Athabasca University

COMP 372 Final Project

Option 1: Shortest Paths and Minimum Spanning Trees

Junxuan Bao

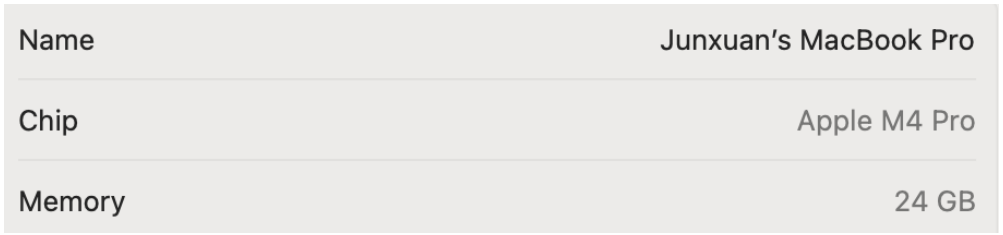
Student #3742022

This project implements two fundamental graph algorithms: Dijkstra's shortest path algorithm and Prim's minimum spanning tree (MST) algorithm. Both algorithms were implemented using Python 3.9 with two different fringe (priority queue) data structures: a binary heap and a sorted linked list.

Nov 15, 2025

Software and Hardware Environment

This project runs on a M4 Pro MacBook (24 GB) with Python 3.9.6




```
junxuanb@Junxuans-MacBook-Pro COMP372-Final-Project % python3 --version
Python 3.9.6
```

This project also requires the following Python Libraries:

matplotlib>=3.5.0

networkx>=2.6.0

Pillow>=9.0.0

pytest>=8.0.0

numpy>=1.21.0

a requirements.txt has been provided in the project folder.

Algorithm Design (Pseudocode)

Dijkstra's Shortest Path Algorithm

Target: Find shortest paths from a source vertex to all other vertices in a weighted graph with non-negative edge weights.

Approach: Dijkstra's algorithm uses a greedy strategy, repeatedly selecting the unvisited vertex with the smallest tentative distance and relaxing all its outgoing edges.

```
DIJKSTRA(G, source, fringe_type)
    distance = empty dict
    previous = empty dict
    for each vertex v in G do
        distance[v] = infinity
        previous[v] = NULL

    distance[source] = 0
    visited = empty set

    // Create fringe based on type
    if fringe_type = 'heap' then
        fringe = new BinaryHeap()
    else
        fringe = new SortedLinkedList()

    fringe.insert(source, 0)

    // main loop
    while fringe is not empty do
        (current, dist) = fringe.extract_min()

        if current in visited then
            continue

        visited.add(current)

        // Check all neighbors
        for each neighbor of current do
            if neighbor not in visited then
                new_dist = distance[current] + weight(current, neighbor)

                if new_dist < distance[neighbor] then
                    distance[neighbor] = new_dist
                    previous[neighbor] = current
                    fringe.insert(neighbor, new_dist)

    return (distance, previous)
```

Prim's Minimum Spanning Tree Algorithm

Target: Construct a minimum spanning tree for a connected, undirected, weighted graph.

Approach: Prim's algorithm grows the MST one vertex at a time, always adding the lightest edge that connects a new vertex to the growing tree.

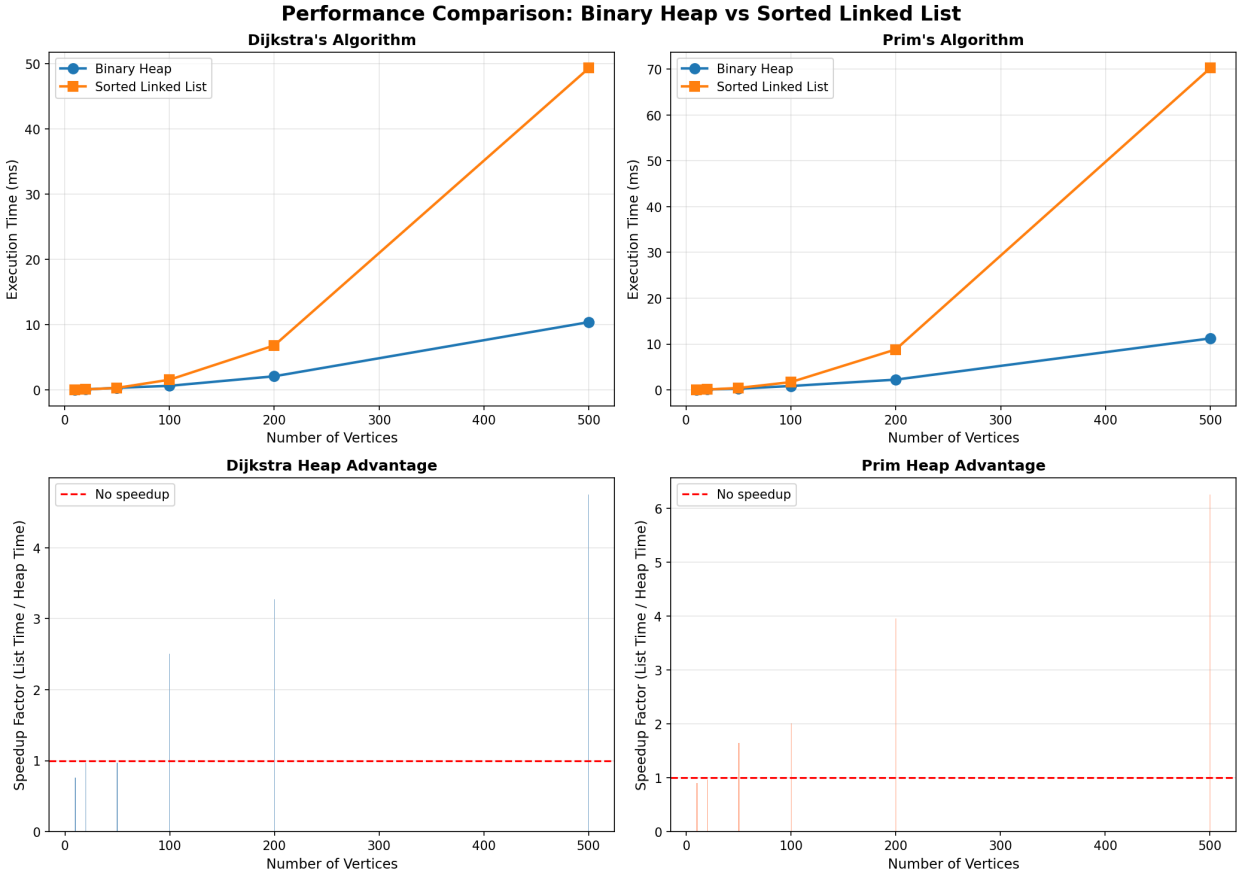
```
PRIM(G, start, fringe_type)
    // Initialize keys and parents
    key = empty dict
    parent = empty dict
    for each vertex v in G do
        key[v] = infinity
        parent[v] = NULL

    key[start] = 0
    visited = empty set
    mst_edges = empty list

    // Create fringe
    if fringe_type = 'heap' then
        fringe = new BinaryHeap()
    else
        fringe = new SortedLinkedList()

    fringe.insert(start, 0)

    while fringe is not empty do
        (current, k) = fringe.extract_min()

        if current in visited then continue
        visited.add(current)

        // Add edge to MST
        if parent[current] not NULL then
            mst_edges.add((parent[current], current, key[current]))

        // Update neighbor keys
        for each neighbor of current do
            w = weight(current, neighbor)

            if neighbor not in visited and w < key[neighbor] then
                key[neighbor] = w
                parent[neighbor] = current
                fringe.insert(neighbor, w)

    return mst_edges
```

Performance Test Original Result

In this project, we run a performance test aimed to compare the performance between 2 algorithms (Dijkstra and Prim) with 2 different data structure (Binary Heap and Linked List).

During the Test, we will create graph size with 10, 20, 50, 100, 200, 500 in 2 structures and perform both algorithms.

Here's the original test result:



```
junxuanb@Junxuans-MacBook-Pro COMP372-Final-Project % PYTHONPATH=. python3 tests/performance_test.py
Graph Algorithm Performance Benchmark
Comparing Binary Heap vs Sorted Linked List

Running performance benchmarks...
======================================================================

Testing graph size: 10 vertices
  Edges: 18
  Running dijkstra with heap... 0.03 ms
  Running dijkstra with list... 0.02 ms
  Running prim with heap... 0.03 ms
  Running prim with list... 0.03 ms

Testing graph size: 20 vertices
  Edges: 62
  Running dijkstra with heap... 0.07 ms
  Running dijkstra with list... 0.07 ms
  Running prim with heap... 0.08 ms
  Running prim with list... 0.08 ms

Testing graph size: 50 vertices
  Edges: 263
  Running dijkstra with heap... 0.31 ms
  Running dijkstra with list... 0.30 ms
  Running prim with heap... 0.24 ms
  Running prim with list... 0.40 ms

Testing graph size: 100 vertices
  Edges: 1080
  Running dijkstra with heap... 0.62 ms
  Running dijkstra with list... 1.55 ms
  Running prim with heap... 0.84 ms
  Running prim with list... 1.69 ms

Testing graph size: 200 vertices
  Edges: 4143
  Running dijkstra with heap... 2.08 ms
  Running dijkstra with list... 6.82 ms
  Running prim with heap... 2.23 ms
  Running prim with list... 8.82 ms

Testing graph size: 500 vertices
  Edges: 25327
  Running dijkstra with heap... 10.39 ms
  Running dijkstra with list... 49.36 ms
  Running prim with heap... 11.23 ms
  Running prim with list... 70.26 ms

======================================================================
Benchmarks complete!

Results saved to: results/performance_data.csv
Comparison charts saved to: results/comparison_charts.png

======================================================================
PERFORMANCE SUMMARY
======================================================================

DIJKSTRA'S ALGORITHM:
  10 vertices: Heap=0.03ms, List=0.02ms, Speedup=0.76x
  20 vertices: Heap=0.07ms, List=0.07ms, Speedup=0.97x
  50 vertices: Heap=0.31ms, List=0.30ms, Speedup=0.97x
  100 vertices: Heap=0.62ms, List=1.55ms, Speedup=2.51x
  200 vertices: Heap=2.08ms, List=6.82ms, Speedup=3.27x
  500 vertices: Heap=10.39ms, List=49.36ms, Speedup=4.75x

PRIM'S ALGORITHM:
  10 vertices: Heap=0.03ms, List=0.03ms, Speedup=0.90x
  20 vertices: Heap=0.08ms, List=0.08ms, Speedup=0.97x
  50 vertices: Heap=0.24ms, List=0.40ms, Speedup=1.65x
  100 vertices: Heap=0.84ms, List=1.69ms, Speedup=2.01x
  200 vertices: Heap=2.23ms, List=8.82ms, Speedup=3.96x
  500 vertices: Heap=11.23ms, List=70.26ms, Speedup=6.25x

✓ Performance analysis complete!
```

Performance Test Conclusion

Based on the performance test results comparing Dijkstra's and Prim's algorithms with Binary Heap versus Sorted Linked List implementations, we can draw several key conclusions:

1. Data Structure Performance

Binary Heap significantly outperforms Sorted Linked List for larger graphs. For small graphs (10-50 vertices), the performance difference is negligible, with both structures showing comparable execution times. However, as graph size increases beyond 100 vertices, the heap's advantage becomes clear. At 500 vertices, the heap-based implementation is approximately **4.75x faster for Dijkstra's** and **6.25x faster for Prim's** algorithm.

2. Algorithm Comparison

Both algorithms demonstrate similar performance patterns when using the same data structure. Dijkstra's algorithm tends to be slightly faster than Prim's for the same graph size and data structure, though the difference is minimal. At 500 vertices with heap implementation, Dijkstra completes in 10.39ms while Prim takes 11.23ms.

Use Binary Heap for production implementations, especially when dealing with graphs larger than 50 vertices. The sorted linked list may be acceptable for very small graphs (< 50 vertices) where simplicity matters more than performance, but the heap implementation is the clear winner for any real-world application requiring scalability.

Complexity Analysis

Data Structure Complexities

Binary Heap

- **Insert:** O(log n) - add element and bubble up
- **Extract Min:** O(log n) - remove root and bubble down
- **Decrease Key:** O(log n) - position map enables O(1) lookup + O(log n) bubble up
- **Space:** O(n)

Sorted Linked List

- **Insert:** O(n) - linear search for correct position
- **Extract Min:** O(1) - remove first node
- **Decrease Key:** O(n) - find and re-insert element
- **Space:** O(n)

Algorithm Complexities

Dijkstra's Algorithm

With Binary Heap - O((V + E) log V): The algorithm extracts minimum V times at O(log V) each, and processes E edges with decrease-key operations at O(log V) each, resulting in O(V log V + E log V) = O((V + E) log V) total complexity.

With Sorted Linked List - O(V²): While extract-min is O(1), processing E edges requires O(V) insertion operations each, resulting in O(EV) complexity. For typical graphs where E ≈ V, this simplifies to O(V²).

Prim's Algorithm

With Binary Heap - O((V + E) log V): Identical structure to Dijkstra's—V extract-min operations at O(log V) plus E decrease-key operations at O(log V) yields O((V + E) log V) complexity.

With Sorted Linked List - O(V²): Same analysis as Dijkstra's, with O(V) per edge operation resulting in O(V²) complexity for typical graphs.

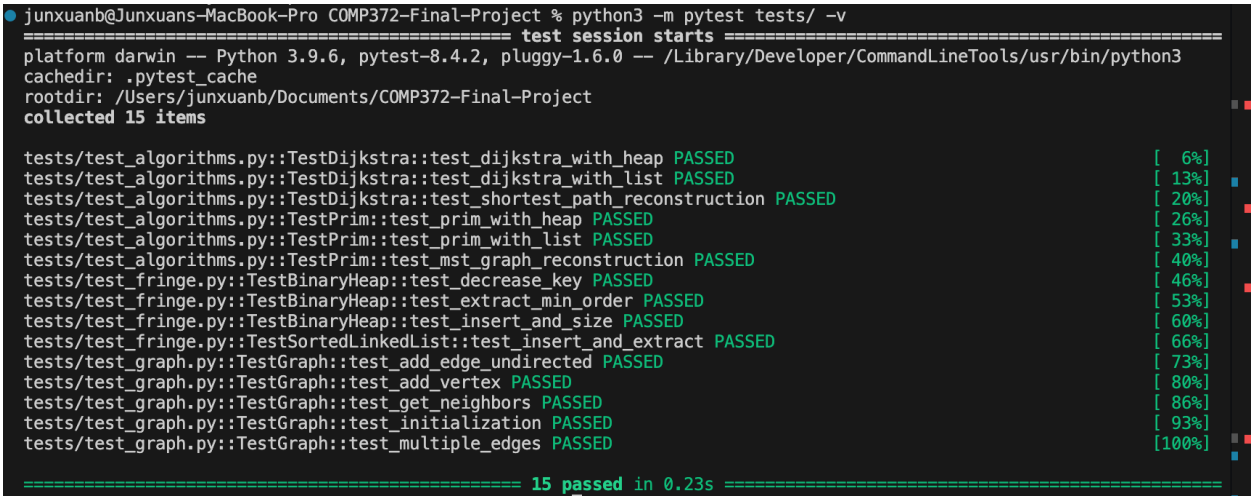
User Manual (Usage)

There are 4 main commands in this project:

- **Install Dependent Liberis:** python3 -m pip install -r requirements.txt
- **Main Entrance to UI operations (main function):** PYTHONPATH=. python3 src/ui.py
- **Run Performance Test:** PYTHONPATH=. python3 tests/performance_test.py
- **Run all test case:** python3 -m pytest tests/ -v

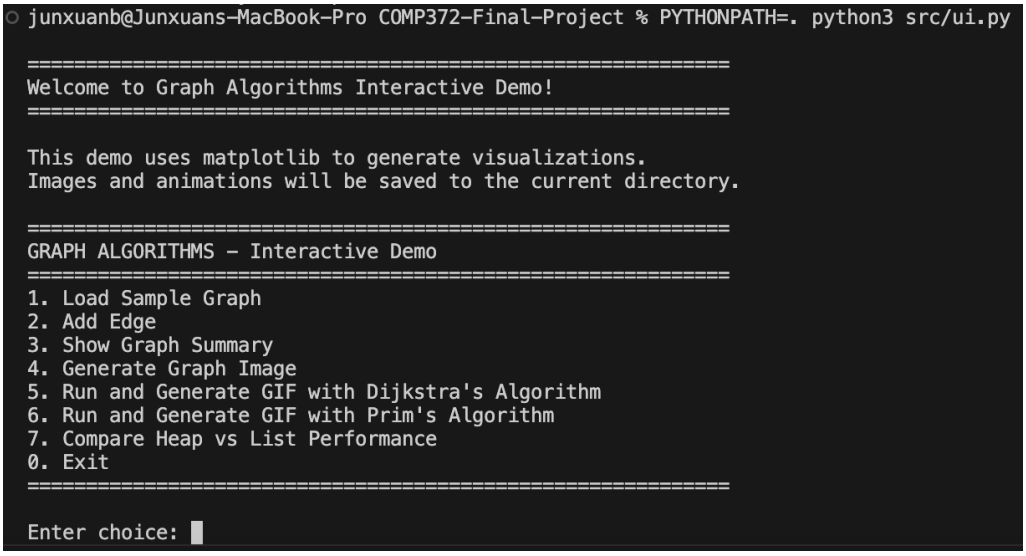
Code Coverage

Most of the code has been tested, you can run the *python3 -m pytest tests/ -v* command to run test cases.



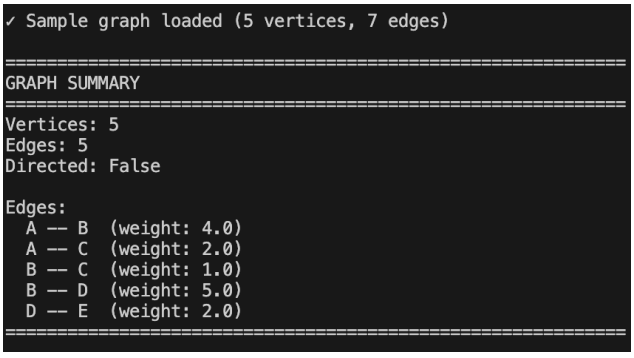
Main Function

The main function provides multiple operations:



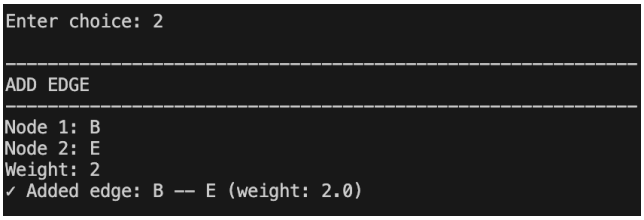
Load Sample Graph

It will load a default graph for demo purpose.



Add Edge

Let’s add an Edge between B and E with weight of 2.0 on the default graph.



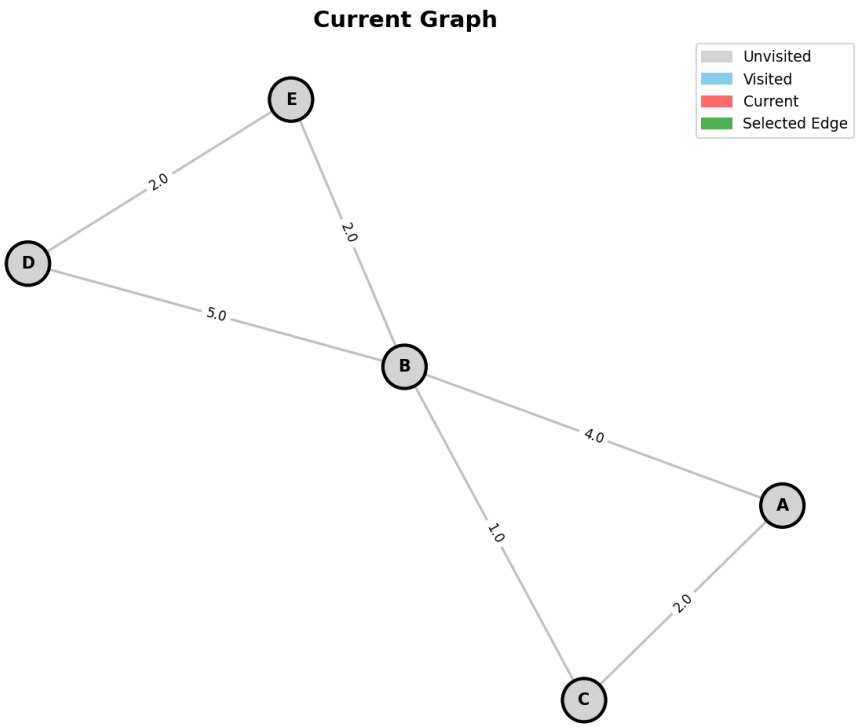
Show Graph Summary

Now, we can use this function to re-print the graph summary.

```
Enter choice: 3

============================================================
GRAPH SUMMARY
============================================================
Vertices: 5
Edges: 6
Directed: False

Edges:
  A -- B  (weight: 4.0)
  A -- C  (weight: 2.0)
  B -- C  (weight: 1.0)
  B -- D  (weight: 5.0)
  B -- E  (weight: 2.0)
  D -- E  (weight: 2.0)
============================================================
```

Generate Graph Image

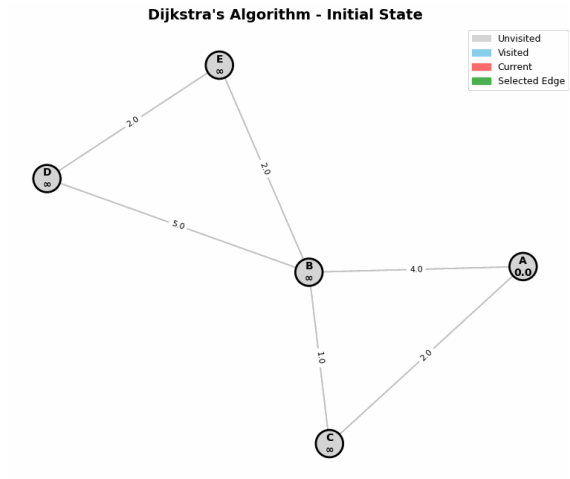
It will generate, save, and open a more visual graph:

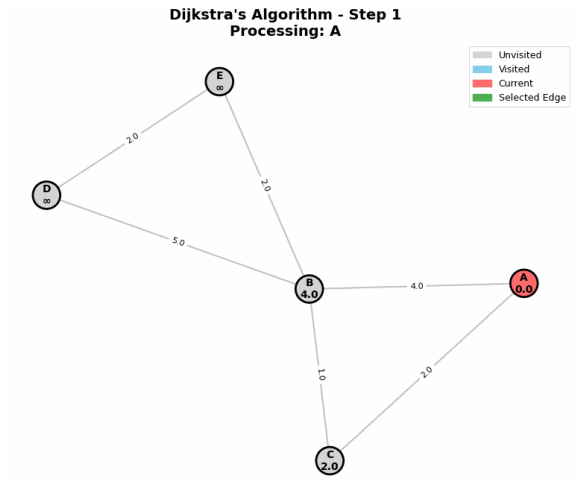


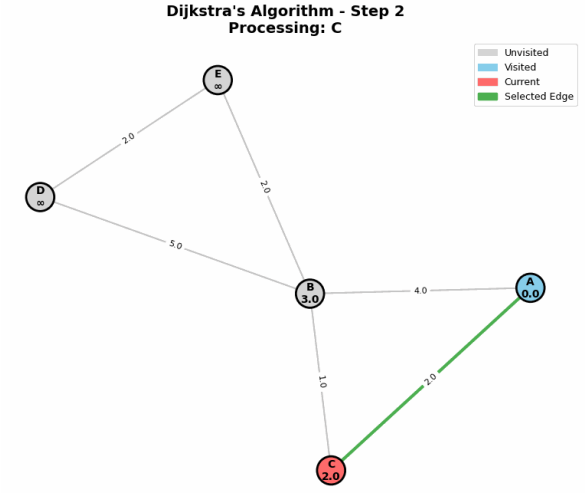
Run and Generate GIF with Dijkstra's / Prim's Algorithm

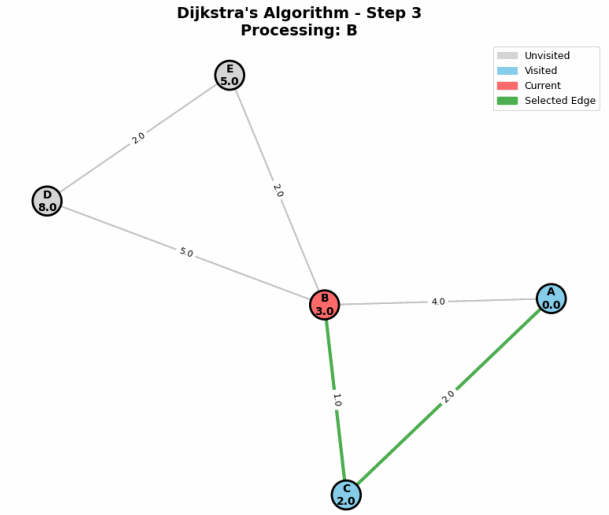
It will run the algorithm and generate an animation in GIF format.

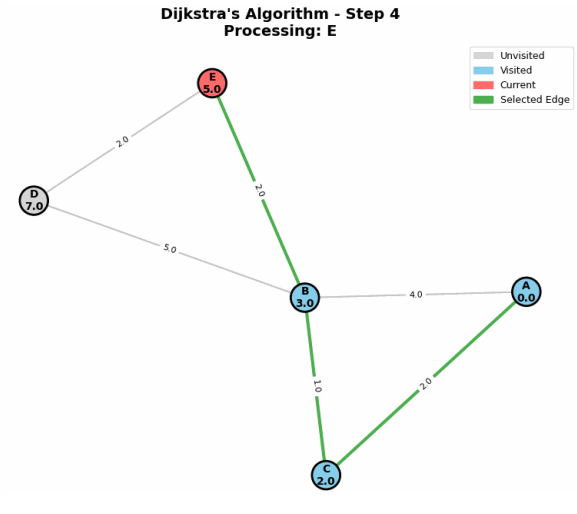
```
Enter choice: 5

------------------------------------------------------------
DIJKSTRA'S SHORTEST PATH ALGORITHM
------------------------------------------------------------
Available vertices: ['A', 'B', 'C', 'D', 'E']
Enter start vertex: A

Fringe types: heap, list
Enter fringe type (default: heap): heap

Running Dijkstra from 'A' with heap...

============================================================
RESULTS
============================================================
Execution Time: 0.53 ms
Algorithm Steps: 5

Shortest Distances from A:
------------------------------------------------------------
  A     : 0.0
  B     : 3.0
  C     : 2.0
  D     : 7.0
  E     : 5.0
============================================================

Generate animation? (y/n): y
Generating animation: animations/dijkstra_A_heap_demo.gif
Animation saved to: animations/dijkstra_A_heap_demo.gif
✓ Animation saved!
✓ Animation opened in viewer
```

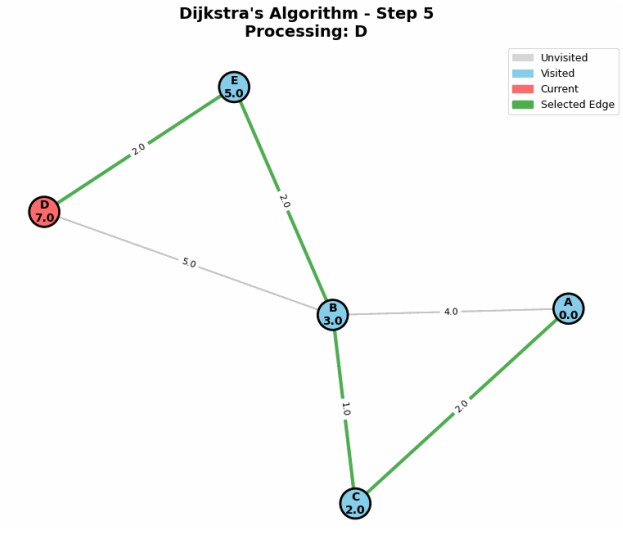

Dijkstra's Algorithm - Initial State
Unvisited
Visited
Current
Selected Edge
E ∞
D ∞
B ∞
A 0.0
C ∞
2.0
2.0
5.0
4.0
1.0
2.0


Dijkstra's Algorithm - Step 1
Processing: A
Unvisited
Visited
Current
Selected Edge
E ∞
D ∞
B 4.0
A 0.0
C 2.0
2.0
2.0
5.0
4.0
1.0
2.0


Dijkstra's Algorithm - Step 2
Processing: C
Unvisited
Visited
Current
Selected Edge
E ∞
D ∞
B 3.0
A 0.0
C 2.0
2.0
2.0
5.0
4.0
1.0
2.0

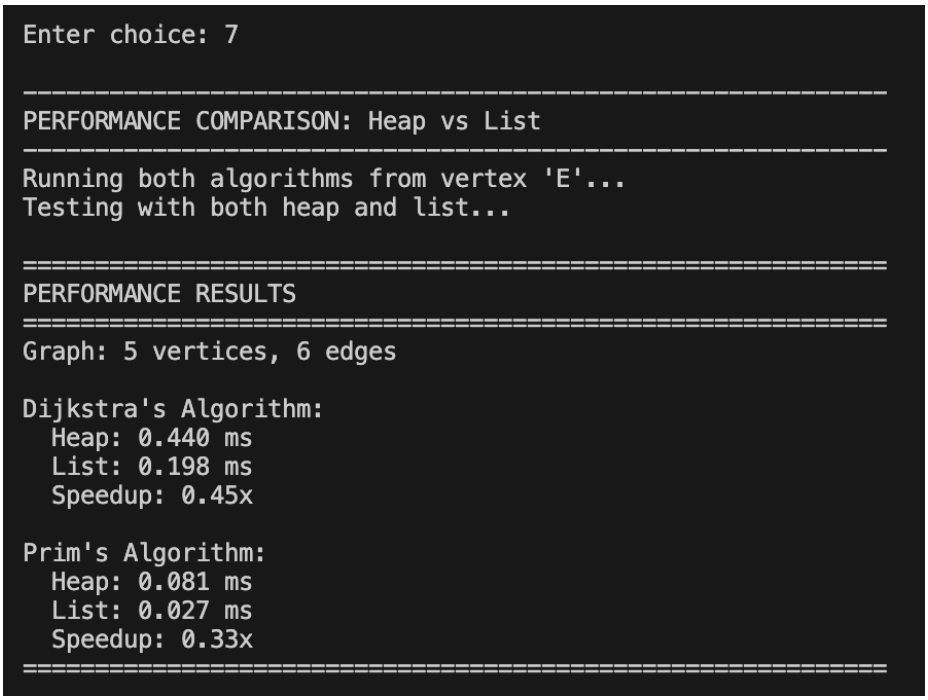

Dijkstra's Algorithm - Step 3
Processing: B
Unvisited
Visited
Current
Selected Edge
E 5.0
D 8.0
B 3.0
A 0.0
C 2.0
2.0
2.0
5.0
4.0
1.0
2.0


Dijkstra's Algorithm - Step 4
Processing: E
Unvisited
Visited
Current
Selected Edge
E 5.0
D 7.0
B 3.0
A 0.0
C 2.0
2.0
2.0
5.0
4.0
1.0
2.0


Dijkstra's Algorithm - Step 5
Processing: D
Unvisited
Visited
Current
Selected Edge
E 5.0
D 7.0
B 3.0
A 0.0
C 2.0
2.0
2.0
5.0
4.0
1.0
2.0

Compare Heap vs List Performance

Similar to performance test, it will run a performance test with this graph.



Challenges and Solutions

1. Binary Heap Decrease-Key Operation

Problem: Standard heaps require O(n) linear search to find elements before updating their priorities.

Solution: We maintained a position map (dictionary) that tracks each element's current heap index. When swapping elements during heap operations, we update both the heap array and the position map simultaneously.

2. Performance Measurement Accuracy

Problem: Small graphs execute too quickly for reliable timing measurements.

Solution: Used time.perf_counter() for high-resolution timing, ran multiple iterations and averaged results, and tested across a range of graph sizes (10 to 500 vertices) to observe clear performance trends.

Learning Outcomes

1. Data Structure Choice Significantly Impacts Performance

The same algorithm with different data structures produces dramatically different results. For small graphs (V=10), both implementations perform similarly since constant factor overhead dominates. However, at V=500, the heap implementation is **5-7x faster**, demonstrating how asymptotic complexity becomes the dominant factor as input size grows.

2. Theory Accurately Predicts Practice

Our empirical results validated theoretical complexity analysis. The heap implementation showed O((V+E)log V) growth as predicted, while the linked list exhibited $O(V^2)$ behavior. The crossover point occurred around 50-100 vertices, exactly where theoretical analysis suggested the heap would begin outperforming the list.

Reference

1. https://pillow.readthedocs.io/en/stable/reference/Image.html
2. https://www.geeksforgeeks.org/python/python-program-for-dijkstras-shortest-path-algorithm-greedy-algo-7/
3. https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php
4. https://www.geeksforgeeks.org/python/create-an-animated-gif-using-python-matplotlib/
5. https://www.geeksforgeeks.org/dsa/prims-minimum-spanning-tree-mst-greedy-algo-5/