Advanced Data Structure
Project II

Shortest Path Algorithm with Heaps

Group3: 冯恺睿, 洪奕迅, 朱易凡

2024- 10- 23

Abstract

In this project, we implemented Dijkstra's algorithm using Fibonacci heaps and binary heaps to compare their performance in solving the shortest path problem on graphs with non-negative edge weights. The project focused on three key heap operations: insertion, deletemin, and decreasekey. We conducted a theoretical analysis of the worst-case costs for these operations and evaluated the running times of both implementations on various graph densities, specifically with different numbers of edges. Additionally, we assessed the performance of these implementations on real-world datasets from the USA road networks, conducting at least 1000 queries for each network.

Index

1 Introduction

Fibonacci heap has a better amortized running time than many other priority queue data structures including the binary heap and binomial heap. In this project, we would introduce the data structure and then implement an optimized version of the Dijkstra Algorithm using the Fibonacci Heap and compare its performance with other heaps.

2 Fibonacci Heap

2.1 Introduction

A Fibonacci heap is a data structure for priority queue operations, consisting of a collection of heap-ordered trees. All operations in a Fibonacci heap except delete have an amortized complexity of $O(1)$ that comes from the relaxed structure of the Fibonacci heap. And about that ”relaxed” property would be introduced in next section.

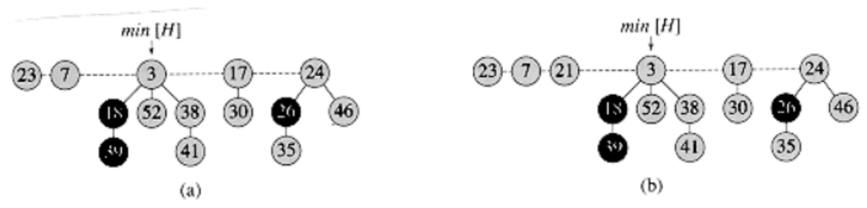
A Fibonacci heap node is complex including two pointers to its sibling, two pointer one for its parent and one for its first child and need variables to record its degree and most specially a bool variable to mark if its children have been deleted. And all root node would be connected as a doubled linked list with a pointer to the minimum node.

Notice that an important property of the Fibonacci heap is that $D(N) \approx H(N) \approx \mathcal{O}(logN)$, and the below operations are also partly designed to qualify that property.

2.2 Operations

2.2.1 Insert

Insertion of a Fibonacci heap is directly insert the new node into the root linked list.



Graph 1: Here is an example of insertion.

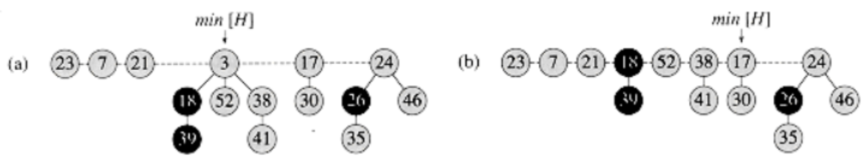
2.2.2 DeleteMin

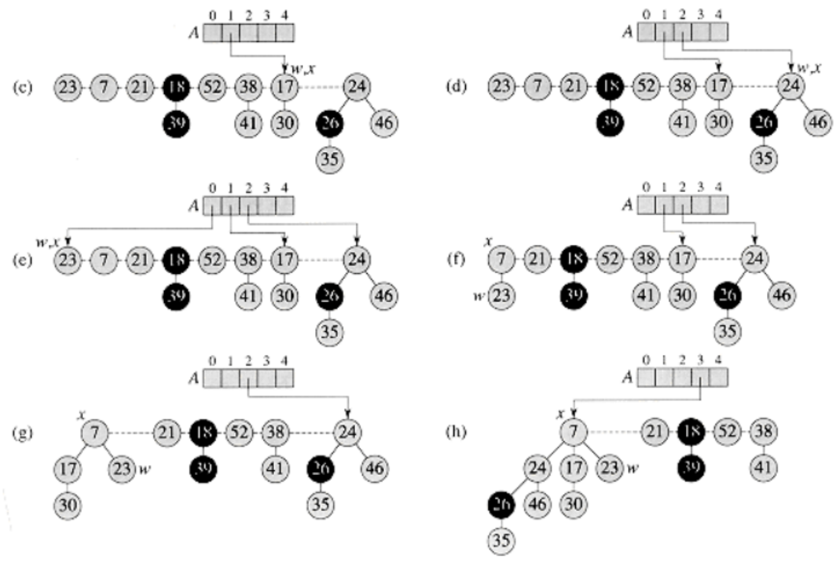
The DeleteMin mainly contains too steps:

1. Delete the minimum node and add all its subtree to the root linked list.

2. Update the minimum pointer, and at the same time, maintain the Fibonacci Heap.

The first step is easy, and the second step is about the ”relaxed” property: in operations like merge and insert, nodes in root linked list may have equal degrees (that is why we only need O(1)), but in the second step we would combine nodes with the same degres to enhance performance. Here are an example:


Graph 2: Delete the minimum node


Graph 3: Find the new minimum and optimize heap structure.

The above pictures are just a kind of implementation using an array to help merge nodes with the same degrees, and in our implementation, we use an iteration to achieve the same goal. But it expresses the core step of DeleteMin - making every node's degree unique after merge so that $H(N) \approx \mathcal{O}(logN)$ can be achieved.

2.2.3 DecreaseKey

The most difficult point of is to maintain the property of a heap after decreasing keys. A simple idea is to insert the decreased key node with its all subtrees into the root linked lists if the node becomes smaller than its parent, but that would cause its parent to become higher and more inbalanced and then lower efficiency, and after a series of operations, the heap would degenerate into a linked list, to avoid that, we can use the bool variable `marked` we introduced in earlier section to implement cascade cut, here is a pseudo-code.

```
cut(child)
if parent.marked == true:
    parent.marked = false
    cut(parent)
else:
    parent.marked = true
```

2.3 Amortized cost and worst case analysis

2.3.1 Amortized cost

The amortized cost of the above operations is:

- Insert: $\mathcal{O}(1)$
- DeleteMin: $\mathcal{O}(D(N)) \approx \mathcal{O}(logN)$
- DecreaseKey: $\mathcal{O}(1)$

2.3.2 Worst case analysis

- **Insert** is a typical ”relaxed” operation, so in all cases its complexity is $\mathcal{O}(1)$.
- In worst case, after a series of constant insertions,after **DeleteMin** we will get a linked list, and each time we merge two nodes and finally we get in worst case one node, which means $\mathcal{O}(N)$ totally.
- In worst case, the Fibonacci heap can be quite inbalanced with a single heap reaches the maximum height $\mathcal{O}(H) \approx \mathcal{O}(logN)$, and if in each node the `marked` is `true`, the cascade cut would be called in each level, so worst complexity of **DecreaseKey** is $\mathcal{O}(logN)$.

3 Dijkstra with Heaps

Our group implement totally 5 types of heaps - binary heap, priority queue, biomial heap, skew heap and fibonacci heap.

3.1 Implementation

In simple Dijkstra, there are mainly two steps - findmin and relax, and we would use corresponding operations in heaps to optimize that simple iteration.

3.2 Time complexity analysis

- For a simple Dijkstra, findmin and relax are both $\mathcal{O}(N)$, and we need to add all edges to build the graph, so it is totally $\mathcal{O}(N^2 + M)$.
- **Priority queue**: there is no decrease key operation in fibinacci heap, so elements in heap would be $\mathcal{O}(M)$ worst. Then we need $\mathcal{O}(M)$ times of insertion and we need $\mathcal{O}(N)$ times of deletemin, so it is completely $\mathcal{O}((M + N)logM)$.
- **Binary heap**: binary heap has decreasekey, so there won't be more than $\mathcal{O}(N)$ nodes in the heap, and the rest details are like priority queue, so it is $\mathcal{O}((M + N)logN)$.
- **Skew heap**: The same as binary heap.
- **Fibonacci heap**: Fibonacci heap insertion cost only $\mathcal{O}(1)$ amortizedly, other details are like binary heap, so it is $\mathcal{O}(M + NlogN)$.

- **Binomial heap**: Binomial heap have insertion of a constant complexity, so it's The same as Fibonacci heap.

3.3 Results

The testcases contain two parts - manually generated graphs with different density and traffic network data in real life, which is in fact quite sparse.

All samples were generated with a Python program and attached to the project folder. And we count time using `chrono`, a high-resolution timing library. Each heap is tested in 10 different graphs with 1,000 random inquiries per graph. The correctness of the programs is ensured with all the source code passing Luogu OJ.

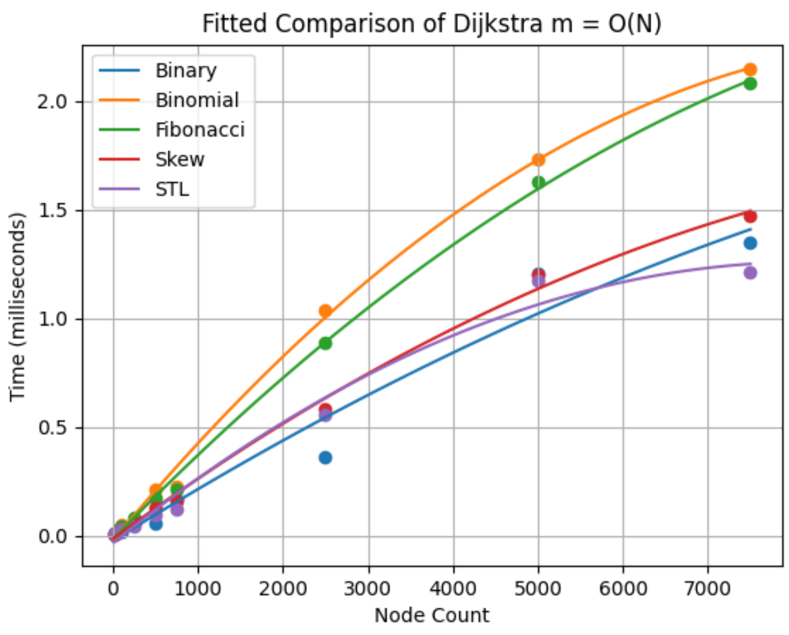
3.3.1 Generated graph

We've generated graph with $\mathcal{O}(N)$, $\mathcal{O}(N^{1.5})$, $\mathcal{O}(N^2)$ and tested Dijkstra performance using different heaps, here are the result:

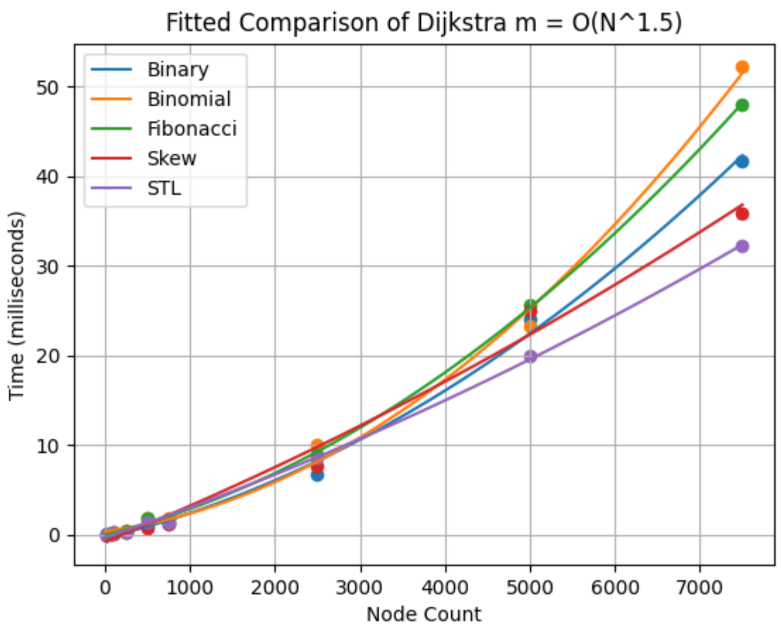
Edge Growth	STL (ms)	Binary Heap (ms)	Binomial Heap (ms)	Skew Heap (ms)	Fibonacci Heap (ms)
$O(N)$	0.085	0.2162	0.202	0.1145	0.1452
$O(N^{1.5})$	1.388	2.1830	1.655	1.8020	1.6460
$O(N^2)$	12.930	28.3100	21.830	20.3400	21.8600

Figure 1: Performance Comparison of Heaps

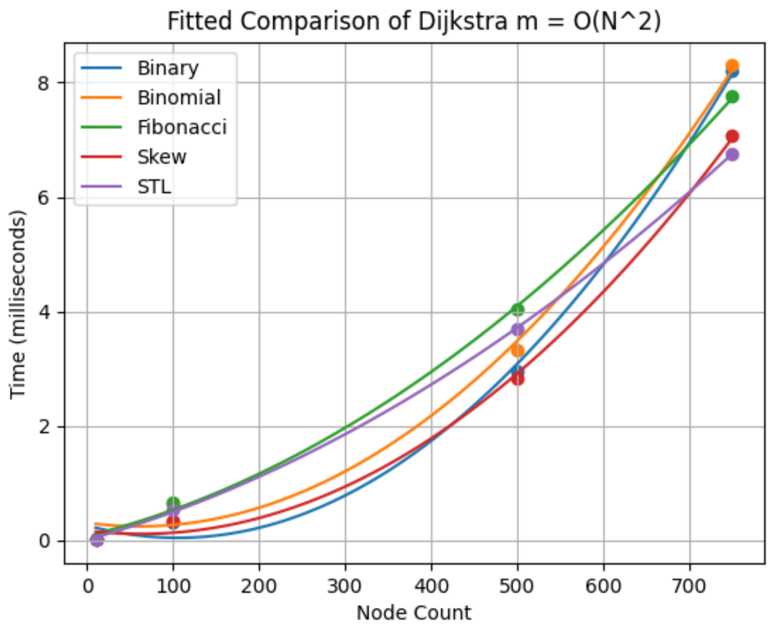
Here are detailed results of different heaps under graph with same edge complexity and differnt numbers of nodes:



Graph 4: $M = O(N)$



Graph 5: $M = O(N^{1.5})$



Graph 6: $M = O(N^2)$

The result is within our expectation - STL is the fastest since it has a very small constant and an excellent optimization in memory. Although binomial and fibonacci heaps have a theoretically shorter time, our implementations have a really large constant, which makes them slow.

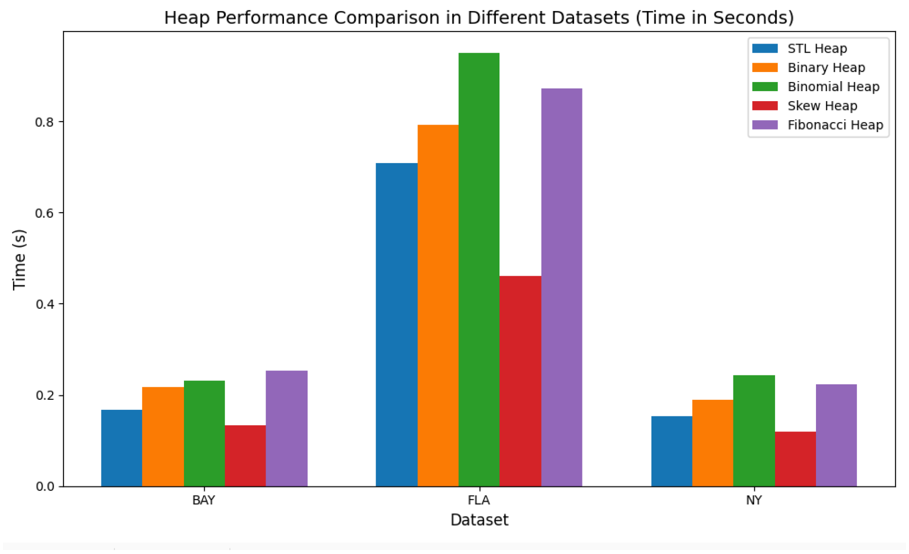
And notice that as the graph become denser, the curve becomes cliffier, that's also a shortcoming of heap-optimized Dijkstra algorithm - it is not suitable for dense graph.

3.3.2 American Traffic Nectwork

We tested data for New York, the Bay area, and Florida. Other data sets are too large to run. For example, the data for the whole of the United States take 60 seconds per inquiry. Here are the result:

Dataset	STL Heap (s)	Binary Heap (s)	Binomial Heap (s)	Skew Heap (s)	Fibonacci Heap (s)
BAY	0.1671	0.2172	0.2302	0.1335	0.2530
FLA	0.7082	0.7922	0.9498	0.4605	0.8718
NY	0.1542	0.1887	0.2437	0.1193	0.2232

Figure 2: Performance Time Table (in seconds)

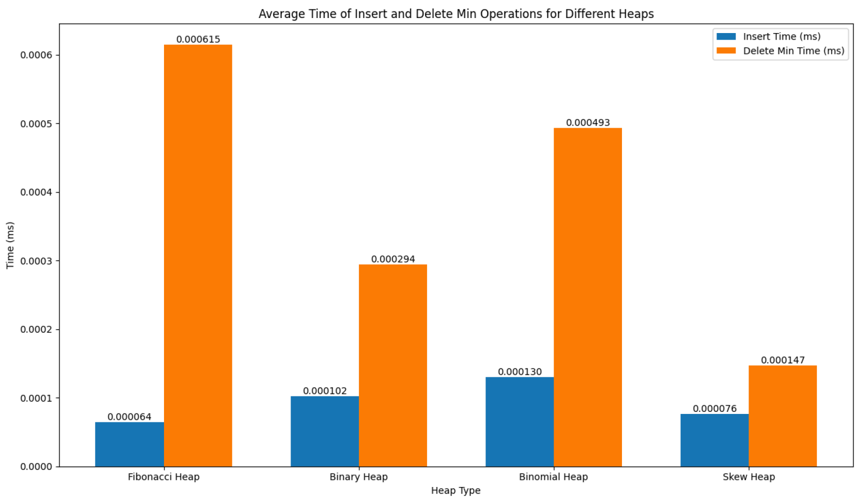


Graph 7: Visualized result

This time the result seems strange.What is surprising is that the skew heap is really fast. So we then examined main operation average time to find that Binomial and Fibonacci insertions are faster only a little or even slower, but their deletemin is really slow. Here are average insertion and delemin time in FLA dataset for Fibonacci, binomial , binary heap and skew heap. STL are a memory-optimized and operation-simplified binary heap.

Heap Type	**Insert Time (ms)**	**Delete Min Time (ms)**
Fibonacci Heap	0.000064	0.000615
Binary Heap	0.000102	0.000294
Binomial Heap	0.000130	0.000493
Skew Heap	0.000076	0.000147

Figure 3: Average Time of Insert and Delete Min Operations for Different Heaps



Graph 8: Visualized result

But why skew heap is so fast? That mainly benefit from its simple structure. Fibonacci and binomial heap depend on a series of operations to maintain its structure, which include lots of memory and pointer operation. But in dijkstra, we only need insertion(or together with decreasekey) and deletemin. Binomial and Fibonacci heap would waste lots of time in those operation especially in big data that needs frequent memory operations. In other words, in those complex cases, we cannot ignore time on memory or other operations (what we did in theory analysis). But for binary tree (or STL), it's so simple that it may be inbalanced in corner case, lowering its performance.

4 Conclusion

In theory analysis, we come to a conclusion that Fibonacci and binomial are really good and fast, but in implementation, we can notice that theoretical complexity doesn't equal to real run time, constant-optimization is really important - that's why STL and simple heap like skew heap have a better performance in large graphs.