Engineering Algorithms for Large Network Applications

Abhishta G. Adyatma

1 Introduction

Large networks are characterized as $G = (V, E)$ where there exists an abundant amount of Vertices (V) and Edges (E). These networks are often used to model real-world structures like roads, railways, air traffic, and social networks. Traversing these vast networks poses a significant challenge, particularly when dealing with millions of vertices and edges. Within this report, the author delves into several approaches aimed at efficiently solving single-source single-target query (shortest path problems) with reducing the search space (number of vertices visited) of the Dijkstra's Algorithm.

As discussed in [4], the resolution of single-source single-target path problems on expansive networks typically revolves around two primary approaches. The first method, termed 'Graph Annotation,' involves adding supplementary information to the vertices and/or edges within the graph. This information facilitates acceleration techniques aimed at discerning which vertices require no further exploration. Variants of this approach may encompass heuristic computations aimed at estimating vertex distances. Additionally, certain variations involve pre-computations that establish constraints and enable the pruning of vertices during the search process [3].

The second approach, termed 'Auxiliary Graph' involves the construction of a secondary graph from the original graph to aid in solving specific problems or performing certain operations more efficiently. Auxiliary graphs simplify complex problems, enable efficient algorithmic solutions, or provide alternative representations for specific analyses. Variants of this approach may encompass the construction of an *Overlay Graph* [1], Spanning Trees, or using methods of Graph Aggregations [2] to fabricate a higher-level abstraction.

This report aims to conduct a benchmarking analysis of multiple approaches using a large network dataset. The goal is to delve into the performance and constraints of previously introduced approaches and their algorithm implementations. The author's focus will be on key metrics, including the Average Visits (number of vertices visited), Average Query Time (time taken to solve a shortest-path problem), and in selected implementations, the Average Resolution Time (time taken to complete a single vertex computation).

2 Dataset

In this report's experimentation, the author chose to benchmark the algorithms using the 9th DIMACS Implementation Challenge - Shortest Paths. This dataset provides 12 representations of the USA road networks. The primary focus of this report is on two specific networks: the New York Road Network, comprising 264,346 vertices and 733,846 edges, and the San Francisco Bay Area Road Network, containing 321,270 vertices and 800,172 edges. Three files are mainly found on each dataset, the Distance, the Time, and the Coordinate of each edges and vertices. The author will focus on the Distance information for benchmarking.

Dataset	Num. Vertices	Num. Edges
NY	264,346	733,846
NY (50K)	20,386	50,000
NY (10K)	4,154	10,000
BAY	321,270	800,172
BAY (50K)	18,322	50,000
BAY (10K)	4,164	10,000

Table 1: Dataset Composition

Additionally, to facilitate memory-intensive pre-computations and/or run-times required by certain algorithms, the author created subset variations of each road network of 50,000 and 10,000 edges. These information are displayed in Table 1.

3 Algorithms

The author experimented with several algorithms specific to Graph Annotation and Auxiliary Graph approaches aimed at minimizing the Average Visits and Average Query Time of the baseline algorithm. The author chose Dijkstra's Algorithm to be the best line since there were no negative weights in the experimented dataset. The following comparison algorithm shows a variety of approaches from Heuristics (Approximation), Graph Annotation through Containerization of Vertices, and Minimizing the Graph through an Auxiliary Graph.

3.1 Dijkstra (Baseline)

Dijkstra's algorithm serves as the baseline of this analysis. It's implemented in a single-source, single-target setup, designed to conclude the search once the target vertex has been visited. This particular implementation employs a Binary Heap Min Priority Queue, offering a time complexity of $O(E\ log\ V)$. Overall, the goal of the next few implementations are to reduce the baseline search space (Average Visits).

3.2 A* Haversine Distance

Algorithm 1 Relaxation with Haversine Distance Heuristic

function RELAX(u, v, e, w, H)
  if $d[v] > d[u] + w(u, v) + H(v, e)$ **then**
    $d[v] = d[u] + w(u, v) + H(v, e)$ ▷ H is the Haversine Formula, e is the End Vertex
  end if
end function

Given the dataset has information on the vertex geographic coordinates, the author pursues an approach to construct a heuristic that estimates the neighboring vertex against the end vertex during the search. These additional information supplemented to the vertices enters the approach of Graph Annotation where it can make calculations to extend the baseline algorithm further. To approximate the distance between neighboring vertex and end vertex, the Haversine Formula (defined by Equation 1 & 2) is used on both geographic coordinates.

$$hav(\theta) = \sin^2\left(\frac{\Delta \text{lat}}{2}\right) + \cos(\text{lat}_1) \cdot \cos(\text{lat}_2) \cdot \sin^2\left(\frac{\Delta \text{lon}}{2}\right) \tag{1}$$

$$distance = 2 \cdot R \cdot \text{atan2}\left(\sqrt{hav(\theta)}, \sqrt{1 - hav(\theta)}\right) \tag{2}$$

This approximation is reliable for every calculation within the experimented dataset because the weight function correlates with the Haversine Distance, giving the algorithm clear approximations between the source and the target. Moreover, Algorithm 1 shows a modified 'RELAX' function with addition of the heuristic of 'H(v, e)' which calculates the distance. 'H(v, e)' will continuously be smaller if the neighboring vertex is closing in to the end vertex. Intuitively, this will decrease the Average Visits per Query with this additional information being calculated. Additionally, the computation of this heuristic have a time complexity of $O(1)$ which with Dijkstra's Algorithm implementation remains $O(E\ log\ V)$.

3.3 Geometric Containers

The approach explored by [3] focuses on minimizing the search space further through methods of pre-computations. This method involves constructing a 'container,' which comprises of sets of vertices strategically chosen to enhance efficiency in subsequent shortest-path computations. In Figure 1, the algorithm iterates over all vertices to enlarge $C(u, v)$ with vertex encountered from the single-source of s. This will include all vertices visited from s to v to enter the container.

These containers play a pivotal role in restricting the search space of Dijkstra's algorithm, this is referred as 'Dijkstra Pruning', depicted in Figure 2. This adaptation of Dijkstra's method integrates the pre-computed containers. Specifically, if the end vertex t belongs to $C(u, v)$, then the algorithm navigates to vertex v. The success of this algorithm heavily hinges on the quality of the containers, which can fluctuate in size relative to the graph.

Moreover, this algorithm falls into the category of approach of Graph Annotation since it is supplying additional information between edges. Further analyzing the algorithms complexity, constructing the containers takes a time complexity of $O((E\ log\ V) * V)$ with a space complexity stated

```
 0   for all s ∈ V do
 1       for all nodes u ∈ V set  dist(u) := ∞
 2       initialize priority queue Q with source s and set  dist(s) := 0
 3       while priority queue Q is not empty
 4           get node u with smallest tentative distance  dist(u) in Q
4a           if u ≠ s enlarge C(A[u]) to contain u
 5           for all neighbor nodes v of u
 7                   set  new-dist :=  dist(u) + w(u, v)
 8                   if  new-dist <  dist(v)
 9                       if  dist(v) = ∞
10                           insert neighbor node v in Q with priority  new-dist
11                       else
12                           set priority of neighbor node v in Q to  new-dist
13                       set  dist(v) :=  new-dist
14                       if u = s
15                           set A[v] := (s, v)
16                       else
17                           set A[v] := A[u]
```

Figure 1: Geometric Container Construction (Wagner et al., 2005)

```
 1   for all nodes u ∈ V set dist(u) := ∞
 2   initialize priority queue Q with source s and set dist(s) := 0
 3   while priority queue Q is not empty
3a       if u = t return
 4       get node u with smallest tentative distance dist(u) in Q
 5       for all neighbor nodes v of u
5a           if t ∈ C(u, v)
 7               set new-dist := dist(u) + w(u, v)
 8               if new-dist < dist(v)
 9                   if dist(v) = ∞
10                       insert neighbor node v in Q with priority new-dist
11                   else
12                       set priority of neighbor node v in Q to new-dist
13                   set dist(v) := new-dist
```

Figure 2: Dijkstra with Pruning (Wagner et al., 2005)

Procedure `min-overlay`(G, ℓ, S)

For each vertex $u \in S$, run Dijkstra's algorithm on G with pairs (ℓ_e, σ_e) as edge weights, where $\sigma_e := -1$ if the tail of edge e belongs to $S \setminus \{u\}$, and $\sigma_e := 0$ otherwise. Addition is done pairwise, and the order is lexicographic. The result of Dijkstra's algorithm are distance labels (ℓ_v, σ_v) at the vertices, where $(\ell_u, \sigma_u) := (0, 0)$ in the beginning. For each $v \in S \setminus \{u\}$ we introduce an edge (u, v) in E' with length ℓ_v if and only if $\sigma_v = 0$.

Figure 3: Min Overlay Procedure (Holzer et al., 2008)

to be $O(E * V)$. This is due to computing for all vertices. As for the search algorithm, the time complexity remains $O(E\ log\ V)$.

3.4 Min Overlay Graph

Constructing an Overlay Graph, defined as a graph $G = (V, E)$ on a subset $S \subseteq V$ is a graph with vertex set S and edges corresponding to shortest paths in G [1], would create a representation of the original Graph with a selection of vertices and edges that is minimal to all-pairs within S. This type of approach is an example of an Auxiliary Graph approach, where a secondary graph is constructed to either aid or solve the problem entirely.

Figure 3 illustrates the construction process of the Min Overlay Graph. This method leverages a customized version of the Dijkstra Algorithm, incorporating adjustments to handle pairs of (ℓ_e, σ_e). These pairs guide the algorithm's decision-making during the search phase to determine whether a specific vertex should be included in the set of edges E' connected to vertex u or not. Specifically, the value of σ is modified to -1 selectively during the exploration process when the algorithm encounters a current vertex $c \in V$ and a neighboring vertex $n \notin S$.

With this algorithm, constructing a Min Overlay Graph would have a time complexity of $O((E\ log\ V) * S)$. However, this report's implementation towards this approach only explores the construction of the Min Overlay Graph such that any query $v \in S$ has a time complexity of $O(1)$. Looking at variations of similar implementations, this algorithm can also be paired with Dijkstra to search a shortest-path between two vertices where $(u, v) \notin S$. Ultimately, this algorithm can speed up a shortest-path problem partially if $S \subset V$ else then Dijkstra with a Min Overlay Prioritization can aid to reduce the search space.

	Dijkstra (Original)		Dijkstra (Baseline)		A* Haversine		Geo. Container	
	Visit	Time	Visit	Time	Visit	Time	Visit	Time
NY	264,346	277.19ms	131,828	132.87ms	**29,195**	**47.6ms**	-	-
NY (50K)	9,346	5.69ms	6,737	4.04ms	5,277	5.5ms	**52**	**2.57ms**
NY (10K)	3,682	2.17ms	1,934	1.12ms	684	651.68µs	**44**	**435.56µs**
BAY	321,270	313.8ms	160,431	155.2ms	**82,375**	**129.64ms**	-	-
BAY (50K)	10,880	7.61ms	6,824	4.87ms	4,961	5.9ms	**62**	**3.31ms**
BAY (10K)	482	323.86µs	425	258.31µs	398	345.546µs	**3**	**17.5µs**

Table 2: Average Visits and Query Time in 5000 Benchmark Sets

4 Experimental Results

4.1 Setup

For this report's experimentation, the author analyzes the aforementioned algorithms using an Apple M1 Chip with 8 GB Memory. All algorithm implementations are using Go 1.18+ with base implementations (no external libraries are used). The scope of analysis this report could achieve is bounded to this hardware specification. The experiment is conducted on 6 Benchmarks randomly created to form a Test Case of 5000 Shortest-Path Problems each. Source code can be found at Github : abhihtagatya/graf

4.2 Average Visits and Query Time

The author's objective to reduce the number of visited nodes per query, which represents a single shortest-path problem is woven into the design of almost all the aforementioned algorithms. The total number of visits and query time per test set iteration are aggregated and subsequently averaged across the benchmark set's size to derive meaningful metrics. The algorithms considered for this metric includes the Dijkstra (Original Implementation), Dijkstra (Baseline), A* Haversine, and the Geometric Container algorithms.

Table 2 displays the performance of each algorithm to perform in each benchmark sets. The author also includes the original implementation of Dijkstra's algorithm compared to the baseline implementation of single-source single-target Dijkstra's algorithm to visualize the how a single visit to the target vertex can conclude the minimal cost between a source vertex and a target vertex.

Comparing the baseline with the A* Haversine algorithm displays that the applied heuristics to estimate the cost between a neighboring vertex to the end vertex can improve the algorithm further by $\sim$ 41%. However, there has been cases recorded that shows the Average Query Time on A* Haversine algorithm is slightly higher than the baseline even though the search space is reduced. Hypothesis towards this might be due to cases where a shortest-path is not found (therefore has a distance of ∞) and thus the trying to compensate by finding unnecessary vertex.

The Geometric Container algorithm has reduced the search space significantly with a reduction of $\sim$ 99% compared to the A* Haversine algorithm. However, this algorithm comes with a heavy pre-computation to be able to be optimized for a specific graph. Unfortunately, this experimental setup is not capable of computing the containers for the New York Road Network and San Francisco Bay Area Road Network. Using this experimental setup, it is estimated to pre-compute both dataset with a computation time of over $\sim$ 73 Days with around 16 GB which is 4,776x the size of the original graph. Additionally, it is stated in [3], that a change of weight or minor changes to layout might require an update on the pre-computed containers which shows the limitation of this algorithm.

	Min. Overlay
	Time
NY	14.13s
NY (50K)	576.45ms
NY (10K)	68.57ms
BAY	14.19s
BAY (50K)	659.8ms
BAY (10K)	19.68ms

Table 3: Average Resolution Time in 5000 Benchmark Sets

4.3 Average Resolution Time

The author's objective to reduce the number of visited nodes per query can also be achieved through methods that does not follow searching iteratively. Through the auxiliary graph approach, it leverages methods that computes the problem (whether entirely or a subset) to create a subgraph that answers queries given some pre-computations. An example of that implementation is the Min Overlay Graph, which creates an overlay graph with all edges being the shortest-path to each. To measure the performance of this algorithm, a metric of Average Resolution Time is introduced where a Resolution Time is the time taken to resolve all queries for a single vertex $u \in S$.

5 Conclusion

In the pursuit of forming solutions for large network applications, both approaches of Graph Annotation and Auxiliary Graph underwent extensive exploration and evaluation within the 9th DIMACS Implementation Challenge of Shortest-Paths Dataset. The findings revealed an interesting discovery: achieving a significant reduction in search space necessitates computationally intensive algorithms and pre-computations. This is clearly demonstrated in the implementation of Geometric Containers, where the trade-off between computational expense and reduced search space is discussed. Moreover, the usability of heuristics is illuminated through the A* Haversine algorithm, emerging as the least computationally demanding method while still significantly reducing the search space. Another method explored was creating a minimal subset representation of the original graph. This concept is exemplified by the implementation of the Min Overlay Graph.

Given the increasing demand for optimized algorithms in handling vast networks, future works in this realm might delve into hybrid implementations of previously studied methodologies or experiment with stacking these implementations and/or heuristics.

References

[1] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multi-level overlay graphs for shortest-path queries. *ACM Journal of Experimental Algorithmics*, 13, 09 2008.

[2] Sami Sieranoja and Pasi Fränti. Adapting k-means for graph clustering. *Knowl. Inf. Syst.*, 64(1):115–142, jan 2022.

[3] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. Geometric containers for efficient shortest-path computation. *ACM J. Exp. Algorithmics*, 10:1.3–es, dec 2005.

[4] Christos Zaroliagis. *Engineering Algorithms for Large Network Applications*, pages 272–274. Springer US, Boston, MA, 2008.