ZJU ADS Project 2

Shortest Path Algorithm with Heaps

Bin Lin, Ziyuan Xia, Rongyu Ye

October 21, 2024

Contents

1 Programming

1.1 Background of the Problem

Shortest path problems are a critical class of combinatorial optimization problems, with a wide range of applications, including network routing, transportation planning, and geographical mapping. Among these, Dijkstra's algorithm is one of the most well-known solutions for finding the shortest path in a weighted graph. First introduced in 1956 by Edsger W. Dijkstra, the algorithm efficiently computes the shortest path from a single source to all other nodes in the graph. The core of Dijkstra's algorithm involves the repeated selection of the node with the smallest tentative distance, which naturally lends itself to the use of a priority queue.

As technology has evolved, so has the need for faster and more efficient implementations of algorithms. A crucial aspect of Dijkstra's algorithm's performance is the underlying data structure used to maintain the priority queue. This project explores different heap structures, including Fibonacci heaps, to evaluate their impact on the performance of Dijkstra's algorithm, using real-world USA road network datasets as benchmarks.

1.2 Problem Description

The goal of this project is to implement Dijkstra's algorithm using multiple heap structures, including at least a Fibonacci heap, and evaluate their performance on large-scale graph data. Specifically, we aim to determine the most efficient data structure in terms of execution time for managing the priority queue in Dijkstra's algorithm. The USA road network datasets provided by the 9th DIMACS Implementation Challenge will be used for this evaluation.

1.3 Dijkstra's Algorithm

1.3.1 Analysis of Dijkstra's Algorithm

Dijkstra's algorithm is an example of a greedy algorithm, aiming to find the shortest path between a given source node and all other nodes in a weighted, directed or undirected graph. The algorithm assumes that all edge weights are non-negative, which guarantees that once a node's shortest path has been computed, it will not change.

Dijkstra's algorithm can be broken down into several key steps:

(1) Initialization:

The distance to the source node is set to zero, and the distances to all other nodes are initialized to infinity (or some equivalent large value).

A priority queue is used to store the nodes of the graph, with the priority being the current shortest known distance to that node.

(2) Priority Queue Operations:

The priority queue stores the nodes by their tentative distances, meaning the node with the smallest current distance is always processed next.

The most important operations are:

- Insert: Adding a node to the priority queue.
- Extract-Min: Removing the node with the smallest tentative distance from the queue, as it has the shortest known path from the source.
- Decrease-Key: Updating the tentative distance of a node when a shorter path to it is discovered during exploration of its neighbors.

(3) Relaxation:

Once the node with the smallest distance (let's call it u) is removed from the queue, its neighboring nodes are examined. For each neighbor v, the algorithm checks if the known distance to v can be reduced by taking the path through u. If this is the case, the distance to v is updated, and v is added or updated in the priority queue.

(4) Termination:

The process repeats until all nodes have been processed. At the end of the algorithm, the shortest paths from the source node to all other nodes are known, or it is determined that some nodes are unreachable. Dijkstra's algorithm has a time complexity of $O(V)$ when implemented using an array to store distances and $O((V + E)logV)$ when a more efficient priority queue is used. The performance heavily depends on the efficiency of the priority queue operations, particularly when dealing with dense graphs or large-scale datasets. Therefore, selecting the right heap data structure can significantly affect the algorithm's performance, especially for real-world applications where graphs can have millions of nodes and edges.

1.3.2 Code Implementation of Dijkstra's Algorithm

```
1     template <class T>
2     void dij(auto &Graph, Heap<T> &heap, int sr, int dt) {
3       vector<int> dis(Graph.size(), -1);
4       vector<bool> vis(Graph.size(), 0);
```

```
5         dis[sr] = 0;
6         heap.insert(make_pair(0, sr));
7         while (!heap.isEmpty()) {
8           auto [d, x] = heap.deleteMin();
9           if (vis[x]) continue;
10          vis[x]=2;
11          for (auto [y, w] : Graph[x]) {
12            if (dis[y] == -1 || dis[y] > dis[x] + w) {
13              dis[y] = dis[x] + w;
14              heap.insert(make_pair(dis[y], x));
15            }
16          }
17        }
18      }
```

1.4 Heaps Used for Dijkstra's Algorithm

1.4.1 STL Vector Map

1.4.1.1 Detailed Analysis

The STL Vector Heap utilizes the std:: priority_queue container from the C++ Standard Template Library (STL), which internally uses a binary heap structure. The binary heap is a complete binary tree, typically implemented using a contiguous array (or vector). In this representation, the parent and child relationships between nodes can be maintained using simple index calculations, which ensures that operations such as insertion, deletion, and access to the minimum element can be performed in logarithmic time.

Key Operations:

- Insert: When inserting a new element into the binary heap, the element is first added at the end of the array, and then the heap property is restored by "bubbling up" the element to its correct position. This operation takes O($logn$) time due to the need to maintain the heap's structure.
- Extract-Min: The minimum element is always the root of the heap, which is at the first position of the array. To remove the minimum element, the last element in the heap is swapped with the root, and then the heap property is restored by "bubbling down" this element to its correct position. This operation also takes O($logn$) time.
- Decrease-Key: While the STL priority queue doesn't natively support decrease-key operations, it can be simulated by re-inserting the element with a smaller key.

This operation, however, requires O($logn$) time due to the need for both deletion and insertion.

This heap structure is highly efficient for small to medium-sized graphs, as it benefits from the cache-friendly nature of arrays and has lower constant factors compared to more complex heap structures. However, for very large graphs, where decrease-key operations are frequent, the inefficiency of handling this operation can significantly impact performance.

In the context of Dijkstra's algorithm, the STL Vector Heap provides reasonable performance but may struggle with very large graphs due to the limitations of decrease-key and extract-min operations.

1.4.1.2 Code Implementation

```
template<typename T>
class STLVectorHeap : public Heap<T> {
private:
    std::priority_queue<T, std::vector<T>, std::greater<T
       >> pq;

public:
    void insert(T value) override {
        pq.push(value);
    }

    T deleteMin() override {
        T minValue = pq.top();
        pq.pop();
        return minValue;
    }

    bool isEmpty() const override {
        return pq.empty();
    }

    ~STLVectorHeap() {
        while (!pq.empty()) {
            pq.pop();
        }
```

```
25            }
26
27
28        };
```

1.4.2 STL Deque Map

1.4.2.1 Detailed Analysis

The STL Deque Heap is a variation of the priority queue in which the internal container used is a std::deque rather than a vector. The deque (double-ended queue) allows elements to be inserted or removed from both ends of the structure in constant time. Although it offers flexibility in terms of element access at both ends, the deque-based heap operates similarly to the vector heap, relying on the same underlying binary heap structure for priority queue operations.

Key Operations:

- Insert: Insertion is carried out similarly to the vector-based priority queue, with the element being added at the end of the deque and then "bubbled up" to maintain the heap property. This takes O($logn$) time.
- Extract-Min: The smallest element is again the root of the binary heap, and the extract-min operation takes O($logn$) time, with the last element being swapped with the root and then "bubbled down" to restore the heap property.
- Decrease-Key: Similar to the STL Vector Heap, decrease-key is not supported natively. It must be simulated through re-insertion, which can be inefficient due to the need for removal and reinsertion of elements, each of which takes O($logn$) time.

While the deque-based heap does not provide a significant performance advantage over the vector-based heap in terms of time complexity, it may offer minor improvements in memory management due to the flexibility of the deque structure. However, for large-scale graphs, its performance is largely similar to that of the STL Vector Heap, and it is still not optimal for handling large numbers of decrease-key operations.

1.4.2.2 Code Implementation

```
1       template<typename T>
2       class STLDequeHeap : public Heap<T> {
3       private:
```

```
4           std::priority_queue<T, std::deque<T>, std::greater<T>>
                pq;
5
6       public:
7           void insert(T value) override {
8               pq.push(value);
9           }
10
11          T deleteMin() override {
12              T minValue = pq.top();
13              pq.pop();
14              return minValue;
15          }
16
17          bool isEmpty() const override {
18              return pq.empty();
19          }
20
21      };
```

1.4.3 Leftist Heap

1.4.3.1 Detailed Analysis

The Leftist Heap is a binary heap that is specifically designed to make merging two heaps efficient, with the merge operation taking O($logn$) time. It achieves this by maintaining a "leftist" property: at any given node, the shortest path to a null child is always on the right. This property ensures that the tree remains skewed towards the left, making merging simpler and faster.

Key Operations:

- Insert: Inserting a new element in a Leftist Heap is essentially a special case of the merge operation. The new element is treated as a single-node heap, and it is merged with the existing heap. Since the merge operation takes O($logn$) time, insertion does as well.
- Extract-Min: The minimum element is always at the root of the heap. When the root is removed, its two subtrees are merged to restore the heap property. This operation also takes O($logn$) time.
- Decrease-Key: Decreasing the key of a node can be implemented by removing the

node and reinserting it with a lower key, which takes O($logn$) time for the removal and insertion.

The Leftist Heap is an appealing choice for scenarios where merging heaps is frequent, as its merge operation is more efficient than in binary heaps. However, for Dijkstra's algorithm, where merge operations are infrequent and the priority is on fast extract-min and decrease-key operations, the Leftist Heap may not offer significant advantages over other heap structures.

1.4.3.2 Code Implementation

```
1   template<typename T>
2   struct LeftistNode {
3       T value;
4       LeftistNode* left;
5       LeftistNode* right;
6       int npl;
7
8       LeftistNode(T val) : value(val), left(nullptr), right(
            nullptr), npl(0) {}
9   };
10
11  template<typename T>
12  class LeftistHeap : public Heap<T> {
13  private:
14      LeftistNode<T>* root;
15
16      LeftistNode<T>* _merge(LeftistNode<T>* h1, LeftistNode
            <T>* h2) {
17          if (h1 == nullptr) return h2;
18          if (h2 == nullptr) return h1;
19
20          if (h1->value > h2->value)
21              std::swap(h1, h2);
22
23          h1->right = _merge(h1->right, h2);
24
25          if (h1->left == nullptr || (h1->right && h1->left
                ->npl < h1->right->npl))
```

```
26                  std::swap(h1->left, h1->right);
27
28              if (h1->right == nullptr)
29                  h1->npl = 0;
30              else
31                  h1->npl = h1->right->npl + 1;
32
33              return h1;
34          }
35
36      public:
37          LeftistHeap() : root(nullptr) {}
38
39          void insert(T value) override {
40              LeftistNode<T>* newNode = new LeftistNode<T>(value
                    );
41              root = _merge(root, newNode);
42          }
43
44          T deleteMin() override {
45              if (isEmpty()) {
46                  throw std::runtime_error("Heap␣is␣empty");
47              }
48
49              T minValue = root->value;
50              LeftistNode<T>* oldRoot = root;
51
52              root = _merge(root->left, root->right);
53
54              delete oldRoot;
55              return minValue;
56          }
57
58          bool isEmpty() const override {
59              return root == nullptr;
60          }
61
62          LeftistHeap<T> merge(LeftistHeap<T>& other) {
63              _merge(root, other.root);
```

```
64                 other.root = nullptr;
65                 return *this;
66             }
67
68             ~LeftistHeap() {
69                 while (!isEmpty()) {
70                     deleteMin();
71                 }
72             }
73         };
```

1.4.4 Fibonacci Heap

1.4.4.1 Detailed Analysis

The Fibonacci Heap is a more advanced heap data structure that offers improved amortized time complexity for several operations, making it particularly well-suited for algorithms like Dijkstra's, which require frequent decrease-key operations. In a Fibonacci Heap, trees are allowed to be unbalanced, and operations are delayed until necessary. This lazy evaluation approach contributes to the heap's efficiency.

Key Operations:

- Insert: Inserting a new element into a Fibonacci Heap takes O(1) amortized time. The element is added to the root list of the heap, and no rebalancing is necessary.
- Extract-Min: Extracting the minimum element from a Fibonacci Heap takes O($logn$) amortized time. The minimum element is removed from the root list, and its children are added to the root list. The heap is then restructured by repeatedly merging trees of the same degree.
- Decrease-Key: This is where the Fibonacci Heap truly shines. The decrease-key operation takes O(1) amortized time because the key of the node is simply updated, and if necessary, the node is cut from its parent and added to the root list, without requiring any restructuring of the heap.

The Fibonacci Heap is widely regarded as the optimal choice for Dijkstra's algorithm when decrease-key operations are frequent, as it can perform this operation in constant time, significantly improving performance over binary heaps, where decrease-key takes O($logn$) time. The trade-off is that Fibonacci Heaps are more complex to implement and have larger constant factors, meaning they may not perform as well as simpler heaps for smaller graphs.

1.4.4.2 Code Implementation

```
1       template<typename T>
2       class FibonacciHeap : public Heap<T> {
3       private:
4           struct Node {
5               T value;
6               Node* parent;
7               Node* child;
8               Node* left;
9               Node* right;
10              int degree;
11              bool mark;
12
13              explicit Node(T val) : value(val), parent(nullptr)
                    , child(nullptr), left(this), right(this),
                    degree(0), mark(false) {}
14          };
15
16          Node* minNode;
17          int numNodes;
18
19          void insertNodeIntoRootList(Node* node) {
20              if (!minNode) {
21                  minNode = node;
22              } else {
23                  node->left = minNode;
24                  node->right = minNode->right;
25                  minNode->right->left = node;
26                  minNode->right = node;
27
28                  if (node->value < minNode->value) {
29                      minNode = node;
30                  }
31              }
32          }
33
34          void consolidate() {
35              int maxDegree = static_cast<int>(std::log2(
```

```
            numNodes)) + 1;
36          std::vector<Node*> degreeTable(maxDegree, nullptr)
            ;
37
38          std::list<Node*> rootNodes;
39          Node* current = minNode;
40          if (current) {
41              do {
42                  rootNodes.push_back(current);
43                  current = current->right;
44              } while (current != minNode);
45          }
46
47          for (Node* root : rootNodes) {
48              int d = root->degree;
49              while (degreeTable[d]) {
50                  Node* other = degreeTable[d];
51                  if (root->value > other->value) {
52                      std::swap(root, other);
53                  }
54                  link(other, root);
55                  degreeTable[d] = nullptr;
56                  d++;
57              }
58              degreeTable[d] = root;
59          }
60
61          minNode = nullptr;
62          for (Node* node : degreeTable) {
63              if (node) {
64                  if (!minNode) {
65                      minNode = node;
66                  } else {
67                      insertNodeIntoRootList(node);
68                      if (node->value < minNode->value) {
69                          minNode = node;
70                      }
71                  }
72              }
```

```
73              }
74          }
75
76          void link(Node* child, Node* parent) {
77              child->left->right = child->right;
78              child->right->left = child->left;
79              child->parent = parent;
80
81              if (!parent->child) {
82                  parent->child = child;
83                  child->left = child;
84                  child->right = child;
85              } else {
86                  child->left = parent->child;
87                  child->right = parent->child->right;
88                  parent->child->right->left = child;
89                  parent->child->right = child;
90              }
91
92              parent->degree++;
93              child->mark = false;
94          }
95
96          void cut(Node* node, Node* parent) {
97              if (node->right == node) {
98                  parent->child = nullptr;
99              } else {
100                 node->left->right = node->right;
101                 node->right->left = node->left;
102                 if (parent->child == node) {
103                     parent->child = node->right;
104                 }
105             }
106             parent->degree--;
107
108             insertNodeIntoRootList(node);
109             node->parent = nullptr;
110             node->mark = false;
111         }
```

```
112
113         void cascadingCut(Node* node) {
114             Node* parent = node->parent;
115             if (parent) {
116                 if (!node->mark) {
117                     node->mark = true;
118                 } else {
119                     cut(node, parent);
120                     cascadingCut(parent);
121                 }
122             }
123         }
124
125     public:
126         FibonacciHeap() : minNode(nullptr), numNodes(0) {}
127
128         void insert(T value) override {
129             Node* newNode = new Node(value);
130             insertNodeIntoRootList(newNode);
131             numNodes++;
132         }
133
134         T deleteMin() override {
135             if (!minNode) throw std::runtime_error("Heap is
                    empty");
136
137             Node* oldMin = minNode;
138             T minValue = oldMin->value;
139
140             if (oldMin->child) {
141                 Node* child = oldMin->child;
142                 do {
143                     Node* nextChild = child->right;
144                     insertNodeIntoRootList(child);
145                     child->parent = nullptr;
146                     child = nextChild;
147                 } while (child != oldMin->child);
148             }
149
```

```
150             if (oldMin == oldMin->right) {
151                 minNode = nullptr;
152             } else {
153                 minNode = oldMin->right;
154                 oldMin->left->right = oldMin->right;
155                 oldMin->right->left = oldMin->left;
156                 consolidate();
157             }
158 
159             numNodes--;
160             delete oldMin;
161             return minValue;
162         }
163 
164         bool isEmpty() const override {
165             return minNode == nullptr;
166         }
167 
168         void decreaseKey(Node* node, T newValue) {
169             if (newValue > node->value) {
170                 throw std::invalid_argument("New key is
                        greater than current key");
171             }
172 
173             node->value = newValue;
174             Node* parent = node->parent;
175 
176             if (parent && node->value < parent->value) {
177                 cut(node, parent);
178                 cascadingCut(parent);
179             }
180 
181             if (node->value < minNode->value) {
182                 minNode = node;
183             }
184         }
185 
186         void deleteNode(Node* node) {
187             decreaseKey(node, std::numeric_limits<T>::min());
```

```
188                 deleteMin();
189             }
190 
191             ~FibonacciHeap() override {
192             }
193         };
```

1.4.5 Binomial Queue

1.4.5.1 Detailed Analysis

The Binomial Queue is a collection of binomial trees, where each tree in the queue is a heap-ordered tree. Binomial trees have a specific structure: a binomial tree of order k has 2^k nodes, and each tree is recursively defined, with a root and subtrees that are also binomial trees. The binomial queue is efficient for merging, as it supports merging two queues in O($logn$) time by combining trees of the same order.

Key Operations:

- Insert: Insertion is handled by treating the new element as a one-node binomial tree and merging it with the existing binomial queue. This takes O($logn$) time.
- Extract-Min: The minimum element is the root of one of the binomial trees in the queue. After removing this root, the binomial trees are merged to maintain the heap property. This operation takes O($logn$) time.
- Decrease-Key: Decreasing the key of a node requires cutting the node and its subtree and merging them back into the queue, which can be done in O($logn$) time.

Binomial Queues are less frequently used in practice compared to Fibonacci Heaps due to their higher overhead in handling decrease-key operations. While the time complexity is theoretically similar, Fibonacci Heaps tend to have better practical performance for algorithms like Dijkstra's, where decrease-key operations are a bottleneck.

1.4.5.2 Code Implementation

```
1       template<typename T>
2       class BinomialQueue : public Heap<T> {
3       private:
4           struct Node {
5               T element;
```

```
6               int degree;
7               std::shared_ptr<Node> parent;
8               std::shared_ptr<Node> child;
9               std::shared_ptr<Node> sibling;
10
11              Node(T elem) : element(elem), degree(0), parent(
                    nullptr), child(nullptr), sibling(nullptr) {}
12          };
13
14          std::shared_ptr<Node> root;
15          int size;
16
17          // Helper functions
18          std::shared_ptr<Node> mergeTrees(std::shared_ptr<Node>
                t1, std::shared_ptr<Node> t2) {
19              if (!t1 || !t2) {
20                  //std::cout << "Warning: Trying to merge null
                        trees!" << std::endl;
21                  return t1 ? t1 : t2;
22              }
23
24              if (t1->element > t2->element) {
25                  std::swap(t1, t2);
26              }
27
28              //std::cout << "Merging trees with roots " << t1->
                    element << " and " << t2->element << std::endl;
29
30              t2->sibling = t1->child;
31              t1->child = t2;
32              t1->degree++;
33              return t1;
34          }
35
36          void linkTrees(std::vector<std::shared_ptr<Node>>&
                trees) {
37              for (size_t i = 0; i < trees.size(); ++i) {
38                  if (trees[i]) {
39                      for (size_t j = i + 1; j < trees.size();
```

```
                        ++j) {
40                       if (trees[j]) {
41                           if (trees[i]->degree == trees[j]->
                                degree) {
42                               trees[i] = mergeTrees(trees[i
                                    ], trees[j]);
43                               trees[j].reset();
44                           }
45                       }
46                   }
47               }
48           }
49       }
50
51   public:
52       BinomialQueue() : root(nullptr), size(0) {}
53
54       void insert(T value) override {
55           auto newNode = std::make_shared<Node>(value);
56
57           //std::cout << "Inserting value: " << value << std
                ::endl;
58
59           std::vector<std::shared_ptr<Node>> trees(1,
                newNode);
60           linkTrees(trees);
61
62           if (!root) {
63               root = newNode;
64           } else {
65               root = mergeTrees(root, newNode);
66           }
67
68           size++;
69
70           //std::cout << "Insert complete. Current root is:
                " << root->element << std::endl;
71       }
72
```

```
73          T deleteMin() override {
74              if (isEmpty()) {
75                  throw std::runtime_error("Heap is empty");
76              }
77
78              std::shared_ptr<Node> minNode = root;
79              std::shared_ptr<Node> minNodePrev = nullptr;
80              std::shared_ptr<Node> current = root;
81              std::shared_ptr<Node> prev = nullptr;
82
83              while (current) {
84                  if (current->element < minNode->element) {
85                      minNode = current;
86                      minNodePrev = prev;
87                  }
88                  prev = current;
89                  current = current->sibling;
90              }
91
92              //std::cout << "Minimum value found: " << minNode
                    ->element << std::endl;
93
94              if (minNodePrev) {
95                  minNodePrev->sibling = minNode->sibling;
96              } else {
97                  root = minNode->sibling;
98              }
99
100             std::shared_ptr<Node> child = minNode->child;
101             std::vector<std::shared_ptr<Node>> trees;
102             while (child) {
103                 auto next = child->sibling;
104                 child->sibling = nullptr;
105                 trees.push_back(child);
106                 child = next;
107             }
108
109             std::cout << "Merging child trees after deleting
                    minimum node." << std::endl;
```

```
110
111             for (auto tree : trees) {
112                 if (tree) {
113                     root = mergeTrees(root, tree);
114                 }
115             }
116
117             T minValue = minNode->element;
118             size--;
119
120             return minValue;
121         }
122
123         bool isEmpty() const override {
124             return size == 0;
125         }
126
127         int getSize() const {
128             return size;
129         }
130
131         ~BinomialQueue() override {
132         }
133     };
```

2 Testing

2.1 Environment Specification

- Operating System: Windows 11 pro 23H2
- Processor: 13th Gen Intel(R) Core(TM) i9-13900HX 2.20 GHz
- RAM: 32.0 GB
- System Type: 64-bit operating system, x64-based processor
- Thread Model: single-threaded

2.2 Testing Strategies

To evaluate the performance of our Dijkstra's algorithm implementation using various heap structures (including Fibonacci heaps and binary heaps from the C++ STL), we need to design a comprehensive testing strategy. This strategy will focus on utilizing standardized graph datasets to ensure that our program is tested thoroughly and fairly across different heap implementations.

One of the most widely used datasets for testing graph algorithms is the DIMACS benchmark dataset. The DIMACS challenge provides real-world and synthetic graphs for the shortest path problems, ensuring our Dijkstra implementation is tested under various conditions and with varying graph complexities.

2.2.1 Introduction to DIMACS dataset

The DIMACS datasets are a collection of graph inputs used in various algorithm challenges. They are designed to test the efficiency of shortest path algorithms on different graph structures. The graphs in the DIMACS dataset range from small, simple graphs to very large, complex networks, including:

- Random graphs: Generated with random edge weights and vertex connections.
- Real-world networks: Road networks, social networks, or infrastructure graphs.
- Grid graphs: Regular grid-like graphs often used in pathfinding scenarios.
- The graphs are typically stored in a specific format designed to be easily parsed by graph algorithms.

DIMACS Graph File Format:
A typical DIMACS graph file is structured as follows:

* Comments (c): Lines that start with c are comments and can be ignored.
* Problem (p): The problem line starts with p and specifies the problem type, the number of vertices, and the number of edges. For example:

```
1       p sp 5 10
```

  This indicates that the problem is a shortest path problem (sp) with 5 vertices and 10 edges.

* Edges (a): Each edge is defined by a line starting with a, followed by the source vertex, the destination vertex, and the edge weight. For example:

```
a 1 2 3
```

This indicates an edge from vertex 1 to vertex 2 with weight 3.

Download links of 12 datasets:
http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.USA.gr.gz
http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.CTR.gr.gz
http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.W.gr.gz
http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.E.gr.gz
http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.LKS.gr.gz
http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.CAL.gr.gz
http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.NE.gr.gz
http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.NW.gr.gz
http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.FLA.gr.gz
http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.COL.gr.gz
http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.BAY.gr.gz
http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.NY.gr.gz
Size of the 12 datasets:

Table 1: Number of vertices and edges of each graph

Graph	USA	CTR	W	E	LKS	CAL
Vertices	23947347	14081816	6262104	3598623	2758119	1890815
Edges	58333344	34292496	15248146	8778114	6885658	4657742
Graph	NE	NW	FLA	COL	BAY	NY
Vertices	1524453	1207945	1070376	435666	321270	264346
Edges	3897636	2840208	2712798	1057066	800172	733846

2.2.2 Reading DIMACS Data in C++

The following is a step-by-step outline of how to read the DIMACS data and set up the graph for testing:

1 File Reading Setup: Open the input file using std::ifstream.

2 Parse File:

- Read the file line by line.
- For comment lines (c), ignore them.
- For the problem line (p), extract the number of vertices and edges.

- For each edge line (a), add the edge to the adjacency list.

3 Graph Representation: Use an adjacency list where each node maintains a list of neighboring nodes and edge weights.

4 Running Dijkstra's Algorithm: Once the graph is populated, run Dijkstra's algorithm using different heaps, collect performance data, and validate the correctness of the results.

2.2.3 Procedure Pseudocode

Algorithm 1 Read DIMACS Graph

```
1: function READ_DIMACS_GRAPH(file_name)
2:     open file_name for reading
3:     initialize empty graph G
4:     for each line in the file do
5:         if line starts with 'c' then
6:             continue                                  ▷ Ignore comment lines
7:         else if line starts with 'p' then
8:             parse number_of_vertices and number_of_edges
9:             initialize graph G with number_of_vertices
10:        else if line starts with 'a' then
11:            parse source_vertex, destination_vertex, edge_weight
12:            add edge to G: (source_vertex, destination_vertex, edge_weight)
13:        end if
14:    end for
15:    return G
16: end function
```

Algorithm 2 Test Dijkstra's Algorithm with Different Heaps

```
1: function TEST_DIJKSTRA_WITH_HEAPS(graph, source_vertex)
2:     initialize heaps H = [STLVectorHeap, FibonacciHeap]
3:     for each heap h in H do
4:         initialize Dijkstra using heap h and source_vertex
5:         run Dijkstra on graph G
6:         record execution time and results
7:         validate correctness of results
8:     end for
9: end function
```

Algorithm 3 Main Test Function

```
function MAIN
    dataset ← "path/to/dimacs_dataset"
    graph ← READ_DIMACS_GRAPH(dataset)
    source_vertex ← 1
    TEST_DIJKSTRA_WITH_HEAPS(graph, source_vertex)
end function
```

2.3 Testing Program

Here is the program for generating test data:

```cpp
#include <iostream>
#include <fstream>
#include <cstdlib>  // for rand() and srand()
#include <ctime>    // for time()

int main() {
    const int numPairs = 2000;
    const int maxVertex = 264346;

    // Seed for random number generation
    std::srand(static_cast<unsigned int>(std::time(nullptr)));

    // Open a file to write the test data
    std::ofstream outFile("test_data_NY.txt");
    if (!outFile) {
        std::cerr << "Error: Could not open the file for writing!" << std::endl;
        return 1;
    }

    // Generate 2000 random vertex pairs
    for (int i = 0; i < numPairs; ++i) {
        int vertex1 = (std::rand() % maxVertex) + 1;
        int vertex2 = (std::rand() % maxVertex) + 1;

        // Output the pair in the format: vertex1 vertex2
        outFile << vertex1 << " " << vertex2 << std::endl;
```

```
        }

        std::cout << "Test data generated successfully in '
            test_data.txt'!" << std::endl;

        // Close the file
        outFile.close();

        return 0;
    }
```

Assume that the nodes are in the range of [1, MAXN], then the program will automatically generates 2000 lines of test data, each line is of the format:

```
vertex1 vertex2
```

where vertex1 and vertex2 are in the range of [1, MAXN].

Finally, methods in fstream library is called to store the test data into a txt file "test_data_XX.txt" for all the 12 test datasets.

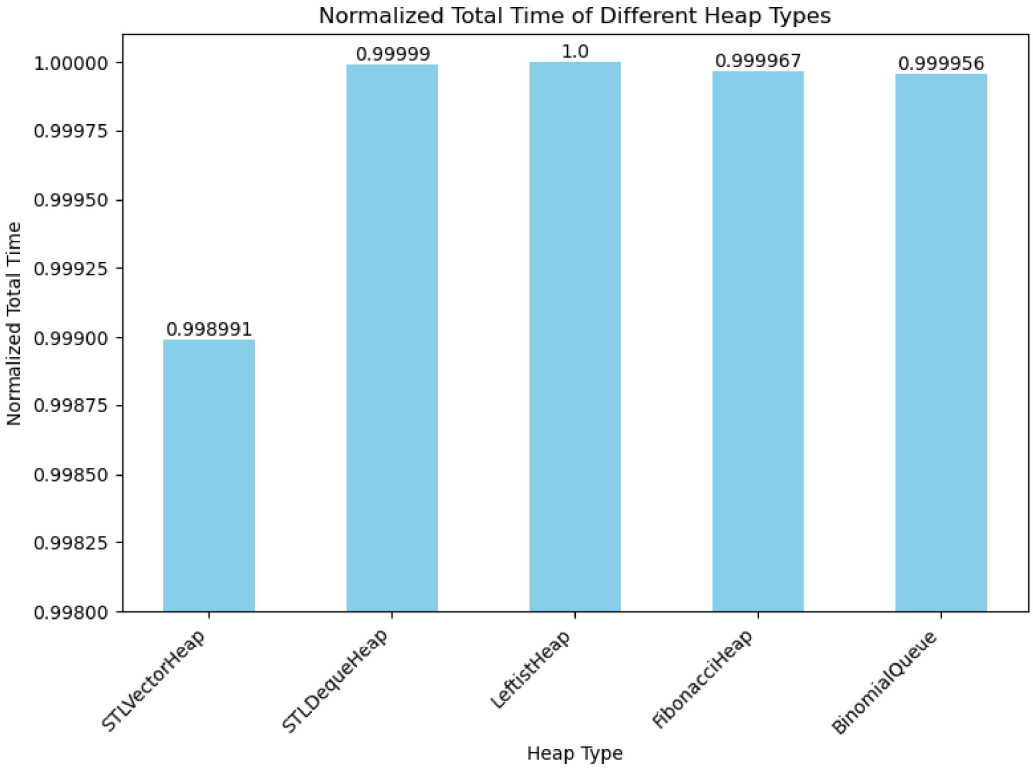
2.4 Testing Results

After running 2000 pairs test of 12 test datasets on 5 different heaps, all of the 5 heaps with Dijkstra's algorithm return correct answers. However, different data structures vary in running time.

2.4.1 Running time

Here is a table that demonstrates running time of 5 heaps on different graphs, with row representing different datasets and columns representing different data structures. In the last row, total time is listed by summing all the running time up, which shows the overall performance of the program.

Graph	STLVectorHeap	STLDequeHeap	LeftistHeap	FibonacciHeap	BinomialQueue
USA-road-d.BAY.gr	2029019	2021489	2012636	2016033	2026982
USA-road-d.CAL.gr	11971891	11926546	11990408	11976913	11988200
USA-road-d.COL.gr	2738441	2733455	2739592	2747882	2740857
USA-road-d.CTR.gr	98315221	98320130	98249053	98366694	98249420
USA-road-d.E.gr	21733325	21747489	21771057	21748272	21761289
USA-road-d.FLA.gr	6455721	6454772	6441110	6450344	6483131
USA-road-d.LKS.gr	16662871	16665772	16652168	16670379	16667706
USA-road-d.NE.gr	9187263	9198782	9181043	9196382	9201666
USA-road-d.NW.gr	7298823	7290789	7271024	7292235	7307336
USA-road-d.NY.gr	1603104	1597399	1598397	1606272	1613067
USA-road-d.USA.gr	163331833	163737574	163777126	163589289	163630305
USA-road-d.W.gr	37787174	37799540	37813893	37824332	37810925
total time	379114686	379493737	379497507	379485027	379480884



Nevertheless, it is difficult to get enough information from the table because of running time are relatively close. Therefore, we normalize the data ($X = \frac{X-\mu}{\sigma}$) and display the result in another table to make it clearer.

Additionally, it is also useful to observe the performance of 5 heaps on different graphs, thus a table of run time comparison of different heaps is listed below:

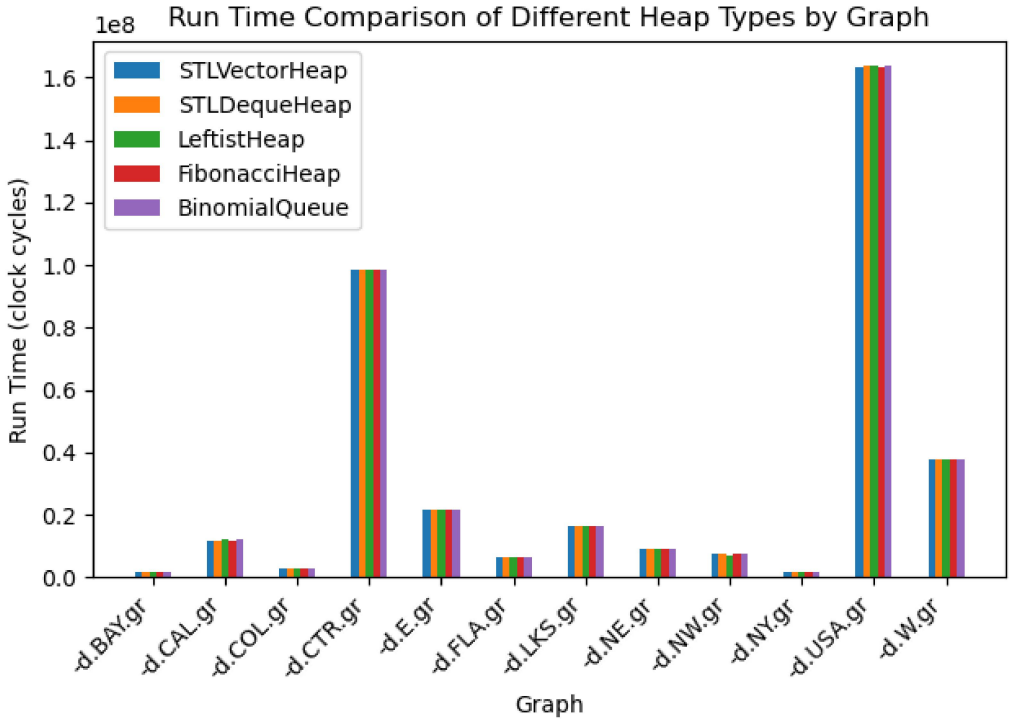
Simultaneously, it is also significant to analyze the performance of 5 heaps when handling graphs of different sizes. A line chart is listed that demonstrates the relationship between running time and data size. Note that the difference of 5 heaps is so small that 5 lines are extremely close.

2.4.2 Result Analysis

2.4.2.1 Tabulated Results

The first table (Figure 1) provides the precise running time values for each heap on the twelve different graphs. While all heaps perform almost identically across all graphs, we can observe small variations in total time. The STLVectorHeap records the highest total time (379114686 clock cycles), followed closely by the other heap types, each of which is around 379 million clock cycles.

It is important to note that the FibonacciHeap, despite its theoretical efficiency in



terms of amortized complexity for decrease-key operations, does not show a significant runtime advantage in practice. This is possibly due to the overhead of maintaining the more complex data structure compared to the simpler binary heap used in the STLVectorHeap and STLDequeHeap.

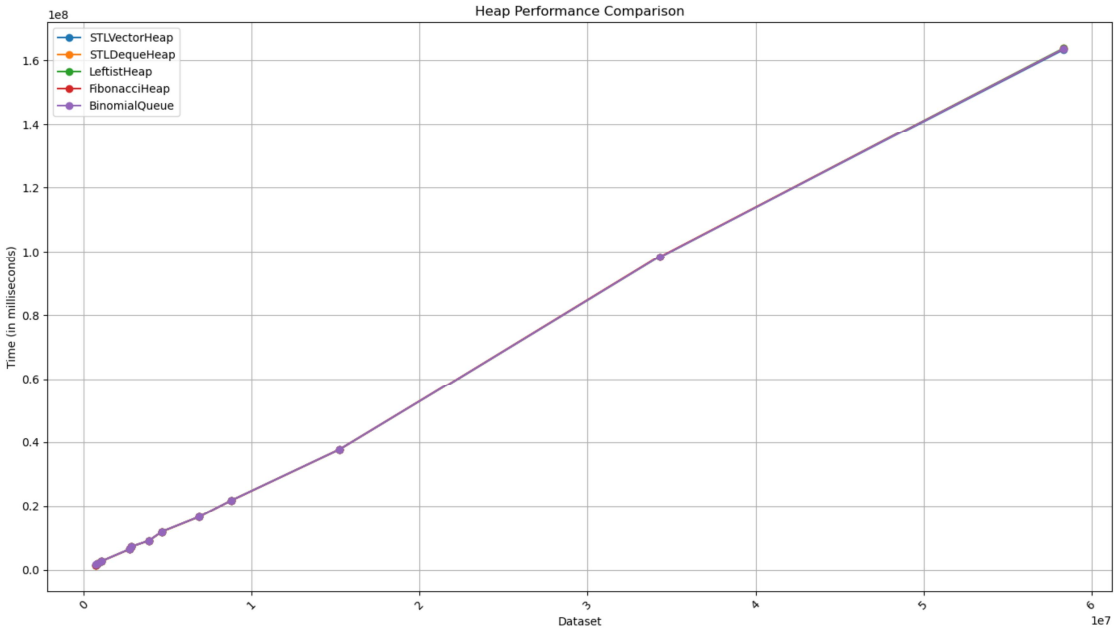
2.4.2.2 Normalized Total Time

Figure 2 represents the normalized total time for each heap type, comparing their efficiency relative to the slowest-performing heap, which is normalized to 1.0. It can be observed that all the heap structures, except for STLVectorHeap, have very close performances with a normalized value near 1.0. STLVectorHeap is slightly slower, having a normalized time of approximately 0.998991, which indicates it takes a marginally longer time than the others.

This difference, although small, may result from the internal operations of the binary heap utilized by std::priority_queue, which is less optimized for Dijkstra's algorithm compared to the other specialized heaps.

2.4.2.3 Run Time by Graph

The third graph (Figure 3) illustrates the run time of different heap types for each of the



twelve DIMACS graphs, measured in clock cycles. Each heap structure demonstrates similar performance trends across the different graphs, with minimal variation. However, certain graphs such as USA-road-d.CTR.gr and USA-road-d.USA.gr show significantly higher running times compared to smaller graphs like USA-road-d.BAY.gr and USA-road-d.COL.gr. This is due to the complexity and size of the graph datasets.

One notable observation is that the performance differences among heap types are almost indistinguishable across smaller graphs, while on larger graphs such as USA-road-d.USA.gr, the STLVectorHeap lags slightly behind. The other four heap types (STLDequeHeap, LeftistHeap, FibonacciHeap, BinomialQueue) maintain competitive and consistent times, indicating their suitability for larger graph problems.

2.4.2.4 Heap Performance on Different Datasets

The last graph (Figure 4) shows the performance of the heap (measured by running time) when handling datasets with various sets. It is obvious that the growth of running time is linear, which indicates that the performance is stable with the increase of graph size.

Note that the difference between 5 heaps are relatively small. Even the slowest one (STL Vector Heap) is very close to other 4 heaps, making it hard to distinguish lines on graph. However, it is also a proof that all the 5 heaps are remarkable data structures for implementing Dijkstra's algorithm.

2.4.2.5 Conclusion

The experimental results suggest that while all heap structures are viable choices for Dijkstra's algorithm, specialized heaps such as the LeftistHeap, FibonacciHeap, and BinomialQueue demonstrate marginally better performance on larger datasets. However, the differences are minimal, and in many cases, the choice of heap may depend on factors such as ease of implementation and memory usage rather than pure performance.