VE281 Homework III

Performance Analysis for Priority Queues Applied in Dijkstra SSSP Algorithm

Name: Tianyi Ge **Stu Number:** 516370910168

1 Introduction

In this report, we discuss the time complexity of three types of different priority queues including *Unsorted heap*, *Binary heap* and *Fibonacci heap*, by testing their average runtime applied in *Dijkstra Single-Source-Shortest-Path Algorithm.* For each test case size, we have five distinct maps, testing the extreme situation where the source and sink locate on the diagonal of the map.

In the test case, the test code `#define TEST` to hide all the unnecessary output part and obtain the direct runtime of *Dijkstra Algorithm.*

It's noteworthy that since *Dijkstra Algorithm* cannot deal with negative weight edges, the randomized weights for each grid is restricted to $[0, 100]$ to prevent the intermediate or final path cost from exceeding the limit of `int`.

2 Performance analysis on Selection Algorithms

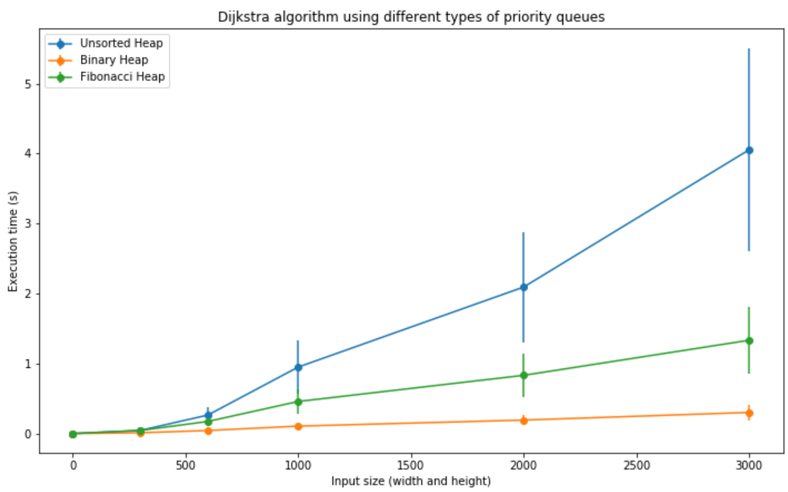

Figure 1: Dijkstra algorithm using different types of priority queues

To graphically demonstrate the outcome, here the standard deviation of tests are kept as error bar.

Theoretically, *Fibonacci heap* take an edge over three of these three priority queues on amortized time complexity. The `enqueue` operation of *Fibonacci heap* is $O(1)$ and the `dequeue_min` operation of which is $O(\log n)$. Nevertheless, the runtime result does not manifest the advantage. *Fibonacci heap* is much faster than *Unsorted heap*, undoubtedly, runs clearly slower than binary tree in practice.

Presumably, the disappointment results, firstly, from the frequent memory operation and the construction of structures like `node_t *` and `coord` (see Appendix). Thus, *rvalue reference* is introduced here. If the memory operation is efficient, the power of *Fibonacci heap* might boost. On the contrary, the binary tree only entails index accessing to the vector. The time cost is relatively small.

Moreover, the constant of the time complexity of *Fibonacci heap* may be larger than *Binary heap* because each operation of the former requires several memory steps. Also, if the root list in *Fibonacci heap* accumulates, then the time complexity of `consolidate` may degrade to $O(n)$.

3 Conclusion

In this report, we have demonstrated the characteristics of three different priority queues applied in Dijkstra Algorithms as well as their performance. *Unsorted heap* is essentially not a good way to maintain the minimum key value. Each time it requires a traversal to get the result. The interesting comparison between *Fibonacci heap* and *Binary heap* shows that the latter one outperform significantly. Theoretically, *Fibonacci heap* seems better but in practice, *Binary heap* is a better choice both on runtime and the difficulty to implement.

4 Appendix

A Source Codes

Program code 1: Dijkstra algorithms using three different heaps

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <string>
#include <getopt.h>
#include "unsorted_heap.h"
#include "binary_heap.h"
#include "fib_heap.h"

#define TEST

#define MAXN 5010
#define MAXM 5010
#define point(u) "(" << u.x << ", " << u.y << ")"

struct coord {
    unsigned x, y;
    int pathcost;
} s, t;

struct compare_t {
    bool operator()(const coord &a, const coord &b) const {
        if (a.pathcost == b.pathcost) {
            if (a.x == b.x) return a.y < b.y;
            return a.x < b.x;
        }
        return a.pathcost < b.pathcost;
    }
};

enum im_t {UNSORTED, BINARY, FIBONACCI, IM_SIZE};

unsigned m, n;
```

```
int map[MAXM][MAXN];
bool reached[MAXM][MAXN];
coord pred[MAXM][MAXN];
const int dx[] = {1, 0, -1, 0};
const int dy[] = {0, 1, 0, -1};

static int v_flag = 0;
static im_t i_flag = IM_SIZE;

const static char *im_name[] = {"UNSORTED", "BINARY", "FIBONACCI"};

priority_queue<coord, compare_t> *heap = NULL;

bool hasNeighbor(const coord &u, unsigned d) {
    int nx = u.x + dx[d];
    int ny = u.y + dy[d];
    return (nx >= 0 && nx < (int)m && ny >= 0 && ny < (int)n);
}

void read() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(0);
    std::cin >> m >> n;
    std::cin >> s.x >> s.y >> t.x >> t.y;
    for (unsigned j = 0; j < n; ++j)
        for (unsigned i = 0; i < m; ++i)
            std::cin >> map[i][j];
}

#ifndef TEST

static unsigned step = 0;

void trace_helper(const coord &u) {
    if (u.x != s.x || u.y != s.y)
        trace_helper(pred[u.x][u.y]);
    std::cout << point(u) << "\n";
}
```

```cpp

void trace_back_path(const int &dist) {
    std::cout << "The shortest path from " << point(s) << " to " << point(t)
    ↪  << " is " << dist << "." << "\n";
    std::cout << "Path:" << "\n";
    trace_helper(t);
}

void log_u(const coord &u) {
    std::cout << "Step " << step++ << "\n";
    std::cout << "Choose cell " << point(u) << " with accumulated length " <<
    ↪  u.pathcost << ".\n";
}

void log_v(const coord &v) {
    std::cout << "Cell " << point(v) << " with accumulated length " <<
    ↪  v.pathcost << " is added into the queue." << "\n";
}

void log_t(const coord &v) {
    std::cout << "Cell " << point(v) << " with accumulated length " <<
    ↪  v.pathcost << " is the ending point." << "\n";
}

#endif

void construct_heap() {
    switch (i_flag) {
        case UNSORTED:
            heap = new unsorted_heap<coord, compare_t>;
            break;
        case BINARY:
            heap = new binary_heap<coord, compare_t>;
            break;
        case FIBONACCI:
            heap = new fib_heap<coord, compare_t>;
            break;
        default:
```

```
                return;
        }
    }

    void destroy_heap() {
        delete heap;
    }

    void dijkstra_heap() {
        reached[s.x][s.y] = true;
        s.pathcost = map[s.x][s.y];
        heap->enqueue(s);
        while (!heap->empty()) {
            coord u = heap->dequeue_min();
    #ifndef TEST
            if (v_flag) log_u(u);
    #endif
            for (unsigned d = 0; d < 4; ++d) {
                if (!hasNeighbor(u, d)) continue;
                unsigned nx = u.x + dx[d], ny = u.y + dy[d];
                if (reached[nx][ny]) continue;
                coord v = {nx, ny, u.pathcost + map[nx][ny]};
                reached[nx][ny] = true;
                pred[nx][ny] = u;
                if (t.x == nx && t.y == ny) {
    #ifndef TEST
                    if (v_flag) log_t(v);
                    trace_back_path(v.pathcost);
    #endif
                    return;
                }
    #ifndef TEST
                if (v_flag) log_v(v);
    #endif
                heap->enqueue(v);
            }
        }
    }
```

```
void getoptions(const int &argc, char **argv) {
    static option long_options[] = {
        {"implementation", required_argument, 0, 'i'},
        {"verbose", no_argument, 0, 'v'},
        {0, 0, 0, 0}
    };

    int option_index = 0, c = -1;
    while ((c = getopt_long(argc, argv, "i:v", long_options, &option_index)) != -1) {
        switch(c) {
            case 'v':
                v_flag = 1;
                break;
            case 'i':
                for (unsigned i = 0; i < IM_SIZE; ++i)
                    if (strcmp(optarg, im_name[i]) == 0) {
                        i_flag = (im_t)i;
                        break;
                    }
                break;
            default:
                break;
        }
    }
}

#ifdef TEST

double bg, runtime;

void set_clock() {
    bg = clock();
}

void get_clock() {
    runtime = (clock() - bg) * 1.0 / CLOCKS_PER_SEC;
```

```
}

#endif

int main(int argc, char **argv) {
    getoptions(argc, argv);
    read();
    construct_heap();

#ifdef TEST
    set_clock();
#endif

    dijkstra_heap();

#ifdef TEST
    get_clock();
    std::cout << runtime << "\n";
#endif

    destroy_heap();
    return 0;
}
```

Program code 2: Test case generator

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import sys
import random
import os
import time


```

```python
# In[7]:


TEST_SIZE = 6
PER_SIZE = 5
INT_MAX = 100;
size = [1, 300, 600, 900, 1200, 1500]
if __name__ == "__main__":
    dirname = '../inputs'
    if not os.path.exists(dirname):
        os.makedirs(dirname)
    for cases in range(TEST_SIZE):
        for i in range(PER_SIZE):
            filename = '../inputs/{}{}.in'.format(cases, i)
            with open(filename, 'w') as w:
                n = size[cases]
                w.write(str(n) + '\n' + str(n) + '\n')
                w.write('0 0\n')
                w.write(str(n-1) + ' ' + str(n-1) + '\n')
                for x in range(n):
                    for y in range(n):
                        w.write(str(random.randint(0, INT_MAX-1)) + ' ')
                    w.write('\n')
            print("Cases{}{}".format(cases, i))


# In[ ]:
```

Program code 3: Test case runner

```zsh
#!/bin/zsh
size=(1 300 600 1000 2000 3000)
for ((i=0; i<6; i++)); do
    for ((j=0; j<5; j++)); do
        ./main -i UNSORTED  < ../inputs/$i$j.in > ../outputs/u$i$j.out
        echo u$i$j
        ./main -i BINARY    < ../inputs/$i$j.in > ../outputs/b$i$j.out
        echo b$i$j
```

```
        ./main -i FIBONACCI < ../inputs/$i$j.in > ../outputs/f$i$j.out
        echo f$i$j
    done
done
```

Program code 4: Plotting program

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:



import matplotlib.pyplot as plt
from scipy.stats import t
import numpy as np


# In[3]:


TEST_SIZE = 6
PER_SIZE = 5
INT_MAX = 100;
size = [1, 300, 600, 1000, 2000, 3000]
im = ['u', 'b', 'f']
plt.figure(figsize=(12,7))

for flag in range(3):
    y=np.array([])
    conf_interval=np.array([])
    for cases in range(TEST_SIZE):
        time=np.array([])
        # get time array per size per implementation
        for i in range(PER_SIZE):
            with open('../outputs/' + im[flag] + '{}{}.out'.format(cases, i),
             ↪  'r') as f:
                data=f.read();
```

```
                data = data.split('\n')
                time = np.append(time, float(data[0]))

        y = np.append(y, np.mean(time))  # one point on one line
        conf_interval = np.append(conf_interval, np.std(y))
    plt.errorbar(size, y, yerr = conf_interval, fmt = '-o')

plt.legend(['Unsorted Heap', 'Binary Heap', 'Fibonacci Heap'], loc = 'upper
↪  left')
plt.xlabel('Input size (width and height)')
plt.ylabel('Execution time (s)')
plt.title('Dijkstra algorithm using different types of priority queues')
plt.savefig('res.png')
plt.show()


# In[ ]:
```