When Asymptotic Complexity Fails: An Empirical and Cost-Based Study of Binary and Fibonacci Heaps in Python

Seungjun Lee

April 14, 2026

Contents

1 Introduction

Dijkstra's algorithm is widely used for finding the shortest path in graphs. Its performance can be improved by implementing a priority queue. Binary heaps and Fibonacci heaps are commonly used priority queue implementations. Theoretically, Fibonacci heaps have a lower asymptotic time complexity than binary heaps. However, empirical studies often show that binary heaps achieve better practical runtime performance. Asymptotic time complexity is widely used to evaluate algorithm performance. Investigating the gap between theoretical complexity and practical performance is therefore important. Therefore, this study investigates why does the theoretical asymptotic advantage of Fibonacci heap in Dijkstra's algorithm not translate into practical runtime improvements in Python implementations.

2 Theoretical Background

2.1 Dijkstra's Algorithm

Dijkstra's algorithm solves the *single-source shortest-path* problem on a finite graph whose edges are assigned nonnegative real weights. Given a designated source vertex, the algorithm computes, for every vertex reachable from the source, the minimum total weight of any path from the source to that vertex. The pseudocode below records a standard sequential formulation in which each iteration selects the next vertex by scanning the entire unvisited set.

```
dist[v] = ∞ for all vertices v
unvisited = set of all vertices

dist[source] = 0

while unvisited is not empty:
  # Main loop executes once per vertex → total O(V)

  cur = unvisited vertex with minimum dist[cur]
  # Select vertex with smallest tentative distance
  # Linear scan over unvisited → O(V) per iteration

  if dist[cur] = ∞:
    break
  remove cur from unvisited
```

```
  for each neighbor nxt of cur:
    # Iterate over all adjacent edges
    # Total number of iterations across algorithm = O(E)
    # Each iteration is counted as a relaxation attempt

    cand = dist[cur] + weight(cur, nxt)
    if cand < dist[nxt]:
      dist[nxt] = cand
        # Relaxation success
    endif
  endfor
endwhile

return dist
```

The algorithm iterates until all vertices have been processed or there are no more connected nodes. In each iteration, the algorithm finds the unvisited node with the shortest distance and updates the distance of its neighbors. Then, for each neighboring node, compare the existing distance with the distance going through the current node. If going through the current node is shorter, change the distance. By doing this, the algorithm can find the shortest path from the source to all other vertices.

2.2 Time Complexity

The total time complexity of this algorithm can be calculated by summing the cost of each operation. First, the main loop repeats once for every vertex, resulting in V iterations. In each interation, it tooks $O(V)$ to find the minimum-distance unvisited vertex. Therefore, the total cost of minimum selection is $O(V^2)$. Additionally, during each iteration, the algorithm iterates all neighboring vertices of the current vertex. As each edge is chosen once, neighbor examinations took total $O(E)$ times. Combining these two, the overall time complexity becomes $O(V^2 + E)$.

2.3 Priority Queues

The pure dijkstra must scan all vertices to find the smallest distance, which takes $O(V)$ time for each iteration. To efficiently handle this process, a priority queue is typically adopted.

A priority queue is a data structure that supports efficient extraction of the element with smallest key. This is used in Dijkstra's algorithm to select the

vertex with smallest tentative distance. The main operations required by Dijkstra's algorithm are insert, decrease-key, and extract-min. As different queues have distinct time complexities in each operation, the choice of priority queue implementation determines the overall time complexity of the algorithm.

2.4 Binary heap

A binary heap is a heap data structure that is implemented in a complete binary tree satisfying either the min-heap or max-heap property.

In Dijkstra's algorithm, it is used as a priority queue that stores vertices with their current tentative distances and supports the operations insert, decrease-key, and extract-min. As a complete binary tree has height of $\log_2 V$, restoring the heap property after an insertion or key modification requires moving a node up or down the tree by at most $\log_2 V$ times. Therefore, the insert, decrease-key, and extract-min operations each run in $O(\log V)$ time.

2.5 Fibonacci Heap

To enhance the performance of the priority queue, a fibonacci heap was introduced.

A fibonacci heap is a collection of trees rather than a single tree. Unlike a binary tree, its trees do not need to be complete binary trees. They only need to satisfy the min-heap or max-heap property.

The Insert operation simply adds a new node to the root list and therefore run in $O(1)$ time.

The decrease-key operation cuts the modified node from its parent and move it to the root list. If the parent has previously lost a child, a cascading cut occurs to maintain structural properties.

The extract-min operation removes the minimum node, which is tracked by a pointer. To trach a new minimum node into the pointer, it performs consolidation process. Although consolidation require significant work, amortized analysis shows that extract-min runs in $O(\log n)$ time.

As a result, insert and decrease-key operations take $O(1)$ amortized time, while extract-min takes $O(\log n)$ amortized time.

```
dist[v] = ∞ for all vertices v
dist[source] = 0
```

```
queue = priority queue containing all vertices
decrease-key in queue source to 0

while queue is not empty:                                    # O(V)
  cur = extract-min from queue                               # Extract-min O(a)

  for each neighbor nxt of cur:                              # Relaxation attempt O(E_cur) <- Sum
    cand = dist[cur] + weight(cur, nxt)
    if cand < dist[nxt]:
      dist[nxt] = cand
      decrease-key in queue nxt to cand                      # Relaxation success O(b)
    endif
  endfor
endwhile

return dist

The entire process is similar to the pure dijkstra.
However, it uses priority queue to select current vertex.
```

2.6 Time Complexity

The time complexity of Dijkstra's algorithm with a priority queue can be expressed in a general form by separating the cost of key operations.

As in the pure version, the main loop iterates once for each vertex, resulting in V iterations. In each iteration, the algorithm performs an extract-min operation to select the current vertex. Let the cost of this operation be $O(A)$, which depends on the choice of priority queue. Therefore, the total cost of minimum selection is $O(AV)$.

In addition, the algorithm performs relaxation on neighboring vertices. Across the entire execution, each edge is examined once, resulting in E relaxation attempts. When a shorter path is found, a decrease-key operation is performed. Let the cost of this operation be $O(B)$. Thus, the total cost of neighbor processing becomes $O(BE)$.

Combining these components, the overall time complexity can be expressed as:

$$O(AV + BE)$$

This formulation allows the effect of different priority queue implementations to be analyzed by substituting their respective operation costs.

For a binary heap, both extract-min and decrease-key operations require $O(\log V)$ time. Substituting $A = \log V$ and $B = \log V$, the total complexity becomes:

$$O(V \log V + E \log V)$$

For a Fibonacci heap, the extract-min operation takes $O(\log V)$ amortized time, while the decrease-key operation requires only $O(1)$ amortized time. Substituting $A = \log V$ and $B = 1$, the total complexity becomes:

$$O(V \log V + E)$$

Structure	Insert	Decrease-key	Extract-min
binary heap	$O(\log n)$	$O(\log n)$	$O(\log n)$
fibonacci heap	$O(1)$ (amortized)	$O(1)$ (amortized)	$O(\log n)$ (amortized)

Table 1: Time complexity of differnent heaps

Therfore, as shown in Table 1, applying binary heaps and fibonacci heaps result in total time complexity of $O(VlogV + ElogV)$ and $O(VlogV + E)amortized.$ Eventhough is reduced to , but E becomes $E \log V$. The reason that dijkstra with queue is faster is that most of graph data in reality are sparse data. Sparse is opposite of dense. Density of graph is calculated as $E/V(V-1)$.

There are also time complexity difference between different queue types. dijkstra with binary heap has time complexity of $O(VlogV + ElogV)$ and one with fibonacci heap has time complexity of $O(VlogV+E)amortized.$ Looking without the concept of amortized, this difference comes from decrease-key operation.

However, despite this theoretical advantage, Dijkstra's algorithm implemented with a binary heap often demonstrates better runtime performance in practice. Several empirical studies report that binary heaps outperform Fibonacci heaps in real-world implementations. This discrepancy between theoretical complexity and practical performance motivates further investigation into the factors affecting runtime behaviour.

3 Methodology

3.1 Experimental Environment

All experiments were conducted using Python 3.12 on a Debian virtual machine running in a Proxmox Virtual Environment, configured with 4 CPU cores and

8 GB of RAM.

Both binary heap and Fibonacci heap implementations were written in Python and executed within the same codebase to ensure a fair comparison. Only the priority queue implementation differed between the two versions of Dijkstra's algorithm.

Standard Python libraries were used for the experiments, and runtime measurements were obtained using Python's built-in timing functions.

3.2 Thesis

decrease-key operation takes place when the algorithm finds faster way to go to neibour nodes. Therfore number executed can be explained as relaxation attempt success. This value is affected by two variables; relax-attempts and relax-success-ratio.

Relaxation attempts is determined by edge numbers. And edge number can be expressed with nodes number and density.

Relaxation success ratio may be affected by lots of variables, but distribution is significant factor. After finding the distribution of road data, by changing its factor, find the differnce.

Therefore, the goal is to create various environments by identifying the two variables of variance and density and the correlation between them, and then to analyze whether the time taken for each operation is similar and how different the relax success is.

3.3 Real Data

The real data to analyze is selected to the DIMACS USA dataset. It was selected from the 9th DIMACS Implementation Challenge: Shortest Paths (2005-2006), organized by Rutgers University's Center for Discrete Mathematics and Theoretical Computer Science, with editors from Microsoft Research and AT&T Labs Research. The USA graph contains 23,947,347 vertices (road intersections) and 58,333,344 directed edges (road segments), derived from the U.S. Census Bureau's official TIGER/Line database. TIGER was developed in collaboration with the U.S. Geological Survey and became the first nationwide digital map of roads in the United States. Due to its scale, real-world structure, and status as the field's de facto standard benchmark, the dataset has facilitated substantial follow-up work and better experimental standards across the shortest path research community. It has since been adopted in hundreds of peer-reviewed studies as the common basis for algorithm comparison.

Based on the (Aradhana Singh, 2025), most of road data follow log-normal distribution. To investigate whether the edge weight distribution of the DIMACS USA road network follows a log-normal distribution, we visualized the distribution using a histogram and a Q-Q plot against the normal distribution. We additionally applied a log transformation to the edge weights and repeated the same visualizations on the transformed data. To quantitatively assess normality, we computed skewness and excess kurtosis for both the original and log-transformed distributions, using 0 as the theoretical reference value for each metric under a normal distribution.

Then, mean and vairance are calulated.

$$\mu = \overline{\text{edge}}$$

variance was calculated using this equation.

$$\text{variance} = \frac{\sum_i (\text{edge}_i - \mu)^2}{\text{edge}}$$

graph density was calculated using this equation.

$$\text{density} = \frac{\text{edge}}{\text{node} \cdot (\text{node} - 1)}$$

3.4 Synthetic Data

To investigate the relationship between graph structure and algorithmic behavior, it is necessary to control key variables such as the number of nodes, graph density, and edge weight distribution. However, real-world road network data does not allow independent control of these variables, as they are inherently fixed and interdependent. Therefore, synthetic graph data was generated based on statistical properties extracted from the real dataset.

The synthetic graphs were designed to preserve the essential characteristics of real-world road networks while enabling systematic variation of individual parameters. In particular, the mean edge weight observed in the real dataset (approximately 2950) was rounded to 3000 for simplicity, as this difference is negligible relative to the overall scale. The standard deviation of edge weights was varied across a range from 1000 to 16000, centered around the observed real-world value (4071), in order to examine the effect of weight dispersion on relaxation behavior.

To model the edge weight distribution, a lognormal parameterization was adopted. Empirical analysis of the real dataset showed a highly right-skewed distribu-

tion of edge weights, and the lognormal distribution provides a reasonable approximation for such positively skewed data. Using this parameterization, edge weights were generated by specifying the mean and standard deviation, allowing controlled variation in variance while maintaining realistic distributional properties.

Graph topology was generated using an out-degree-based approach. Each vertex was assigned a number of outgoing edges determined by the target density, ensuring that the overall number of edges satisfied $E \approx d \cdot V(V-1)$, where d is the density. This method was chosen because it allows direct control over graph density while maintaining consistent local connectivity across vertices. Compared to purely random edge sampling, the out-degree approach provides a more stable and interpretable structure for analyzing algorithm behavior.

The number of nodes was varied exponentially as $[2000, 4000, 8000, 16000]$ to evaluate scalability while maintaining computational feasibility. Similarly, density values were selected using logarithmic spacing across multiple orders of magnitude, ranging from 10^{-7} to 3×10^{-2}. This logarithmic sampling was used to efficiently capture behavioral changes across both extremely sparse and moderately dense graphs, while providing sufficient intermediate resolution to identify transitional effects.

Although real-world road networks exhibit extremely low density (approximately), such sparse graphs produce limited variation in relaxation behavior and decrease-key frequency. Therefore, density was intentionally expanded beyond real-world values to explore a broader range of algorithmic conditions. This allows the analysis to identify how graph connectivity influences the frequency of key operations such as relaxation and decrease-key, which are central to the theoretical performance differences between heap implementations.

Overall, this synthetic data generation approach enables controlled experimentation across a wide range of graph conditions, making it possible to isolate and analyze the impact of structural and statistical variables on Dijkstra's algorithm performance.

3.5 Algorithm Implementation

To ensure a fair and controlled comparison between different priority queue implementations, Dijkstra's algorithm, binary heap, and Fibonacci heap were implemented from scratch in Python.

Existing library implementations were not used, as they may differ in internal optimizations, data structures, and implementation details. Such differences could introduce uncontrolled variables into the experiment, making it difficult

to isolate the effect of the underlying data structure on performance. By implementing all components within a unified environment and following standard algorithmic definitions, the comparison focuses solely on the theoretical characteristics of each data structure.

All implementations were written using only Python's built-in features without external optimization libraries. This ensures consistency across implementations and minimizes the influence of language-specific optimizations or hidden performance enhancements.

In addition to measuring overall runtime, several internal operation metrics were recorded to analyze the behavior of the algorithm in detail. These metrics include:

- **Runtime**: Total execution time of the algorithm.
- **Extract-min calls**: The number of times the minimum element is removed from the priority queue.
- **Relaxation attempts**: The number of edge relaxations attempted during execution.
- **Relaxation successes**: The number of times a shorter path is found, resulting in a distance update.
- **Reached nodes**: The number of nodes that were reached from the source node during execution.

These measurements allow for a more detailed analysis beyond overall runtime, enabling the investigation of how graph structure influences the frequency of key operations such as extract-min and decrease-key. In particular, relaxation successes correspond directly to decrease-key operations in the priority queue, providing a bridge between theoretical complexity and observed runtime behavior.

3.6 Analysis

Plot relationship between $E = V(V-1) \cdot Density$ and relax attempts. Then Find curve that fit the most. Do same thing for variance and relax success ratio.

Calculate each operation cost using multivariate linear regression. For each type of queue, set add,extract-main,relaxation-attempts, and decrease-key as dependent variables and runtime is independent variable. Using data collected, find the coefficient of each dependent variables.

4 Experimental results

4.1 Real Data Properties

This subsection reports exploratory analysis of edge weights in the DIMACS USA instance described in Section 3.3; the conclusions inform the lognormal edge-weight specification used for synthetic graph generation.

Figure 1 displays the marginal distribution on the original scale, and Figure 2 compares sample quantiles to those of a reference normal distribution with matching mean and variance. The histogram is strongly right-skewed with a pronounced upper tail. The Q-Q plot shows systematic upward curvature relative to the diagonal, indicating heavier right-tail behavior than a Gaussian model and motivating a monotone transformation of the strictly positive weights.

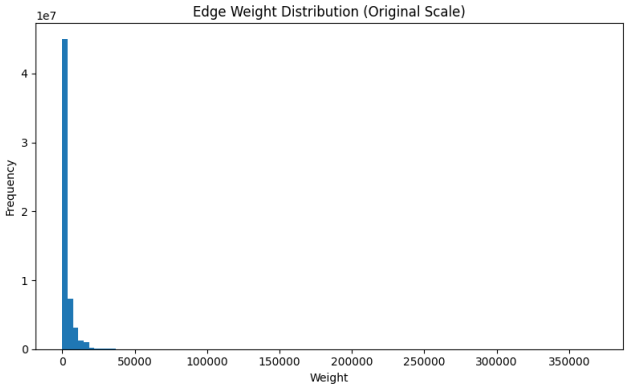

Figure 1: Histogram of DIMACS USA edge weights on the original scale.

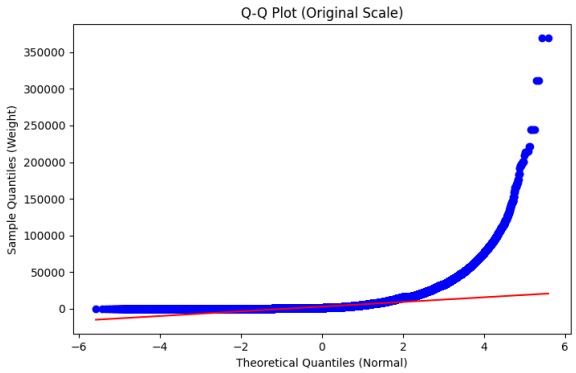

Figure 2: Normal Q-Q plot of DIMACS USA edge weights on the original scale.

Figure 3 and Figure 4 repeat the diagnostics after applying a logarithmic transformation. The histogram becomes approximately symmetric and unimodal, and the Q-Q plot follows the reference line closely except for minor deviations in the extremes, which is consistent with approximate normality of the transformed weights and hence with a lognormal model on the original scale.

The Q-Q plot shows strong linearity across the central and upper quantiles, confirming that the log-transformed weights closely follow a normal distribution. The left tail, however, deviates noticeably — sample quantiles cluster near zero rather than tracking the theoretical line. This reflects a boundary effect common in lognormal distributions with large sigma, where probability mass concentrates near zero on the original scale, causing a collapse of low-end values upon log transformation.

Importantly, this deviation is confined to a small fraction of lower-end observations. The linear trend holds across the bulk of the data, and the alignment with the reference line confirms that the distributional body satisfies the lognormal assumption. The left-tail departure is therefore an artifact of the data generation process, not evidence against lognormality.

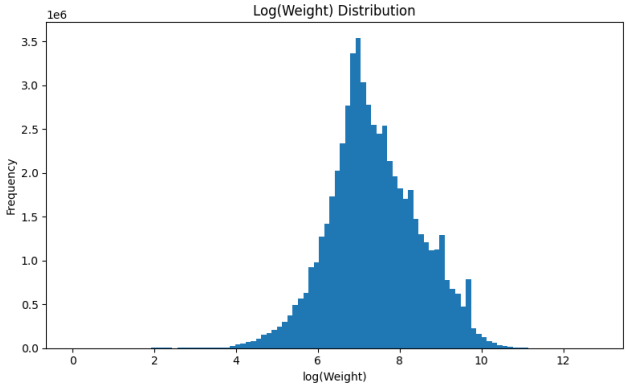

Figure 3: Histogram of DIMACS USA edge weights after a logarithmic transformation.

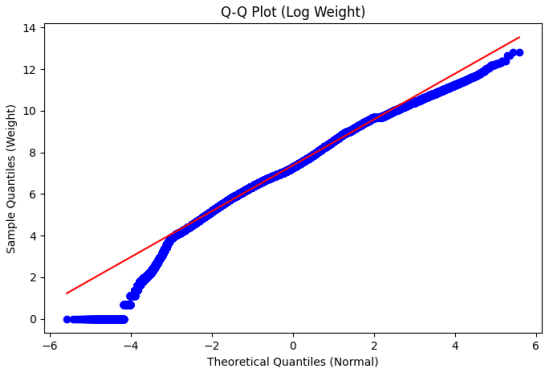

Figure 4: Normal Q-Q plot of DIMACS USA edge weights after a logarithmic transformation.

Statistic	Original	Log-transformed
Skewness	4.1154	0.0281
Excess kurtosis	39.0225	0.2218

Table 2: Sample skewness and excess kurtosis of DIMACS USA edge weights before and after a logarithmic transformation. For a normal distribution, both quantities equal zero.

Table 2 quantifies the change in shape: skewness and excess kurtosis both move

sharply toward the normal benchmarks of zero on the transformed scale, corroborating the graphical evidence.

The empirical sample mean and sample standard deviation of the edge weights are approximately 2950 and 4071, respectively. For ease of exposition and to set round nominal parameters in the synthetic experiments that follow, these estimates are represented by 3000 and 4000.

4.2 Experimental Implementation

```
nodes,density,std,sigma,trial,seed,time,algorithm,reached,extract_min_calls,relax_attempts,r
2000,1e-07,1000,0.32459284597450133,1,60566251,0.0006364700384438038,binary,False,2,0,0
2000,1e-07,1000,0.32459284597450133,1,60566251,0.0012478521093726158,fibonacci,False,2,0,0
2000,1e-07,1000,0.32459284597450133,2,60566248,0.0004217512905597687,binary,False,2,0,0
2000,1e-07,1000,0.32459284597450133,2,60566248,0.0010348870418965816,fibonacci,False,2,0,0
2000,1e-07,1000,0.32459284597450133,3,60566249,0.0004464490339159965 5,binary,False,2,0,0
2000,1e-07,1000,0.32459284597450133,3,60566249,0.0013218028470873833,fibonacci,False,2,0,0
2000,1e-07,1000,0.32459284597450133,4,60566254,0.0004335441626608372,binary,False,2,0,0
2000,1e-07,1000,0.32459284597450133,4,60566254,0.001101895235478878,fibonacci,False,2,0,0
2000,1e-07,1000,0.32459284597450133,5,60566255,0.0004320708103477955,binary,False,2,0,0
2000,1e-07,1000,0.32459284597450133,5,60566255,0.0010821172036230564,fibonacci,False,2,0,0
```

4.3 Analysis

```
Todo

1. 상관관계
  - E <-> Relax attempts
- Density, Sigma <-> Relax success ratio
- V, Density, Sigma <-> Relax success(Decrease key)
- Decrease key <-> runtime ratio

2. 분포
- decrease key up -> runtime ratio > 1 up
-> regression result

3. calculation
- 현실 데이터 기반으로 V, Density, Sigma -> Decrease key -> runtime ratio 도
출.
```

Edges and relaxation attempts show almost perfect linear relationship.

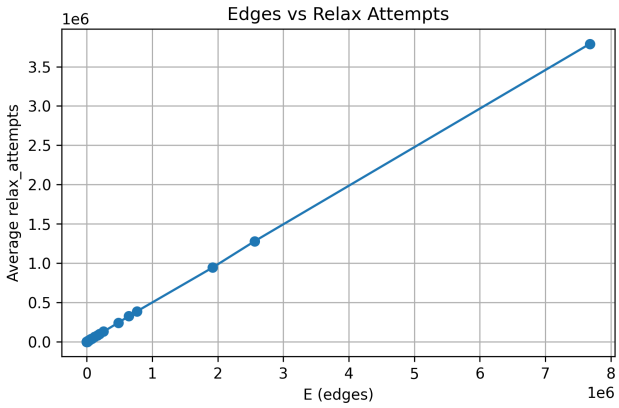

Figure 5: Relationship between edges and relaxation attempts

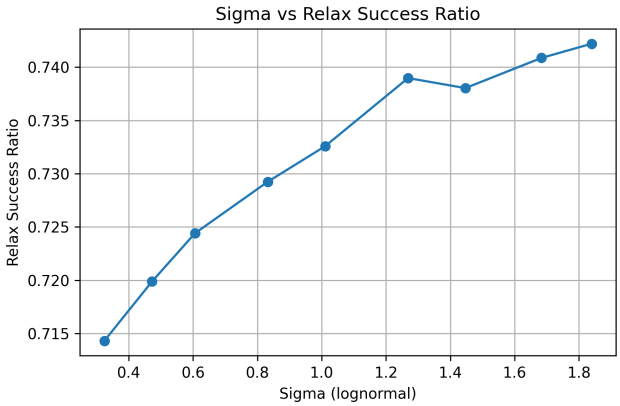

Figure 6: Relationship between variance and relaxation success ratio

variance and relaxation success ratio shows positve relationship. Specifically, it looks like log or Exponential Saturation function.

Therefore, it is clear that decrease-key operations occurs more often when density or variance is higher.

However, unlike this results, in every case binary heap was faster than finonacci heap. Therfore, regression for each operation is considered together.

Table 3 reports the fitted regression, and all operation coefficients except decrease-key are of comparable magnitude across heaps, as expected. However, shockingly, there was high difference in coef decrease key in opposite direction. Fi-

bonacci heap tooks much more time to operate decrease-key. There was almost 10 times difference to finish single decrease-key operation.

Heap	Intercept	Add	Extr.$\cdot \log N$	Relax	Decrease	R^2	6
Binary	0.000086	-2.449305×10^{-7}	2.984790×10^{-7}	9.549140×10^{-8}	1.762933×10^{-7}	0.978281	6
Fibonacci	-0.001343	6.499882×10^{-7}	3.999607×10^{-7}	9.479877×10^{-8}	1.579612×10^{-6}	0.977307	6

Table 3: Multivariate linear regression of wall-clock runtime (seconds) on four predictors: add-call count; extract-min calls multiplied by $\log N$ (natural logarithm of the number of nodes, as in the implementation); relaxation-attempt count; and a decrease-key term that multiplies decrease-key calls by $\log N$ for the binary heap but uses raw decrease-key calls for the Fibonacci heap. Coefficients are seconds per unit of the corresponding predictor; n is the number of observations.

This cause make binary heap perform better in the practice. The reason this happens is based on how it developed. In python binary heap is made with array. So it has small memory overhead and modifying is fast. However, in fibonacci heap, each node is saved as individual object. This causes huge memory overhead and tooks long to modify.

Therfore there are no situation that fibonaci heap is faster.

5 Discussion

The theoretical advantage of the Fibonacci heap in Dijkstra's algorithm arises from its lower asymptotic complexity for the decrease-key operation. While a binary heap requires $O(\log V)$ time for this operation, the Fibonacci heap reduces it to $O(1)$ amortized time. Based on this analysis, the Fibonacci heap is expected to outperform the binary heap, particularly in graphs with a large number of edges.

However, the experimental results show that the binary heap consistently achieves better runtime performance than the Fibonacci heap under the tested conditions. This discrepancy indicates that asymptotic complexity alone is insufficient to explain practical performance.

A key explanation for this phenomenon lies in the difference in constant factors associated with each data structure. Although the Fibonacci heap has a better theoretical bound, it relies on a more complex structure involving multiple trees, pointer-based node connections, and cascading operations. These features introduce significant overhead in each operation. In contrast, the binary heap is implemented using a simple array-based structure, allowing efficient memory

access and minimal overhead.

This difference can be interpreted through a more detailed runtime model. The total runtime of Dijkstra's algorithm can be expressed as the sum of operation counts multiplied by their respective costs:

$$T = c_1 \cdot (\text{extract-min operations}) + c_2 \cdot (\text{decrease-key operations})$$

where c_1 and c_2 represent the actual cost of each operation. Although the Fibonacci heap reduces the asymptotic cost of the decrease-key operation, the corresponding constant c_2 is significantly larger due to implementation overhead. As a result, the practical runtime is dominated by these constant factors rather than asymptotic differences.

From the perspective of algorithm engineering, this result highlights an important limitation of asymptotic analysis. Big-O notation describes the growth rate of an algorithm but ignores constant factors and low-level implementation details, which can have a substantial impact on performance in real-world environments. Therefore, an algorithm with better theoretical complexity does not necessarily guarantee superior practical performance.

In addition, the programming environment further amplifies these effects. In Python, object-oriented structures and pointer-based data manipulation incur additional overhead compared to contiguous array-based structures. The Fibonacci heap, which heavily relies on such operations, becomes less efficient in this context. On the other hand, the binary heap benefits from Python's optimized list operations and memory locality.

It is important to note that these findings may not generalize across all programming languages. In lower-level languages such as C or C++, where memory management and pointer operations can be more efficiently controlled, the relative performance of Fibonacci heaps may differ. Therefore, the observed performance gap is influenced not only by the algorithm itself but also by the implementation environment.

Overall, the results suggest that the practical inefficiency of the Fibonacci heap arises not from its asymptotic complexity, but from large constant factors and implementation overhead. This explains why binary heaps often outperform Fibonacci heaps in real-world applications of Dijkstra's algorithm, despite their inferior theoretical complexity.

6 Conclusion

I tried to figure about the gap between empirical practice and the time complexity theory.

Extract properties from real data and synthesize various situation. Run dijkstra with each heap type.

In result, binary heaps outperform everytime. This is due to the constant(coefficient).

Thererfore, it shows limitation of asymptotic time complexity theory.

However, there are limitation of the study too. - Environment - Data