Shortest Path Algorithm with Heaps

Group number: Jiefeng Wu Jiajun Qin Wenjie Huang

April 8, 2023

Abstract

This report explores the performance of various heap data structures in optimizing Dijkstra's shortest path algorithm. The importance of the problem is highlighted, as Dijkstra's algorithm is a fundamental algorithm in graph theory with widespread applications in fields such as transportation networks, computer networking, and social networks. The methods used in the report include a thorough analysis of the time and space complexity of the algorithm using different heap data structures, including binary heaps, leftist heaps, binomial heaps, and Fibonacci heaps. Fibonacci heaps offer the best theoretical time and space complexity for Dijkstra's algorithm, with a time complexity of $\mathrm{O}(E + V \log V)$ and a space complexity of $O(V)$. However, in practice, the high constant factors associated with the Fibonacci heap operations may impact its performance. Therefore, it is important to consider other factors such as the size and structure of the graph, other heaps like leftist heap, binomial heap can also be taken consideration into, too. The implications and significance of this project are discussed, including how this information can be applied in real-world scenarios to improve the efficiency and accuracy of graph-based computations. Overall, this report provides valuable insights for researchers and practitioners who are looking to optimize Dijkstra's algorithm using the most efficient heap data structures.

1 Introduction

Shortest path problems are classic and common problems in graph theory. One of the most important algorithms for the shortest path problems is Dijkstra, which we can optimize by min-priority queue. Therefore, the goal of the project is to find the best data structure for the Dijkstra's algorithm.

2 Data Structure / Algorithm Specification

2.1 Data Structures For Graph Representation

As we all know, the comman data structures for graph representation are adjacency list, adjacency matrix and incidence matrix and so on.

Concering the size of nodes(can be at most 23,947,347) and the size of edges(at most 58,333,344), we choose adjacency list[4] as our data structure for graph representation, with the space complexity of $O(|V|+|E|)$, which is good for a sparse graph.

For convenience, we use the STL `vector` to achieve adjacency list, which is a container in C++. Every node v has a `vector` which stores the adjacency node, indicating there is an edge starting from v. If the edge has values, we can set `struct` as the elements of `vector`, with the attributes of nodes and the edge values.

2.2 The Algorithm For the Shortest Path Problems – Dijkstra

Dijkstra's algorithm[1] is an algorithm for finding the shortest paths between nodes in a weighted graph. It is usually used for solving single-source problems.

2.2.1 Algorithm Procedure

1. Mark all nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
2. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. During the run of the algorithm, the tentative distance of a node v is the length of the shortest path discovered so far between the node v and the starting node.
3. For the current node, consider all of its unvisited neighbors and calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the one currently assigned to the neighbor and assign it the smaller one.
4. When we are done considering all of the unvisited neighbors of the current node, mark the current node as visited and remove it from the unvisited set.
5. If the destination node has been marked visited or if the smallest tentative distance among the nodes in the unvisited set is infinity, then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new current node, and go back to step 3.

The algorithm can be described by the pseudo-code below.

```
function Dijkstra(Graph, source)
    for each vertex v in Graph.Vertices do
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    end for
    dist[source] ← 0
    while Q is not empty do
        u ← in Q with min dist[u]
        remove u from Q
        for each neighbor v of u in Q do
            alt ← dist[u] + Graph.Edges(u, v)
            if alt < dist[v] then
                dist[v] ← alt
                prev[v] ← u
            end if
        end for
    end while
    return dist[ ], prev[ ]
end function
```

2.2.2 Heap Optimization

A min-priority queue is an abstract data type that provides 3 basic operations: `add_with_priority()`, `decrease_priority()` and `extract_min()`. With the help of the min-priority queue, we can find the node u with the minimum $dist[u]$ just in $O(\log N)$, which can reduce the time complexity of Dijkstra.

And the algorithm with the heap ptimization can be described by the pseudo-code below.

```
function Dijkstra(Graph, source)
    for each vertex v in Graph.Vertices do
        if v ≠ source then
            dist[v] ← INFINITY
            prev[v] ← UNDEFINED
        end if
    end for
    Q.add_with_priority(v, dist[v])
    dist[source] ← 0
    while Q is not empty do
        u ← Q.extract_min()
        for each neighbor v of u in Q do
            alt ← dist[u] + Graph.Edges(u, v)
            if alt < dist[v] then
                dist[v] ← alt
                prev[v] ← u
                Q.add_with_priority(v, dist[v])
            end if
        end for
    end while
    return dist[ ], prev[ ]
end function
```

2.3 Heap

As mentioned above, we can use heaps to optimize Dijkstra algorithm. However, there are many types of heaps, like leftist heap, binomial queue and Fibonacci heap and so on. Not all of them are suitable for this scene since they disgree about performance of some functions.

Operation	FindMin	DeleteMin	Insert	DecreaseKey
Binary	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$
Binomial	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$
Fibonacci	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$

Table 1: time complexities of various heap data structures

In our implementation of the project, we use the binary heap, binomial heap, leftist heap and the Fibonacci heap. Here are the brief introductions.

2.3.1 Binary Heap

A binary heap[5] is defined as a binary tree with two additional constraints:

- Shape property: a binary heap is a complete binary tree. Except the deepest level, all internal nodes are fully filled, and the nodes of the deepest level are filled from left to right.
- Heap property: the key stored in each node is either greater than or equal to (max-priority) or less than or equal to (min-priority) the keys in the node's children, according to some total order.

Since the implementation and the analysis have been covered in the course "Fundamental of Data Structures", the report will skip it. And the complexity of the comman operations has been listed in the table above. For convenience, we use the STL `priority_queue` to achieve binary heap, which is a container in C++ too.

2.3.2 Leftist Heap

The leftist heap[6] property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.

And its common operations are `LEFTIST-HEAP-MERGE`.

- `Merge two leftist heaps`

```
function BINOMIAL-HEAP-MERGE(H_1, H_2)
    if H_1 == NIL then
        return H_2
    end if
    if H_2 == NIL then
        return H_1
    end if
    if H_1.key > H_2.key then
        return BINOMIAL-HEAP-MERGE(H_2, H_1)
    end if
    H_1.right ← BINOMIAL-HEAP-MERGE(H_1.right, H_2)
    if H_1.left == NIL then
        SWAP(H_1.left, H_1.right)
        H_1.s_value ← 1
        return H_1
    end if
    if H_1.right.s_value > H_1.left.s_value then
        SWAP(H_1.left, H_1.right)
    end if
    H_1.s_value ← H_1.right.s_value + 1
    return H_1
end function
```

- **Deleting the minimum key or Inserting a new key** Other operations are based on `LEFTIST-HEAP-MERGE`. The minimum key for the heap is the value stored in the root. If we want to perform `DeleteMin`, just delete it and merge the subtrees.

  If we want to perform `Insert`, take the new key as a new leftist heap with a single node, then merge them.

2.3.3 Binomial Heap

A binomial heap[2] is implemented as a set of binomial trees (compare with a binary heap, which has a shape of a single binary tree), which are defined recursively as follows

- A binomial tree of order 0 is a single node
- A binomial tree of order k has a root node whose children are roots of binomial trees of orders $k-1, k-2, \ldots, 1, 0$.

And its comman operations are `BINOMIAL-HEAP-MINIMUM`, `BINOMIAL-HEAP-UNION`, and `BINOMIAL-HEAP-EXTRACT-MIN`

- **Finding the minimum key**
  The minimum key is in one of the roots.

```
function Binomial-Heap-Minimum(H)
    y ← NIL
    x ← head[H]
    min ← +∞
    while x ≠ NIL do
        if key[x] < min then
            min ← key[x]
            y ← x
        end if
        x ← sibling[x]
    end while
    return y
end function
```

- **Uniting two binomial heaps**

```
function Binomial-Heap-Union(H1, H2)
    H ← Make-Binomial-Heap
    head[H] ← Binomial-Heap-Merge(H1, H2)
    free the object H1 and H2 but not the lists they point to
    if head[H] == NIL then return H
    end if
    prev − x ← NIL
    x ← head[H]
    next − x ← sibling[x]
    while next − x ≠ NIL do
        if (degree[x] ≠ degree[next − x]) or (sibling[next − x] ≠ NIL and degree[sibling[next − x]] ==
degree[x]) then
            prev − x ← x
            x ← next − x
        else
            if key[x] ≤ key[next − x] then
                sibling[x] ← sibling[next − x]
                Binomial-Link(next − x, x)
            else
                if prev − x == NIL then
                    head[H] ← next − x
                elsesibling[next − x] ← next − x
                    Binomial-Link(x, next − x)
                end if
            end if
        end if
        next − x ← sibling[x]
    end while
    return H
end function
```

- **Inserting a node**
  `Insert` can be implemented as a special case of `Merge`. Just take the node as a binomial queue B_0, then merge it with the current heap$B$.

- **Extracting the node with minimum key**
  Find root x with min key in root list of H, and delete. Then the rest part H' and H are two binomial heap, we just need to merge them.

```
function Binomial-Heap-Extract-Min(H)
```

```
    find the root x with the minimum key in the root list of H, and remove x from the root list of H
    H' ← MAKE-BINOMIAL-HEAP
    reverse the order of the linked list of x's children, and set head[H'] to point to the head of the resulting list.
    H ← BINOMIAL-HEAP-UNION(H, H')
    return x
end function
```

2.3.4 Fibonacci Heap

A Fibonacci heap[3] is a collection of rooted trees that are min-heap ordered.

- **Creating a new Fibonacci heap**

  To make an empty Fibonacci heap, the `MAKE-FIB-HEAP` procedure allocates and returns the Fibonacci heap object H,where $H.n = 0$ and $H.min =$ NIL;there are no trees in H.

- **Inserting a node**

  The following procedure inserts node x into Fibonacci heap H, assuming that the node has already been allocated and that $x.key$ has already been filled in.

```
function FIB-HEAP-INSERT(H,x)
    x.degree ← 0
    x.p ← NIL
    x.child ← NIL
    x.mark ← FALSE
    if H.min == NIL then
        Create a root list for H containing just x
        x.min ← x
    else
        Insert x into H's root list
        if x.key < H.min.key then H.min ← x
        end if
    end if
    H.n ← H.n + 1
end function
```

- **Finding the minimum node**

  The minimum node of a Fibonacci heap H is given by the pointer $H.min$,sowe can find the minimum node in $O(1)$ actual time.

- **Uniting two Fibonacci heaps**

  It simply concatenates the root lists of H_1 and H_2 and then determines the new minimum node. Afterward, the objects representing H_1 and H_2 will never be used again.

```
function FIB-HEAP-UNION(H)
    z ← H.min
    if thenz ≠ NIL
        for each child x of z do
            Add x to the root list of H
            x.p ← NIL
        end for
        Remove z from the root list of H
        if z == z.right then
            H.min == NIL
```

```
        else
            H.min == z.right
            CONSOLIDATE(H)
        end if
        H.n ← H.n − 1
    end if
    return z
end function
```

- **Extracting the minimum node**

  `FIB-HEAP-EXTRACT-MIN` works by first making a root out of each of the minimum node's children and removing the minimum node from the root list. It then consolidates the root list by linking roots of equal degree until at most one root remains of each degree.

```
function FIB-HEAP-EXTRACT-MIN(H1,H2)
    H ← MAKE-FIB-HEAP()
    H.min ← H1.min
    Concatenate the root list of H2 with the root list of H
    if H1.min == NIL or (H2.min ≠ NIL and H2.min.key < H1.min.key) then
        H.min ← H2.min
    end if
    H.n ← H1.n + H2.n return H
end function
```

The next step, in which we reduce the number of trees in the Fibonacci heap, is consolidating the root list of H, which the call `CONSOLIDATE(H)` accomplishes.

```
procedure CONSOLIDATE(H)
    Let A[0...D(H.n)] be a new array
    for i = 0 to D(H.n) do
        A[i] ← NIL
    end for
    for each node w in the root list of H do
        x ← w
        d ← x.degree
        while A[d] ≠ NIL do
            y ← A[d]                    ▷ another node with the same degree as x
            if x.key > y.key then
                Exchange x with y
            end ifFIB-HEAP-LINK(H, y, x)
            A[d] ← NIL
            d ← d + 1
        end while
        A[d] ← x
    end for
    H.min ← NIL
    for i = 0 to D(H.n) do
        if A[i] ≠ NIL then
            if H.min == NIL then
                Create a root list for H containing just A[i]
                H.min ← A[i]
            else
                Insert A[i] into H's root list
                if A[i].key < H.min.key then
                    H.min ← A[i]
```

```
                end if
            end if
        end if
    end for
end procedure
procedure FIB-HEAP-LINK(H, y, x)
    Remove y from the root list of H
    Make y a child of x, incrementing x.degree
    y.mark ← FALSE
end procedure
```

- **DecreaseKey**

  In the following pseudocode for the operation `FIB-HEAP-DECREASE-KEY`, we assume as before that removing a node from a linked list does not change any of the structural attributes in the removed node.

  If min-heap order has been violated, many changes may occur. The `CUT` procedure "cuts" the link between x and its parent y, making x a root.

  We are not yet done, because x might be the second child cut from its parent y since the time that y was linked to another node. The `CASCADING-CUT` procedure recurses its way up the tree until it finds either a root or an unmarked node.

```
function FIB-HEAP-DECREASE-KEY(H, x, k)
    if k > x.key then
        Error("new key is greater than current key")s
    end if
    x.key ← k
    y ← x.p
    if y ≠ NIL and x.key < y.key then
        CUT(H, x, y)
        CASCADING-CUT(H, y)
    end if
    if x.key < H.min.key then
        H.min ← x
    end if
end function
procedure CUT(H, x, y)
    Remove x from the child list of y, decrementing y.degree
    Add x to the root list of H
    x.p ← NIL
    x.mark ← FALSE
end procedure
procedure CASCADING-CUT(H, y)
    if z ≠ NIL then
        if y.mark == FALSE then
            y.mark ← TRUE
        else
            CUT(H, y, z)
            CASCADING-CUT(H, z)
        end if
    end if
end procedure
```

3 Testing Results

3.1 Test the Running Time Versus Input Sizes

First, we use the graph generated by ourselves, where every node has 3 edges starting from it. The target for this test is to see the performance of Dijkstra algorithm with heap optimization.

It's worth noting that the test environment is Windows 10, Lenovo Yoga 14s. And the results are shown in the tables and diagrams below.

Query Times	Size	Time(total)	Time(per)	Operations(total)[1]	Operations(per)
2000	1000	$0.41s$	$0.205ms$	$7,998,000$	$3,999$
	10000	$7.039s$	$3.5195ms$	$7,998,000$	$3,999$
	25000	$15.064s$	$7.532ms$	$199,998,000$	$9,999$
	50000	$23.625s$	$10.812ms$	$399,996,000$	$199,998$

[1] Here we record all operations about heaps. The target is to avoid an inapproriate comparsion between different heaps. For example, the operations for heap A is far less than B, and the time of A is also shorter. However, we can not say heap A is better than B since their operations have obvious difference.

Table 2: Performance of Dijkstra with Binary Heap

Query Times	Size	Time(total)	Time(per)	Operations(total)	Operations(per)
2000	1000	$1.962s$	$0.981ms$	$6,665,340$	$3,332$
	10000	$235.183s$	$117.592ms$	$6,665,334$	$3,333$
	25000	$1634.627s$	$817.313ms$	$166,665,334$	$83,332$
	50000	$5652.095s$	$2.826s$	$399,996,000$	$199,998$

Table 3: Performance of Dijkstra with Fibonacci Heap

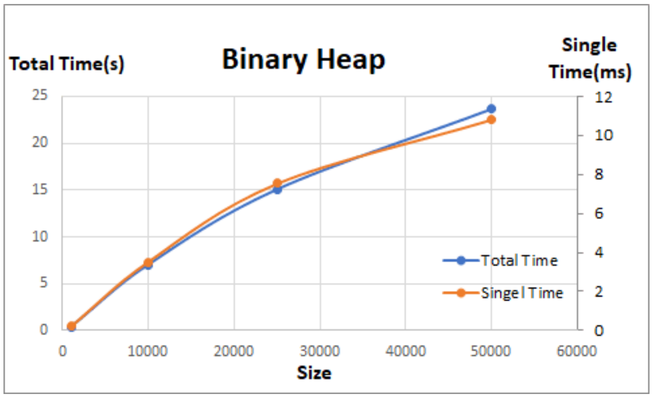
Query Times	Size	Time(total)	Time(per)	Operations(total)	Operations(per)
2000	1000	$1.493s$	$0.747ms$	$7,002,000$	$3,501$
	10000	$15.463s$	$7.732ms$	$7,002,000$	3.501
	25000	$31.770s$	$15.885ms$	$175,002,000$	$87,501$
	50000	$108.682s$	$54.341ms$	$349,998,000$	$167,999$

Table 4: Performance of Dijkstra with Binomial Heap

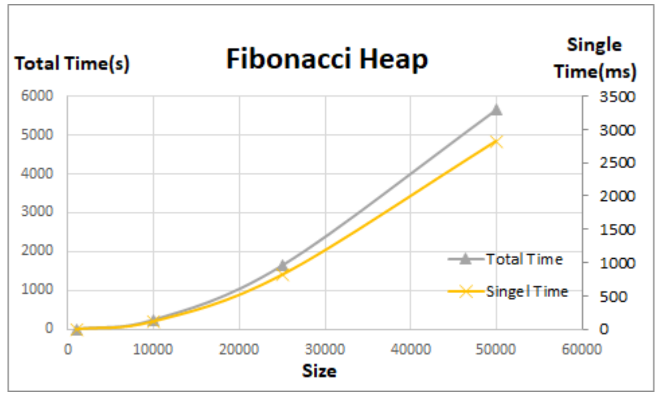
Query Times	Size	Time(total)	Time(per)	Operations(total)	Operations(per)
2000	1000	$0.250s$	$0.125ms$	$6,666,000$	$3,333$
	10000	$2.533s$	$1.267ms$	$6,666,000$	$3,333$
	25000	$7.599s$	$3.800ms$	$166,668,000$	83.334
	50000	$16.765s$	$8.383ms$	$333,336,000$	$166,668$

Table 5: Performance of Dijkstra with Leftist Heap

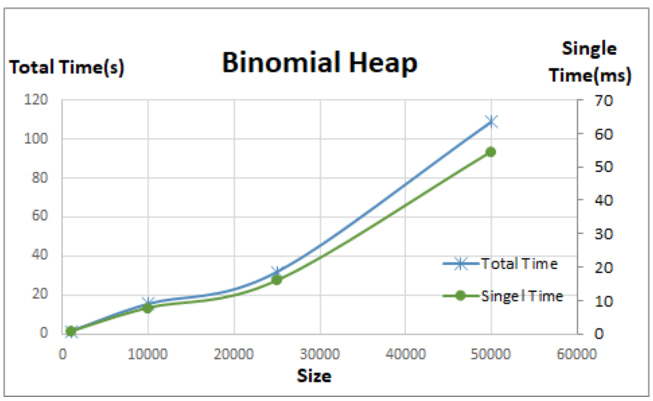
Based on the data above, we can plot the diagrams of the running time versus input sizes.



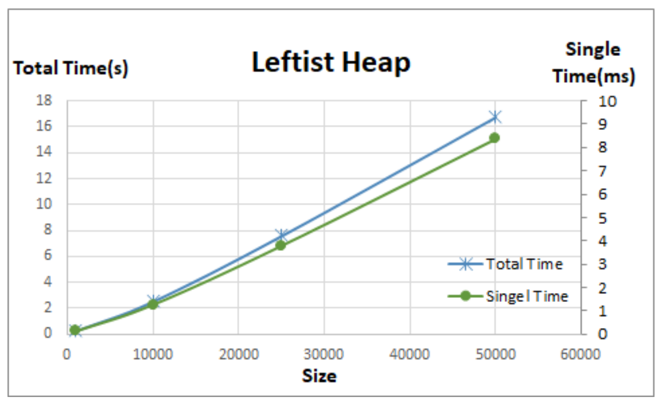
(a) Binary



(b) Fibonacci



(c) Binomial



(d) Leftist

Figure 1: Diagram for the Running Time Versus Input Sizes

Obviously, with input size increasing, the time increases too. And among our four heap optimization, leftist heap is the best in this test.

3.2 Test the USA Road Networks

This test data is given on the website.

It's worth noting that the time shown below in a unit of milliseconds and the test environment is MacbookPro 14 m1 pro.

name	used time long	used times
Fibonacci_heap	524793682	73258335
prior_queue	17622015	76977720
binomial_heap	19056702	76977678
Leftist_Heap	12128688	76977801

Table 6: USA(23,947,347 nodes and 58,333,344 arcs)

name	used time long	used times
Fibonacci_heap	1041877	1331915
prior_queue	261823	1398261
binomial_heap	282846	1398270
Leftist_Heap	149239	1398279

Table 7: COL(435,666 nodes and 1,057,066 arcs)

name	used time long	used times
Fibonacci_heap	4724075	3288061
prior_queue	669613	3517677
binomial_heap	740225	3517671
Leftist_Heap	395829	3517677

Table 8: FLA(1,070,376 nodes and 2,712,798 arcs)

name	used time long	used times
Fibonacci_heap	35547024	11010549
prior_queue	2400380	11561061
binomial_heap	2635003	11561031
Leftist_Heap	1544018	11561007

Table 9: E(3,598,623 nodes and 3,598,623 arcs)

name	used time long	used times
Fibonacci_heap	14751139	5794859
prior_queue	1198659	6123609
binomial_heap	1307724	6123600
Leftist_Heap	705394	6123615

Table 10: CAL(1,890,815 nodes and 1,890,815 arcs)

name	used time long	used times
Fibonacci_heap	19425865	8461248
prior_queue	1740881	8970867
binomial_heap	1872000	8970810
Leftist_Heap	1029884	8970831

Table 11: LKS(2,758,119 nodes and 6,885,658 arcs)

name	used time long	used times
Fibonacci_heap	15312337	4687273
prior_queue	1041698	4988169
binomial_heap	1157084	4988196
Leftist_Heap	665619	4988190

Table 12: NE(1,524,453 nodes and 1,524,453 arcs)

name	used time long	used times
Fibonacci_heap	4794502	3680666
prior_queue	763564	3829974
binomial_heap	820310	3829971
Leftist_Heap	446185	3829923

Table 13: NW(1,524,453 nodes and 1,524,453 arcs)

name	used time long	used times
Fibonacci_heap	66010751	19155944
prior_queue	4353301	20149614
binomial_heap	4839815	20149572
Leftist_Heap	2847256	20149647

Table 14: W(6,262,104 nodes and 6,262,104 arcs)

name	used time long	used times
Fibonacci_heap	363634589	43077157
prior_queue	11800386	45266625
binomial_heap	11803926	45266574
Leftist_Heap	7505323	45266646

Table 15: CTR(14,081,816 nodes and 14,081,816 arcs)

And we plot all situations into one diagram. (Note that since the time of Fibonacci is far greater than the other three heaps, we use a different axis for Fibonacci heap.)

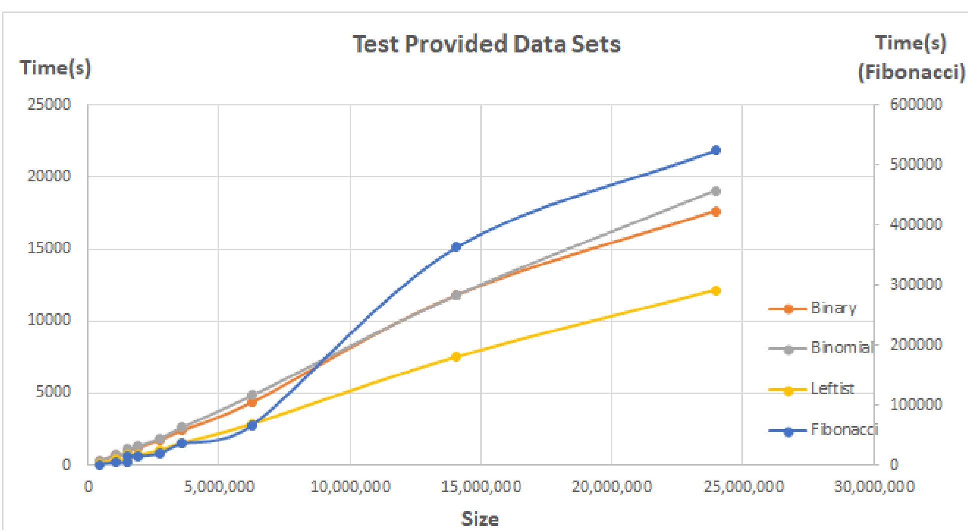


Figure 2: Test Provided Data Sets

From the diagram, we can draw similar conclusion as 3.1, which is that the leftist heap has the best performance while Fibonacci heap is worse than other heaps.

4 Analysis and Comments

4.1 Time Complexity

- **For Dijkstra's algorithm with a binary heap, a binomial heap or a leftist heap**

  Time complexity of $O((E+V)\log V)$, where E is the number of edges and V is the number of vertices in the graph.

  In this case, the algorithm maintains a heap of vertices by their distance from the source vertex. The insertion of a new vertex requires $O(\log V)$ time complexity in the worst case, while the decrease key operation and the extraction of the minimum vertex require $O(\log V)$ time complexity each. Since these operations are performed a maximum of $E+V$ times, the total time complexity is $O((E+V)\log V)$.

- **For Dijkstra's algorithm with a Fibonacci heap**

  Time complexity of $O(E+V\log V)$, where E is the number of edges and V is the number of vertices in the graph.

  To take advantage of the decrease key operation in Dijkstra's algorithm, you need to maintain a reference to each vertex in the heap. The decrease key operation in Fibonacci heap has an amortized time complexity of $O(1)$, which means that it is a constant-time operation on average.

  In this case, the algorithm maintains a Fibonacci heap of vertices by their distance from the source vertex. The insertion of a new vertex and the decrease key operation both have an amortized time complexity of $O(1)$, while the extraction of the minimum vertex has an amortized time complexity of $O(\log V)$. Since these operations are performed a maximum of $E+V$ times, the total time complexity is $O(E+V\log V)$. In detail, building the heap is $O(V)$, extracting minimum vertex is $O(V\log V)$, updating the distances is $O(E)$, so the result is $O(E+V\log V)$.

  Besides, although STL

- **Summary**

  As we can see, in theory, Fibonacci heap has a better time complexity than binary heap for some operations, including decrease key and merge, especially for sparse graphs with many unreachable vertices.

  However, in practice, Fibonacci heap may not always be the best choice for Dijkstra's algorithm due to its higher constant factor and more complex implementation compared to binary heap, which can explain why in our test the performance of Fibonacci heap is much worse than the binary heap.

  The choice between Fibonacci heap and binary heap should depend on the specific characteristics of the graph and the performance requirements of the application.

4.2 Space Complexity

In Dijkstra with heap optimization, space complexity is $O(V+E)$, where E is the number of edges and V is the number of vertices in the graph.

In this case, the algorithm maintains a heap of vertices by their distance from the source vertex. The heap requires space proportional to the number of vertices, which is $O(V)$. Additionally, the algorithm maintains an adjacency list to store the edges and their weights. The space required for the adjacency list or matrix is $O(E)$. Besides, for Fibonacci heap, we need an additional array to record the position stored in the heap of each node, so that we can call `DecreaseKey` without searching for the position, which also cost $O(V)$. Therefore, the total space complexity is $O(V+E)$.

5 Author list

The code and report are finished by all of us.

JieFeng Hu finish the code.
Jiajun Qin finish the report.
Wenjie Huang finish the PPT of the presentation.

Declaration

We hereby declare that all the work done in this project titled ”Shortest Path Algorithm with Heaps” is of our independent effort as a group.

6 Signatures

We hereby declare that all the work done in this project titled ”Shortest Path Algorithm with Heaps” is of our independent effort as a group.

黄文杰

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms, third edition. In *Single-Source Shortest Paths*, chapter 24. The MIT Press, 3rd edition, 2009.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms, third edition. In *Binomial Heaps*, chapter 19. The MIT Press, 3rd edition, 2009.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms, third edition. In *Fibonacci Heaps*, chapter 20. The MIT Press, 3rd edition, 2009.

[4] G. R. . A. L. in C++. *Graph Representation – Adjacency List in C++*.

[5] M. A. Weiss. Data structures and algorithm analysis in c (2nd ed.). In *Priority Queues(Heaps)*, chapter 6. Addison-Wesley Longman Publishing Co., Inc., USA, 1996.

[6] M. A. Weiss. Data structures and algorithm analysis in c (2nd ed.). In *Amortized Analysis*, chapter 11. Addison-Wesley Longman Publishing Co., Inc., USA, 1996.

A Source Code (if required)

```
#include <iostream>
#include <fstream>
#include <vector>
#include <queue>
#include <time.h>
#include <random>
using namespace std;
long queue_use;
long time_second;
const int INF = 0x3f3f3f3f;
//2^30
const long CAPITAL = 1073741824;

//The definition of point, as concise as possible, name is the position of node
//dis is the distance from the source point
struct node{
    int name;
    int dis;
    node(int n=-1,int d=INF){name=n;dis=d;}
    //Need to overload the comparison symbol
    //It was originally a maximum heap, so it is a minimum heap
    bool operator <( const node &r )const{
        return dis < r.dis ;
    }
bool operator ==( const node &r )const{
        return dis == r.dis;
    }
    bool operator !=( const node &r )const{
        return dis != r.dis;
    }
};

//The definition of point, as concise as possible, name is the position of node
//dis is the distance from the source point
//pri is specially resized, turning the largest heap into the smallest heap
struct node_pri{
```

```
    int name;
    int dis;
    node_pri(int n=-1,int d=INF){name=n;dis=d;}
    //Need to overload the comparison symbol
    //It was originally a maximum heap, so it is a minimum heap
    bool operator <( const node_pri &r )const{
        return dis > r.dis ;
    }
    bool operator ==( const node_pri &r )const{
return dis == r.dis;
    }
    bool operator !=( const node_pri &r )const{
        return dis != r.dis;
    }
};

//Edge definition, as concise as possible, only destination and distance
struct edge {
    int mute;
    int distance;
    edge(int e,int d){mute=e;distance=d;}
};

/*Fibonacci Heap Section*/
struct FibNode{
        //The content is key, the format is node
        node key;
        //Double linked list pointers of the left and right siblings
        FibNode*rbro;
        FibNode*lbro;
        //Parent and child pointers
FibNode*father;
        FibNode*child;
        //degree of node
        int degree;
        //Did you lose a child
        bool mark;
        void Init();
};

//initialize the point
void FibNode::Init(){
    //The left and right brothers point to themselves
    rbro = lbro = this;
    //Both parent and child are empty
    father = child = NULL;
    //the degree of the node is 0
    degree = 0;
    //makr is also 0
    mark = false;
}

class FibHeap{
    public:
        FibHeap(){min=NULL;number=0;}
        FibNode* push(node a);
        node pop();
        bool empty();
void decrease( FibNode*x, int k );
```

```
    private:
        //point to the smallest node
        FibNode*min;
        //The number of all points in the tree
        int number;
        void cut( FibNode*x, FibNode*y );
        void cascading_cut( FibNode*y );
        void consolidate();
};

FibNode*cat_pointer( FibNode*a, FibNode*b){
    //if one of them is empty
    //directly return
    if( a == NULL ) return b;
    if( b == NULL ) return a;
    //define a temporary pointer
    FibNode*temp;
    //connect two doubly linked lists
    temp = a->rbro;
    a->rbro = b->rbro;
b->rbro->lbro = a;
    b->rbro = temp;
    temp->lbro = b;
    return a;
}

//insert operation
FibNode*FibHeap::push( node a ){
    //New request for a space to store
    FibNode*x = (FibNode*)malloc(sizeof(FibNode));
    //initialize and assign
    x->Init();
    x->key = a;
    //Insert x into the forest
    //If it is empty, insert it directly
    if( min == NULL ) min = x;
    //Otherwise, insert into the linked list
    else {
        //Insert into root's doubly linked list
        cat_pointer( min , x );
        //update min
        if( x->key < min->key ) min = x;
    }
    //update number
    number ++;
    //return pointer for easy lookup
return x;
}

//Remove the point from the doubly linked list
void delete_pointer( FibNode*a ){
    //Interconnection of left and right sub-chains
    a->lbro->rbro = a->rbro;
    a->rbro->lbro = a->lbro;
    //The left and right brothers connect themselves
    a->rbro = a->lbro = a;
    a->father = NULL;
}
```

```
//Link one point to another's sublist
void heap_link( FibNode*child, FibNode*father ){
    //child is out of the original linked list
    delete_pointer( child );
    //Include child in farther's sublist
    father->child = cat_pointer( father->child, child );
    //Update child's father and father's degree
    father->degree ++;
    child->father = father;
return;
}

//Adjust the properties of the heap
void FibHeap::consolidate(){
    //No need to adjust if min is null
    if( min == NULL ) return ;
    //Create a new heap to buffer
    FibNode*A[number];
    //clean up memory
    for( int i=0; i<number; i++ ) A[i] = NULL;
    //Create a temporary pointer
    FibNode*now_pointer = min;
    //traverse all root doubly linked lists
    do {
        //Create a temporary variable
        FibNode*x = now_pointer;
        int degree = x->degree;
        //Start to find the appropriate area in A
        while( A[degree] != NULL ){
            //Create temporary changes
            FibNode*y = A[degree];
//Determine whose value is smaller to be the parent
            if( y->key < x->key ){
                //exchange
                FibNode*temp = x;
                x = y;
                y = temp;
            }
            //link two
            heap_link( y, x );
            //Clean up the original
            A[degree] = NULL;
            //update degree
            degree++;
        }
        //Found and put in
        A[degree] = x;
    }while( now_pointer != min );
    //erase the original forest
    min = NULL;
    //rescale the forest
    for( int i=0; i<number; i++ ){
if( A[i] != NULL ){
            //If min doesn't exist yet, use it as min
            if( min == NULL ) min = A[i];
            //Otherwise, join min's doubly linked list
            else {
                cat_pointer( min, A[i] );
                //update min
```

```
                if( A[i]->key < min->key ) min = A[i];
            }
        }
    }
    return;
}

//Pop the smallest point
node FibHeap::pop(){
    //first separate min
    FibNode*z = min;
    //if the heap is not empty
    if( z != NULL ){
        //Transfer all children of z to root
        while( z->degree != 0 ){
//mark the child to be detached
            FibNode*child = z->child;
            //The child's pointer points to null or the right brother
            if( z->degree == 1 ) z->child = NULL;
            else z->child = z->child->rbro;
            //sub out
            delete_pointer( child );
            //update the degree of z
            z->degree --;
            //Link the child to the main chain
            cat_pointer( min, child );
        }
        //first judge min
        if( z->rbro == z ) min=NULL;
        else min = z->rbro;
        //remove z
delete_pointer( z );
        //Adjustment
        consolidate();
        //update number
        number --;
    }
    return z->key;
}

//Auxiliary function, extract x to the top, adjust the minimum heap properties
void FibHeap::cut( FibNode*x, FibNode*y ){
    //Determine whether the child node of y has only one x or other
    if( y->degree == 1 ) y->child = NULL;
    else y->child = x->rbro;
    //Update the degree of y
    y->degree --;
    //extract x from y
    delete_pointer( x );
    //Add x to the main linked list
    cat_pointer( min , x );
    x->mark = false;
}

//Auxiliary function, adjust the minimum heap properties
void FibHeap::cascading_cut( FibNode*y ){
    //Create a temporary variable
    FibNode*z = y->father;
    //if y is not in the main linked list
```

```
    if( z != NULL ){
        //If the mark of y is
        if( y->mark == false ) y->mark = true;//adjusted
        else {
            //recursively do cut
            cut( y , z );
            cascading_cut( z );
        }
    }
}

//Decrement the value of the specified pointer
void FibHeap::decrease( FibNode*x, int k ){
    //can only reduce
    if( k > x->key.dis ) return;
    //renew
    x->key.dis = k;
    //Create a temporary variable
    FibNode*y = x->father;
//If x is not the top level, and x is smaller than y
    if( y!= NULL && x->key < y->key ){
        //cut and cascading-cut two operations
        cut( x, y );
        cascading_cut( y );
    }
    //If the value of x is smaller than min, replace min
    if( x->key < min->key ) min = x;
}

bool FibHeap::empty(){
    return number==0;
}
/*FIBONACCI HEAP PART END*/

/*Binomial Heap Section*/
//point of binary heap
struct binonode{
    //The key of the node is node_pri, because the heap is the largest heap
    node key;
    //parent node, left and right child nodes
    binonode*nsibil;
    binonode*lchild;
};

//binary heap
class binoheap{
    public:
        binoheap(){number=0;for(int i=0;i<30;i++) forest[i]=NULL;}
        node pop();
        bool empty();
        void push( node in );
        void combineForest( binoheap H2 );
        binonode*combineTree( binonode*T1, binonode*T2 );
    private:
        //number of heaps
        int number;
        //array of forests
binonode*forest[30];
};
```

```

//merge two trees
binonode*binoheap::combineTree( binonode*T1, binonode*T2 ){
    //The one with the smaller key is the father
    if( T2->key < T1->key ) return combineTree( T2, T1 );
    //Make T2 a child of T1
    //T2 becomes the leftmost son of T1
    T2->nsibil = T1->lchild;
    T1->lchild = T2;
    //return parent
    return T1;
}

//Merge two forests, one of which is this class forest
void binoheap::combineForest( binoheap H2 ){
    //define temporary variable
    binonode*T1;
    binonode*T2;
    binonode*Carry=NULL;
    //throw an error if out of bounds
if( number + H2. number > CAPITAL ) exit(1);
    //Prepare to transfer everything to T1
    number += H2.number;
    //The maximum upper limit is lognumber
    int i,j;
    for( i=0,j=1; j<=number; i++,j*=2 ){
        //Intercept the current tree of T1 and T2
        T1 = forest[i];T2 = H2.forest[i];
        //Man-made three-bit binary judgment is 0
        switch ( 4*(!!Carry) + 2*(!!T2) + (!!T1) )
        {
        //All 0s do not need any operation, because there is nothing in all three
        case 0:break;
        //001 Only T1 is not NULL, just skip it
        case 1: break;
        //010 is only available in T2, it needs to be transferred to T1
case 2:forest[i]=T2;H2.forest[i]=NULL;break;
        //011 Both T1 and T2 have to be integrated and go to the next layer
        case 3:Carry=combineTree( T1, T2 );forest[i]=H2.forest[i]=NULL;break;
        //100 is only available for carry, drive here
        case 4:forest[i]=Carry;Carry=NULL;break;
        //101, only Carry and T1 have it, merged to the next layer
        case 5:Carry=combineTree(Carry,T1);forest[i]=NULL;break;
        //110, only Carry and T2 have it, merged to the next layer
        case 6:Carry=combineTree(Carry,T2);H2.forest[i]=NULL;break;
        //111 All three have Carry to the forest, T1T2 merges into the next layer
case 7:forest[i]=Carry;Carry=combineTree(T1,T2);H2.forest[i]=NULL;break;
        }
    }
    return;
}

//insert a node
void binoheap::push( node in ){
    //If binoheap is empty, just add it to 1
    //create new point
    binonode*newNode = (binonode*)malloc(sizeof(binonode));
    //initialization
    newNode->lchild = NULL;newNode->nsibil = NULL;
```

```
    //assignment
    newNode->key = in;
    //If the forest is empty, it can be directly added to 0
    if( number == 0 ){
        forest[0] = newNode;
        number ++;
return;
    }
    else {
        //Create a forest fusion with only this point
        binoheap temp;
        temp.number=1;
        temp.forest[0] = newNode;
        combineForest( temp );
    }
    //Finish
    return;
}

//Push out the minimum point
node binoheap::pop(){
    //Create a new heap to host child nodes
    binoheap Delete;
    //Create a temporary variable
    binonode*DeleteTree;
    binonode*OldRoot;
    //the minimum value to be pushed out
    node MinItem;
    //If it is empty, report an error directly
    if( number == 0 ) exit(1);
    //define temporary variable
    int i,j,MinTree;
    //Traverse all roots to find the root point of the tree where the smallest value is located
for( int i=0; i<30; i++ ){
        //This point is not NULL, but smaller than MinItrm
        if( forest[i] != NULL && forest[i]->key < MinItem ){
            //MinItem is this, update
            MinItem = forest[i]->key;
            MinTree = i;
        }
    }
    //Determine the point to delete
    DeleteTree = forest[MinTree];
    //delete in the forest
    forest[MinTree] = NULL;
    //ready to transfer
    OldRoot = DeleteTree;
    DeleteTree = DeleteTree->lchild;
    //liberate the original node
    free(OldRoot);
    //The number of points is the power of 2 minus the deleted
    Delete.number =(1 << MinTree)-1;
//start adding value
    for( j=MinTree-1; j>=0; j--){
        Delete.forest[j] = DeleteTree;
        DeleteTree = DeleteTree->nsibil;
        Delete.forest[j]->nsibil = NULL;
    }
```

```
    //Fusion
    //first update the original value
    number -= ( Delete.number + 1 );
    //reintegrate
    combineForest( Delete );
    //return minimum value
    return MinItem;
}

//test if there is content
bool binoheap::empty(){
    return number==0;
}

/*Binomial Heap Section Ends*/
/*Left tilted pile section*/
//Point structure of Lefist heap node
struct LeftistNode{
    node key;
    //Npl attribute, used to adjust the nature of the heap
    int Npl;
    //left and right children
    LeftistNode*lchild;
    LeftistNode*rchild;
};

//Leftist heap
class LefistTree{
    public:
        LefistTree(){tree=NULL;number=0;}
        node pop();
        void push( node in );
        bool empty();
        LeftistNode*Merge( LeftistNode*H1, LeftistNode*H2 );
private:
        LeftistNode*Merge_( LeftistNode*H1, LeftistNode*H2 );
        int number;
        LeftistNode*tree;
};

//Auxiliary function, the actual operation of merging two trees
LeftistNode*LefistTree::Merge_( LeftistNode*H1, LeftistNode*H2 ){
    //single Node, direct fusion
    if( H1->lchild == NULL ) H1->lchild = H2;
    //otherwise convert it
    else {
        //Fusion
        H1->rchild = Merge( H1->rchild, H2 );
        //transform the child
        if( H1->lchild->Npl < H1->rchild->Npl ){
            LeftistNode*temp = H1->lchild;
H1->lchild = H1->rchild;
            H1->rchild = temp;
        }
        //update Npl
        H1->Npl = H1->rchild->Npl + 1;
    }
    //return
    return H1;
```

```
}

//Merge the two trees
LeftistNode*LefistTree::Merge( LeftistNode*H1, LeftistNode*H2 ){
    //Judge whether fusion is needed, if one side is null and return directly, it is OK
    if( H1 == NULL ) return H2;
    if( H2 == NULL ) return H1;
    //determine how to link
    if( H1->key < H2->key ) return Merge_( H1, H2 );
    else return Merge_( H2, H1 );
}

node LefistTree::pop(){
    //find the smallest point
LeftistNode*Min = tree;
    //Create a temporary variable
    LeftistNode*Ltree = tree->lchild;
    LeftistNode*Rtree = tree->rchild;
    //Delete the tree and create a new tree
    tree = Merge( Ltree, Rtree );
    //return the smallest
    return Min->key;
}

//push the element into the heap
void LefistTree::push( node in ){
    //is to push the new one in as a new tree
    //create new point
    LeftistNode*newNode = (LeftistNode*)malloc(sizeof(LeftistNode));
    newNode->key = in;
    newNode->lchild = newNode->rchild = NULL;
    newNode->Npl = 1;
    //push the new point in
    tree = Merge( tree, newNode );
return;
}

//determine if it is empty
bool LefistTree::empty(){
    return tree==NULL;
}

/*Left Slope Pile Section End*/


//Whether vis dis and pre have been detected, the previous point is represented by the global, otherwise it cannot be read by queue
int*vis;
int*pre;
int*dis;
//nodeLib is used to store pointers to different points, easy to find in O(1)
FibNode**nodeLib;

//The map is stored in a vector
vector<edge>*map;

//initialization
void Init(int NodeNum){
    //Initialize vis and pre
```

```
    if( vis ) free(vis);
    if( pre ) free(pre);
    if( dis ) free(dis);
//Vis is initialized without vis
    vis = (int*)malloc(sizeof(int)*NodeNum);
    //The initialization of pre is -1. That is, there is no
    pre = (int*)malloc(sizeof(int)*NodeNum);
    //The initialization of dis is infinity
    dis = (int*)malloc(sizeof(int)*NodeNum);
    for(int i=0; i<NodeNum; i++){
        vis[i]=0;
        pre[i]=-1;
        dis[i]=INF;
    }
    return;
}

//Dijkstra algorithm implementation, returns the running time in double form
double Dijkstra_fib(int source, FibHeap q ) {
    //time, count
    clock_t start, end;
    start = clock();

    //Function implementation part
/*---------------------------------------------------------------------------------------------------------*/
    //The starting point is 0
    //push the first point
    q.push(node(source,0));
    queue_use++;
    //The map of the first point is set to 0
    dis[source] = 0;
    node Temp;
    //As long as it hasn't been traversed yet
    while(!q.empty()){
        //squeeze out the closest one
        Temp = q. pop();
        queue_use += 2;
        //Check to see if it has been traversed
        if( vis[Temp.name] ) continue;
        else vis[Temp.name] = 1;
        //Start to find the edge of the point one by one
        int num = map[Temp.name].size();
        //Find edges one by one
for(int i=0; i<num; i++ ){
            //This edge is the edge of the current operation
            edge tmp = map[Temp.name][i];
            //point of edge connection
            int mute = tmp.mute;
            //If the edge has also been traversed, it means that there must be no further optimization, just skip it
            if( vis[mute] ) continue;
            //edge distance
            int distance = tmp.distance;
            //If you go from this point, there can be optimization
            if( dis[mute] > dis[Temp.name]+ distance ){
 //If it is not in the heap, it is still INF
                if( dis[mute] == INF ){
                    //update distance
```

```
                    dis[mute] = dis[Temp.name] + distance;
                    //add to heap
                    //join the map
                    nodeLib[mute] = q.push(node(mute,dis[mute]));
                }
                //If in the heap, update the heap directly
                else {
                    //update distance
                    dis[mute] = dis[Temp.name] + distance;
                    //Find the pointer in the map and update
q.decrease( nodeLib[mute], dis[mute] );
                }

                queue_use++;
            }
        }
    }

/*----------------------------------------------------------------------------------------
    */
    end = clock();
    //cout << "begin " << start << " end " << end << endl;
    return (double) (end -start);
}

//Dijkstra algorithm implementation, returns the running time in double form
//T as a heap. There are several basic functions
double Dijkstra(int source, priority_queue<node_pri> q ) {
    //time, count
clock_t start, end;
    start = clock();

    //Function implementation part
/*----------------------------------------------------------------------------------------
    */
    //The starting point is 0
    //push the first point
    q.push(node_pri(source,0));
    queue_use++;
    //The map of the first point is set to 0
    dis[source] = 0;
    node_pri Temp;
    //As long as it hasn't been traversed yet
    while(!q.empty()){
        //squeeze out the closest one
        Temp = q. top();
        q. pop();
        queue_use += 2;
        //Check to see if it has been traversed
        if( vis[Temp.name] ) continue;
else vis[Temp.name] = 1;
        //Start to find the edge of the point one by one
        int num = map[Temp.name].size();
        //Find edges one by one
        for(int i=0; i<num; i++ ){
            //This edge is the edge of the current operation
            edge tmp = map[Temp.name][i];
            //point of edge connection
            int mute = tmp.mute;
```

```
            //If the edge has also been traversed, it means that there must be no further optimization, just skip it
            if( vis[mute] ) continue;
            //edge distance
            int distance = tmp.distance;
            //If you go from this point, there can be optimization
if( dis[mute] > dis[Temp.name] + distance ){
                dis[mute] = dis[Temp.name] + distance;
                q.push(node_pri(mute,dis[mute]));
                queue_use++;
            }
        }
    }

/*----------------------------------------------------------------------------------------*/
    end = clock();
    //cout << "begin " << start << " end " << end << endl;
    return (double) (end -start);
}

//Dijkstra algorithm implementation, returns the running time in double form
//T as a heap. There are several basic functions
template<class T>
double Dijkstra_bino(int source, T q ) {
    //time, count
    clock_t start, end;
    start = clock();

    //Function implementation part
/*----------------------------------------------------------------------------------------*/
    //The starting point is 0
    //push the first point
    q.push(node(source,0));
    queue_use++;
    //The map of the first point is set to 0
    dis[source] = 0;
    node Temp;
    //As long as it hasn't been traversed yet
    while(!q.empty()){
        //squeeze out the closest one
        Temp = q. pop();
        queue_use += 2;
        //Check to see if it has been traversed
if( vis[Temp.name] ) continue;
        else vis[Temp.name] = 1;
        //Start to find the edge of the point one by one
        int num = map[Temp.name].size();
        //Find edges one by one
        for(int i=0; i<num; i++ ){
            //This edge is the edge of the current operation
            edge tmp = map[Temp.name][i];
            //point of edge connection
            int mute = tmp.mute;
            //If the edge has also been traversed, it means that there must be no further optimization, just skip it
            if( vis[mute] ) continue;
            //edge distance
```

```
            int distance = tmp.distance;
            //If you go from this point, there can be optimization
if( dis[mute] > dis[Temp.name] + distance ){
                dis[mute] = dis[Temp.name] + distance;
                q.push(node(mute,dis[mute]));
                queue_use++;
            }
        }
    }

/*----------------------------------------------------------------------------------------
    */
    end = clock();
    //cout << "begin " << start << " end " << end << endl;
    return (double) (end -start);
}


//read map file
int readFile( string filePath ){
    //read data stage
ifstream mapFile;
    //cout << "begin" << endl;
    //if(argv[1][0] == 'a' ) mapFile.open("./USA-road-d.E.gr",ios::in);
    //else mapFile.open("./USA-road-d.NY.gr",ios::in);
    mapFile.open(filePath,ios::in);
    //Read the first few lines of comments
    char l[100];
    for( int i=0; i<4; i++ ){
        mapFile.getline(l,100);
    }
    //read key information
    char type;char sign[3];int num_1,num_2;
    //read the file at the beginning
    mapFile >> type >> sign >> num_1 >> num_2;
    //reread the useless gaze
    mapFile.getline(l,100);
mapFile.getline(l,100);
    mapFile.getline(l,100);
    //build data structure
    //build the map
    //cout << sizeof(vector<edge>)*num_1 << endl;
    map = (vector<edge>*)std::malloc(sizeof(vector<edge>)*num_1);

    //There are num_2 lines in total
    char type_2;
    int node_1, node_2, length;
    for( int i=0; i<num_2; i++ ){
        mapFile >> type_2 >> node_1 >> node_2 >> length;
        if( type_2 != 'a') return 1;
        map[node_1].push_back(edge(node_2,length));
    }
    cout << num_2 << endl;
//cout << "end" << endl;
    mapFile. close();
    //end of reading data
    //The data is stored in the map in the form of edge
    return num_1;
}
```

```
int main(){

    //read the path in the file
    int node_number;
    //Read the relative path of the path
    string filePath = "./USA-road-d.E.gr";
    node_number = readFile(filePath);

    //run a few random points
    int test_number = 1;

//test the Fibonacci heap
//Test data USA map, randomly find 1000 points, and find out all its paths and shortest paths
//Test the number of times the output queue is used, and there is still time
/*-----------------------------------------------------------------------------------------*/

    time_second = 0;
    queue_use = 0;
    for( int i=0; i<test_number; i++ ){
        //start testing
        Init(node_number);
nodeLib = (FibNode**)malloc(sizeof(FibNode*)*node_number);
        // try with prior_queue first
        FibHeap Fib;
        time_second += Dijkstra_fib(i*100+2301,Fib);
    }
    cout << "The queue is fibonacci_queue" << endl;
    cout << "time is : " << time_second << endl;
    cout << "queue use : " << queue_use << endl;
    cout << "\n" << endl;

/*-----------------------------------------------------------------------------------------*/

//□□std::prior_queue
//Test data USA map, randomly find 1000 points, and find out all its paths and shortest paths
//Test the number of times the output queue is used, and there is still time
/*-----------------------------------------------------------------------------------------*/

    time_second = 0;
    queue_use = 0;
    for( int i=0; i<test_number; i++ ){
        //start testing
        Init(node_number);
        //Try it with prior_queue first
        priority_queue<node_pri> prior;
        time_second += Dijkstra(i*100+2301,prior);
    }
    cout << "The queue is prior_queue" << endl;
cout << "time is : " << time_second << endl;
    cout << "queue use : " << queue_use << endl;
    cout << "\n" << endl;

/*-----------------------------------------------------------------------------------------*/
```

```
//test the binomal heap
//Test data USA map, randomly find 1000 points, and find out all its paths and shortest paths
//Test the number of times the output queue is used, and there is still time
/*------------------------------------------------------------------------------------------*/

    time_second = 0;
    queue_use = 0;
    for( int i=0; i<test_number; i++ ){
        //start testing
Init(node_number);
        //Try it with prior_queue first
        binoheap bino;
        time_second += Dijkstra_bino(i*100+2301,bino);
    }
    cout << "The queue is binomial_queue" << endl;
    cout << "time is : " << time_second << endl;
    cout << "queue use : " << queue_use << endl;
    cout << "\n" << endl;

/*------------------------------------------------------------------------------------------*/

//test leftist heap
//Test data USA map, randomly find 1000 points, and find out all its paths and shortest paths
//Test the number of times the output queue is used, and there is still time
/*------------------------------------------------------------------------------------------*/

    time_second = 0;
    queue_use = 0;
    for( int i=0; i<test_number; i++ ){
        // start testing
        Init(node_number);
        // try with prior_queue first
        LefistTree leftist;
        time_second += Dijkstra_bino(i*100+2301,leftist);
    }
    cout << "The queue is leftist_queue" << endl;
    cout << "time is : " << time_second << endl;
    cout << "queue use : " << queue_use << endl;
cout << "\n" << endl;

/*------------------------------------------------------------------------------------------*/
    return 0;
}
```