Efficient Route Planning

matkr, cemn

November 2024

1 Introduction

This assignment focuses on implementing and analyzing algorithms for the shortest path problem across large datasets. Starting with Dijkstra's algorithm, we implement our own version, including an early stopping criteria, that terminates the search once shortest path to the target vertex has been reached. To further improve the running time, we introduce a Bidirectional Dijkstra implementation, which improves efficiency by simultaneously searching from both the source and target vertices. Furthermore, we explore contraction hierarchies preprocessing to produce an augmented graph to significantly speed up query times on large graphs by reducing the search space. At last, we compare the performance and correctness of our implementations, and discuss the choices made in the implementations. Implementation can be found here.

2 Dijkstra and Bidirectional Dijkstra

The Dijkstra implementation is based on the algs4 implementation, and is extended by adding an early stopping criteria. The stopping criteria, stops the algorithm early, and still ensures that the optimal path is found as soon as the target node is settled. This is ensured by stopping the search, as soon as a shortest path is found, and all other possible routes are longer than this path. To test the implementation, we ran 1,000 different source/target pairs on the graph of Denmark, asserting the shortest path found remains the same as the algs4 implementation. The average running time of the algorithm is 130 ms with an average number of relaxed edges at 592,685.
To further improve the Dijkstra implementation, we introduce a Bidirectional Dijkstra, also with an early stopping criteria. In theory, this halves the running time. By benchmarking this implementation with the same 1,000 source/target pairs, we now have an average running time of 95 ms with an average number of relaxed edges at 431,397. In theory the running time of bidirectional dijkstra should be around half of the dijkstra implementation.

3 Contraction Hierarchies

Contraction in this context of route planning is a preprocessing step in which vertices in a graph are progressively contracted to optimize query times. When a vertex is contracted, shortcuts are added between its neighboring vertices, when no other shorter path exists, to preserve shortest paths that would have been removed traversing it. This step minimizes the number of vertices and edges involved during queries, reducing query times.

3.1 Implementation of the preproccessing phase

3.1.1 Ordering of vertices

In our implementation, the preprocessing phase starts by simulating a contraction of all vertices, to give them all a rank, based on the amount of simulated shortcuts and the degree of the vertex, namely the edge difference. This gives a preliminary ordering of the vertices, which we use when actually contracting vertices. The ranks of the vertices are then again calculated, now re-ranking them one at a time with the extra heuristic of deleted neighbors. The ranking heuristics are dynamically updated throughout the process, which changes the order of which vertices should be contracted. If the newly calculated rank of the top vertex in the queue is higher than the next vertex's rank, we postpone a full recalculation of the ranks in the queue by moving the current top vertex back into the queue and proceed with the next vertex instead. This approach allows the queue to adapt to the new ranks without immediately recalculating the entire queue. However, we cap these lazy updates to 25 occurrences, after which we fully recalculate the queue to prevent ranks from drifting too far out of sync with their preliminary ranks. This optimization reduces the overhead of frequently recalculating the entire queue, while avoiding the costly task of continuously updating each vertex's position in the queue lazily.
The rank of a vertex is stored in an integer array of size equal to the amount of vertices in the graph. Storing the ranks like this allows getting the rank or updating the rank using array accesses in constant time.

3.1.2 Adding and removing edges

When contracting a vertex, we evaluate pairs of adjacent vertices to determine if shortcuts are needed between them. For each neighbor pair of the vertex, we check if their ranks are higher than the current vertex's, which preserves the hierarchy. If so, the method considers creating a shortcut with a weight equal to the sum of the two adjacent edges. A local Dijkstra search for a witness path, between the two adjacent vertices is performed, while ignoring the vertex being contracted, to check if a shorter or equal path already exists between the neighbors. If no such path exists, the shortcut is added.

Shortcut edges are added to the graph with an `isShortcut` flag, distinguishing them from original edges. This flag, which is added as an extension to the edge class, defaults to false for all original edges. Upon contracting a vertex,

we aim to remove its edges to prevent them from being reconsidered in future contractions. Since removing an edge from a large graph data structure can be costly if we naively traverse the adjacency list of a node in linear time, we instead ignore edges that have already been visited by some other local search, excluding them from subsequent searches during future vertex contractions. Edges are marked as visited, which also is an extension to the original Edge class from algs4. For each contraction, we check whether the adjacent edges are visited or not, and we then avoid to perform a local search, for any edges that already has been visited.

3.1.3 Efficient local search

We denote the amount of vertices in the graph as `V`. The Dijkstra local search is executed $O(V^2)$ times during the preprocessing phase, where each search typically explores only a small portion of the graph. Although these searches are localized, the full graph must still be available, as the search boundaries are not predetermined. Initializing a new Dijkstra instance for each local search across the entire graph would be highly inefficient, since each local search would have to execute a linear amount of work resetting an array. To address this, we reuse a single Dijkstra instance and efficiently reset it between searches, enhancing the preprocessing running time.

This is achieved through the use of epochs. At the start of the preprocessing phase, we initialize one Dijkstra instance. An integer array, `epoch`, of the same size as the number of vertices, is used alongside an integer `currentEpoch` to track each new search cycle. Every time a local search begins, `currentEpoch` is incremented to represent a new search phase. When an edge is relaxed, we check if the vertex's epoch value matches the `currentEpoch` . If it does not, we reset the vertex's `distTo` value in the array, performing a lazy update of `distTo` when we operate in a different epoch. This optimization significantly reduces running time by avoiding eagerly resetting the `distTo` array for each search.

In our local searches, we assume that they remain localized and therefore computationally lightweight. However, this assumption can break down in cases where long-distance edges are involved, potentially making some local searches more time-consuming. To address this, we introduce specific limits that truncate overly extensive local searches, while still preserving accurate shortest path distances. Although these constraints may lead to the creation of some superfluous shortcuts, the integrity of shortest path calculations and preprocessing is maintained.

To prevent excessive running times in local searches, we enforce a limit of 100 settled nodes. This limit means that each search will relax edges from a maximum of 100 nodes before termination, thereby avoiding prolonged searches. As a trade-off, this may lead to additional shortcuts that could become unnecessary as other nodes are contracted later. Nonetheless, this approach strikes a balance between faster preprocessing and manageable shortcut generation.

Additionally, we implement a staged hop limit, which scales with the average

degree of the node being contracted. As recommended in [1], we start with a low hop limit for initial searches and increase it based on node degree. For nodes with an average degree of up to 3, comprising roughly the 200,000 least significant vertices, we apply an initial hop limit of 5. This limit increases to 30 hops for nodes with a degree of 3, to 50 hops for degree 4, and finally to 200 hops for nodes with degrees higher than 4.

While introducing these limits may produce some superfluous shortcuts, the combined effect of quicker preprocessing and the re-evaluation of redundant shortcuts in later contractions improves the overall performance of the algorithm without compromising the accuracy of shortest path calculations.

A visual representation of the shortcuts created for the Denmark graph can be found in the appendix. 1

3.2 Testing of contraction

To test that our implementation works as intended, we have implemented various unit tests to confirm this. Simple test cases include that our parser logic is correctly by asserting the number of edges and vertices are equal to the numbers in the test file, and that the graph contains the edges specified in the file. To test whether the query on the contracted graph works, we use a Bidirectional Dijkstra on the augmented graph with shortcuts, and asserts that we get the same shortest path as the Dijkstra and the Bidirectional Dijkstra finds on the original graph. This is a crucial test, for an overall view at whether our implementation works. Unit tests for more specific subparts of the algorithm can be found at (app/src/test/java/org/algo-bench/algorithms/shortestpath/ShortestPathTest.java).

3.3 Performance

Unfortunately, our implementation of the contraction hierarchy has a bug, meaning that it does not find the correct shortest path. We either find an almost shortest path or the correct shortest path when using the augmented graph with shortcuts, which means that the bug most likely lies in the early stopping criteria of the localsearch. More precisely, the local search is limited by settling at max 100 nodes. If that limit is reached, we return true which results in no shortcuts created. We are aware that this specific case is our bug, and that in this case we should instead break from the loop, and create a shortcut if the shortest path is not found. Although, when trying out this solution, our algorithm runs for too long, and does not terminate. For preprocessing the graph The query time of searching in the augmented graph, with our modified bidirectional dijkstra which only searches upwards in rank, is on average 12 ms per query and it relaxes 14,762 edges.

References

[1] DOMINIK SCHULTES ROBERT GEISBERGER, PETER SANDERS and CHRISTIAN VETTER. Exact routing in large road networks using contraction hierarchies. *Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany.*

Appendix

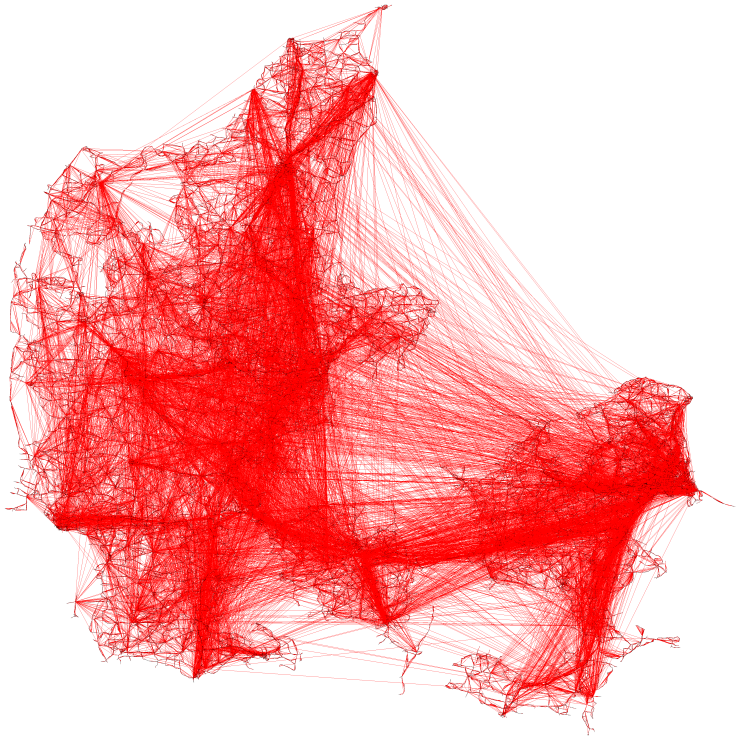

Figure 1: Denmark graph showing all shortcuts in red, original graph in black.