Project 2

Shortest Path Algorithm with Heap

Group 17

Date:2022-04-09

Chapter 1: Introduction

Shortest path problems are ones of the most fundamental combinatorial optimization problems with many applications, both direct and as subroutines in other combinatorial optimization algorithms. Algorithms for these problems have been studied since 1950's and still remain an active area of research. The most well-known algorithmic solution to this problem is Dijkstra's Algorithm, which uses a priority queue to keep track of the vertices that have been visited and their current distances from the source. Dijkstra's Algorithm is used to find the shortest route from one vertex, called the source, to all others in a weighted graph, but it can be adapted to focus the distance to one node, called a target. Fibonacci heaps are sometimes referred to as “lazy” data structures because their structure is less strictly managed as they perform operations than in many other data structures. But this “laziness” allows them to have extremely fast amortized time complexities for common heap operations.

In this project we are supposed to compute the shortest paths using Dijkstra's algorithm. And the implementation is based on a min-priority queue, Fibonacci heap. We are going to find the best data structure for the Dijkstra's algorithm, using the USA road networks for evaluation.

Chapter 2: Algorithm Specification

Main program

The main program is to read the data and then use Dijkstra's algorithm based on binary heap and Fibonacci Heap to find the shortest paths.

```
Procedure readdata()
    ifstream fin("data/USA-road-d.NY.gr");
    while fin >> str do
        if str = "sp" then break;
    fin >> n >> m;
    dist := new int[n + 1];
    head := new edge*[n + 1];
    fill(head, head + n + 1, nullptr);
    while fin >> str do
        if str = "arcs" then break;
    fin >> str;
    for i = 0 To m-1 Step 1
    {
        fin >> str >> s >> t >> w;
        addedge(s, t, w);
    }
    OUTPUT "Data Read Finished..."

main(signed, char**, char**)
{
    readdata();
    genquery();
    SP_Bin();
    SP_Fib();
}
```

ShortestPath

Initialize distance array and get the vertex with minimum distance on Fibonacci heap.

```
Procedure Fib_Dijkstra(x,t:integer)
    for i = 1 To n Step 1  if i!= x
        dist[i] := inf;
```

```
        pos[i] := H.insert(i, inf);
    Next i

    dist[x] := 0;
    pos[x] := H.insert(x, 0);

    while H.m_numOfNodes!=0 do
    {
         now := H.m_minNode->V;
        if now = t then
        {
            while H.m_numOfNodes do
                H.extract_min();
            return;
        }
        H.extract_min();
        for (edge *p = head[now];p;p = p->next) {
            if dist[now] + p->weight < dist[p->target] then
            {
                dist[p->target] := dist[now] + p->weight;
                H._decrease_key(pos[p->target], dist[p->target]);
            }
        }
     }

```

Binheap

Initialize distance array and get the vertex with minimum distance on binary heap.

The BinHeap is realized by using array, as it's simple, the realization part is omitted. You can see it in the codes.

```
Procedure Bin_Dijkstra(x,t:integer)
    for  i = 1 To n Step 1  if i!= x
        dist[i] := inf;
    H.clear(n);
    dist[x] := 0;
    H.decrease_key(x, 0);

    while H.Size!=0 do
        now := H.top();
```

```
        if now = t then
            return;
        H.pop();
        for (edge *p = head[now];p;p = p->next) {
            if dist[now] + p->weight < dist[p->target] then
            {
                dist[p->target] := dist[now] + p->weight;
                H.decrease_key(p->target, dist[p->target]);
            }
        }
    }


Procedure pop
    data[1].key := inf;
    pair ret := data[1];
    now := 1;
    while 1 do
    {
        if (now << 1) <= Size then
        {
            if (now << 1 | 1) <= Size && data[now << 1].key > data[now << 1 |
                nxt := now << 1 | 1;
            else
                nxt := now << 1;

            data[now] := data[nxt];
            idpos[data[nxt].id] := now;
            now := nxt;
        }
        else
        {
                data[now] := ret;
                idpos[ret.id] := now;
                break;
        }
    }


Procedure decrease_key(id,key:integer)
    pair x{ id,key };
    t := idpos[id], i := t;
    while data[i >> 1].key > key do
```

```
    {
        data[i] := data[i >> 1];
        idpos[data[i].id] := i;
        i >>= 1;
    }
    data[i] := x;
    idpos[id] := i;
```

Fibheap

Fibonacci heap, whose theories can be found in textbook.

You shall try read the code to get more comments.

```
/*insert*/
FibHeapNode* FibHeap::insert(id, newKey:integer)
    FibHeapNode* newNode := _create_node(id, newKey);
    _insert_node(newNode);
    return newNode;

/*merge*/
FibHeapNode* FibHeap::_merge(FibHeapNode* a, FibHeapNode* b)
        if a = nullptr then
            return b;
        if b = nullptr then
            return a;
        if a->key > b->key
        {
            FibHeapNode* temp := a;
            a := b;
            b := temp;
        }
        FibHeapNode* aRight := a->right;
        FibHeapNode* bLeft := b->left;
        a->right := b;
        b->left := a;
        aRight->left := bLeft;
        bLeft->right := aRight;
        return a;


/*create node*/
```

```
FibHeapNode* FibHeap::_create_node(id, newKey:integer)
    FibHeapNode* newNode := mp->getnew();
    newNode->V := id;
    newNode->key := newKey;
    newNode->left := newNode;
    newNode->right := newNode;
    newNode->parent := nullptr;
    newNode->child := nullptr;
    newNode->degree := 0;
    newNode->mark := false;
    return newNode;


/*remove*/
FibHeap::_remove_from_circular_list(FibHeapNode* x)
    if x->right = x then
        return;
    FibHeapNode* leftSib := x->left;
    FibHeapNode* rightSib := x->right;
    leftSib->right := rightSib;
    rightSib->left := leftSib;

```

Chapter 3: Testing Results

Since random numbers are involved in the algorithm, on the premise of saving time, we repeat the query operation for many times and then calculate the average value, so as to make the test data more real and effective as far as possible.

The testing results are as follows (The running time of the code is the average result):

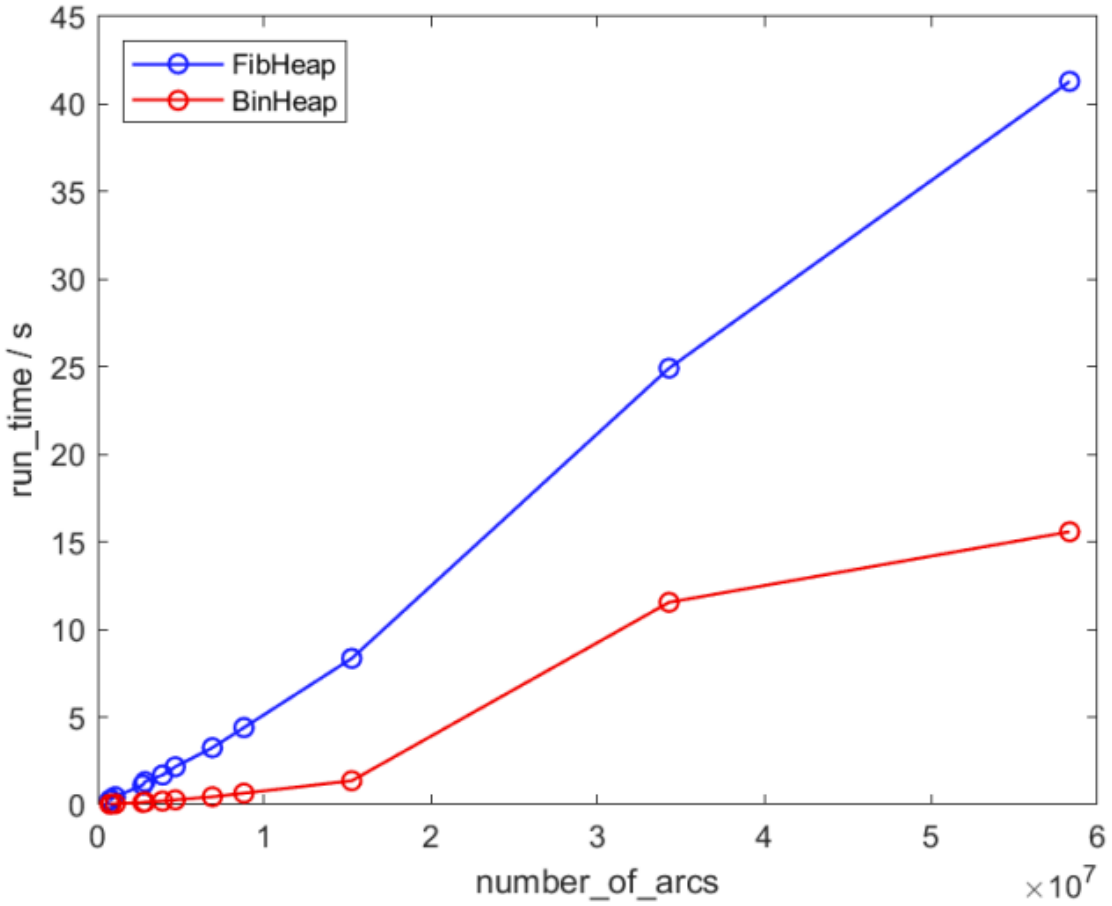
Name	nodes	arcs	BinHeap/s	FibHeap/s	Query times
USA	23,947,347	58,333,344	15.584	41.301	5
CTR	14,081,816	34,292,496	11.554	24.925	10
W	6,262,104	15,248,146	1.364	8.352	50
E	3,598,623	8,778,114	6.546	4.399	100
LKS	2,758,119	6,885,658	0.450	3.251	100
CAL	1,890,815	4,657,742	0.272	2.162	100
NE	1,524,453	3,897,636	0.203	1.693	100
NW	1,207,945	2,840,208	0.151	1.311	100
FLA	1,070,376	2,712,798	0.124	1.123	100
COL	435,666	1,057,066	0.047	0.452	100
BAY	321,270	800,172	0.030	0.299	100
NY	264,346	733,846	0.028	0.254	100

> here are the download links in the table:
>
> USA: http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.USA.gr.gz
>
> CTR: http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.CTR.gr.gz

> W: http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.W.gr.gz
>
> E: http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.E.gr.gz
>
> LKS: http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.LKS.gr.gz
>
> CAL: http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.CAL.gr.gz
>
> NE: http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.NE.gr.gz
>
> NW: http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.NW.gr.gz
>
> FLA: http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.FLA.gr.gz
>
> COL: http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.COL.gr.gz
>
> BAY: http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.BAY.gr.gz
>
> NY: http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.NY.gr.gz

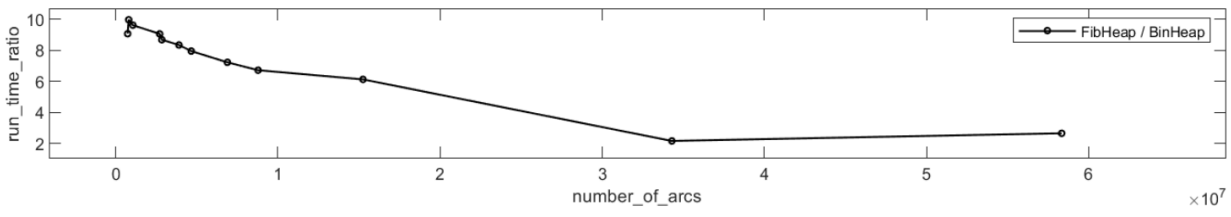
Graphs

Here is some pictures about the performance. To simplify, we just use 'the number of arcs' as x-axis.



As we can see in the following graph, the runtime ratio (FibHeap/BinHeap) is generally going down when input size increases.

If we have much bigger data, FibHeap will be faster than BinHeap theoritically. However our data (and our computer) cannot suit that conditions.



Chapter 4: Analysis and Comments

- We define the size of nodes: n, the size of edges: m

Space Complexity

- Apparently, Using BinHeap or FibHeap both uses $O(n+m)$ space. The heap size won't exceed n, however to store the graph the program already uses $O(n+m)$.

Time Complexity

- In the simplest Dijkstra implementation we use arrays (or linked lists) to store the set Q of all vertices, and the time complexity is $O(n^2)$.
- Dijkstra algorithm using a binomial heap can optimize the time complexity to $O((m+n)logn)$, for $n*O(logn)$ insertions and $m*(logn)$ decrease_key operations.
- Fibonacci Heap further optimizes time complexity to $O(n+mlogn)$, for $n*O(1)$ insertions and $m*(logn)$ decrease_key operations.

The Fibonacci heap is a collection of ordered trees of the smallest heaps in computer science. It has similar properties to the binomial heap and can be used to implement merge priority queues. Operations that do not involve deleting elements have $O(1)$ amortized time. The number of extract-min and Delete is more efficient when compared to others. Each Decrease-key in a dense graph requires only $O(1)$ amortization time, which is a huge improvement over $O(logn)$ in binomial heap. The key idea of Fibonacci heaps is to delay heap maintenance as much as possible. Theoretically Fibonacci heaps perform well, but from a practical point of view, Fibonacci heaps perform poorly in terms of constant factors and programming complexity compared to binary heaps for most applications except for those that need to manage large amounts of data.

Appendix(readme.txt)

We suppose reading the source files for more comments. The files:

Main Program and Data Generator - main.cpp & gen.h. By Zuo.
Dijkstra using Binomial Heap - binheap.cpp & binheap.h. By Zuo.
Dijkstra using Fibonacci Heap - ShortestPath.cpp, fibheap.cpp & fibheap.h. By Han.
Memory pool for Fibonacci Heap - memorypoll.cpp & memorypoll.h. By Zuo.

This report is mainly written by Pan and Shi.

We recommend using DEV-C++ v5.15 to open the project, and TDM-GCC 9.2.0 to compile. The C++ standard shall be C++11 or above.
The .o files are compiled by TDM-GCC 9.2.0 in 64-bit. You may need delete them to re-compile.