Optimizing Dijkstra: A Comparative Analysis of Priority Queues, Bidirectional Search, and Contraction Hierarchies

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. We also examine a bidirectional Dijkstra variant that employs skewed search frontiers. Finally, we implement Contraction Hierarchies (CH), a hierarchical speed-up technique. All data structures are implemented from scratch in Python and evaluated on large randomly generated weighted graphs designed to approximate real-world network structures. Theoretical time and space bounds are contrasted with empirical behaviour, using runtime and peak memory usage as the primary metrics.

Additional Key Words and Phrases: Dijkstra's Algorithm, Binary Heap, Radix Heap, Fibonacci Heap, Bidirectional Skewness, Contraction Hierarchy, 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 bucket-based priority queue for monotone integer keys, exploiting the fact that Dijkstra's distances never decrease. It provides amortized $O(1)$ insert and `decrease_key` and $O(\log C)$ `extract_min`, where C is the maximum key difference.

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.

Finally, we implement **Contraction Hierarchies**, a speed-up technique that relies on preprocessing the graph to create a hierarchy of nodes, allowing for extremely fast queries by ignoring unimportant nodes during the search.

Overall, this project aims to provide both a theoretical and empirical comparison of these data structures and algorithms, evaluating their runtime, scalability, and memory efficiency on 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

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.

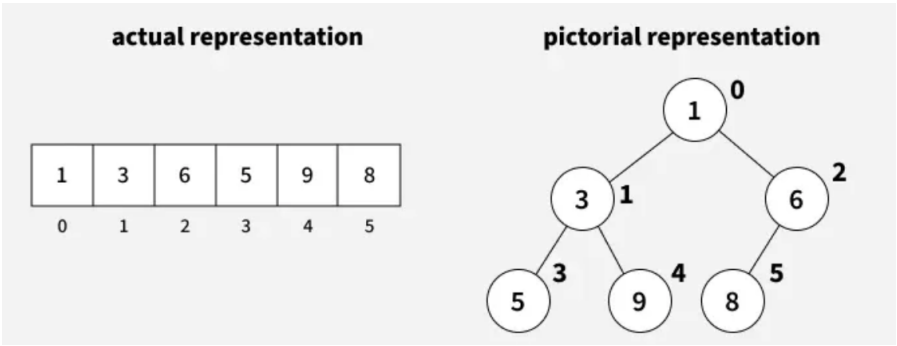

Fig. 1. Binary Heap: array-based and pictorial representations (adapted from GeeksforGeeks [4]).

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.

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.

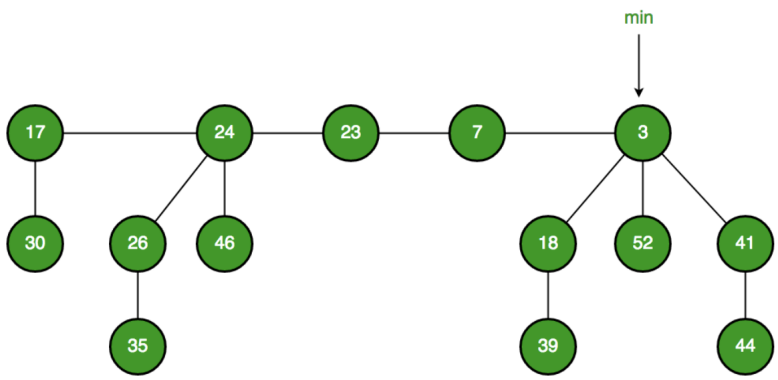

Fig. 2. Example of a Fibonacci Heap with multiple trees in the root list. The node with key 3 is the global minimum (adapted from GeeksforGeeks [5]).

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.

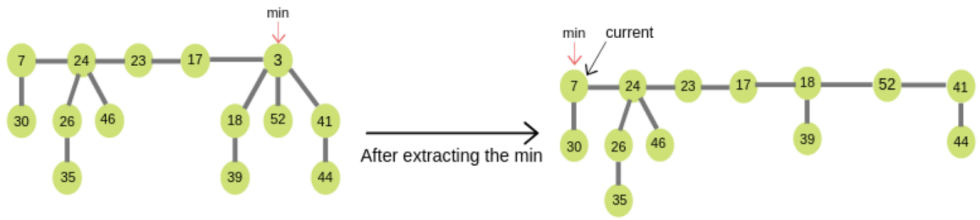

Fig. 3. Illustration of the `extract_min` operation in a Fibonacci Heap. The minimum node (3) is removed, and its children are added to the root list, followed by consolidation of trees with equal degree (adapted from OpenGenus [3]).

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.

Space Complexity. The space usage of a Fibonacci Heap is primarily determined by the number of nodes and the auxiliary pointers stored in its tree structure. Each node maintains:

- a pointer to its parent,
- a pointer to one of its children,
- left and right pointers for the circular doubly linked root list,
- a degree value (number of children),
- a mark bit for cascading cuts.

Every node therefore requires a constant amount of additional metadata. Since the total number of nodes in the heap is n, the space complexity is linear:

$$S(n) = O(n)$$

The consolidation process during `extract_min` temporarily requires an auxiliary array of size $O(\log n)$ to store pointers to trees grouped by degree. However, this memory is negligible compared to the $O(n)$ main structure and does not affect the overall asymptotic bound.

In summary, Fibonacci Heaps trade increased constant-factor memory usage for improved amortized time complexity, storing more structural information than Binary Heaps but remaining within the same linear space bound.

2.4 Radix Heap

A **Radix Heap** is a priority queue designed for algorithms such as Dijkstra's where extracted keys are *monotone*, meaning they never decrease. Instead of maintaining a tree structure, a Radix Heap organizes elements into a sequence of *buckets*, where each bucket corresponds to an exponentially increasing range of integer keys. This bucket-based structure allows the heap to efficiently locate the minimum key while taking advantage of the monotonicity property inherent to shortest-path algorithms with non-negative integer edge weights.

The Radix Heap was introduced by Ahuja, Mehlhorn, Orlin, and Tarjan [1], who showed that this structure supports efficient priority queue operations when the difference between consecutive extracted keys is bounded. Figure 4 illustrates an example of the bucket layout, where each bucket maintains keys whose values fall within a specific integer interval.

Insertion. To insert a key, the heap identifies the bucket whose range $[b_i, b_{i+1})$ contains the key and appends the element to that bucket. Because bucket ranges grow exponentially and keys only increase relative to the last extracted minimum, each element moves to lower-index buckets only a limited number of times. Amortized over all operations, insertion runs in constant time:

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

`extract_min`. The minimum key is always located in the first bucket. If this bucket is empty, the heap performs a *refill* step by locating the next non-empty bucket, determining the minimum key within it, updating the bucket boundaries, and redistributing its contents into earlier buckets. Although this redistribution may involve scanning an entire bucket, each element participates in only a small number of such redistributions. The amortized time for `extract_min` is therefore logarithmic in the maximum key difference:

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

where C is the maximum difference between successive extracted keys (often bounded by the maximum edge weight).

`decrease_key`. The `decrease_key` operation inserts a new, smaller key for an existing element while marking the old entry as stale. Since new keys can only move to buckets of lower or equal index, and each element shifts buckets only a limited number of times, the amortized cost of `decrease_key` matches that of insertion:

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

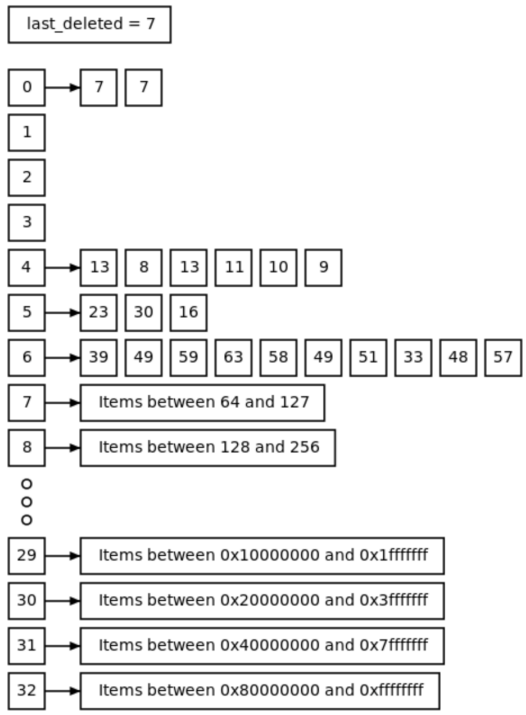

Fig. 4. Radix Heap bucket structure: each bucket represents an exponentially increasing range of integer keys (adapted from ImpulseTrain [8]).

Space Complexity. A Radix Heap contains

$$B = \lfloor \log_2 C \rfloor + 2$$

buckets, each storing elements whose keys fall within an associated integer interval. Because each element appears only once (aside from lazily removed stale entries), and the number of buckets is logarithmic in C, the overall space complexity is linear in the number of elements:

$$S(n) = O(n).$$

Radix Heaps provide strong practical performance for Dijkstra's algorithm on graphs with bounded integer edge weights. Their bucket-based organization avoids the pointer-heavy overhead of Fibonacci Heaps while reducing the logarithmic costs found in Binary Heaps. Theoretical performance follows the bound $O(m + n \log C)$ for one-level Radix Heaps, with excellent efficiency when the key range is small.

2.5 Bidirectional Dijkstra

Bidirectional Dijkstra [9] is an optimization of the standard Dijkstra algorithm that executes two simultaneous searches: a **forward search** from the source node and a **backward search** from the target node. The algorithm terminates when the two search frontiers meet, often significantly reducing the number of visited nodes compared to the standard, unidirectional search.

Search Strategy. In a standard Dijkstra search, the explored area expands as a circle centered at the source. If the target is at distance r, the search visits approximately πr^2 nodes. In contrast, Bidirectional Dijkstra expands two smaller circles of radius $r/2$, visiting approximately $2 \times \pi(r/2)^2 = \pi r^2/2$ nodes, potentially halving the search space.

Skewed Expansion. Standard bidirectional search typically alternates between the forward and backward heaps in a round-robin order (1:1). However, this may be inefficient if one frontier expands into a sparse region while the other is stuck in a dense one. **Skewed Bidirectional Dijkstra** addresses this by dynamically balancing the search. The direction of expansion is chosen based on the relative sizes of the priority queues:

$$|Q_f| \cdot (1 - \sigma) \leq |Q_b| \cdot \sigma$$

where $|Q_f|$ and $|Q_b|$ are the sizes of the forward and backward heaps, and $\sigma \in [0, 1]$ is a skew parameter.

- $\sigma = 0.5$: Balanced expansion (expands the smaller frontier).
- $\sigma < 0.5$: Biased towards backward expansion.
- $\sigma > 0.5$: Biased towards forward expansion.

This heuristic tries to keep the search frontiers balanced in terms of cost rather than just distance.

Termination. The algorithm maintains a value μ, representing the shortest path distance found so far connecting the two search trees. The search terminates when the sum of the minimum keys from both heaps exceeds μ:

$$\min(Q_f) + \min(Q_b) \geq \mu$$

At this point, no shorter path can exist, and μ is returned as the optimal distance.

2.6 Contraction Hierarchies

Contraction Hierarchies (CH) [6] is a speed-up technique designed for static road networks. It relies on a computationally expensive **preprocessing phase** to accelerate the following **query phases**. The core idea is to create a hierarchy of nodes where queries only need to visit nodes that are "more important" than the current node, minimizing the search space.

Preprocessing Phase. The preprocessing phase orders all nodes by "importance" and iteratively **contracts** them. Contracting a node u involves removing it from the graph and adding **shortcuts** between its neighbors $\{v_i\}$ to preserve shortest path distances. A shortcut (v_i, v_j) with weight $w(v_i, u) + w(u, v_j)$ is added if the path through u is the unique shortest path between v_i and v_j.

The importance of a node is determined using a heuristic that penalizes adding too many shortcuts:

$$I(u) = (\text{Shortcuts Added}) - (\text{Edges Removed}) + (\text{Contracted Neighbors})$$

Nodes are contracted in increasing order of importance (from least to most important). This process effectively pushes local, unimportant nodes (like residential streets) to the bottom of the hierarchy, while major hubs remain at the top.

Query Phase. The query phase executes a bidirectional Dijkstra search on the augmented graph (original edges + shortcuts), but with a strict rank constraint:

- **Forward Search:** Only relaxes edges (u, v) where $\text{rank}(u) < \text{rank}(v)$.
- **Backward Search:** Only relaxes edges (u, v) in the reverse graph where $\text{rank}(u) < \text{rank}(v)$.

This "upward-only" search strategy ensures that the algorithm quickly ascends the hierarchy, ignoring the majority of low-ranking nodes. The forward and backward searches meet at the node with the highest rank on the shortest path.

Complexity.

- **Preprocessing:** Heuristic but typically takes many minutes for millions of nodes.
- **Query:** Extremely fast, often sub-millisecond on road networks, as the search space is reduced from millions of nodes to just a few hundred or thousand.

3 Methodology

3.1 Heap Implementation

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.

3.2 Algorithm Implementation

3.2.1 Dijkstra. 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.

3.2.2 Bidirectional Dijkstra. The `BidirectionalDijkstra` class manages two separate priority queues, `f_heap` and `b_heap`, with corresponding distance dictionaries `f_dist` and `b_dist`. To support skewed expansion, the algorithm dynamically selects which frontier to expand based on the skew parameter σ, as described in section 2.5. The decision logic is implemented as:

$$|Q_f| \times (1 - \sigma) \leq |Q_b| \times \sigma$$

This allows the search to prioritize the smaller frontier or bias towards one direction. The search terminates when the sum of the minimum keys from both heaps exceeds the shortest path found so far (μ):

$$\min(Q_f) + \min(Q_b) \geq \mu$$

3.2.3 Contraction Hierarchies. The `ContractionHierarchy` class is divided into two distinct phases: preprocessing and querying.

Preprocessing. The `preprocess()` method iteratively contracts nodes based on their importance. Importance is calculated dynamically using the heuristic discussed in section 2.6:

$$I(u) = \text{shortcuts\_added} - \text{edges\_removed} + \text{contracted\_neighbors}$$

To determine if a shortcut is necessary between two neighbors u and v when contracting node w, the algorithm runs a local search (`_local_dijkstra`) to check if an alternative path exists that is shorter than or equal to the path through w.

Querying. The `query()` method executes a modified bidirectional Dijkstra search. It enforces the rank property by only relaxing edges (u, v) where $\text{rank}(u) < \text{rank}(v)$ in the forward search, and similarly in the reverse graph for the backward search. This ensures the search space is restricted to the "upward" path in the hierarchy.

3.3 Graph Generation

To evaluate scalability and performance, large directed weighted graphs were generated using custom functions implemented in `graph_generator.py`. Each graph is represented using a standard adjacency-list format:

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

where each node u maps to a list of outgoing edges (v, w), with weights w sampled uniformly from $[1, 10]$.

Two different random graph models were implemented:

- **Erdős–Rényi (ER) Model** — Edges are added uniformly at random. This produces graphs with Poisson degree distributions, making ER suitable for controlled benchmarking due to predictable sparsity and generation speed.
- **Barabási–Albert (BA) Model** — A preferential-attachment model that produces heavy-tailed, scale-free degree distributions similar to many real-world networks. This model offers improved structural realism at the cost of significantly higher generation time for large n.

Choice of Graph Model for Experiments. While both ER and BA generators were implemented, the full experimental evaluation uses the **ER model** exclusively. The BA model becomes computationally expensive at large scales (hundreds of thousands to millions of nodes) due to the need to maintain and repeatedly sample from a dynamically weighted degree distribution. On typical hardware, generating BA graphs of size comparable to ER graphs can take several minutes to hours.

Thus, BA graphs were used only for smaller exploratory tests, while all large-scale benchmarking relies on ER graphs for consistent generation speed and reproducibility.

Generation Parameters. For each experiment, the number of edges was selected using:

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

yielding sparse graphs suitable for evaluating shortest-path algorithms. Progress bars were displayed using `tqdm` [2] during large-scale generation.

The main experimental dataset consists of ER graphs of the following sizes:

- 10K nodes → ≈80K edges
- 50K nodes → ≈500K edges
- 100K nodes → ≈1.2M edges
- 500K nodes → ≈7.5M edges
- 1M nodes → ≈20M edges
- 2M nodes → ≈50M edges

Larger ER 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.

4 Results

All experiments were executed on a MacBook Pro M2 with 16 GB of unified memory. Both runtime and peak memory usage were recorded for each heap implementation using wall-clock

time and Python's tracemalloc module. For graphs with up to 500,000 nodes, each configuration (algorithm–heap pair) was executed three times, and we report the average runtime and peak memory to reduce noise from background processes. For larger graphs with 1M and 2M nodes, each configuration was run once due to the substantial computation time and memory required.

Table 1. Average runtime (in seconds) of Dijkstra and Bidirectional Dijkstra with Binary, Radix, and Fibonacci Heaps. For $V \leq 500$K, values are averaged over three runs; for $V \geq 1$M, a single run is reported.

Heap Type	Algorithm	10K	50K	100K	500K	1M	2M
Binary Heap	Dijkstra	0.14s	0.96s	2.24s	14.87s	33.48s	200.38s
	Bidirectional	0.05s	0.39s	1.00s	7.99s	22.73s	470.46s
Radix Heap	Dijkstra	0.12s	1.07s	2.76s	18.39s	44.53s	271.37s
	Bidirectional	0.06s	0.43s	1.11s	8.87s	24.39s	482.72s
Fibonacci Heap	Dijkstra	0.13s	0.67s	1.61s	9.74s	21.71s	123.56s
	Bidirectional	0.05s	0.40s	1.01s	8.08s	21.82s	313.88s

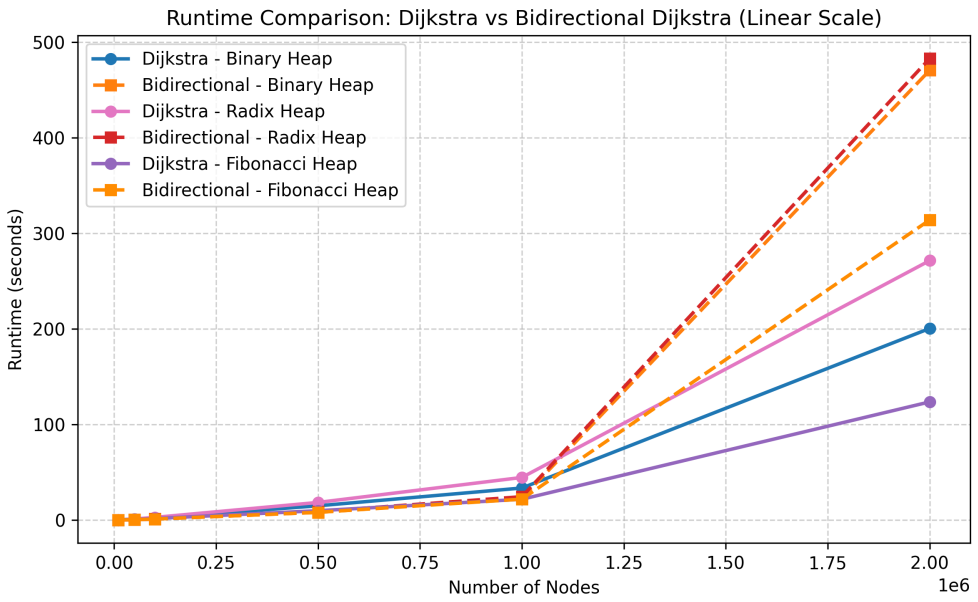

Fig. 5. Runtime comparison of Dijkstra and Bidirectional Dijkstra (skewed search) using Binary, Radix, and Fibonacci Heaps on a linear scale.

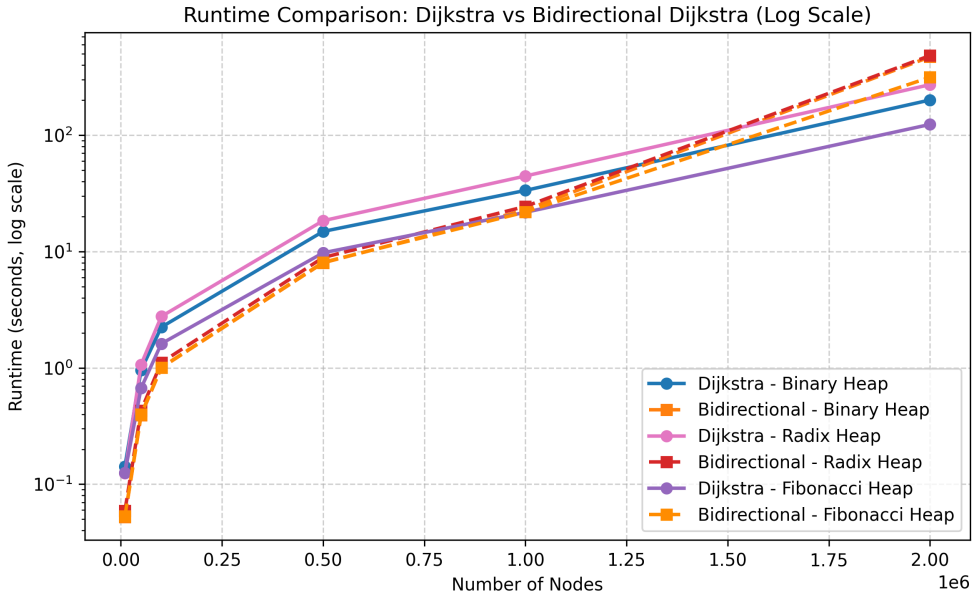

Fig. 6. Log-scale runtime comparison of Dijkstra and Bidirectional Dijkstra across increasing graph sizes for all heap types.

Table 2. Peak memory usage (in MiB) of Dijkstra and Bidirectional Dijkstra with Binary, Radix, and Fibonacci Heaps. For $V \leq 500$K, values correspond to the stable peak observed across three runs; for $V \geq 1$M, a single run is reported.

Heap Type	Algorithm	10K	50K	100K	500K	1M	2M
Binary Heap	Dijkstra	1.07	9.20	18.96	79.30	160.08	321.93
	Bidirectional	5.92	37.41	86.89	522.97	1351.56	3294.57
Radix Heap	Dijkstra	1.60	11.76	25.86	152.86	319.40	579.09
	Bidirectional	5.90	37.15	86.58	519.41	1342.25	3296.11
Fibonacci Heap	Dijkstra	1.36	10.53	21.71	91.63	185.27	375.67
	Bidirectional	5.91	37.52	87.01	523.98	1353.41	3294.99

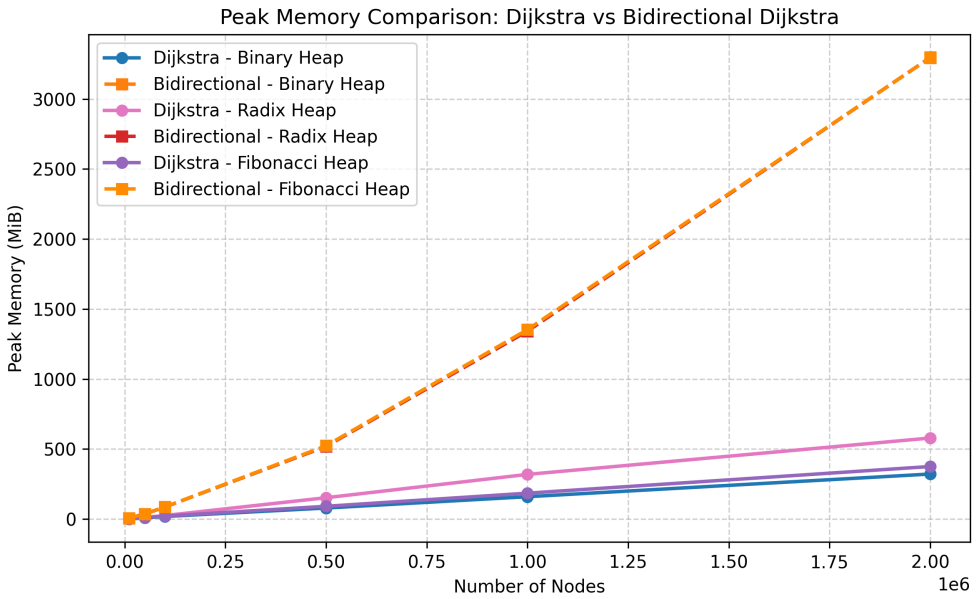

Fig. 7. Peak memory usage for Dijkstra and Bidirectional Dijkstra using Binary, Radix, and Fibonacci Heaps.

4.1 Contraction Hierarchies Results

Due to the high computational cost of the preprocessing phase in Python, Contraction Hierarchies were evaluated on smaller graph sizes. Table 3 illustrates the trade-off: while preprocessing time grows rapidly, the resulting query times are orders of magnitude faster than standard Dijkstra.

Table 3. Contraction Hierarchies performance on small graphs. Note the rapid increase in preprocessing time compared to the sub-millisecond query times.

Nodes	Preprocessing Time (s)	Avg Query Time (s)
100	0.21	0.00005
500	53.71	0.00073
1000	759.47	0.00156

Based on the observed data, the preprocessing time T scales with the number of edges E according to the power law:

$$T \approx 1.59 \times 10^{-8} \cdot E^{2.74}$$

This super-quadratic growth ($O(E^{2.74})$) shows that preprocessing becomes prohibitively expensive for larger graphs in our Python implementation, despite the query times remaining extremely low. The high exponent is likely due to the overhead of repeated local Dijkstra searches and the dynamic addition of shortcut edges, which increases the graph density during the contraction process.

5 Discussion

The empirical results reveal clear performance differences among the Binary Heap, Radix Heap, and Fibonacci Heap when used within Dijkstra's algorithm. While all three structures scale roughly

as expected theoretically, real-world implementation factors, memory usage, and workload characteristics significantly influence their actual performance.

Across all graph sizes, the Fibonacci Heap consistently delivered the fastest runtimes, particularly on larger graphs where `decrease_key` operations dominate computation. This matches theoretical expectations: with amortized $O(1)$ `decrease_key` and $O(\log n)$ `extract_min`, Fibonacci Heaps become increasingly advantageous as the edge count grows.

The Binary Heap showed predictable and stable performance, with runtimes growing steadily with graph size. Its simple contiguous array structure, low constant factors, and cache-friendly memory layout make it competitive at small and moderate scales. Nevertheless, as the graph size reached the upper bounds of the experiment (1–2M nodes), its $O(\log n)$ `decrease_key` penalties accumulated and caused the runtime to grow faster than that of the Fibonacci Heap. Despite this, Binary Heaps remained significantly more memory-efficient due to requiring only one integer and one pointer per entry.

The Radix Heap, although theoretically promising for monotone integer priority workloads, consistently performed worse than both other heaps in these experiments. There are several reasons for this behavior:

- **Bucket Redistribution Overhead:** Radix Heaps require periodic "refill" operations that scan bucket contents, recompute bucket boundaries, and reinsert elements. Although amortized complexity remains low, the constant cost of bucket redistribution is substantial in Python.
- **Python Implementation Limitations:** The Radix Heap relies heavily on list operations, repeated scans, and dictionary lookups for stale-entry handling. These operations are significantly slower in Python than the pointer-based operations in Fibonacci Heaps.
- **Large Key Range:** Since key ranges in dense Erdős–Rényi graphs grow rapidly as distances accumulate, the number of buckets increases ($\lfloor \log_2 C \rfloor + 2$). Larger bucket arrays lead to more redistribution work and more frequent boundary updates.
- **High Memory Footprint:** The Radix Heap used nearly twice as much memory as the Binary Heap, leading to poorer cache locality and additional overhead during Python's memory management cycles.

These factors explain why the Radix Heap, despite excellent performance in low-level languages like C or C++, performs less favorably in Python. In this environment, the higher constant factors overshadow its theoretical monotone advantages.

Memory usage results further reinforce these conclusions. The Binary Heap consumed the least memory due to its simple array representation, followed by the Fibonacci Heap, which requires multiple pointers per node but uses them efficiently during consolidation. The Radix Heap consumed the most memory, sometimes nearly double, due to its dictionary-based stale-entry tracking and large bucket arrays. This elevated memory usage not only increases the overall footprint but also contributes to slower runtime through decreased cache efficiency and increased garbage collection activity.

Overall, the experiments highlight that while theoretical bounds are informative, real-world performance is shaped by constant factors, implementation language, memory behavior, and workload characteristics. Under Python, the Fibonacci Heap offers the best runtime scalability, the Binary Heap provides predictable and memory-efficient performance, and the Radix Heap, though

algorithmically suitable for monotone priority queues, incurs enough overhead to underperform relative to the other two structures.

5.1 Bidirectional vs. Unidirectional Search

Bidirectional Dijkstra with skewed expansion demonstrated a clear performance advantage over standard Dijkstra for graph sizes up to 1 million nodes, often reducing runtime by 30-50%. This speedup is attributed to the reduced search space—expanding two smaller circles rather than one large one. However, at 2 million nodes, the trend reversed. The memory consumption of Bidirectional Dijkstra spiked to over 3 GB (compared to ≈300 MB for standard Dijkstra), likely causing cache thrashing and memory management overhead that outweighed the algorithmic benefits. This highlights a critical practical limitation: bidirectional search requires maintaining two distinct distance maps and priority queues, effectively doubling the memory footprint and increasing the cost of set operations.

5.2 Contraction Hierarchies Performance

Our experiments with Contraction Hierarchies (CH) revealed the extreme trade-off between preprocessing and query time. While queries on the CH graph were near-instantaneous (often sub-millisecond), the preprocessing phase was computationally expensive. In our Python implementation, preprocessing even small graphs (e.g., 500-1000 nodes) took significantly longer than running a standard Dijkstra search on a much larger graph. This confirms that CH is highly effective for static graphs where the cost of preprocessing can be amortized over millions of queries, but it is ill-suited for dynamic graphs or scenarios where the graph changes frequently. The overhead of the Python interpreter further exacerbates the preprocessing cost, suggesting that a lower-level language like C++ is essential for practical CH implementations on large networks.

6 Conclusion

This study provides a comprehensive comparison of priority queue structures and search strategies for Dijkstra's algorithm on large sparse graphs. Across all experiments, the Fibonacci Heap delivered the strongest runtime performance in Python, particularly at scale, due to its amortized $O(1)$ `decrease_key` and efficient consolidation. The Binary Heap remained a robust and memory-efficient baseline, offering predictable behavior and outperforming the other heaps in terms of space usage, though its $O(\log n)$ updates limited its scalability on the largest graphs.

The Radix Heap, while theoretically appealing for monotone integer keys, consistently underperformed in Python due to bucket redistribution overhead and high memory consumption—factors that overshadowed its amortized constant-time operations. Bidirectional Dijkstra provided notable runtime improvements on small and medium-sized graphs but became impractical at very large scales due to its doubled memory footprint. Finally, Contraction Hierarchies achieved exceptional query performance but required prohibitively expensive preprocessing, making them suitable only for static networks where preprocessing can be amortized.

Overall, the results highlight how implementation details, memory behavior, and language-level overhead shape the practical performance of theoretically efficient algorithms.

7 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.

A demo of the visualization tool can be found at https://youtu.be/A_zrF0KwM_s

8 Team Contributions

- **Aarav Gosalia:** Implemented the data structures, Dijkstra, and other utility tools (graph generator, runtime analysis).
- **Riley Eaton:** Implemented Bidirectional Dijkstra, Contraction Hierarchies, and developed the visualization application.

Both team members contributed to writing the report, preparing slides, and reviewing each other's sections.

GitHub Repository

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

Acknowledgements

We would like to acknowledge the use of ChatGPT [7] for assisting with documentation, debugging, and report language refinement.

References

[1] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, and Robert E. Tarjan. 1990. Faster Algorithms for the Shortest Path Problem. *J. ACM* 37, 2 (1990), 213–223. https://doi.org/10.1145/77600.77615 Primary source for Radix Heaps as monotone priority queues for Dijkstra's algorithm.

[2] 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..

[3] OpenGenus Foundation. 2023. *Fibonacci Heap and its Operations Explained.* https://iq.opengenus.org/fibonacci-heap/

[4] GeeksforGeeks. 2023. *Binary Heap - Data Structures and Algorithms.* https://www.geeksforgeeks.org/dsa/binary-heap/

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

[6] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Experimental Algorithms (Lecture Notes in Computer Science, Vol. 5038).* Springer, 319–333. https://doi.org/10.1007/978-3-540-68552-4_24

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

[8] Soren Sandmann Pederson. 2013. *Radix Heaps.* https://ssp.impulsetrain.com/radix-heap.html Explanation and diagrams used for understanding bucket boundaries and monotone queue behavior.

[9] Ira Pohl. 1971. Bi-directional Search. In *Machine Intelligence 6*, Bernard Meltzer and Donald Michie (Eds.). Edinburgh University Press, 127–140.