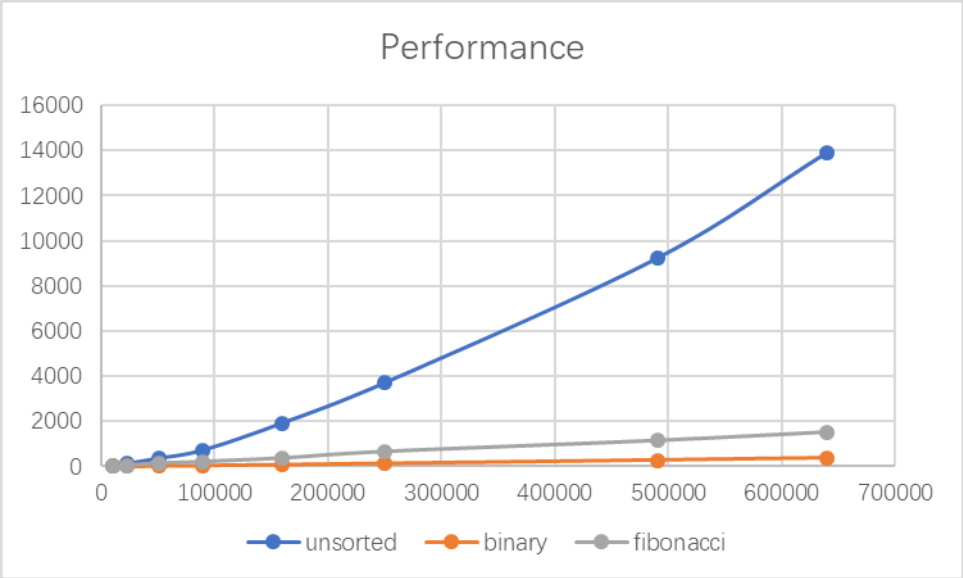
Lab05-PriorityQueuesAndApplication

VE281 - Data Structures and Algorithms, Xiaofeng Gao, TA: Li Ma, Autumn 2019

* Name: Jintian Ge Student ID: 517021911142 Email: gejintian@sjtu.edu.cn

1. The following is the performance examination. From the graph, unsorted heap takes much more time than other two algorithms. The curve looks like a quadratic function, which is $O(n^2)$. And the fibonacci curve and binary curve are both about $O(\log n)$. This result coincides with our expectation.



2. The following is my code: There are three files and I put them into one lstlisting.

```
    #ifndef INC_5_GAME_H
#define INC_5_GAME_H

#include "priority_queue.h"
#include "unsorted_heap.h"
#include "binary_heap.h"
#include "fib_heap.h"
using namespace std;

struct point_t{
    unsigned int x;//horizontal
    unsigned int y;//vertical
    bool is_reached;//true for reached, false for not reached
    int cost;
    int pathcost;
    int predecessor;
    bool is_start = false;
};

struct compare_t
{
    bool operator()(const point_t &a, const point_t &b) const
    {
        if(a.pathcost<b.pathcost) return true;
```

```
25          else if(a.pathcost == b.pathcost){
26              if(a.x == b.x) return a.y < b.y;
27              else return a.x < b.x;
28          }
29          else return false;
30      }
31  };
32
33  void trace_back_path(point_t &N, vector<point_t> &grid, int start_x
        , int start_y, int end_x, int end_y, int width, bool verbose);
34
35  void check_point(point_t *C, point_t *N, int width);//Return true
        if trace_back_path().
36
37  void play(vector<point_t> &grid, int start_x, int start_y, int
        end_x, int end_y, priority_queue<point_t, compare_t> *PQ,
38            unsigned int height, unsigned int width, bool verbose);
39
40
41  #endif //INC_5_GAME_H
42  #include <iostream>
43  #include "game.h"
44  #include "priority_queue.h"
45  using namespace std;
46
47  static void trace_helper(point_t N, vector<point_t> &grid){
48      if(N.is_start){
49          cout<<"("<<N.x<<", "<<N.y<<")"<<endl;
50          return;}
51      trace_helper(grid[N.predecessor],grid);
52      cout<<"("<<N.x<<", "<<N.y<<")"<<endl;
53  }
54
55
56  void trace_back_path(point_t &N, vector<point_t> &grid, int start_x
        , int start_y, int end_x, int end_y){
57      cout<<"The shortest path from ("<<start_x<<", "<<start_y<<") to
            ("<<end_x<<", "<<end_y<<") is ";
58      cout<<N.pathcost<<".\n"<<endl;
59      cout<<"Path:"<<endl;
60      trace_helper(N, grid);
61  }
62
63  void check_point(point_t *C, point_t *N, int width){
64      N->pathcost = C->pathcost + N->cost;
65      N->is_reached = true;
66      N->predecessor = C->x + width*C->y;
67  }
68
69  void play(vector<point_t> &grid, int start_x, int start_y, int
```

```
    end_x, int end_y, priority_queue<point_t, compare_t> *PQ,
70        unsigned int height, unsigned int width, bool verbose){//
              index = x + y*width
71    //Start_point.pathcost=start_point.cellweight
72    grid[start_x + start_y*width].pathcost = grid[start_x + start_y
          *width].cost;
73    //Mark start_point as reached
74    grid[start_x + start_y*width].is_reached = true;
75    grid[start_x + start_y*width].is_start = true;
76    //PQ.enqueue(start_point)
77    PQ->enqueue(grid[start_x + start_y*width]);
78    int n = 0;
79    //cout<<PQ->empty()<<endl;
80    while(!PQ->empty()){
81        point_t C = PQ->dequeue_min();
82        if(verbose){
83            cout<<"Step "<<n<<endl;
84            cout<<"Choose cell ("<<C.x<<", "<<C.y<<") with
                  accumulated length "<<C.pathcost<<"."<<endl;
85            n++;
86        }
87        //cout<<"C points at ("<<C.x<<", "<<C.y<<")."<<endl;
88        //if(C.predecessor!= nullptr)cout<<"C's predecessor is at
              "<<C.predecessor->x<<", "<<C.predecessor->y<<endl;
89        //Right
90        if(C.x + 1 < width && !grid[C.x + 1 + width*C.y].is_reached
              ){
91            //cout<<"Right neighbor is at ("<<grid[C.x + 1 + width*
                  C.y].x<<", "<<grid[C.x + 1 + width*C.y].y<<")."<<
                  endl;
92            check_point(&C, &grid[C.x + 1+ width*C.y], width);
93            if(grid[C.x + 1+ width*C.y].x == end_x && grid[C.x + 1+
                  width*C.y].y == end_y){
94                if(verbose){cout<<"Cell ("<<grid[C.x + 1+ width*C.y
                      ].x<<", "<<grid[C.x + 1+ width*C.y].y<<") with
                      accumulated length ";
95                cout<<grid[C.x + 1+ width*C.y].pathcost<<" is the
                      ending point."<<endl;}
96                trace_back_path(grid[C.x + 1+ width*C.y], grid,
                      start_x, start_y, end_x, end_y);
97                return;
98            }
99            else PQ->enqueue(grid[C.x + 1+ width*C.y]);
100           if(verbose){
101               cout<<"Cell ("<<grid[C.x + 1+ width*C.y].x<<", "<<
                      grid[C.x + 1+ width*C.y].y<<") with accumulated
                      length ";
102               cout<<grid[C.x + 1+ width*C.y].pathcost<<" is added
                      into the queue."<<endl;
103           }
```

```
104         }
105         //Down
106         if(C.y + 1 < height && !grid[C.x+ width*(C.y + 1)].
                is_reached){
107             //cout<<"Down neighbor is at ("<<grid[C.x+ width*(C.y +
                    1)].x<<", "<<grid[C.x+ width*(C.y + 1)].y<<")."<<
                endl;
108             check_point(&C, &grid[C.x+ width*(C.y + 1)], width);
109             if(grid[C.x+ width*(C.y + 1)].x == end_x && grid[C.x+
                    width*(C.y + 1)].y == end_y){
110                 if(verbose){cout<<" Cell ("<<grid[C.x+ width*(C.y +
                        1)].x<<","<<grid[C.x+ width*(C.y + 1)].y<<") 
                        with accumulated length ";
111                 cout<<grid[C.x+ width*(C.y + 1)].pathcost<<" is the
                        ending point."<<endl;}
112                 trace_back_path(grid[C.x+ width*(C.y + 1)], grid,
                        start_x, start_y, end_x, end_y);
113                 return;
114             }
115             else PQ->enqueue(grid[C.x+ width*(C.y + 1)]);
116             if(verbose){
117                 cout<<" Cell ("<<grid[C.x+ width*(C.y + 1)].x<<","
                        <<grid[C.x+ width*(C.y + 1)].y<<") with 
                        accumulated length ";
118                 cout<<grid[C.x+ width*(C.y + 1)].pathcost<<" is 
                        added into the queue."<<endl;
119             }
120         }
121         //Left
122         if(C.x >= 1 && !grid[C.x - 1 + width*C.y].is_reached){
123             //cout<<"Left neighbor is at ("<<grid[C.x - 1 + width*C
                    .y].x<<", "<<grid[C.x - 1 + width*C.y].y<<")."<<endl
                    ;
124             check_point(&C, &grid[C.x - 1 + width*C.y], width);
125             if(grid[C.x - 1 + width*C.y].x == end_x && grid[C.x - 1
                    + width*C.y].y == end_y){
126                 if(verbose){cout<<" Cell ("<<grid[C.x - 1 + width*C.
                        y].x<<","<<grid[C.x - 1 + width*C.y].y<<") with
                        accumulated length ";
127                 cout<<grid[C.x + width*(C.y - 1)].pathcost<<" is 
                        the ending point."<<endl;}
128                 trace_back_path(grid[C.x - 1 + width*C.y], grid,
                        start_x, start_y, end_x, end_y);
129                 return;
130             }
131             else PQ->enqueue(grid[C.x - 1 + width*C.y]);
132             if(verbose){
133                 cout<<" Cell ("<<grid[C.x - 1+ width*C.y].x<<","<<
                        grid[C.x - 1+ width*C.y].y<<") with accumulated 
                        length ";
```

```
134                 cout<<grid[C.x - 1+ width*C.y].pathcost<<" is added
                        into the queue."<<endl;
135             }
136         }
137         //Up
138         if(C.y >= 1 && !grid[C.x + width*(C.y - 1)].is_reached){
139             //cout<<"Up neighbor is at ("<<grid[C.x + width*(C.y -
                    1)].x<<", "<<grid[C.x + width*(C.y - 1)].y<<")."<<
                    endl;
140             check_point(&C, &grid[C.x + width*(C.y - 1)], width);
141             if(grid[C.x + width*(C.y - 1)].x == end_x && grid[C.x +
                    width*(C.y - 1)].y == end_y){
142                 if(verbose){cout<<"Cell ("<<grid[C.x + width*(C.y -
                        1)].x<<", "<<grid[C.x + width*(C.y - 1)].y<<") 
                        with accumulated length ";
143                 cout<<grid[C.x + width*(C.y - 1)].pathcost<<" is 
                        the ending point."<<endl;}
144                 trace_back_path(grid[C.x + width*(C.y - 1)], grid,
                        start_x, start_y, end_x, end_y);
145                 return;
146             }
147             else PQ->enqueue(grid[C.x + width*(C.y - 1)]);
148             if(verbose){
149                 cout<<"Cell ("<<grid[C.x + width*(C.y - 1)].x<<", "
                        <<grid[C.x + width*(C.y - 1)].y<<") with 
                        accumulated length ";
150                 cout<<grid[C.x + width*(C.y - 1)].pathcost<<" is 
                        added into the queue."<<endl;
151             }
152         }
153     }
154 }
155 #include <iostream>
156 #include <unistd.h>
157 #include <getopt.h>
158 #include "game.h"
159 #include "priority_queue.h"
160 #include "unsorted_heap.h"
161 #include "binary_heap.h"
162 #include "fib_heap.h"
163 using namespace std;
164 
165 int main(int argc, char **argv) {
166     std::ios::sync_with_stdio(false);
167     std::cin.tie(0);
168     //Input part
169     int verbose_flag = 0;
170     int i_flag = 0;
171     char *cvalue = NULL;
172     int c;
```

```
173     int option_index = 0;
174     while(true){
175         static struct option long_options[] = {
176                 {"verbose", no_argument, &verbose_flag, 1},
177                 {"implementation", required_argument, nullptr, 'i'
                    },
178                 {0,0,0,0}
179         };
180         c = getopt_long(argc, argv,"+vi:", long_options, &
               option_index);
181         if(c == -1) break;
182         switch(c){
183             case 'i':
184                 cvalue = optarg;
185                 i_flag = 1;
186                 break;
187             case 'v':
188                 verbose_flag = 1;
189                 break;
190             default:
191                 break;
192         }
193     }
194     if(i_flag == 0) return 0;
195     string str(cvalue);
196     priority_queue<point_t ,compare_t> *PQ;
197     if(str == "BINARY") PQ = new binary_heap<point_t ,compare_t>;
198     else if(str == "UNSORTED") PQ = new unsorted_heap<point_t ,
           compare_t>;
199     else PQ = new fib_heap<point_t ,compare_t>;
200     bool verbose;
201     if(verbose_flag == 1) verbose = true;
202     else verbose = false;
203     //
204     unsigned int height, width;
205     cin>>width>>height;
206     unsigned int start_x, start_y, end_x, end_y;
207     cin>>start_x>>start_y>>end_x>>end_y;
208     vector<point_t> grid;
209     for(unsigned int y = 0; y < height; y++){
210         for(unsigned int x = 0; x < width; x++){
211             point_t point;
212             point.x = x;
213             point.y = y;
214             cin>>point.cost;
215             point.is_reached = false;
216             grid.push_back(point);
217         }
218     }
219     //cout<<"Start point is ("<<grid[start_x + start_y*width].x<<",
```

```
        "<<grid[start_x + start_y*width].y<<")."<<endl;
220     //
221     play(grid, start_x, start_y, end_x, end_y, PQ, height, width,
            verbose);
222     delete PQ;
223     return 0;
224 }
```