Shortest Path Algorithm Comparisons: Dijkstra with Advanced Heaps and Bidirectional Skewness

AARAV GOSALIA, University of British Columbia Okanagan, Canada
RILEY EATON, University of British Columbia Okanagan, Canada

This report presents a comparative analysis of several variants of Dijkstra's shortest path algorithm, focusing on both priority queue design and search strategy. We implement Dijkstra's algorithm with three different heap-based priority queues: Binary Heap, Radix Heap, and Fibonacci Heap, and also study a bidirectional Dijkstra variant with skewed search frontiers. All data structures are implemented from scratch in Python and evaluated on large randomly generated weighted graphs designed to approximate real-world network structure. Theoretical time and space bounds are contrasted with empirical behaviour, using both runtime and peak memory usage as primary metrics.

[Results summary to be completed once experiments are finalized.]

Additional Key Words and Phrases: Dijkstra's Algorithm, Binary Heap, Radix Heap, Fibonacci Heap, Bidirectional Skewness, Graph Algorithms, Priority Queues

1 Introduction

Dijkstra's shortest path algorithm remains one of the most influential and widely used algorithms in graph theory, powering applications in transportation networks, communication systems, robotics, and large-scale optimization. Although the algorithmic structure of Dijkstra's method is conceptually simple, its real-world performance is deeply dependent on the efficiency of the priority queue used to repeatedly extract the next closest vertex and perform decrease_key operations.

This report examines and compares three priority queue data structures that can be used to optimize Dijkstra's algorithm:

(1) **Binary Heap** – A widely used baseline implementation offering $O(\log n)$ extract_min and decrease_key, forming the standard version taught in most courses.
(2) **Fibonacci Heap** – A theoretically optimal structure with $O(1)$ amortized decrease_key and $O(\log n)$ extract_min, often cited for achieving Dijkstra's best-known theoretical bound of $O(m + n \log n)$.
(3) **Radix Heap** – A monotone integer priority queue tailored for Dijkstra on graphs with non-negative edge weights. It exploits the fact that extracted distances are non-decreasing, achieving $O(1)$ amortized insert and decrease_key, and $O(\log C)$ for extract_min, where C is the maximum key difference. This makes it highly efficient on graphs with bounded or small integer weights.

In addition to these data structures, this project will also extend the analysis to include **Bidirectional Dijkstra with Skewed Expansion**, an optimization technique that explores the graph simultaneously from the source and the target. (ELABORATE HERE)

Overall, the goal of this project is to provide both a theoretical and empirical comparison of these structures, evaluating their runtime behavior, scalability, and memory characteristics when applied to large synthetic graphs.

Authors' Contact Information: Aarav Gosalia, University of British Columbia Okanagan, Kelowna, Canada; Riley Eaton, University of British Columbia Okanagan, Kelowna, Canada.

2 Background and Theory

This section discusses the underlying principles of Dijkstra's algorithm, the priority queue data structures used to optimize it, and bidirectional skewness.

2.1 Dijkstra's Algorithm Overview

Dijkstra's algorithm finds the shortest path from a source node to all other nodes in a weighted graph with non-negative edge weights. It repeatedly extracts the node with the smallest tentative distance and updates its neighbors.

The efficiency of Dijkstra's algorithm largely depends on how the "next smallest distance" is retrieved and updated, i.e., on the efficiency of the `extract_min` and `decrease_key` operations. Therefore, the overall performance of Dijkstra's algorithm depends directly on the choice of priority queue.

2.2 Binary Heap

A **Binary Heap** is a complete binary tree that satisfies the *heap property* — each parent node's key is smaller than or equal to the keys of its children (in a min-heap). Internally, it is most often implemented using an array where the element at index i has:

- Left child at index $2i + 1$
- Right child at index $2i + 2$
- Parent at index $\lfloor (i - 1)/2 \rfloor$

This array-based representation eliminates the need for pointers, making Binary Heaps both memory efficient and cache friendly. Figure 1 illustrates how the same heap structure can be represented in both array and tree form.

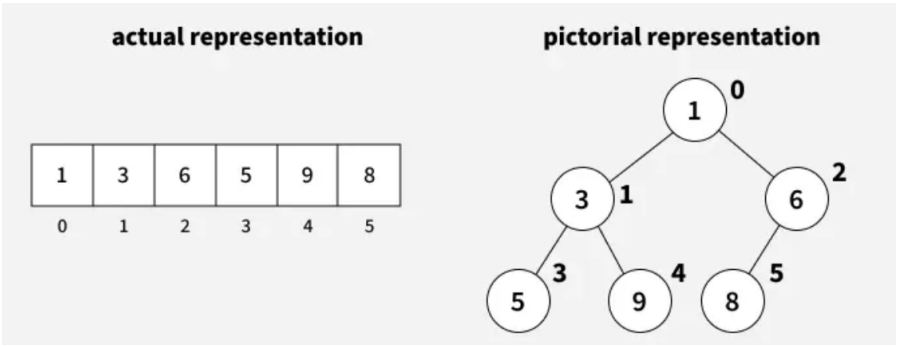

Fig. 1. Binary Heap: array-based and pictorial representations (adapted from GeeksforGeeks [3]).

Insertion. When a new element is inserted, it is first placed at the end of the array (to maintain the complete tree property). Then, it is repeatedly compared with its parent and swapped upward until the heap property is restored. This upward movement is known as a **sift-up** or **bubble-up**. Since each level in the tree can contain twice as many nodes as the previous one, and the heap's height is $\log_2 n$, the insertion process requires at most $\log n$ swaps.

$$T_{\text{insert}} = O(\log n)$$

Extract_Min. Extracting the minimum element involves removing the root (the smallest key). To maintain completeness, the last element in the array is moved to the root position. Then, it is

repeatedly swapped with the smaller of its two children until the heap property is restored. This process is called **sift-down** or **heapify**. Each swap moves the element one level deeper, and the tree height is $\log n$, so the operation takes logarithmic time.

$$T_{\text{extract\_min}} = O(\log n)$$

decrease_Key. The `decrease_key` operation lowers the key value of an element in the heap. Because the key becomes smaller, the node may violate the heap property with respect to its parent. Therefore, the element is moved upward using the same **sift-up** procedure as insertion. In the worst case, it moves up to the root, resulting in logarithmic time.

$$T_{\text{decrease\_key}} = O(\log n)$$

Space Complexity. A Binary Heap stores all elements in a single contiguous array, which makes its space usage straightforward to analyze. The heap requires one array slot per element, leading to a total memory footprint of

$$O(n)$$

where n is the number of elements in the heap.

Because the array representation avoids pointers entirely, Binary Heaps have excellent spatial locality and minimal per-node overhead compared to pointer-based structures such as Fibonacci Heaps. This contiguous layout also makes them cache-friendly, which contributes to their strong practical performance despite having asymptotically slower operations than more advanced heaps.

Binary Heaps are conceptually simple, have small constant factors, and perform well in practice due to their contiguous memory layout. They are the default choice for most implementations of Dijkstra's algorithm in production systems. However, their performance can degrade for dense graphs where the number of `decrease_key` operations is large, motivating more advanced structures such as the **Radix Heap** and **Fibonacci Heap**.

2.3 Fibonacci Heap

A **Fibonacci Heap** is an advanced data structure that improves the efficiency of priority queue operations through a collection of *heap-ordered trees*. Unlike Binary Heaps, it performs most operations in constant amortized time by deferring expensive structural adjustments until absolutely necessary.

Each tree in a Fibonacci Heap obeys the *min-heap property*: the key of every node is greater than or equal to that of its parent. The heap maintains a circular doubly linked list of all tree roots (the *root list*), with a pointer to the minimum key node.

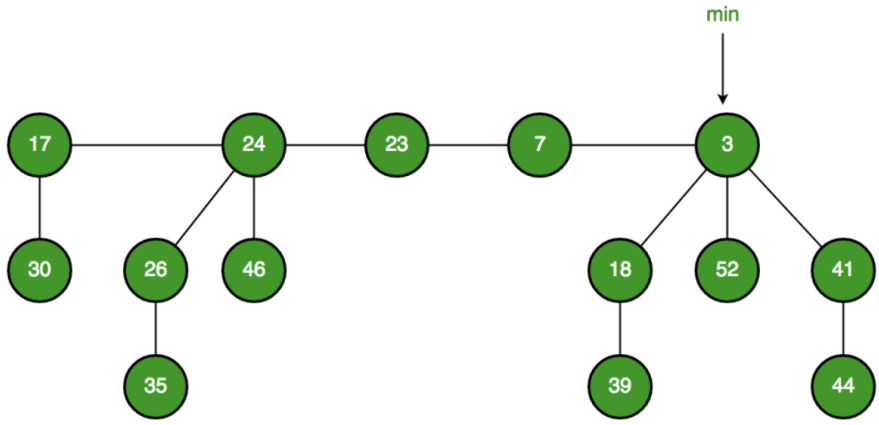

Fig. 2. Example of a Fibonacci Heap with multiple trees in the root list. The node with key 3 is the global minimum (adapted from GeeksforGeeks [4]).

Insertion. To insert a new key, a single-node tree is created and added directly to the root list. The minimum pointer is updated if the new key is smaller than the current minimum. Since no tree restructuring or traversal is required, the insertion operation runs in **constant time**.

$$T_{\text{insert}} = O(1)$$

Extract_Min. The `extract_min` operation removes the node pointed to by the minimum pointer. Its children are then added to the root list, effectively promoting them to become separate trees. To restore the heap's structure, trees in the root list with the same degree (number of children) are **consolidated** by linking one tree as a child of another with a smaller key.

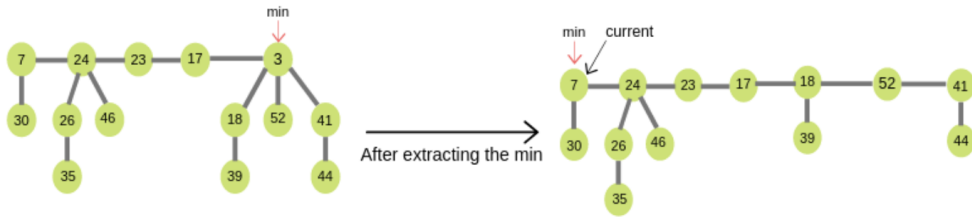

Fig. 3. Illustration of the `extract_min` operation in a Fibonacci Heap. The minimum node (3) is removed, and its children are added to the root list, followed by consolidation of trees with equal degree (adapted from OpenGenus [2]).

The consolidation process ensures that the number of trees in the heap is logarithmic with respect to n, the total number of nodes. Therefore, while individual link operations are constant time, the entire `extract_min` runs in amortized logarithmic time.

$$T_{\text{extract\_min}} = O(\log n)$$

decrease_Key. The `decrease_key` operation reduces the key value of a node. If the new key violates the heap property (i.e., becomes smaller than its parent), the node is **cut** from its parent and added to the root list. To maintain balance, if a parent loses more than one child, it is also cut, a process known as a **cascading cut**.

Since cuts involve constant-time pointer manipulations, the amortized time for this operation remains constant.

$$T_{\text{decrease\_key}} = O(1)$$

Fibonacci Heaps achieve remarkable theoretical efficiency, particularly for algorithms like Dijkstra's, where `decrease_key` is frequent. The overall complexity improves to:

$$T(V, E) = O(E + V \log V)$$

This is better than the $O(E \log V)$ bound of Binary Heaps.

However, in practice, Fibonacci Heaps might be slower due to their large constant factors and complex pointer manipulations. They are mainly of theoretical interest and in applications where the number of `decrease_key` operations is exceptionally high. The implementation complexity also makes debugging and memory management more challenging compared to simpler structures like Binary Heaps.

2.4 Radix Heap

2.5 BiDirectional Skewness

3 Methodology

All three heaps, Binary, Radix, and Fibonacci, were implemented entirely in **Python 3** from scratch. Each heap defines a consistent interface to support Dijkstra's algorithm, ensuring fair benchmarking and identical algorithmic behavior across all data structures.

Each heap class implements the following core methods:

- `insert(node, priority)` – Inserts a node into the heap with its associated distance value.
- `extract_min()` – Removes and returns the node with the smallest priority value.
- `decrease_key(node, new_priority)` – Updates a node's priority when a shorter path is found.
- `is_empty()` – Checks whether the heap contains any remaining nodes.

A unified implementation of Dijkstra's algorithm was created in `dijkstra.py`, where the heap type can be switched dynamically:

```
dijkstra(graph, source, heap_type="binary")
```

This modular design allows the same Dijkstra function to operate seamlessly with any heap type, enabling controlled comparisons of performance and runtime behavior under identical conditions.

3.1 Graph Generation

To test scalability, large random graphs were generated using a custom `graph_generator.py` script located in the `utils` directory. Each graph is represented as a directed, weighted adjacency list:

$$G = \{u : [(v, w)], \ldots\}$$

where each node u maps to a list of tuples (v, w), representing a directed edge from u to v with weight w randomly drawn from the range $[1, 10]$.

Generation Process. For each graph size, a specified number of nodes and an average number of outgoing edges per node were chosen. The generator computes the total number of edges approximately as:

$$E \approx \text{avg\_edges\_per\_node} \times V$$

For example, a graph with $V = 500{,}000$ nodes and 15 average edges per node produces roughly 7.5 million directed edges. Edges and weights are randomly assigned, and the process is visualized using the `tqdm` [1] progress bar for large-scale runs.

The final experiments were conducted on four large-scale graphs of increasing size to evaluate scalability and runtime growth across different heap implementations.

- 100K nodes $\longrightarrow \approx$ 1.2M edges
- 500K nodes $\longrightarrow \approx$7.5M edges
- 1M nodes $\longrightarrow \approx$20M edges
- 2M nodes $\longrightarrow \approx$50M edges

Larger graphs beyond 2 million nodes were not tested due to hardware constraints, specifically, the system's 16GB memory capacity and the significant runtime required for dense graphs at that scale. Each graph was processed by the Dijkstra implementation using all three heap structures to record total execution time, enabling both theoretical and empirical performance comparisons.

4 Results

All experiments were executed on a MacBook Pro M2 (16 GB RAM) using wall-clock time to measure total runtime for each heap implementation. Each experiment was repeated **twice** to ensure consistency and reproducibility of results, minimizing the effect of system background processes or transient performance variations.

Table 1. Runtime comparison of Dijkstra's Algorithm using different heap implementations across two runs.

Heap Type	Run	100K	500K	1M	2M
Binary Heap	Run 1	0.77s	6.55s	15.29s	237.31s
	Run 2	0.91s	5.77s	14.20s	143.59s
Radix Heap	Run 1	0.76s	6.31s	13.80s	152.58s
	Run 2	0.78s	5.60s	12.79s	73.23s
Fibonacci Heap	Run 1	0.90s	6.46s	15.66s	78.97s
	Run 2	0.86s	5.91s	14.73s	50.16s

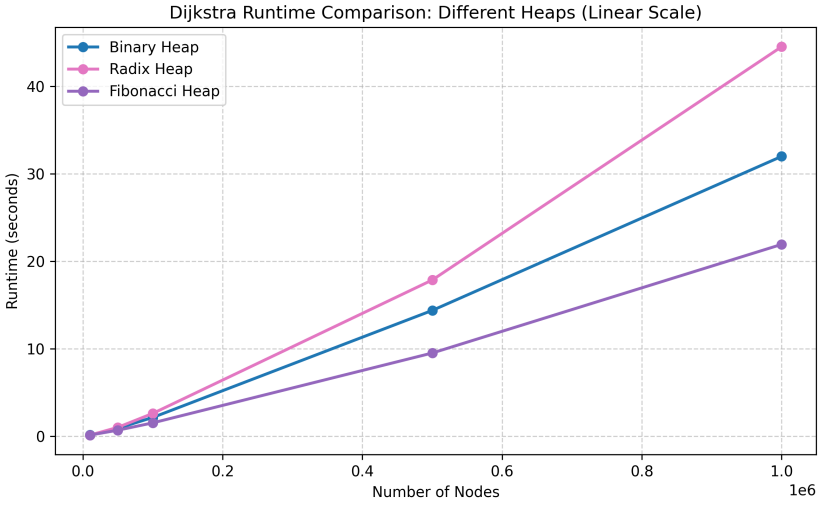

Fig. 4. Runtime comparison of Dijkstra's Algorithm (Run 1)

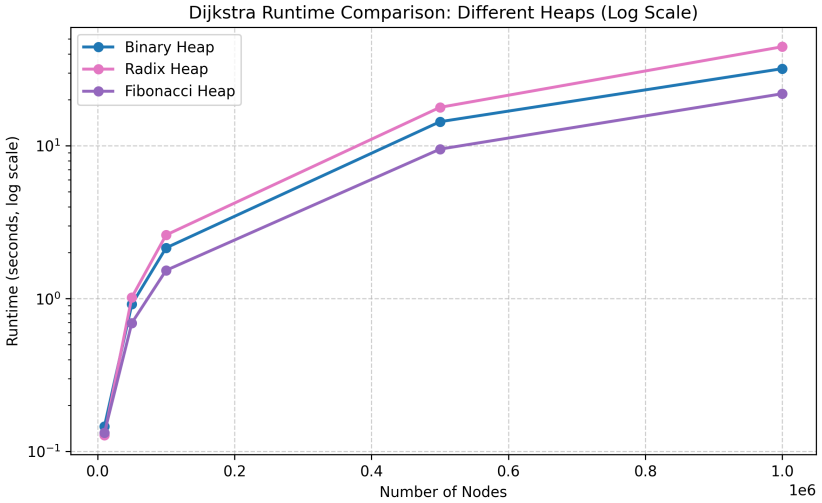

Fig. 5. Runtime comparison of Dijkstra's Algorithm (Run 2)

5 Visualization

The visualization system uses the Pygame library to provide an interactive, real-time comparison of shortest path algorithms on a customizable grid-based graph.

Each algorithm is implemented as a Python generator that yields intermediate states at each iteration. The grid is represented as an $N \times N$ matrix of nodes, which is converted to an adjacency list for algorithm execution. During visualization, nodes are color-coded to reflect their state: orange for the start node, turquoise for the end, black for obstacles, red for visited nodes, green for frontier nodes, and purple for the final shortest path. This allows users to observe how each algorithm

explores the search space differently—Dijkstra expands uniformly from the source, Bidirectional Dijkstra searches from both endpoints simultaneously, and Contraction Hierarchies shows very little exploration due to its preprocessing. For Contraction Hierarchies specifically, the visualization includes gold lines connecting nodes with shortcuts, demonstrating how the algorithm "jumps" across the graph using preprocessed edges before unpacking them into the complete path.

To provide accurate performance comparisons without visualization overhead, each algorithm runs twice: first non-visually to capture timing metrics, then as a generator for step-by-step rendering. The system supports command-line configuration of grid size and heap type (binary, Fibonacci, or radix), allowing users to easily compare different configurations. Users interact using mouse clicks to place walls and start/end nodes, with keyboard controls for algorithm selection and execution. Preset mazes and random wall generation allow for repeatable and rapid testing, respectively.

This visualization tool serves as both an educational resource for understanding algorithmic behavior and a practical benchmarking environment to analyze performance for different graph structures.

6 Discussion

7 Conclusion

GitHub Repository

The full implementation (data structures, Dijkstra's algorithm, unit tests, and benchmark plots) is available at: https://github.com/aaravg31/shortest_path_comparison

Acknowledgements

We would like to acknowledge ChatGPT [5] for assistance with code explanations, debugging, and report language refinement.

References

[1] Carlos da Costa-Luis et al. 2024. tqdm: A Fast, Extensible Progress Bar for Python and CLI. https://tqdm.github.io. Used for progress visualization during graph generation..

[2] OpenGenus Foundation. 2023. *Fibonacci Heap and its Operations Explained.* https://iq.opengenus.org/fibonacci-heap/ Accessed: 2025-10-24.

[3] GeeksforGeeks. 2023. *Binary Heap - Data Structures and Algorithms.* https://www.geeksforgeeks.org/dsa/binary-heap/ Accessed: 2025-10-24.

[4] GeeksforGeeks. 2023. *Fibonacci Heap - Introduction and Operations.* https://www.geeksforgeeks.org/dsa/fibonacci-heap-set-1-introduction/ Accessed: 2025-10-24.

[5] OpenAI. 2025. ChatGPT (Mar 2025 Version). Large language model assistance used for explanation, debugging, and report editing.. https://chat.openai.com.