Assignment 3: Contraction Hierarchies

Tomas Vemola, Miguel Angel Castro Gimenez, Urszula Matysiak

2025-11-12

1 Introduction

Finding the shortest and most efficient route in large-scale road networks is a fundamental problem in computer science, with applications ranging from navigation systems to logistics and traffic management. Traditional algorithms such as Dijkstra's algorithm guarantee optimal routes but become computationally expensive when applied to large graphs, such as real-world maps containing hundreds of thousands of intersections and roads.

To address these scalability issues, advanced preprocessing techniques like Contraction Hierarchies (CH) have been developed. CH accelerates shortest path queries by introducing shortcut edges and hierarchical node ordering, significantly reducing query times while maintaining exact results.

In this project, we implement and evaluate a variant of the Contraction Hierarchies algorithm proposed by Geisberger et al. (2012) on a real-world dataset representing the road network of Denmark. The work involves developing both a bidirectional Dijkstra algorithm and an optimized CH-based search, comparing their performance in terms of preprocessing time, query efficiency, and the number of edges inspected during searches.

2 Implementation

Our implementation is structured around three core classes: `Graph`, `BidirectionalDijkstra`, and `ContractionHierarchy`, which together provide the necessary functionality for efficient shortest path computation and Contraction Hierarchies pre-processing.

Graph class

The `Graph` class implements the core data structure representing the road network and provides utilities for contraction and pre-processing. Key meth-

ods include:

- `contract(long u)` – Performs the contraction of a vertex by removing it and adding shortcut edges between its neighbors, where necessary.
- `getEdgeDifference(long v)` – Calculates the change in the number of edges (added shortcuts minus removed edges) when contracting a vertex; used to determine contraction order.
- `preprocess(int checkInterval)` – Executes the pre-processing phase using the edge difference heuristic and lazy updates to build the contraction hierarchy.
- `AugmentedGraph(int checkInterval)` – Generates a copy of the original graph and augments it with all shortcuts and node ranks from pre-processing.

BidirectionalDijkstra class

The `BidirectionalDijkstra` class implements a bidirectional variant of Dijkstra's shortest path algorithm for more efficient distance queries. Key methods include:

- `distance(Graph g, long from, long to)` – Runs simultaneous forward and backward searches from the source and target. Uses two priority queues and terminates when the searches meet at a common node. Returns a `Result` object containing the distance, execution time, and number of relaxed edges.

ContractionHierarchy class

The `ContractionHierarchy` class handles the creation, storage, and querying of the augmented graph after pre-processing. Key methods include:

- `query(long s, long t)` – Performs a bidirectional CH query that only follows edges with increasing rank, reducing search space, and improving query speed.
- `storeGraph(Graph g, String filename)` – Writes the augmented graph to a file, including vertex ranks and shortcut information for visualization and validation.

In general, when implementing, we followed the task description and the provided paper. We had to do minor adjustments compared to pseudo-code when writing Bidirectional Dijkstra - instead of having one settled set we keep two maps - one for the left search and one for the right search.

Last but not least, even though the contracted nodes exist in the graph structure of our implementation, they are effectively bypassed by the rank-based filtering and shortcuts.

2.1 Pre-processing and architectural choices

The pre-processing of the Denmark graph took 1h and 35 minutes. The results are visible in Figure 1. It produced an augmented graph containing all original edges plus the added shortcuts, resulting in a total of `1,148,493` undirected edges (compared to `587,643` before pre-processing). During contraction, Dijkstra searches are performed locally between all pairs of neighbors of the vertex being contracted, using a distance limit equal to the potential shortcut weight; this ensures that a shortcut is only added if no shorter path already exists. In the final search, the query is performed on the fully augmented graph, which includes all original nodes and shortcut edges. The `query` method enforces the contraction hierarchy by only following edges that go upward in rank in the forward search and upward in reverse in the backward search. As a result, if a shortcut exists, it is automatically used to traverse contracted nodes indirectly without explicitly visiting them, ensuring correctness while efficiently reducing search space.

3 Tests and Experiments

3.1 Unit Tests Overview

We wrote unit tests for each method, and put them in the four test files, as described below:

- **DijkstraTest:** Tests the correctness of Dijkstra's shortest path implementation. Checks include basic path computations, direct connections, unreachable vertices, and self-path queries.
- **GraphTest:** Validates core graph operations such as vertex and edge addition, neighbor retrieval, graph copying, edge difference computation, vertex contraction, preprocessing, and augmented graph creation. Moreover, tests that stored graphs are correctly written to a file.

- **TestBidirectionalDijkstra:** Verifies the bidirectional Dijkstra implementation. Tests include basic shortest paths, larger example graphs, direct connections, and distance queries from a node to itself.
- **TestContractionHierarchies:** Checks the correctness of the contraction hierarchy preprocessing and query system. Tests include comparing CH query results with standard Dijkstra results, ensuring accurate distance computations for basic, larger, and direct paths, as well as self-node queries. It also verifies that preprocessed graphs can be correctly loaded from file.

3.2 Experiments

We ran experiments 2 experiments to compare Dijkstra vs Bidirectional Dijkstra and Dijkstra vs Contraction Hierarchies.

3.2.1 Methodology

For both experiments the same methodology was used. 1000 random pairs of vertices (start, target pairs) were selected from the input file, then fed to the main program with the following data: file to recreate the graph from, algorithm to be used ("CH", "D", "BD" or "PREPROCESS" to trigger the pre-processing of the graph to be used with Contraction Hierarchies) and the start, target pair (use "0 0" for pre-processing). The output from the main program consists of start vertex, target vertex, algorithm, time in ns, number of relaxed edges and distance. This output is recorded by our experiments and the results of the experiments we ran can be found in the following files:

- `/contraction-hierarchies-template/experiment_resultsCH.csv`
- `/contraction-hierarchies-template/experiment_resultsBDvsD.csv`

3.2.2 Dijkstra vs Bidirectional Dijkstra

As visible in Table 1 and Figure 2 Bidirectional Dijkstra reduces edge relaxations by 62.4% compared to standard Dijkstra (225,340 vs 599,755), confirming the theoretical advantage of meeting-in-the-middle search strategies. However, execution times are nearly identical (173 ms vs 169 ms), suggesting that managing two priority queues and coordinating bidirectional exploration offsets the benefits of fewer relaxations. Both algorithms paths of almost identical length, however, Bidirectional Dijkstra does not always find the shortest path, hence the average path distance is slightly higher.

	Time (ms)	Relaxed edges	Path distance
Dijkstra	169	599,755	9,597.83
Bidirectional Dijkstra	173	225,340	9,598.40

Table 1: Average results for a random seed of 1000 different pairs of vertices (start, target pairs) run on both algorithms.

3.2.3 Dijkstra vs Contraction Hierarchies

As seems obvious from Table 2 and Figure 3 Contraction Hierarchies achieve exceptional performance improvements over Dijkstra, reducing edge relaxations by 99.86% (400 vs 293,553) and execution time by 96.6% (6 ms vs 177 ms) while maintaining optimal path quality. The 29.5× speedup demonstrates that CH's preprocessing overhead translates directly into query-time efficiency, with the hierarchical structure enabling searches to explore only high-importance nodes. The log-log plot reveals clear separation between the two algorithms, with CH (green) forming a distinct low-complexity cluster compared to Dijkstra's (blue) higher computational demands. This dramatic improvement validates the core insight of Contraction Hierarchies: investing in preprocessing to create a node hierarchy and shortcuts yields substantial returns for repeated shortest-path queries on static graphs.

	Time (ms)	Relaxed edges	Path distance
Dijkstra	177	293,553	9,597.83
Contraction Hierarchies	6	400	9,597.83

Table 2: Average results for a random seed of 1000 different pairs of vertices (start, target pairs) run on both algorithms.

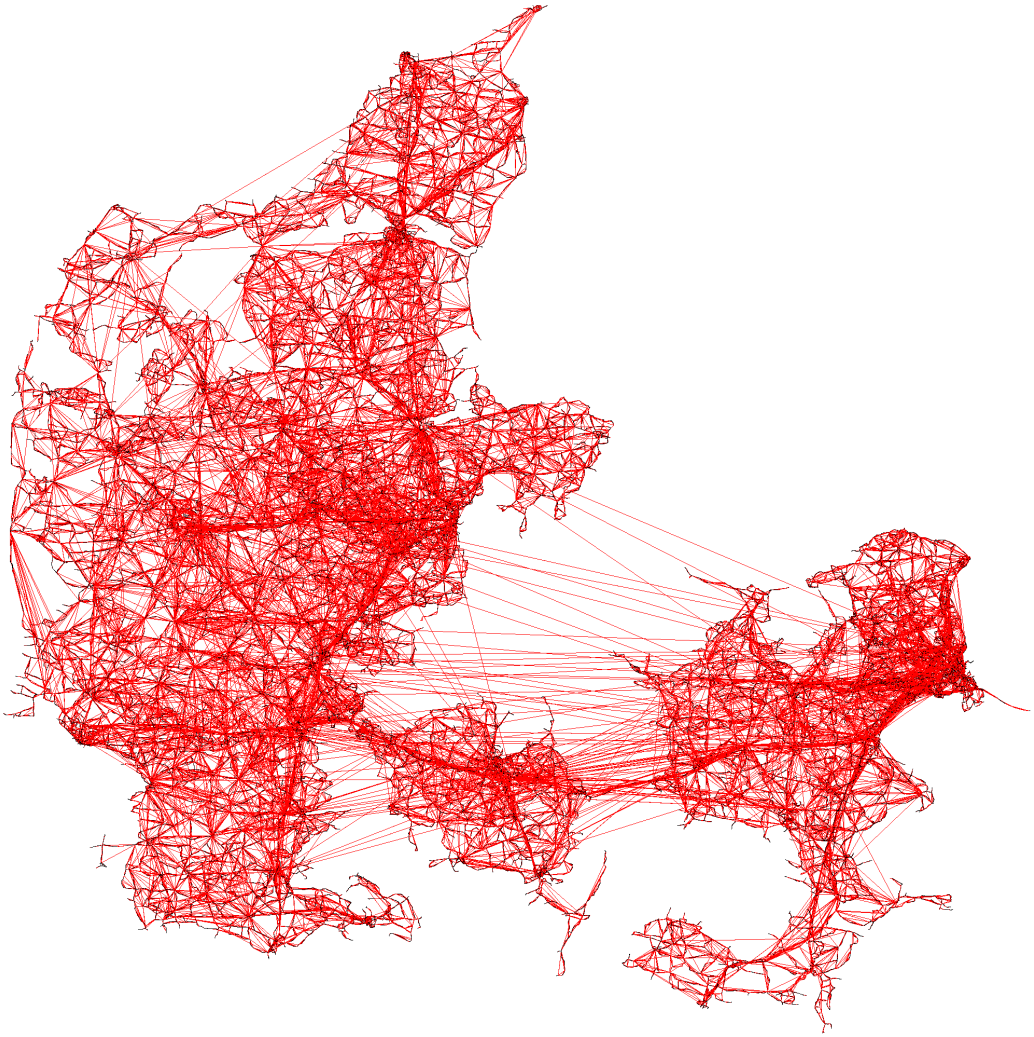

Figure 1: Augmented graph of Denmark with original edges, plus shortcuts resulting from pre-processing.

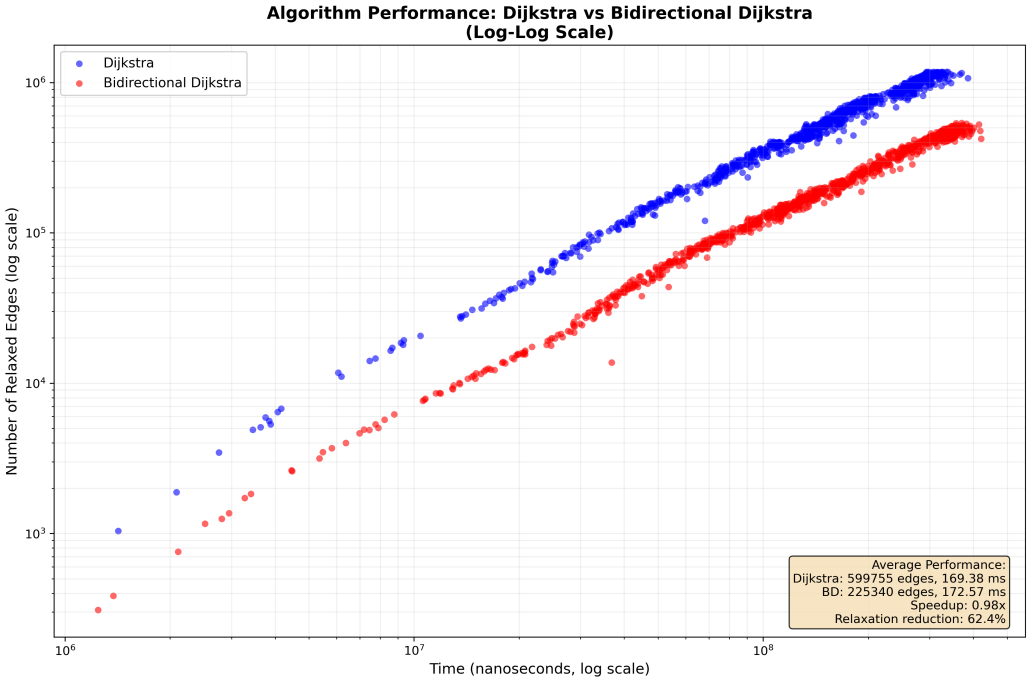

Figure 2: Log-log scatter plot comparing Dijkstra and Bidirectional Dijkstra algorithms. Each point represents a query between two random vertices, with Bidirectional Dijkstra (red) achieving significant reduction in relaxed edges compared to standard Dijkstra (blue).

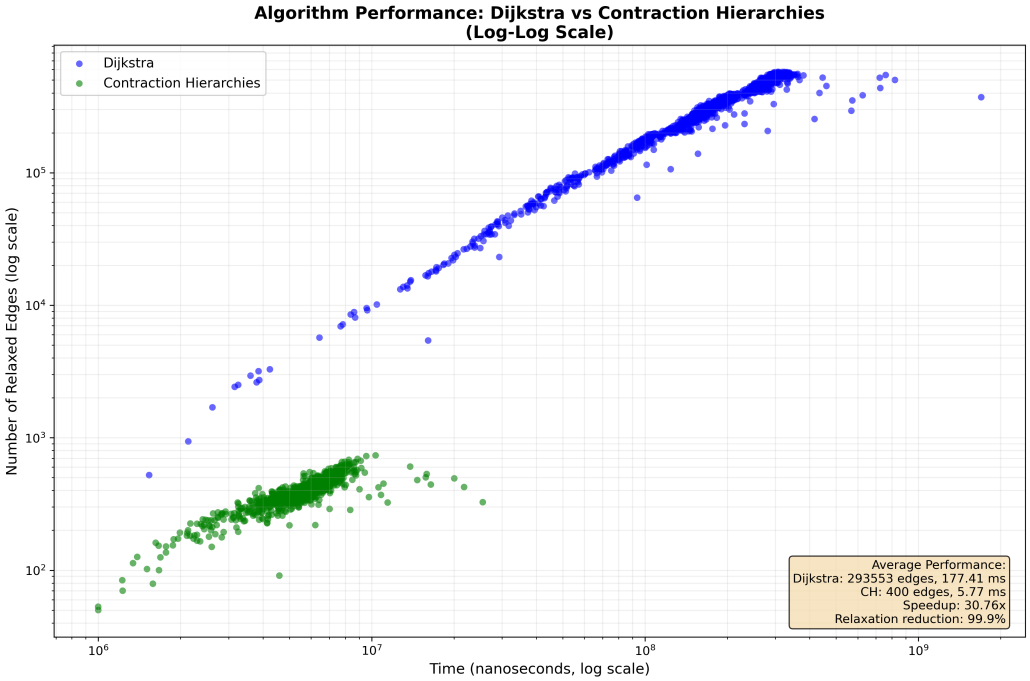

Figure 3: Log-log scatter plot comparing Dijkstra and Contraction Hierarchies algorithms. Each point represents a query between two random vertices, with Contraction Hierarchies (green) demonstrating dramatic performance improvement over standard Dijkstra (blue) with significantly fewer edge relaxations.