Comparative Analysis of Binary, Pairing, and Fibonacci Heaps in Dijkstra's Shortest Path Algorithm

AARAV GOSALIA, University of British Columbia Okanagan, Canada

This report presents a comparative analysis of three advanced priority queue data structures, Binary Heap, Pairing Heap, and Fibonacci Heap, which were implemented from scratch in Python and evaluated using Dijkstra's shortest path algorithm. Each structure was analyzed theoretically and empirically for runtime, efficiency, and scalability. Random weighted graphs of increasing sizes (100K–2M nodes) were generated to benchmark performance. Results demonstrate that while Fibonacci Heaps achieve the best theoretical asymptotic bounds, Pairing Heaps outperform in practice due to lower constant factors and simpler pointer manipulations. All experiments, source code, and plots are available in the public GitHub repository.

Additional Key Words and Phrases: Dijkstra's Algorithm, Binary Heap, Pairing Heap, Fibonacci Heap, Data Structures, Graph Algorithms, Priority Queues

1 Introduction

Dijkstra's shortest path algorithm is a cornerstone of graph theory and network optimization, used in applications ranging from GPS routing to network packet scheduling. Its efficiency depends heavily on the data structure used to maintain the priority queue of vertices. This report explores and compares three different heap implementations used to optimize Dijkstra's algorithm:

(1) **Binary Heap** – A classic implementation used in most practical systems.
(2) **Pairing Heap** – A self-adjusting heap offering efficient amortized operations.
(3) **Fibonacci Heap** – A theoretically optimal structure with $O(1)$ amortized decrease_key.

The objective of this assignment was to implement each data structure *from scratch*, integrate it with Dijkstra's algorithm, and analyze both theoretical and empirical performance on randomly generated graphs.

2 Background and Theory

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

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 $2i + 1$

Author's Contact Information: Aarav Gosalia, University of British Columbia Okanagan, Kelowna, Canada.

- 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.

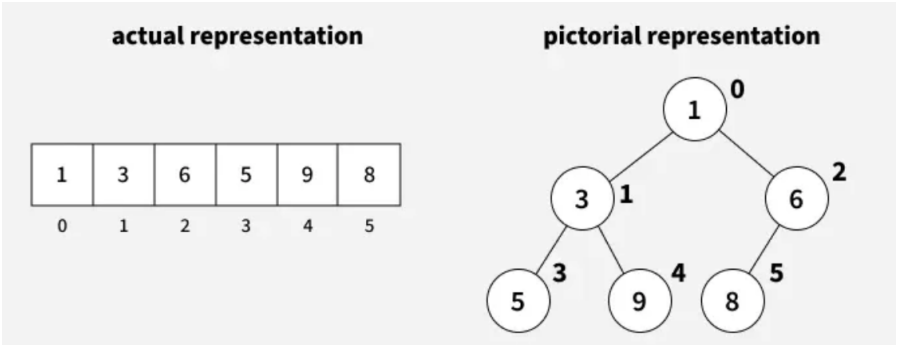

Fig. 1. Binary Heap: array-based and pictorial representations (adapted from GeeksforGeeks [3]).

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.

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

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 **Pairing 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.

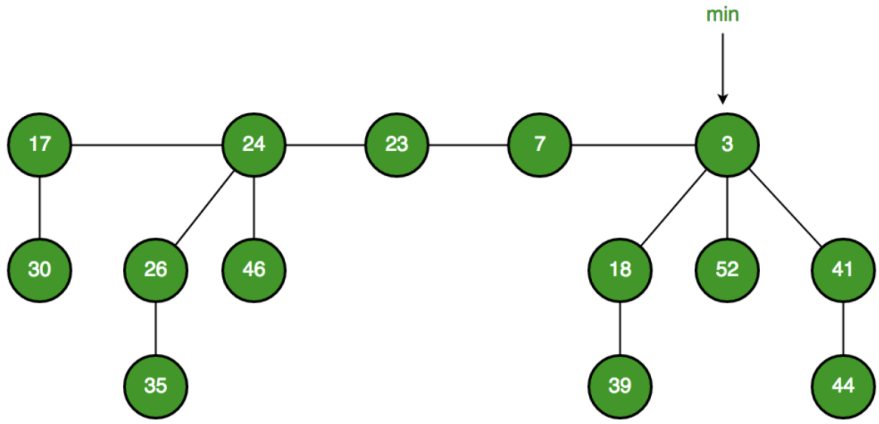

Fig. 2. Example of a Fibonacci Heap with multiple trees in the root list. The node with key 3 is the global minimum (adapted from GeeksforGeeks [4]).

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.

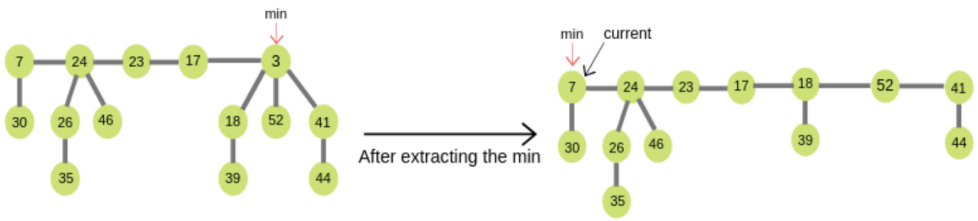

Fig. 3. Illustration of the `extract_min` operation in a Fibonacci Heap. The minimum node (3) is removed, and its children are added to the root list, followed by consolidation of trees with equal degree (adapted from OpenGenus [2]).

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 or Pairing Heaps.

2.4 Pairing Heap

A **Pairing Heap** is a self-adjusting heap structure that combines simplicity with efficient amortized performance. Unlike Binary or Fibonacci Heaps, it uses a flexible multiway tree structure and performs structural adjustments dynamically after each operation rather than maintaining strict balancing rules. Each node in a Pairing Heap may have multiple children, stored as a linked list of siblings. The structure evolves naturally based on *pairing* operations rather than explicit rebalancing.

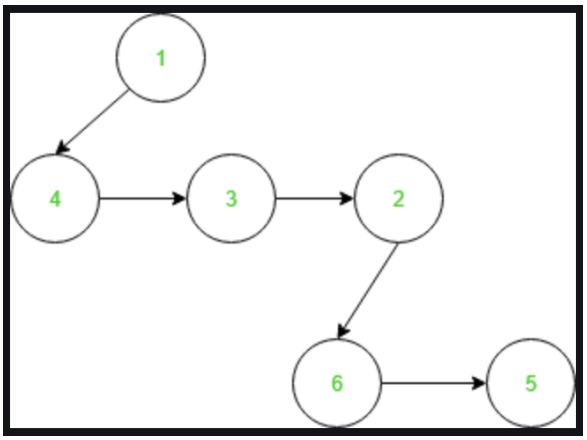

Fig. 4. Example structure of a Pairing Heap showing a multiway tree layout (adapted from GeeksforGeeks [5]).

The Pairing Heap relies on two fundamental procedures:

- **Compare and Link:** When combining two heaps, the root with the smaller key becomes the parent, and the other becomes its leftmost child. This is the basic building block for all heap operations.
- **Combine Siblings:** During deletion, after removing the minimum element, its subtrees (children) are merged pairwise in a left-to-right pass and then combined again in a right-to-left pass. This two-phase merging ensures logarithmic behavior over multiple operations.

Insertion. To insert a new element, a single-node heap is created and then linked to the existing root using the **compare and link** operation. Since only one comparison and linking are performed, insertion is extremely efficient in practice and runs in constant amortized time:

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

Extract_Min. The `extract_min` operation removes the smallest key, which is always located at the root. After removal, all children of the root become separate subtrees. These subtrees are then merged using the **combine siblings** process, which repeatedly applies the **compare and link** operation in two passes: first pairing adjacent siblings from left to right, and then merging the resulting heaps from right to left.

This two-phase merging process ensures logarithmic amortized complexity since each node participates in only a limited number of link operations over time:

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

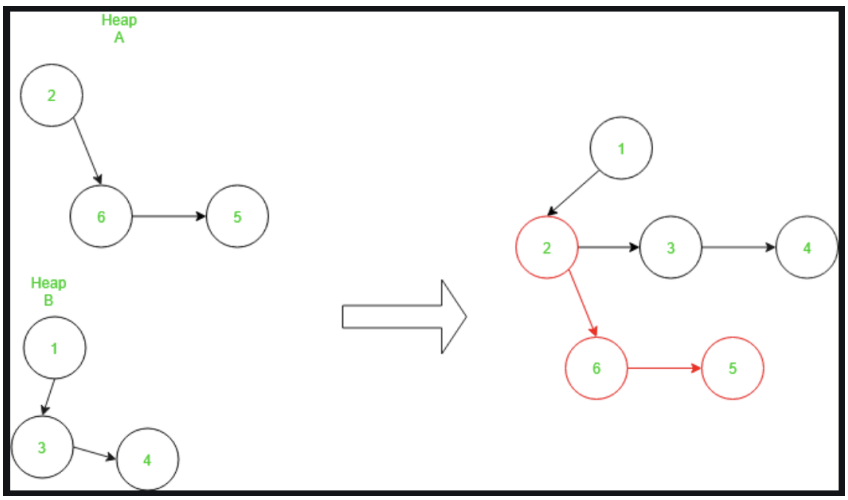

Fig. 5. Illustration of the Pairing Heap merge process after `extract-min`. Two heaps (A and B) are combined using the **compare and link** operation, where the smaller root becomes the parent, and the resulting subtrees are linked in a left-child/right-sibling structure (adapted from GeeksforGeeks [5]).

decrease_Key. To decrease the key of a node, the node is cut from its parent (if necessary) and merged with the root using the **compare and link** operation. Unlike Fibonacci Heaps, the Pairing Heap does not rely on cascading cuts or auxiliary data structures, making this process conceptually simpler and maybe faster in practice.

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

Pairing Heaps strike a balance between simplicity and efficiency. They are easy to implement, require minimal pointer manipulation, and adapt dynamically to usage patterns.

While their theoretical guarantees are not as strong as Fibonacci Heaps, they might perform comparably or even faster in real-world settings, particularly in applications such as Dijkstra's algorithm where `decrease_key` operations dominate.

3 Methodology

All three heaps, Binary, Pairing, 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
Pairing 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

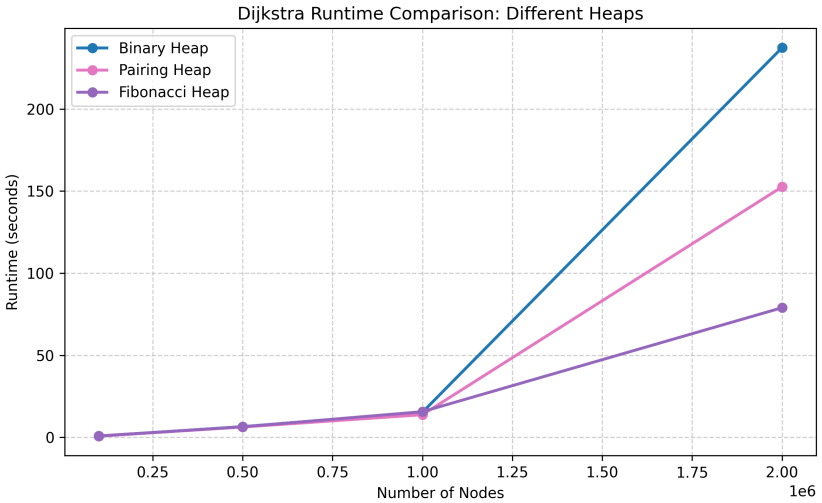

Fig. 6. Runtime comparison of Dijkstra's Algorithm (Run 1)

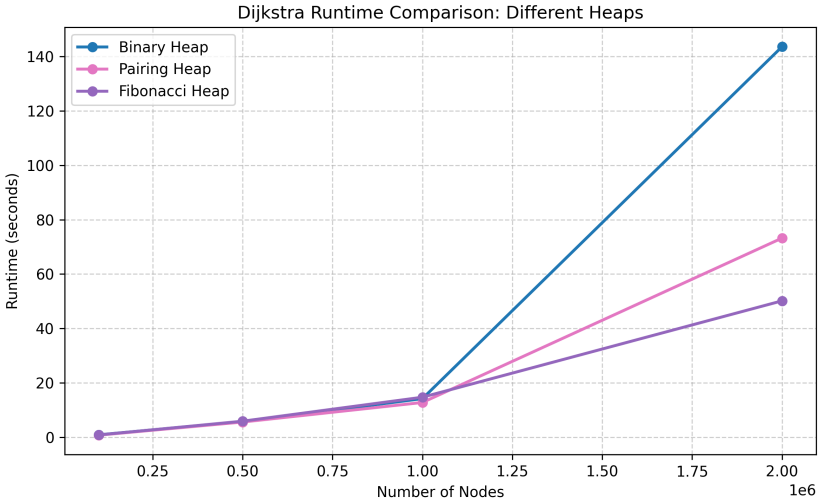

Fig. 7. Runtime comparison of Dijkstra's Algorithm (Run 2)

5 Discussion

The experimental results align closely with theoretical expectations but also highlight how implementation details and constant factors shape real-world performance.

At smaller graph sizes (up to one million nodes), all three heaps performed nearly identically. At this scale, the simplicity of the Binary Heap provides minimal overhead, while the advanced pointer manipulation in the Pairing and Fibonacci Heaps offers no significant advantage. All priority queue operations execute quickly, and the effect of amortized improvements remains negligible.

As graph size increased, performance differences became clearer. The **Binary Heap**, though conceptually straightforward and efficient for moderate input sizes, exhibited the steepest rise in runtime. This is because each `decrease_key` and `extract_min` operation incurs a logarithmic restructuring cost, which compounds quickly in dense graphs.

In contrast, the **Pairing Heap** maintained stable and predictable scaling across both runs. Its self-adjusting nature and minimal bookkeeping make it efficient in practice, despite slightly weaker theoretical bounds. Its compact structure also benefits from cache locality, helping it perform consistently well on large datasets without significant overhead.

The **Fibonacci Heap** began to show its theoretical advantage only at the largest tested scale (2 million nodes). Its amortized $O(1)$ `decrease_key` operation becomes significant when edge relaxations dominate computation. Because restructuring is deferred until `extract_min`, the heap avoids unnecessary adjustments during intermediate operations, making it more effective as problem size and edge density grow. However, this efficiency comes at the cost of higher constant factors and more complex memory management, explaining its slower performance on smaller graphs.

Taken together, these observations suggest that the optimal heap depends on graph scale and density, as well as the practical overheads of the implementation environment.

6 Conclusion

This study compared three heap-based priority queue implementations, Binary, Pairing, and Fibonacci, within Dijkstra's shortest path algorithm. Each structure was implemented from scratch and evaluated on randomly generated graphs of increasing size to analyze both theoretical efficiency and real-world performance.

The findings reveal a clear trade-off between simplicity and scalability. The **Binary Heap** remains fast and practical for small to medium-sized graphs but scales poorly as graph density increases. The **Pairing Heap** offers the most balanced performance, combining low overhead with steady runtime growth across all graph sizes. Finally, the **Fibonacci Heap** demonstrates its theoretical advantage only at the largest scale (2 million nodes), where the number of `decrease_key` operations becomes large enough for amortized gains to outweigh its overhead.

- **Binary Heap:** Best for small and moderate graphs where simplicity and speed matter most.
- **Pairing Heap:** Consistently efficient and practical across all graph sizes.
- **Fibonacci Heap:** Ideal for very large, dense graphs with many relaxations.

In conclusion, theoretical complexity provides a strong foundation for algorithm choice, but real-world performance ultimately depends on implementation overhead and data characteristics. Smaller and moderate graphs may benefit from simpler structures like the Binary or Pairing Heap, whereas the Fibonacci Heap becomes the most efficient choice for very large and dense graphs where its amortized advantages are fully realized.

GitHub Repository

The full implementation (data structures, Dijkstra's algorithm, unit tests, and benchmark plots) is available at: https://github.com/aaravg31/cosc520_A2

Acknowledgements

I would like to acknowledge ChatGPT [6] 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] GeeksforGeeks. 2023. *Pairing Heap - Data Structures and Algorithms.* https://www.geeksforgeeks.org/dsa/pairing-heap/ Accessed: 2025-10-24.

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