Project III:
Shortest Path Algorithm with Heap
Report

xxx

2023-4-2

1 Introduction

Dijkstra algorithm is one of the most efficient algorithms in solving "The Shortest Path" problem by using a heap (priority queue) to support its greedy method. In this project, we uses 4 different types of heap to implement Dijkstra Shortest Path Algorithm (Dijkstra algorithm for short) and compare their efficiency using a dataset from The 9th DIMACS Implementation Challenge.

The 4 types of heap we tried are **Binary Heap**, **Leftist**, .**Skew Heap** and **Fibonacci heap**. These heaps all adopt a tree-like structure but have different performances in various operations including inserting, finding maximum, deleting maximum, merging and increasing key.

2 Algorithm Specification

2.1 Dijkstra algorithm

To describe dijkstra algorithm, we first make some definiton:

- V: the set of vertices
- E: the set of edges
- S: the source vertex
- $dis(u), u \in V$: the distance from S to vertex u
- T: the set of vertices where $\forall u \in T, dis(u)$ is the shortest path from S to u

The Djkstra Algorithm is based on a conclusion that if $\forall v \notin T, dis(u) < dis(v), u \notin T$, then $dis(u)$ cannot be updated by other vertices anymore and we can therefore move u into T. The conclusion can be easily prooved by contradiction. With this, we can use a structure which supporting quick querying and deleting of the

minimum element and inserting a new element to imitate the set V/T. Heap is therefore a perfect choice of this task. The detailed description of the algorithm is as below.

Algorithm 1 Dijkstra

```
For all u ∈ V, Set dis(u) to ∞
dis(S) ← 0
Push (0, S) into heap
while heap is not empty do
    (d, x) ← minimum element of heap
    Delete the minimum element of heap
    if d > dis(x) then
        Continue
    for For all (x, y, z) ∈ E do
        if dis(y) > d + z then
            dis(y) ← d + z
            Push (dis(y), y) into heap
```

2.2 Binary Heap

The binary heap is implemented by a complete binary tree, which can be stored in an array in C/C++. When inserting a new element, we first append it to the end of the complete binary tree and adjust it upwards until it is larger than its root. In the same way, when we want to delete the minimum element, we first swap it with the last element in the complete binary tree and reduce the heap size by 1. Then, all we need to do is adjust the root downward to until it is smaller than both of its sons. When moving downward, we simply choose the smaller one of its sons to swap with it.

The pseudocode of Binary Heap can be found on the lecture powerpoint and is therefore omitted.

2.3 Leftist Heap

The Leftist Heap is an adjustment of the traditional heap to give support for merging operation. It does not use a complete binary tree or a balance tree to guarantee the height. In contrast, it tries to make the left part of the binary tree large. We define the null-path-length of vertex u ($NPL(u)$) as the length of shortest path from u to a NULL child. Leftist Heap tries to guarantee that $NPL(LeftChild(u)) >= NPL(RightChild(u)),\ \forall u$. When merging two heaps, it compares the two root and merge one heap with the right subtree of the other. On recalling, it check whether the restriction holds and swap the left and right child if is violated.

The inserting and deleting operation can both be converted into merging, and are therefore also supported by Leftist Heap.

The pseudocode of Lefist Heap can be found on the lecture powerpoint and is therefore omitted.

2.4 Skew Heap

The Skew Heap is similar to Leftist Heap but do not need to store NPL information. The merging operation is all the same with Letftist Heap except that Skew Heap will always swap the left and right child of all the vertices on the way of recalling. The complexity is guaranteed by amortized analysis and will be shown in following chapters.

The pseudocode of Lefist Heap can be found on the lecture powerpoint and is therefore omitted.

2.5 Fibonacci Heap

The Fibonacci Heap does not maintain a tree, but a forest (a set of trees) instead. The root of all the trees are organized in the form of list to support quick insering of a new tree. Similarly, the children of a vertex is also organized in lists to quickly

merge a tree into the children of another tree. At last, we maintain a pointer (min-pointer) which points to the minimum root.

When inserting a new vertex or merging two heap, we simply merge their roots, which is done by list operation. When querying for the minimum element, we simply return the content of the min-pointer.

The only trouble lies in the operation of deleting minimum. Firstly, we delete the content of the min-pointer and its children then form many new tree in the forest. However, this time we need additional operation to adjust the forest and find the new minimum root. We iterate over all the trees from small to large and merge every two trees with the same sizes. In this processing, we can also find the minimum root and update the min-pointer. After this adjustment, no two trees have the same sizes.

The pseudocode is as below:

Algorithm 2 Merging two Fibonacci Heaps

1: $A =$ root list of $Heap_1$
2: $B =$ root list of $Heap_2$
3: Merge lists A and B into C
4: $minPointer(C) = softmax(minPointer(A), minPointer(B))$
5: Return C

Algorithm 3 Deleting minimum of Fibonacci Heap

1: $u \leftarrow contentofmin - pointer$
2: **for** $v \in Child(u)$ **do**
3: Add subtree of v to root list
4: **for** $v \in rootlist$ **do**
5: **while** $C(Size(v)) \neq NULL$ **do**
6: Merge $C(Size(v))$ into $Child(v)$
7: $C(Size(v)) \leftarrow v$
8: Update $minPointer$ with $Value(v)$
9: Re-organize root list

3 Testing Results

3.1 Data

The testing are separated into two parts, one for the correctness, and the other one for runtime comparison.

To test the correctness of our implementation of the four heaps and dijkstra algorithm, we used the online judges luogu and loj. The problem we used are *luogu P4779* 【模板】单源最短路径（标准版） and *LOJ 119.* 单源最短路, which can be found in the references part. All 4 versions of heaps passed these problems.

To test the runtime of different heaps, we used the data from The 9th DIMACS lmplementation Challenge. We downloaded the dataset of Colorado with 435666 vertices and 1057066 edges. For each of the 4 types of heap, we randomly generated 1000 querys and computed the average runtime for a single query. To make the result more prooving, we added a control group with used the C++ STL priority_queue, which is implemented using a heap.

3.2 Testing

There are 3 versions of main functions in our code, with two of them being comments. The 3 main functions serves for different purposed. One for the correctness check, that is, it is designed for the data format in LOJ119. One for the runtime test, which does the task of random data generating and time clocking. The last one is for user testing, in which users can type in querys themselves and get the result in the standard output. In the second and third version, graph data is loaded from input.txt in a certain format, which will be described in Readme.md.

3.3 Results

The correctness has been shown above.

Below is the runtime information of the 5 different implementation of heaps in miliseconds.

Table 1: Time for a single query/ms

Binary	92.757
Leftist	155.568
Skew	144.919
Fibonacci	181.938
STL	263.247

From the table, we can see that all our implementation of heaps run faster than STL, meaning that they should have a right time complexity. The Binary Heap is the fastest, maybe because it is so easy to implement that the time constant is much smaller than the other structures.

4 Analysis and Comments

4.1 dijkstra & Binary Heap & Lefist Heap & Skew Heap

The time complexity of calculating the shortest path from a single source vertex to all the other vertices is $O(mlogn)$, where $m = |E|,\ n = |V|$. Since all the queries may have different sources vertex, the time complexity for a single query is also $O(mlogn)$.

The time complexity for operations of the 3 heap are as below.

The time complexity analysis of above structures and algorithms have all been taught on class and can still be found in lecture powerpoints, so we imitated detailed proof.

Table 2: Time complexity

	insert	delete-min	query-min	merge
Binary	O(logn)	O(logn)	O(1)	O(n)
Leftist	O(logn)	O(logn)	O(1)	O(logn)
Skew	O(logn)	O(logn)	O(1)	O(logn)

4.2 Fibonacci Heap

Let's put our attention on the time complexity of Fibonacci heap.

Obviously, the time complexity of inserting, merging and mininum querying are all $O(1)$ because they only needs list connecting. With the benefit of lists, we can solve them without any iteration.

In minimum deleting, we first need to iterate over all children of the minimum root, offering a time complexity of $O(d)$ where $d = |Child(minPointer)|$. Then, we iterate over all the root and merge the ones with the same sizes. This gives a time complexity of $O(m)$, where $m = |rootlist|$. To implement this, we need a bucket whose size is the large size of the trees. Because all the sizes are powers of 2, so we only need $log(Max(size))$ buckets. Therefore, the time complexity of this part is no more than $O(logn)$

The only question is the amortized time complexity of iterating the root list. Notice that each node can be merged as a root into the children of another tree for at most once. So the overall time complexity of this part won't exceed the number of insering operation. If we compute this time complexity part in inserting, then inserting would still remain $O(1)$, but deleting minimum can then be prooved to have an amortized time complexity of $O(logn)$.

Therefore, the whole table is as below:

Table 3: Time complexity

	insert	delete-min	query-min	merge
Binary	O(logn)	O(logn)	O(1)	O(n)
Leftist	O(logn)	O(logn)	O(1)	O(logn)
Skew	O(logn)	O(logn)	O(1)	O(logn)
Fibonacci	O(1)	O(logn)	O(1)	O(1)

5 Appendix

Since the code is not short, we think it's better not to copy it here. You can better understand the structure of our code with the Readme.md.

6 References

[1] Dataset downloaded from http://www.diag.uniroma1.it//challenge9/download.shtml

[2] Online judge: luogu:https://www.luogu.com.cn/problem/P4779

[3] Online judge: LOJ:https://loj.ac/p/119

7 Declaration

I hereby declare that all the work done in this project titled "xxx" is of our independent effort as a group.