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

Seungjun Lee

March 11, 2026

Abstract

뒷부분 작성을 덜하여 최신 수정 버전은 아래 링크에 업데이트하고 있습니다.

https://gitea.seung6lee.com/seung6lee/IB-CS-EE/src/branch/main/latex/build/main.pdf

Contents

1 Introduction

- Dijkstra 알고리즘 소개
- Priority queue 소개
- Binary heap 소개 (간단한 설명과 시간 복잡도)
- Fibonacci heap 소개 (간단한 설명과 시간 복잡도)
- 현실과 이론의 괴리 설명
- 연구 질문 명시
- Why does the theoretical asymptotic advantage of Fibonacci heap in Dijkstra's algorithm not translate into practical runtime improvements in Python implementations?

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

- Dijkstra 알고리즘 원리
  - 특정 노드로 갈 수 있는 최단거리 계속 수정
  - 현재 기준 가장 이동거리 짧은 노드 (여기서 priority queue 연결)
- Priority queue
  - 이게 뭔지 설명. add, extract_min, decrease_key 3개 설명
  - Binary tree
  - Fibonacci tree
- 시간 복잡도
  - 어떤 부분에서 차이가 나는지

2.1 Dijkstra's Algorithm

Dijkstra's algorithm computes the shortest path from a source node to all other nodes in a graph with non-negative edge weights. At each step, it selects the unvisited vertex with the smallest distance and relaxes the edges adjacent to that vertex. This process is repeated until all vertices have been processed.

To efficiently select the next vertex with the smallest tentative distance, Dijkstra's algorithm typically uses a priority queue. Without a priority queue, the algorithm must scan all vertices to find the smallest distance, which takes $O(V)$ time for each iteration. Since this operation is performed once for each vertex, the total cost of selecting the next vertex becomes $O(V^2)$. Additionally, each edge is relaxed once during the algorithm, which requires $O(E)$ time. Therefore, the total time complexity becomes $O(V^2 + E)$. Using a priority queue can significantly reduce this cost.

2.2 Priority Queues

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.

2.3 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 $\log_2 n$, restoring the heap property after an insertion or key modification requires moving a node up or down the tree by at most $\log_2 n$ times. Therefore, the insert, decrease-key, and extract-min operations each run in $O(\log n)$ time.

2.4 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.

2.5 Time Complexity

Originally Dijkstra's algorithm needs $O(V)$ time to select next vertex in each iteration. By using a priority queue, this process can be significantly more efficient.

Using a binary heap reduces this cost because the extract-min operation takes $O(\log V)$ time. Yet, cost of each decrease-key operation increases from $O(1)$ to $O(\log V)$. The extract-min operation occurs at most V times, while the decrease-

key operation occurs up to E times. Therefore, the overall time complexity becomes $O((V + E) \log V)$. Since number of edges are greater than number of vertices in most graphs,

Using a fibonacci heap can therotically further reduce the cost. The decrease-key operation requires $O(\log V)$ time in a binary heap. A fibonacci heap reduces this cost to $O(1)$ amortized time. Therefore entire time complexity becomes $O(V \log V + E)$ amortized.

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

The difference in time complexity between the two heaps mainly arises from the cost of the decrease-key operation. As shown in Table 1, the Fibonacci heap has a significantly lower cost for the decrease-key operation. This implies that the Fibonacci heap becomes much more efficient as the number of decrease-key operations increases.

However, despite the theoretical advantage of Fibonacci heaps, Dijkstra's algorithm implemented with a binary heap often demonstrates better runtime performance in practical implementations. Several empirical studies, including (Idowu et al., 2025), report that binary heaps tend to outperform Fibonacci heaps in real-world applications. This discrepancy between theoretical complexity and practical performance motivates a closer investigation of the factors affecting the runtime behaviour of these priority queue implementations. The following sections investigate this issue through empirical experiments and runtime analysis.

3 Methodology

- Experimental Environment
- 데이터
  - Dimacs에서 추출
  - 데이터 개수가 적음 → 실제 데이터의 형태만 파악하고 이를 바탕으로 가상 데이터 생성
- 그래프 생성
  - outdegree 방식
  - 방향 그래프
  - 평균, 분포, 밀도
- 알고리즘 적용
  - dijkstra w/ binary heap
  - dijkstra w/ fibonacci heap
- 측정 변수
  - runtime
  - extract_min_calls
  - relax_success (decrease_key_call)
  - relax_attempts
- 분석
  - correlation
  - regression

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 Graph Data

The key difference between binary heaps and Fibonacci heaps lies in the cost of the decrease-key operation. Therefore, evaluating the performance of the two priority queues requires graphs where the number of decrease-key operations varies significantly.

However, real-world graph datasets rarely allow precise control over the number of decrease-key operations. As a result, synthetic graph data was generated for the experiments.

Two major factors influence the frequency of decrease-key operations during the execution of Dijkstra's algorithm. First, graph density affects the number of relaxation attempts because each edge may trigger a relaxation operation. Graphs with higher density therefore produce more relax-attempts. Second, the variance of edge weights influences the probability that a relaxation succeeds. Higher variance in edge weights increases the likelihood that newly discovered paths produce shorter distances, resulting in more decrease-key operations.

To construct realistic synthetic graphs, structural properties of real distance graphs were first analyzed, including the distribution, mean, and variance of edge weights. Based on these observations, graphs were generated by controlling both edge density and weight variance.

The generation process first creates a set of edge weights following a specified distribution with given mean and variance. These weights are then assigned to randomly selected edges between vertices while maintaining the desired graph density.

3.3 Algorithm Implementation

Using the synthesized graph data, Dijkstra's algorithm was executed with different priority queue implementations.

Existing Python implementations of Dijkstra's algorithm and heap libraries were not used because their internal optimizations and implementation details could affect runtime performance and make a fair comparison difficult. Therefore, Dijkstra's algorithm, as well as both binary heap and Fibonacci heap data structures, were implemented directly in Python.

To ensure a fair comparison, both implementations shared the same Dijkstra framework and differed only in the priority queue structure. All algorithmic principles followed the original descriptions in the foundational papers. Only standard Python libraries were used in the implementation.

All source codes are provided in the Appendix.

For each set of graph parameters, synthetic graphs were generated with varying densities and weight distributions. On each generated graph, Dijkstra's algorithm was executed using both heap implementations under identical conditions. Each experiment was repeated multiple times to reduce measurement noise.

During execution, several metrics were recorded, including the number of extract-min operations, the number of decrease-key operations, and the total runtime.

4 Experimental results

- real data analysis result
- graph synthesize variable settings
- graph structure experiment (Ex. sigma Vs. relax_success_ratio)
- runtime comparison

4.1 Real Data Properties

Dimacs

5 Analysis

- operation cost multiple linear regression
- crossover condition
- Is it possible in real-world?

6 Discussion

- asymptotic complexity 한계
- constant factor 중요성
- algorithm engineering 관점
- Python implementation 영향
- 다른 언어에서는 달라질 가능성

7 Conclusion

- main result summary
- answer RQ
- ending

References

Idowu, A. I., Olabiyisi, S. O., Alo, O. O., Adeleke, I. A., Jokotoye, A. A., & Omotade, A. L. (2025). Comparative performance analysis of some priority queue variants in dijkstra's algorithm. *12*(8). https://doi.org/10.51244/IJRSI.2025.120800078