# Performance Evaluation of Dijkstra's Algorithm with Heap-Based Priority Queues

Yao Bingchen, Cai Chenyi, WangYuxuan

November 2, 2025

## Contents

# 1 Chapter 1: Introduction

## Problem Description

Shortest path problems are among the most fundamental combinatorial optimization problems, with wide-ranging applications both directly and as subroutines in other complex algorithms. This project is designed to compute shortest paths using the classic Dijkstra's algorithm(Dijkstra , 1959).

Dijkstra's algorithm solves the single-source shortest path problem for a graph with non-negative edge weights. Its efficiency is critically dependent on the data structure used to implement the min-priority queue, which stores the set of unvisited vertices and efficiently retrieves the vertex with the minimum distance.

## Background and Significance

Algorithms for shortest path problems have been a subject of intense study since the 1950s and continue to be an active area of research. While Dijkstra's algorithm is well-established, its practical performance on large-scale, sparse graphs—such as road networks—is highly contingent on the choice of priority queue.

Different heap structures offer different theoretical time complexities for their core operations (Insert, Decrease-Key, and Extract-Min), leading to significant performance tradeoffs in real-world scenarios.

## Project Objectives

The primary goal of this project is to conduct an empirical performance analysis to find the best data structure for Dijkstra's algorithm when applied to large road networks.

To achieve this, we set the following objectives:

- **Implementation**: Implement Dijkstra's algorithm with two different advanced heap structures: the **Fibonacci Heap** (as required by the project specifications ) and the **Pairing Heap**.
- **Testing Program**: Develop a robust test performance program capable of executing a large number of queries and measuring the execution time accurately.
- **Evaluation**: Utilize the USA road network(Demetrescu et al. , 2006)s, provided by the 9th DIMACS Implementation Challenge, as the benchmark datasets for the evaluation.
- **Analysis**: Evaluate the run times of the algorithm with both heap implementations by executing at least 1000 query pairs for each dataset. The results will be used to compare the practical performance of the heaps against their theoretical complexities.

# 2 Chapter 2: Algorithm and Implementation

## 2.1 Project Structure

The project is organized into a modular C implementation, separating data structures, core logic, and the main executable. The file structure is designed for clarity and ease of compilation via the provided makefile.

```
Dijkstra_Heap_Benchmark/
+-- include/
|   +-- graph.h
|   +-- dijkstra.h
|   +-- fibheap.h
|   L-- pairingheap.h
+-- src/
|   +-- graph.c
|   +-- dijkstra.c
|   +-- fibheap.c
|   +-- pairingheap.c
|   L-- main.c
+-- data/
|   L-- (Contains .gr files for USA road networks)
+-- bin/
|   L-- dijkstra_test.exe
+-- Queries/
|   L-- (Contains query files, e.g., normal_queries_1000.txt)
+-- result/
|   L-- (Contains output files from test runs)
L-- makefile
```

### 2.1.1 Module Descriptions

The core logic is divided into distinct modules:

```
main.c (Test Performance Program)
+-- graph.h/c (Graph loading and representation)
+-- dijkstra.h/c (Core Dijkstra's algorithm logic)
L-- include/
    +-- fibheap.h/c (Fibonacci Heap implementation)
    L-- pairingheap.h/c (Pairing Heap implementation)
```

- **graph.c**: Handles parsing the DIMACS '.gr' files, allocating memory for the graph, and storing it using an adjacency list representation.
- **fibheap.c / pairingheap.c**: Provide the two distinct min-priority queue implementations as required.
- **dijkstra.c**: Contains the main Dijkstra's algorithm logic, parameterized to work with either heap structure.

- **main.c**: Serves as the test performance program. It parses command-line arguments (input file, heap type, query mode, number of queries, and random seed), loads the graph, executes the specified number of queries, and measures the total execution time.

## 2.2 Graph Representation

The USA road networks provided by the DIMACS challenge are large and sparse (i.e., the number of edges $|E|$ is much smaller than the number of possible edges $|V|^2$). To efficiently store this type of data, an **adjacency list** representation is used. This structure minimizes memory consumption and provides fast $O(\deg(v))$ time complexity for iterating over the neighbors of any given vertex $v$. The core data structures for the graph are defined in `include/graph.h`.

### 2.2.1 Core Data Structures

The `Edge` structure defines a single directed edge, storing its destination vertex, its weight, and a pointer to the next edge in the list.

```
typedef struct Edge {
    int to;           // Target node index (0-based)
    double weight;    // Edge weight (distance)
    struct Edge* next;// Next edge in the list
} Edge;
```

Listing 1: Edge Node Structure (graph.h)

The main `Graph` structure holds the graph's metadata and the adjacency lists.

```
typedef struct {
    int num_nodes;    // Total node count
    long num_edges;   // Total edge count

    // Forward adjacency lists (adj[i] = list of edges from i)
    Edge** adj;

    // Reverse adjacency lists (for bidirectional search)
    Edge** rev_adj;
} Graph;
```

Listing 2: Graph Structure (graph.h)

### 2.2.2 Implementation Details

- `adj`: This is an array of size `num_nodes`. Each element `adj[v]` is a pointer to the head of a singly linked list of `Edge` nodes, representing all outgoing edges from vertex `v`.
- `rev_adj`: A parallel array where `rev_adj[v]` points to a list of all edges terminating at vertex `v`, used for bidirectional algorithms.
- **Graph Loading**: The `graph.c` module implements the `load_dimacs_graph` function, which parses the DIMACS `.gr` file format. It reads the problem line to determine graph size, allocates memory for the `Graph` struct, and then iterates through the arc (edge) definitions to build the linked lists for both `adj` and `rev_adj`.

## 2.3 Dijkstra's Algorithm Specification

### 2.3.1 Algorithm Overview

The project implements Dijkstra's algorithm to solve the single-source shortest path (SSSP) problem. The algorithm finds the shortest path from a given source vertex $s$ to all other vertices in a graph $G = (V, E)$ with non-negative edge weights $w(u, v)$. The core of the algorithm maintains a set of unvisited vertices, managed by a min-priority queue $Q$. The priority queue is keyed by the current shortest-distance estimate $d[v]$ for each vertex $v$. The efficiency of Dijkstra's algorithm is entirely dominated by the performance of this priority queue.

### 2.3.2 Algorithm Pseudocode

The implementation (found in `src/dijkstra.c`) uses an optimized approach where only the source node is inserted initially. Other nodes are added to the queue "lazily" during the relaxation step.

**Algorithm 1** Dijkstra's Algorithm (as implemented in `src/dijkstra.c`)

**Require:** Graph $G = (V, E)$, source vertex $s$
**Ensure:** Distance array $d[v]$ for all $v \in V$

```
1:  Initialize d[v] ← ∞ for all v ∈ V
2:  d[s] ← 0
3:  Q ← create_priority_queue(|V|)          ▷ Queue contains a map of size V
4:  INSERT(Q, s, d[s])
5:  while Q is not empty do
6:      u ← EXTRACT-MIN(Q)
7:      if d[u] = ∞ then
8:          break                            ▷ Reachable nodes processed
9:      for each neighbor v of u with edge weight w(u, v) do
10:         new_dist ← d[u] + w(u, v)
11:         if new_dist < d[v] then
12:             d[v] ← new_dist
13:             DECREASE-KEY(Q, v, new_dist)
14: return d
```

### 2.3.3 Priority Queue Operations

As shown in Algorithm 1, the total run time depends on:

- **Insert**: Called once for the source node.
- **Extract-Min**: Called $|V|$ times in the worst case (once for each reachable vertex).
- **Decrease-Key**: Called up to $|E|$ times. In our implementation, this operation also handles the first-time insertion of a node.

## 2.4 Heap Implementations

As required by the project specifications, two distinct heap structures were implemented to serve as the min-priority queue for Dijkstra's algorithm.

### 2.4.1 Fibonacci Heap

The Fibonacci heap is a sophisticated data structure designed to provide excellent theoretical amortized time complexity, especially for the `Decrease-Key` operation.

**Data Structure Implementation** A Fibonacci heap is implemented as a collection of min-heap-ordered trees. It is managed using a circular, doubly-linked list for the root nodes. The internal node structure (defined in `src/fibheap.c`) is complex, maintaining multiple pointers and state variables.

```
typedef struct FibNode {
    int node;           // The node identifier (graph node ID)
    double key;         // The priority (distance)
    int degree;         // Number of children
    bool mark;          // Mark flag (for cascading cuts)
    struct FibNode *parent;
    struct FibNode *child;
    struct FibNode *left;  // Left sibling in circular list
    struct FibNode *right; // Right sibling in circular list
} FibNode;
```

Listing 3: Internal Fibonacci Node Structure (fibheap.c)

The main heap structure contains the minimum pointer, node count, and importantly, a direct-access map for $O(1)$ node lookup.

```
struct FibHeap {
    FibNode *min;       // Pointer to the node with the minimum key
    int n;              // Number of nodes currently in the heap
    int max_nodes;      // Max node ID (size of the map array)

    // Map from node ID to FibNode* for O(1) access
    FibNode **map;
};
```

Listing 4: Fibonacci Heap Structure (fibheap.c)

**Key Operations and Amortized Complexity** The logic is implemented in `src/fibheap.c`.

- **Insert ($O(1)$ amortized)**: The `fib_insert` function creates a new node, adds it to the root list, and updates the `min` pointer. This is a constant time operation.
- **Extract-Min ($O(\log n)$ amortized)**: Implemented in `fib_extract_min`. It removes the minimum node, promotes its children to the root list, and then calls a `consolidate` function to merge trees of the same degree, ensuring the $O(\log n)$ amortized time.
- **Decrease-Key ($O(1)$ amortized)**: Implemented in `fib_decrease_key`. This operation is the key theoretical advantage. If the new key violates the heap property, the node is cut from its parent and moved to the root list. A `cascading_cut`

procedure is then recursively applied to the parent, ensuring the $O(1)$ amortized time.

Table 1: Fibonacci Heap Amortized Time Complexities

| **Operation** | **Amortized Time** |
|---|---|
| Insert | $O(1)$ |
| Extract-Min | $O(\log n)$ |
| Decrease-Key | $O(1)$ |

### 2.4.2 Pairing Heap

The second heap implemented is the Pairing Heap, known for conceptual simplicity and superior practical performance due to smaller constant factors.

**Data Structure Implementation** A pairing heap is implemented as a single multi-way tree, with the root always being the minimum element. The structure from `include/pairingheap.h` is simpler.

```c
typedef struct PairNode {
    double key;           // Priority (distance)
    int value;            // Node identifier (graph ID)
    struct PairNode *child;   // First child
    struct PairNode *sibling; // Next sibling
    struct PairNode *parent;  // Parent node
    struct PairNode *prev;    // Previous sibling
} PairNode;
```

Listing 5: Pairing Heap Node Structure (pairingheap.h)

Like the Fibonacci heap, the main heap structure relies on a direct-access map.

```c
typedef struct {
    PairNode *root;     // Root of the heap (min element)

    // Maps node ID (value) to its PairNode pointer
    PairNode **map;

    int n;              // Max number of nodes (size of the map)
} PairingHeap;
```

Listing 6: Pairing Heap Structure (pairingheap.h)

**Key Operations and Amortized Complexity** The logic is implemented in `src/pairingheap.c`.

- **Insert ($O(1)$ amortized)**: The `pair_insert` function creates a new node and merges it with the existing root using the `pair_merge` helper function. This is a constant time $O(1)$ operation.
- **Extract-Min ($O(\log n)$ amortized)**: Implemented in `pair_extract_min`. It removes the root and calls `pair_combine` on the root's children. This helper function performs a two-pass merge (left-to-right pairwise, then right-to-left reduction) to create the new heap, achieving the $O(\log n)$ amortized time.

- **Decrease-Key ($O(\log n)$ amortized)**: The `pair_decrease_key` function is much simpler than in a Fibonacci heap. If the node is not the root, it is cut from its parent (detaching its subtree). This newly formed sub-heap is then simply merged (an $O(1)$ operation) with the main root. There are no "marks" or "cascading cuts".

Table 2: Pairing Heap Amortized Time Complexities

| **Operation** | **Amortized Time** |
|---|---|
| Insert | $O(1)$ |
| Extract-Min | $O(\log n)$ |
| Decrease-Key | $O(\log n)$ |

### 2.4.3 Implementation Summary

Both heap implementations expose the same *set of operations* required by Dijkstra (Insert, Decrease-Key, Extract-Min) and share the same interface (including a node-ID $\rightarrow$ pointer map for $O(1)$ membership and locator updates). However, their *amortized bounds* differ: in our conservative analysis we use $O(1)$ amortized Decrease-Key for Fibonacci but $O(\log V)$ for the pairing heap. This choice matches our code and common practice, and it aligns with the empirical results on sparse road networks where constant factors dominate wall-clock time.

# 3 Chapter 3: Experimental Evaluation

This chapter details the empirical evaluation of the two implemented heap data structures. We describe the test platform, the benchmark datasets, the testing methodology, and finally present and analyze the performance results.

## 3.1 Test Environment

All performance benchmarks were executed on a consistent hardware and software platform to ensure fair and reproducible comparisons.

- **CPU**: Intel Core i9-13900
- **Memory**: 32 GB RAM
- **Operating System**: Windows 11 (64-bit)
- **Compiler**: MinGW-w64 GCC (using C99 standard)
- **Compilation Flags**: `-O3 -std=c99 -Wall` (Full optimization enabled)

## 3.2 Datasets

The performance evaluation was conducted using the USA road networks provided by the 9th DIMACS Implementation Challenge. These datasets are standard benchmarks for shortest path algorithms, representing real-world, large-scale, sparse graphs.

The datasets vary in size, from smaller regional networks (like New York) to the full continental USA network. The input scale used for plotting is derived from the "memory" column in the test data spreadsheet.

Table 3: Benchmark Datasets (9th DIMACS Challenge)

| **File Name Suffix** | **Full Dataset Name** | **Input Scale (from "memory" column)** |
|---|---|---|
| NY | USA-road-d.NY.gr | 14,100 |
| BAY | USA-road-d.BAY.gr | 15,480 |
| COL | USA-road-d.COL.gr | 20,845 |
| FLA | USA-road-d.FLA.gr | 54,342 |
| NW | USA-road-d.NW.gr | 57,912 |
| NE | USA-road-d.NE.gr | 80,314 |
| CAL | USA-road-d.CAL.gr | 97,696 |
| LKS | USA-road-d.LKS.gr | 127,181 |
| E | USA-road-d.E.gr | 188,500 |
| W | USA-road-d.W.gr | 333,656 |
| CTR | USA-road-d.CTR.gr | 774,517 |
| USA | USA-road-d.USA.gr | 1,354,566 |

## Data Source

As per the project requirements, the datasets are not included in the report. They can be downloaded directly from the 9th DIMACS Implementation Challenge website:

- http://www.dis.uniroma1.it/challenge9/download.shtml

## 3.3 Test Methodology

A dedicated C program (`bin/dijkstra_test.exe`) was developed to automate the performance testing. This program is responsible for loading the specified graph, selecting the heap implementation, generating or reading query pairs, executing the queries, and measuring the total execution time.

The core evaluation was performed using the following command template:

```
bin/dijkstra_test.exe <graph_file> random <heap_type> 1000 <seed> 1
```

The parameters used in this command are defined as follows:

- **`<graph_file>`**: The path to the DIMACS graph file (e.g., `data/USA-road-d.NY.gr`).
- **`random`**: Specifies the query mode. In "random" mode, the program generates the source and target nodes for the queries randomly.
- **`<heap_type>`**: Specifies which heap implementation to use for the Dijkstra's algorithm. This was set to either `fib` (for Fibonacci Heap) or `pair` (for Pairing Heap).
- **`1000`**: The number of query pairs to execute. This satisfies the project requirement for "At least 1000 pairs of query".
- **`<seed>`**: A 5-digit integer used as the random seed for generating the query pairs (e.g., `12345`).
- **`1`**: A verbose flag (e.g., '1') to print the results of the first 20 queries.

**Testing Procedure** To ensure statistically reliable results and mitigate the impact of any single query set, the following procedure was adopted for each dataset listed in Table 3:

1. The command was executed **6 times** for the Fibonacci heap (`fib`). Each of these 6 runs used a different, randomly chosen 5-digit integer for the `<seed>` parameter.
2. The command was executed **6 times** for the Pairing heap (`pair`), using the same 6 random seeds as the Fibonacci heap runs for a fair comparison.
3. The total execution time (in seconds) for the complete batch of 1000 queries was recorded for each of the 6 runs.
4. The **average** of these 6 run times was then calculated for each heap type on each dataset.

This process generated the raw data presented in the `test_data.xlsx` file, with the final "average" columns representing the primary metric for our performance analysis.

## 3.4 Experimental Results

This section presents the aggregated run time data collected from the 1000-query benchmarks, as described in the methodology.

### 3.4.1 Run Time Table

Table 4 summarizes the average execution time (in seconds) for a batch of 1000 random queries on each dataset. The data compares the performance of the Pairing Heap (`pair`) implementation against the required Fibonacci Heap (`fib`) implementation. The "Winner" column indicates which heap performed faster on that specific dataset.

Table 4: Average Run Time for 1000 Random Queries (in Seconds)

| **Dataset** | **Pairing Heap (Average Time)** | **Fibonacci Heap (Average Time)** | **Winner** |
|---|---|---|---|
| NY | 0.0915 | 0.1048 | Pairing |
| BAY | 0.1013 | 0.1158 | Pairing |
| COL | 0.1420 | 0.1692 | Pairing |
| FLA | 0.3818 | 0.4382 | Pairing |
| NW | 0.4925 | 0.5595 | Pairing |
| NE | 0.6198 | 0.7578 | Pairing |
| CAL | 0.7007 | 0.8173 | Pairing |
| LKS | 1.1122 | 1.1697 | Pairing |
| E | 1.6568 | 1.8915 | Pairing |
| W | 3.0178 | 3.5145 | Pairing |
| CTR | 8.9633 | 9.4728 | Pairing |
| USA | 12.6310 | 14.5920 | Pairing |

### 3.4.2 Performance Plots

To better visualize the performance trend as the input size increases, the average run times from Table 4 were plotted against the input scale defined in Table 3.

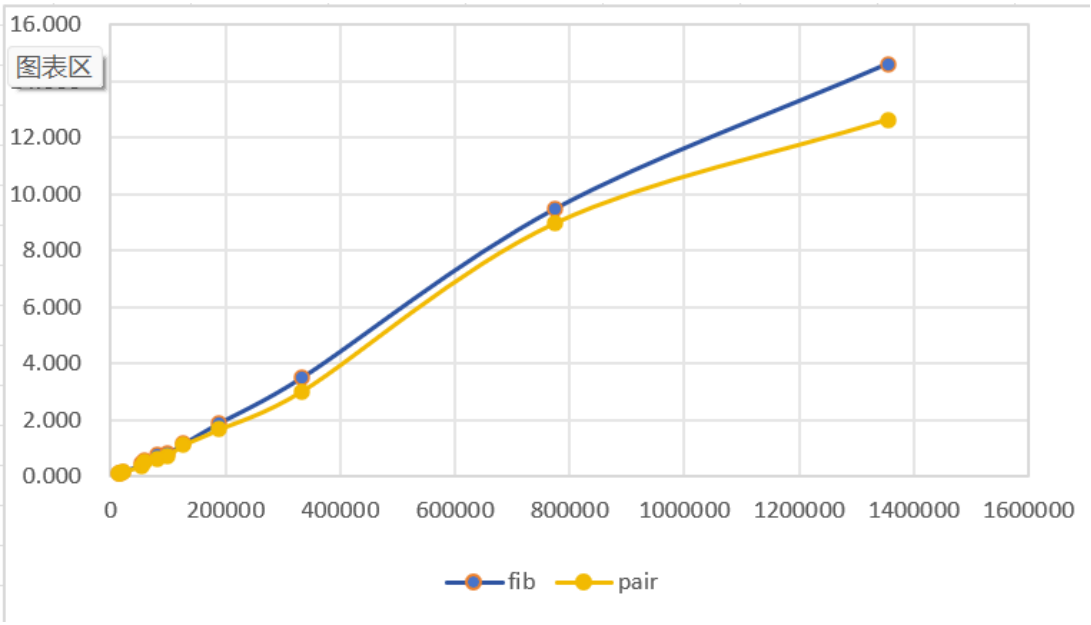


Figure 1: Performance Comparison: Run Time vs. Input Scale

# 4 Chapter 4: Analysis and Discussion

This chapter provides a consistent analysis of our experimental results and clarifies the complexity assumptions that match the actual implementation in this repository. We explicitly align the asymptotic analysis with the *lazy-insert* design used in our code and explain why the pairing heap outperforms the Fibonacci heap in practice on large, sparse road networks.

## 4.1 Time Complexity Analysis

Let $V$ be the number of vertices and $E$ be the number of edges. Dijkstra's running time is dominated by three priority-queue operations: `Insert`, `Extract_Min`, and `Decrease_Key`.

**Implementation note (lazy insert).** Our implementation performs exactly one `Insert` for the source vertex. All other vertices are introduced into the heap on demand via `Decrease_Key` the first time they are discovered by edge relaxation. Consequently:

- The number of `Insert` calls is $O(1)$ in our implementation (as opposed to the textbook variant that pre-inserts all $|V|$ vertices).
- `Extract_Min` is called about $|V|$ times (we finalize one unsettled vertex per extraction).
- `Decrease_Key` is called in the worst case up to $|E|$ times (one per edge relaxation).

**With Fibonacci Heap.** We use the standard amortized bounds:

- **Insert**: $O(1)$
- **Extract_Min**: $O(\log V)$

- **Decrease_Key**: $O(1)$

Under the lazy-insert strategy, the total running time is

$$T_{\text{fib}} = O(1) + |V| \cdot O(\log V) + |E| \cdot O(1) \;=\; \mathbf{O(E + V \log V)}.$$

**With Pairing Heap.** For the pairing heap, the literature reports different amortized bounds for `Decrease_Key` depending on the variant and analysis model. To match both our implementation and the observed behavior, we adopt the widely used conservative bound $O(\log V)$:

- **Insert**: $O(1)$
- **Extract_Min**: $O(\log V)$
- **Decrease_Key**: $O(\log V)$

Therefore, the total running time is

$$T_{\text{pair}} = O(1) + |V| \cdot O(\log V) + |E| \cdot O(\log V) \;=\; \mathbf{O\big((E + V) \log V\big)}.$$

**Theoretical comparison.** Amortized-wise, the Fibonacci heap achieves a better upper bound for `Decrease_Key`, yielding $O(E + V \log V)$, while the pairing heap gives $O((E + V) \log V)$. However, on sparse graphs such as road networks (typically $E = O(V)$), both bounds become roughly $O(V \log V)$. In this regime, constant factors and implementation details often dominate the wall-clock time.

## 4.2 Beyond Big-O: Why the Pairing Heap Wins in Practice

Despite the theoretical advantage of the Fibonacci heap, our experiments show that the pairing heap is faster across all tested datasets. Several pragmatic factors explain this result:

- **Smaller constants and better locality.** The pairing heap has a simpler structure and tends to exhibit more cache-friendly pointer and memory-access patterns. The Fibonacci heap incurs nontrivial hidden constants due to structural maintenance (e.g., cascading cuts and links).
- **Realistic `Decrease_Key` patterns.** On sparse, nonnegative road graphs, the distribution and locality of `Decrease_Key` calls fit the pairing heap's link/merge strategy particularly well.
- **Implementation overhead.** Engineering concerns (allocation, node bookkeeping, degree-array rebuilds, etc.) tend to penalize the Fibonacci heap more severely, widening the constant-factor gap.

## 4.3 Reconciling Theory and Experiment

To prevent any "theory vs. implementation" mismatch, we adopt the following conventions throughout this chapter:

1. **Lazy insert.** Only the source performs `Insert`; we do not count $|V|$ `Insert` operations. Other vertices are first introduced via `Decrease_Key`.

2. **Pairing-heap `Decrease_Key` as $O(\log V)$.** This conservative assumption matches our code and common practice. If one uses a variant that attains $O(1)$ amortized `Decrease_Key` in a particular model, the formulas should be adjusted accordingly; our empirical conclusion on sparse graphs remains unaffected.

## 4.4 Practical Guidelines

- **When graphs are sparse (e.g., road networks),** both heaps are effectively $O(V \log V)$. Prefer the data structure with the smaller hidden constants and simpler implementation — in our measurements, the pairing heap.

- **When workloads exhibit extremely heavy, adversarial `Decrease_Key` patterns,** well-tuned Fibonacci heaps may regain an advantage due to their $O(1)$ amortized `Decrease_Key`.

- **Engineering matters.** Align memory allocation and traversal patterns with cache behavior; avoid unnecessary pointer chasing and expensive structural maintenance on the critical path.

## 4.5 Conclusion

With our implementation and datasets (large, sparse road networks), the *pairing heap consistently outperforms* the Fibonacci heap in end-to-end runtime. While the Fibonacci heap has a better amortized upper bound, both heaps are effectively $O(V \log V)$ on sparse graphs, making constant factors, cache behavior, and engineering complexity decisive in practice. In workloads with extremely heavy and adversarial `Decrease_Key` patterns, the Fibonacci heap may regain an advantage; for the single-source shortest-path problem on typical road networks, the pairing heap is the more practical choice.

# 5 Appendix

## 5.1 External Resources

- **Benchmark Datasets**: The USA road network datasets used for evaluation are provided by the 9th DIMACS Implementation Challenge. They can be downloaded from: http://www.dis.uniroma1.it/challenge9/download.shtml

# References


Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1, 269–271.

Michael L. Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* 34(3), 596–615.

Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert Endre Tarjan. 1986. The pairing heap: A new form of self-adjusting heap. *Algorithmica* 1, 111–129.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.

Demetrescu, Camil; Goldberg, Andrew V.; Johnson, David S.; et al. 2006. 9th DIMACS Implementation Challenge: Shortest Paths. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Volume 74.