Assignment 3: Contraction Hierarchies

Andreas Riget Bagge & Marcus Skjold Pedersen

2024-12-04

This report documents implementation of the Contraction Hierarchies shortest path algorithm (CH algorithm) [1]. We have also implemented three common Dijkstra shortest path algorithm, and we will use them as reference for comparison.

1 The graph

Our experiments were designed to work on a graph of Denmark, provided as a list of nodes with coordinates and bidirectional weighted edges, representing a road network with some sort of travel time between connected locations.

We represent graphs as edge-weighted directed graphs through implementations of the interface `IndexedGraph`. For sake of space-efficiency, nodes are represented as array-indeces to their corresponding adjacency lists of edges, as seen in `WeightedDiGraph`. To handle locations of nodes, we store translations between locations and array-indeces in `LocationsGraph`. The graph of Denmark is represented through this implementation. The final `ContractedGraph` can contain either of the other graphs, but overrides the edge getting methods such that they only return edges leading the caller to higher ranks. This approach allows our bidirectional query algorithms to be decoupled from the specific graph, and we have reduced the CH problem to one of producing a correct contracted graph, given a bidirectional Dijkstra query algorithm.

2 Shortest path query algorithms

All our implementations are concerned with solving the shortest path problem (SPP), that is, given a source node, s, and a target node, t, finding a shortest path (if any) from s to t.

Dijkstra's algorithm solves the SPP by starting from a source node, s, and traveling through its edges in order of lowest total distance from s, computing distances to s from those nodes. In our (`DijkstraSimple`) implementation, the algorithm runs until all reachable nodes are settled. In (`DijkstraEarlyStop`), we add an early stopping criteria, such that the algorithm stops as soon as the target node has been settled.

We implemented two similar variants of a bidirectional Dijkstra algorithm to used for the CH algorithm. These work by running a Dijkstra query from both source node and target node. The one from the target node runs on the nodes *to* the target rather than *from* it. The algorithm stops after the two searches encounter the same node, and it is guaranteed that they cannot meet on another node with a shorter combined path. The shortest path is then the path from the source to the meeting node followed by the path from the meeting node to the target node. Our `DijkstraBidirectional` variant proceeds by order of the shortest distance from either of the source or target, whereas our `DijkstraInterleaving`, inspired by [1, p. 391], alternates between searching from the source and target.

We implemented a common `ShortestPathAlgorithm` interface. A single query using any class implementing this interface is represented as a single instance of that class. A query is supposed to be constructed with a matching graph. Every one of our algorithm runs on "normal" graphs, but only the bidirectional implementations work on "contracted" graphs (because the contracted graphs only return edges pointing to nodes of higher rank). The interface prescribes the following use of a query: A shortest path may be calculated by calling `calculate()`, and the results may be investigated by using any of `relaxedEdges()`, `distance()`, or `retrievePath()`. This interface-based approach made it easier to compare tests and experiments between different algorithms. However, the rather narrow API defined, and the decision to make each query an object with a state caused us to take a relatively more black-box approach to testing, making it harder for us to ensure correctness.

3 Contraction Hierarchies

Our implementation of contraction hierarchies (see `ContractedGraph.contractGraph()`), follows the approach presented in [1]. This consists of first assigning an initial *importance* value to each node, then contracting each node in order of that value, updating the values of neighboring nodes during each contraction. After contracting each node, we give each node a list of edges pointing *to* a node of higher rank, and a list of edges coming *from* a

node of higher rank. These correspond to the upward and downward graphs mentioned in [1, p. 391].

Performing a bidirectional Dijkstra search on this graph is expected to be significantly faster than any Dijkstra algorithm on a normal graph. In the following we present our implementation of this approach.

Contraction: We contract a node by adding shortcuts connecting such that all of its uncontracted neighbors remain connected with a shortest path of the same distance [1, pp. 388, 390]. Then we assign the node a unique rank marking how many nodes have been contracted before it.

We implement this by performing a shortest path search (implemented in `LocalDijkstra`), for each of the neighbouring nodes, ignoring the node under contraction, to find *witness paths* [1, p. 390]. When no witness paths are found, we add a `ContractedGraph.Shortcut` edge. For efficiency, we reuse the same query instance, and "reset" it by emptying its priority quere, and keeping track of a the search "generation", ignoring the distances found in earlier search generations.

Importance: Calculated as the sum of *priority terms*. We have attempted multiple, but as of now only managed to get improvements by using these two: The simulated edge difference of contracting the node, and the number already contracted neighboring nodes [1, pp. 392-393].

Lazy updates: It is impractical to always keep the importance values up to date. We lazily update the values in two cases – (1) As we are picking the next node to contract, and (2) after contracting a node [1, p. 392].
1: Uncontracted nodes are picked from a minimum ordered priority queue, ranked by their importance value. We choose the next node to contract by picking the minimum key from the priority queue, recalculating its importance, and then contracting it if it is still the least important. If too many attempts fail, we recalculate the importance values for every remaining node.
2: After a contraction, we recalculate the importance for each of its uncontracted neighbors.

Printing: We have made a print-client, `printGraphLocs` that prints the contracted graph. The contracted Denmark graph can be found in `contracted_denmark.graph`.

4 Testing

We have provided unit tests for most of the components in our project using JUnit. The graph classes and the first three SPP algorithms are thoroughly tested. We have tested against edge cases such as empty graphs, graphs with multiple shortest paths, disconnected graphs etc. For the stopping

conditions of bidirectional dijkstra, we have also tested for cases, where the first mutually reached node isn't necessarily on the shortest path.

We have tested components of our CH implementation, such as `LocalDijkstra`, finding uncontracted neighbors to a node and querying a contracted graph. The way we designed our API for CH made testing the components of CH difficult, and we have mostly evaluated in terms of the correctness of querying the contracted graph, making testing and bug fixing difficult. This is definitely something we will take into considerations in future projects, making more stateful API's etc.

5 Experiments and results

All experiments can be recreated by running the `Main` class manually or through the gradle `run` task.

Generating the graph from `denmark.graph` took less than 5 seconds, and contracting that graph took less than 14 seconds. We compared our implementations, by using them to compute the shortest path between the same 1000 pairs of randomly selected nodes in the Denmark graph. We measured their performance in terms of mean edge-relaxations and mean query-processing time. The experiment was run on a 2023 MacBook Pro (14 inc, Apple M3 processor, 24GB ram) connected to a power supply. The results can be seen in table 1.

Table 1: *Comparison of performance for our implementations of shortest path algorithms (average over 1.000 random point queries)*

Graph	Algorithm	Query time (ms)	Edges relaxed
Normal	DijkstraSimple	103.59	575241
Normal	DijkstraEarlyStop	53.24	287632
Normal	DijkstraBidirectional	50.29	232396
Normal	DijkstraInterleaving	45.11	208568
Contracted	DijkstraBidirectional	0.10	338
Contracted	DijkstraInterleaving	0.11	350

As expected, the simple algorithm performed worst, the others performed better, and algorithms on the contracted graphs perform orders of magnitude better.

Interestingly, there was not much of a difference between the early stopping variant and the bidirectional variant when running on the normal graph. We suspect this is because in some cases, a bidirectional search may perform

worse than a unidirectional. E.g. if the source node is located at a sparse point in the graph, with long distances and with many paths pointing to the source, and the target node is located at the edge (toward the source) of a dense hub with short distances (like a city). In this case, the bidirectional algorithm will relax many unnecessary edges inside the city. These cases may partially counteract the improvements of bidirectional search.

This theory might also explain why the interleaving variant performs better than the bidirectional on the normal graph, but not on the contracted. The inefficiencies of the above examples would be less pronounced in the interleaving example, as it does not give priority to many shorter edges of the dense network.

References

[1] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, August 2012.