Salesman Problem. In: Trautmann, N., Gnägi, M. (eds) Operations Research Proceedings 2021. OR 2021. Lecture Notes in Operations Research. Springer, Cham. https://doi.org/10.1007/978-3-031-08623-6_10

Zhong, Y., Lin, J., Wang, L., & Zhang, H. (2018). Discrete comprehensive learning particle swarm optimization algorithm with metropolis acceptance criterion for traveling salesman problem. Swarm and Evolutionary Computation, 42, 77–88. https://doi.org/10.1016/j.swevo.2018.02.017

Branch and Bound DFS

Clausen, Jens. "Branch and bound algorithms-principles and examples." Department of Computer Science, University of Copenhagen (1999): 1-30.

Kask, Kalev. “Branch and Bound DFS.”

“Travelling Salesman Problem: Greedy Approach.” GeeksforGeeks, 5 Jan. 2022, https://www.geeksforgeeks.org/travelling-salesman-problem-greedy-approach/ .

Zhang, Weixiong. "Truncated and anytime depth-first branch and bound: A case study on the asymmetric traveling salesman problem." AAAI Spring Symposium Series: Search Techniques for Problem Solving Under Uncertainty and Incomplete Information. Vol. 99. AAAI Press Menlo Park, CA, USA, 1999.