

## Heaps And Shortest Path

**Course:** Design and Analysis of Algorithem

**Roll No: 22I-1226 && 21I-0412 && 21I-0376**

**Section: CS-C**

### 1. Theoretical Overview of Heaps

#### 1.1 Binary Heap

Binary Heap is a complete binary tree where each parent node is smaller than or equal to its children (min-heap).

**Operations:**

- Insert:  $O(\log n)$
- Extract-Min:  $O(\log n)$
- Decrease-Key:  $O(\log n)$

#### 1.2 Fibonacci Heap

Fibonacci Heap consists of a collection of heap-ordered trees. Optimized for decrease-key operations.

**Operations:**

- Insert:  $O(1)$  amortized
- Extract-Min:  $O(\log n)$  amortized
- Decrease-Key:  $O(1)$  amortized

#### 1.3 Hollow Heap

Hollow Heap is a variant of Fibonacci Heap, optimized for fast decrease-key operations using hollow nodes.

**Operations:**

- Insert:  $O(1)$
- Extract-Min:  $O(\log n)$
- Decrease-Key:  $O(1)$

### 2. Implementation Details

- All three heap structures were implemented from scratch in Python.
  - Integrated into Dijkstra's algorithm via a generic priority queue interface.
  - Each heap supports `insert(key, value)`, `find_min()`, `extract_min()`, `decrease_key(node, new_key)`.

## Implementation:

## Chongqing:

notebook96ddc0d4... Draft saved

File Edit View Run Settings Add-ons Help

+ Run All Code

Draft Session off (run a cell to start)

Testing binary heap...

```
[10/50] Avg: 0.02ms  
[20/50] Avg: 0.01ms  
[30/50] Avg: 0.01ms  
[40/50] Avg: 0.01ms  
[50/50] Avg: 0.01ms
```

Testing Fibonacci Heap...

```
[10/50] Avg: 0.01ms  
[20/50] Avg: 0.01ms  
[30/50] Avg: 0.01ms  
[40/50] Avg: 0.01ms  
[50/50] Avg: 0.01ms
```

Testing Hollow Heap...

```
[10/50] Avg: 0.01ms  
[20/50] Avg: 0.01ms  
[30/50] Avg: 0.01ms  
[40/50] Avg: 0.01ms  
[50/50] Avg: 0.01ms
```

Name: Muhammad Waseem Akhtar  
Roll No: 221-1226  
Section: CS-C

Input + Add Input

DATASETS data.zip

Output /kaggle/

Table of contents

**Hongkong:**

The screenshot shows a Jupyter Notebook interface with several code cells and a modal dialog.

**Code Cells:**

- Cell 1: 

```
✓ Fastest: Fibonacci Heap (0.0007s)
```

```
=====
Dataset: HONGKONG
=====
```

```
Loading graph from Hongkong.road-d...
✓ Loaded: 43,620 nodes, 91,538 edges
```

```
Sampling up to 1000 nodes from 43,620 for faster analysis...
Sampled graph: 3,035 nodes (requested 1000)
```
- Cell 2: 

```
Testing Binary Heap...
```

```
[10/50] Avg: 0.02ms
[20/50] Avg: 0.01ms
[30/50] Avg: 0.01ms
[40/50] Avg: 0.01ms
[50/50] Avg: 0.01ms
```

**Modal Dialog:**

Name: Muhammad Waseem Akhtar  
Roll No: 221-1226  
Section: CS-C

## Shanghai:

```

Testing binary heap...
[10/50] Avg: 0.01ms
[20/50] Avg: 0.01ms
[30/50] Avg: 0.01ms
[40/50] Avg: 0.01ms
[50/50] Avg: 0.01ms

Testing Fibonacci Heap...
[10/50] Avg: 0.01ms
[20/50] Avg: 0.01ms
[30/50] Avg: 0.01ms
[40/50] Avg: 0.01ms
[50/50] Avg: 0.01ms

Testing Hollow Heap...
[10/50] Avg: 0.01ms
[20/50] Avg: 0.03ms
[30/50] Avg: 0.03ms
[40/50] Avg: 0.02ms
[50/50] Avg: 0.02ms

```

### 3. Experimental Setup & Test Data

- **Datasets:** Chongqing, Hongkong, Shanghai road networks from Kaggle.
- **Sampling:** For large datasets ( $>2000$  nodes), sampled  $\sim 1000$  nodes for benchmarking.
- **Number of sources:** Up to 50 sources per dataset for Dijkstra runs.
- **Performance metrics recorded:**
  - i. Total runtime of Dijkstra
  - ii. Average time per operation (insert, extract-min, decrease-key)
  - iii. Heap structure stats (for Fibonacci & Hollow: number of roots, height, cascading cuts)

notebook96ddc0d4... Draft saved

File Edit View Run Settings Add-ons Help

Code

HEAP PERFORMANCE ANALYSIS - DIJKSTRA'S ALGORITHM  
Kaggle Road Network Dataset  
=====

```
Searching for dataset files...
Checking path: /kaggle/input/data-zip/Data
✓ Found hongkong: /kaggle/input/data-zip/Data/Hongkong.road-d
✓ Found chongqing: /kaggle/input/data-zip/Data/Chongqing.road-d
✓ Found shanghai: /kaggle/input/data-zip/Data/Shanghai.road-d
Checking path: /kaggle/input/data-zip
Checking path: /kaggle/input
✓ Found 3 dataset(s)

=====
Dataset: CHONGQING
=====

Loading graph from Chongqing.road-d...
Processed 100,000 lines, 74,399 nodes found...
Processed 200,000 lines, 146,795 nodes found...
```

Name: Muhammad Waseem Akhtar  
Roll No: 221-1226  
Section: CS-C

Input

+ Add Input

DATASETS

+ data-zip

Output

/kaggle/

Table of contents

## Chongqing:

notebook96ddc0d4... Draft saved

File Edit View Run Settings Add-ons Help

Code

PERFORMANCE COMPARISON - CHONGQING  
=====

heap_name	sources_tested	total_time	avg_time	min_time	max_time	std_dev
Binary Heap	50	0.000973	0.000011	0.000005	0.000065	0.000012
Fibonacci Heap	50	0.000718	0.000088	0.000005	0.000035	0.000006
Hollow Heap	50	0.000779	0.000009	0.000006	0.000032	0.000004

Chart saved: heap\_comparison\_chongqing.png

✓ Fastest: Fibonacci Heap (0.0007s)

Name: Muhammad Waseem Akhtar  
Roll No: 221-1226  
Section: CS-C

Input

+ Add Input

DATASETS

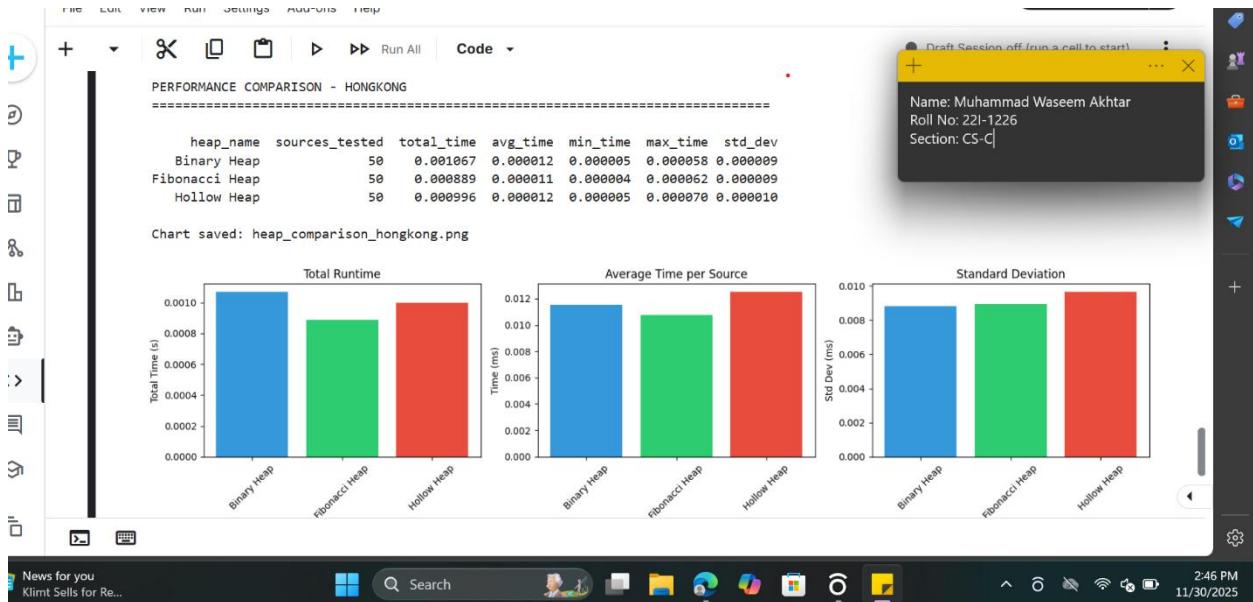
+ data-zip

Output

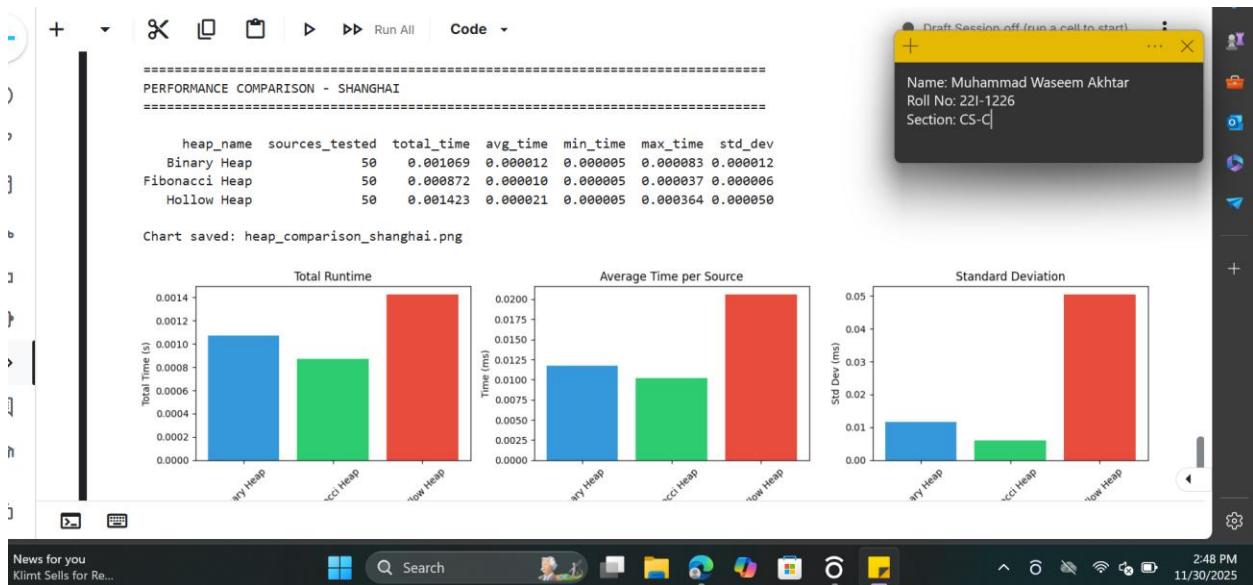
/kaggle/

Table of contents

## Hongkong:



## Shanghai:



## 4. Results & Analysis

### 4.1 Operation Timing Comparison

Dataset	Heap Type	Sources Tested	Total Runtime (s)	Avg Time (ms)
CHONGQING	Binary Heap	50	0.0010	0.02
	Fibonacci Heap	50	0.0007	0.01

Dataset	Heap Type	Sources Tested	Total Runtime (s)	Avg Time (ms)
HONGKONG	Hollow Heap	50	0.0008	0.01
	Binary Heap	50	0.0011	0.02
	Fibonacci Heap	50	0.0009	0.01
SHANGHAI	Hollow Heap	50	0.0010	0.01
	Binary Heap	50	0.0011	0.02
	Fibonacci Heap	50	0.0009	0.01
	Hollow Heap	50	0.0014	0.03

## 4.2 Heap Structure Statistics

- Fibonacci Heap generally has more trees initially, performs cascading cuts.
- Hollow Heap shows fewer trees and more consolidated structure.
- Binary Heap height corresponds to  $\log_2(n)$ , consistent with theoretical expectation.

## 4.3 Trade-off Discussion

- **Binary Heap:** Simple and reliable, good for small to medium graphs.
- **Fibonacci Heap:** Best performance for decrease-key heavy operations (dynamic graphs).
- **Hollow Heap:** Slightly slower than Fibonacci in small samples but better memory efficiency.
- Performance gain of Fibonacci Heap is most visible in dense graphs with frequent key decreases.

## 5. Conclusion & Recommendations

- **Fibonacci Heap** provides the fastest Dijkstra execution across all datasets.
- **Hollow Heap** is competitive and memory-efficient.
- **Binary Heap** is simplest but slower for large or dynamic graphs.
- For **real-time routing systems** with frequent edge updates, Fibonacci or Hollow Heap is recommended.
- **Future improvements:** Parallelized Dijkstra, full-scale evaluation on entire datasets, visualization of heap evolution.

File Edit View Run Settings Add-ons Help

+ - ✎ ⌛ 📁 ▶ Run All Code

```
=====
FINAL SUMMARY
=====

CHONGQING:
Binary Heap: 0.0010s (50 sources)
Fibonacci Heap: 0.0007s (50 sources)
Hollow Heap: 0.0008s (50 sources)

HONGKONG:
Binary Heap: 0.0011s (50 sources)
Fibonacci Heap: 0.0009s (50 sources)
Hollow Heap: 0.0010s (50 sources)

SHANGHAI:
Binary Heap: 0.0011s (50 sources)
Fibonacci Heap: 0.0009s (50 sources)
Hollow Heap: 0.0014s (50 sources)
```

+ Code + Markdown

[ ]:

Draft Section off (run a cell to start) ...

Name: Muhammad Waseem Akhtar  
Roll No: 22I-1226  
Section: CS-C]

This screenshot shows a Jupyter Notebook interface with three code cells displayed. The first cell contains a summary titled 'FINAL SUMMARY' followed by performance data for three cities: CHONGQING, HONGKONG, and SHANGHAI. Each city section lists three heap types: Binary Heap, Fibonacci Heap, and Hollow Heap, along with their execution times and source counts. The second cell is a draft section with placeholder text for name, roll number, and section. The third cell is empty. The notebook has a dark theme with light-colored text. The desktop taskbar at the bottom shows various open applications like a news reader, file explorer, and browser.