Lab05-Priority Queues and Application

VE281 - Data Structures and Algorithms, Xiaofeng Gao, Autumn 2019

∗ Name:Wu Jiayao Student ID:517370910257 Email: jiayaowu1999@sjtu.edu.cn

1 Performance Comparison

1.1 Testing settings

1. For this report, 11 different width are used to compare the performance, listed as follows

$$50, 230, 410, 590, 770, 950, 1130, 1310, 1490, 1670, 1850$$

2. For each of the five size, 30 grids are generated then tested before getting an average running time for this size.

3. The figure is plotted through Excel.

1.2 Performance of Unsorted Heap

Size	Time for Unsorted(s)
50	0.001
230	0.023
410	0.124
590	0.384
770	1.035
950	1.891
1130	2.927
1310	4.641
1490	7.268
1670	11.495
1850	17.518

Table 1: Running time of grids with 11 different size by Unsorted Heap

1.3 Performance of Binary Heap

Size	Time for Binary(s)
50	0.000
230	0.010
410	0.024
590	0.069
770	0.127
950	0.178
1130	0.243
1310	0.327
1490	0.476
1670	0.654
1850	0.836

Table 2: Running time of grids with 11 different size by Binary Heap

1.4 Performance of Fibonacci Heap

Size	Time for Fibonacci(s)
50	0.001
230	0.031
410	0.121
590	0.283
770	0.583
950	1.073
1130	1.548
1310	2.389
1490	3.416
1670	4.920
1850	6.651

Table 3: Running time of grids with 11 different size by Fibonacci Heap

1.5 Overall Comparison

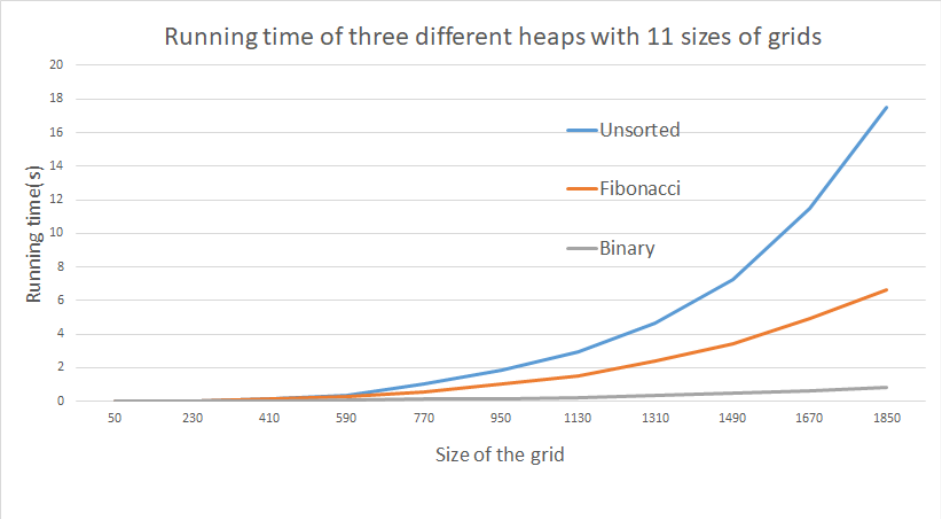

Figure 1: Running time of grids with 11 different size

2 Conclusion and Discussion

In Conclusion, in grids with relatively large size, priority queue implemented with binary heap performs the best with the shortest average running time, followed by that by Fibonacci Heap, the last the unsorted heap.

Theoratically, Fibonacci heap should be of similar or less running time with/than binary heap, since they have the same theoratical time complexity for **dequeue_min**, while **enqueue** operatrion is constant time for Fibonacci heaps but $\mathcal{O}(\log n)$ for binary heaps.

Type	enqueue	dequeue_min
Unsorted	$\Theta(1)$	$\Theta(n)$
Binary	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Fibonacci	$\Theta(1)$	$\mathcal{O}(\log n)$

Table 4: Theoratical running time of two operations for different types of priority queues

The reason that Fibonacci heap performs much worse than binary heap is that in this path searching algorithm, quite a large amount of **dequeue_min** operations are needed to complete such computing. For Fibonacci heap, **dequeue_min** takes a great effort since all its maintainence work is done during **dequeue_min**.

Appendix

priority_queue.h

```
#ifndef PRIORITY_QUEUE_H
#define PRIORITY_QUEUE_H

#include <functional>
#include <vector>

// OVERVIEW: A simple interface that implements a generic heap.
//           Runtime specifications assume constant time comparison and
//           copying. TYPE is the type of the elements stored in the priority
//           queue. COMP is a functor, which returns the comparison result of
//           two elements of the type TYPE. See test_heap.cpp for more details
//           on functor.
template<typename TYPE, typename COMP = std::less<TYPE> >
class priority_queue {
public:
    typedef unsigned size_type;

    virtual ~priority_queue() {}

    // EFFECTS: Add a new element to the heap.
    // MODIFIES: this
    // RUNTIME: O(n) - some implementations *must* have tighter bounds (see
    //           specialized headers).
    virtual void enqueue(const TYPE &val) = 0;

    // EFFECTS: Remove and return the smallest element from the heap.
    // REQUIRES: The heap is not empty.
    //           Note: We will not run tests on your code that would require it
    //           to dequeue an element when the heap is empty.
    // MODIFIES: this
    // RUNTIME: O(n) - some implementations *must* have tighter bounds (see
    //           specialized headers).
    virtual TYPE dequeue_min() = 0;

    // EFFECTS: Return the smallest element of the heap.
    // REQUIRES: The heap is not empty.
    // RUNTIME: O(n) - some implementations *must* have tighter bounds (see
    //           specialized headers).
    virtual const TYPE &get_min() const = 0;
```

```
40
41      // EFFECTS: Get the number of elements in the heap.
42      // RUNTIME: O(1)
43      virtual size_type size() const = 0;
44
45      // EFFECTS: Return true if the heap is empty.
46      // RUNTIME: O(1)
47      virtual bool empty() const = 0;
48
49  };
50
51  #endif //PRIORITY_QUEUE_H
```

binary_heap.h

```
1   #ifndef BINARY_HEAP_H
2   #define BINARY_HEAP_H
3
4   #include "priority_queue.h"
5   #include <algorithm>
6
7   // OVERVIEW: A specialized version of the 'heap' ADT implemented as a
        binary
8   //           heap.
9   template <typename TYPE, typename COMP = std::less<TYPE>>
10  class binary_heap : public priority_queue<TYPE, COMP>
11  {
12  public:
13      typedef unsigned size_type;
14
15      // EFFECTS: Construct an empty heap with an optional comparison
            functor.
16      //          See test_heap.cpp for more details on functor.
17      // MODIFIES: this
18      // RUNTIME: O(1)
19      binary_heap(COMP comp = COMP());
20
21      // EFFECTS: Add a new element to the heap.
22      // MODIFIES: this
23      // RUNTIME: O(log(n))
24      virtual void enqueue(const TYPE &val);
25
26      // EFFECTS: Remove and return the smallest element from the heap.
27      // REQUIRES: The heap is not empty.
28      // MODIFIES: this
29      // RUNTIME: O(log(n))
30      virtual TYPE dequeue_min();
31
32      // EFFECTS: Return the smallest element of the heap.
```

```
33      // REQUIRES: The heap is not empty.
34      // RUNTIME: O(1)
35      virtual const TYPE &get_min() const;
36
37      // EFFECTS: Get the number of elements in the heap.
38      // RUNTIME: O(1)
39      virtual size_type size() const;
40
41      // EFFECTS: Return true if the heap is empty.
42      // RUNTIME: O(1)
43      virtual bool empty() const;
44
45  private:
46      // Note: This vector *must* be used in your heap implementation.
47      std::vector<TYPE> data;
48      // Note: compare is a functor object
49      COMP compare;
50
51  private:
52      virtual void percolate_up(int id);
53      virtual void percolate_down(int id);
54
55  private:
56      // Add any additional member functions or data you require here.
57  };
58
59  template <typename TYPE, typename COMP>
60  binary_heap<TYPE, COMP>::binary_heap(COMP comp)
61  {
62      compare = comp;
63      // Fill in the remaining lines if you need.
64  }
65
66  template <typename TYPE, typename COMP>
67  void binary_heap<TYPE, COMP>::enqueue(const TYPE &val)
68  {
69      data.push_back(val);
70      percolate_up(int(size())-1);
71  }
72
73  template <typename TYPE, typename COMP>
74  TYPE binary_heap<TYPE, COMP>::dequeue_min()
75  {
76      TYPE res = data.front();
77      data[0] = data.back();
78      data.pop_back();
79      if (!empty())
80      {
81          percolate_down(0);
82      }
```

```
83      return res;
84  }
85
86  template <typename TYPE, typename COMP>
87  const TYPE &binary_heap<TYPE, COMP>::get_min() const
88  {
89      return data.front();
90  }
91
92  template <typename TYPE, typename COMP>
93  bool binary_heap<TYPE, COMP>::empty() const
94  {
95      return data.empty();
96  }
97
98  template <typename TYPE, typename COMP>
99  unsigned binary_heap<TYPE, COMP>::size() const
100 {
101     return data.size();
102 }
103
104 template <typename TYPE, typename COMP>
105 void binary_heap<TYPE, COMP>::percolate_up(int id)
106 {
107     while (id > 0 && compare(data[id],data[(id-1) / 2]))
108     {
109         TYPE tmp = data[(id-1) / 2];
110         data[(id-1) / 2] = data[id];
111         data[id] = tmp;
112         id = (id-1) / 2;
113     }
114 }
115
116 template <typename TYPE, typename COMP>
117 void binary_heap<TYPE, COMP>::percolate_down(int id)
118 {
119     for (int j = id*2 + 1; j < size(); j = id*2+1)
120     {
121         if (j < int(size())-1 && compare(data[j+1],data[j]))
122         {
123             j++;
124         }
125         if (compare(data[id],data[j]))
126         {
127             break;
128         }
129         TYPE tmp = data[id];
130         data[id] = data[j];
131         data[j] = tmp;
132         id = j;
```

```
133     }
134 }
135 #endif //BINARY_HEAP_H
```

unsorted_heap.h

```
1  #ifndef UNSORTED_HEAP_H
2  #define UNSORTED_HEAP_H
3
4  #include "priority_queue.h"
5  #include <algorithm>
6
7  // OVERVIEW: A specialized version of the 'heap' ADT that is
       implemented with
8  //           an underlying unordered array-based container. Every time
       a min
9  //           is required, a linear search is performed.
10 template <typename TYPE, typename COMP = std::less<TYPE>>
11 class unsorted_heap : public priority_queue<TYPE, COMP>
12 {
13 public:
14     typedef unsigned size_type;
15
16     // EFFECTS: Construct an empty heap with an optional comparison
           functor.
17     //          See test_heap.cpp for more details on functor.
18     // MODIFIES: this
19     // RUNTIME: O(1)
20     unsorted_heap(COMP comp = COMP());
21
22     // EFFECTS: Add a new element to the heap.
23     // MODIFIES: this
24     // RUNTIME: O(1)
25     virtual void enqueue(const TYPE &val);
26
27     // EFFECTS: Remove and return the smallest element from the heap.
28     // REQUIRES: The heap is not empty.
29     // MODIFIES: this
30     // RUNTIME: O(n)
31     virtual TYPE dequeue_min();
32
33     // EFFECTS: Return the smallest element of the heap.
34     // REQUIRES: The heap is not empty.
35     // RUNTIME: O(n)
36     virtual const TYPE &get_min() const;
37
38     // EFFECTS: Get the number of elements in the heap.
39     // RUNTIME: O(1)
40     virtual size_type size() const;
```

```
41
42      // EFFECTS: Return true if the heap is empty.
43      // RUNTIME: O(1)
44      virtual bool empty() const;
45
46  private:
47      // Note: This vector *must* be used in your heap implementation.
48      std::vector<TYPE> data;
49      // Note: compare is a functor object
50      COMP compare;
51
52  private:
53      // Add any additional member functions or data you require here.
54  };
55
56  template <typename TYPE, typename COMP>
57  unsorted_heap<TYPE, COMP>::unsorted_heap(COMP comp)
58  {
59      compare = comp;
60      // Fill in the remaining lines if you need.
61  }
62
63  template <typename TYPE, typename COMP>
64  void unsorted_heap<TYPE, COMP>::enqueue(const TYPE &val)
65  {
66      data.push_back(val);
67  }
68
69  template <typename TYPE, typename COMP>
70  TYPE unsorted_heap<TYPE, COMP>::dequeue_min()
71  {
72      auto min = data.begin();
73      for (auto it = data.begin(); it != data.end();it++)
74      {
75          if(compare((*it),(*min)))
76          {
77              min = it;
78          }
79      }
80      TYPE res = *min;
81      *min = data.back();
82      data.pop_back();
83      return res;
84  }
85
86  template <typename TYPE, typename COMP>
87  const TYPE &unsorted_heap<TYPE, COMP>::get_min() const
88  {
89      return data[0];
90  }
```

```
91
92 template <typename TYPE, typename COMP>
93 bool unsorted_heap<TYPE, COMP>::empty() const
94 {
95     return data.empty();
96 }
97
98 template <typename TYPE, typename COMP>
99 unsigned unsorted_heap<TYPE, COMP>::size() const
100 {
101     return data.size();
102 }
103
104 #endif //UNSORTED_HEAP_H
```

fib_heap.h

```
1  #ifndef FIB_HEAP_H
2  #define FIB_HEAP_H
3
4  #include "priority_queue.h"
5  #include <algorithm>
6  #include <cmath>
7
8  // OVERVIEW: A specialized version of the 'heap' ADT implemented as a
9  //           Fibonacci heap.
10 template <typename TYPE, typename COMP = std::less<TYPE>>
11 class fib_heap : public priority_queue<TYPE, COMP>
12 {
13 public:
14     typedef unsigned size_type;
15
16     // EFFECTS: Construct an empty heap with an optional comparison
       functor.
17     //          See test_heap.cpp for more details on functor.
18     // MODIFIES: this
19     // RUNTIME: O(1)
20     fib_heap(COMP comp = COMP()):compare(comp),heapSize(0),min(NULL){};
21
22     // EFFECTS: Deconstruct the heap with no memory leak.
23     // MODIFIES: this
24     // RUNTIME: O(n)
25     ~fib_heap();
26
27     // EFFECTS: Add a new element to the heap.
28     // MODIFIES: this
29     // RUNTIME: O(1)
30     virtual void enqueue(const TYPE &val);
31
```

```
32      // EFFECTS: Remove and return the smallest element from the heap.
33      // REQUIRES: The heap is not empty.
34      // MODIFIES: this
35      // RUNTIME: Amortized O(log(n))
36      virtual TYPE dequeue_min();
37
38      // EFFECTS: Return the smallest element of the heap.
39      // REQUIRES: The heap is not empty.
40      // RUNTIME: O(1)
41      virtual const TYPE &get_min() const;
42
43      // EFFECTS: Get the number of elements in the heap.
44      // RUNTIME: O(1)
45      virtual size_type size() const;
46
47      // EFFECTS: Return true if the heap is empty.
48      // RUNTIME: O(1)
49      virtual bool empty() const;
50
51  private:
52      // Note: compare is a functor object
53      COMP compare;
54
55  private:
56      // Add any additional member functions or data you require here.
57      // You may want to define a strcut/class to represent nodes in the
            heap and a
58      // pointer to the min node in the heap.
59      struct Node
60      {
61          TYPE val;
62          Node *left;
63          Node *right;
64          Node *parent;
65          Node *child;
66          int degree;
67          Node(const TYPE &_val) : val(_val)
68          {
69              left = right = this;
70              child = parent = NULL;
71              degree = 0;
72          }
73      };
74      size_type heapSize;
75      Node *min;
76
77      void consolidate();
78
79      void link(Node *y, Node *x);
80
```

```
81      void clear(Node* x);
82  };
83
84  // Add the definitions of the member functions here. Please refer to
85  // binary_heap.h for the syntax.
86  template <typename TYPE, typename COMP>
87  fib_heap<TYPE, COMP>::~fib_heap()
88  {
89  //  while (!empty())
90  //  {
91  //      dequeue_min();
92  //  }
93      clear(min);
94  }
95
96  template <typename TYPE, typename COMP>
97  void fib_heap<TYPE, COMP>::enqueue(const TYPE &val)
98  {
99      Node *node = new Node(val);
100     if (min == NULL)
101     {
102         min = node;
103     }
104     else
105     {
106         node->left = min;
107         node->right = min->right;
108         min->right->left = node;
109         min->right = node;
110         if (compare(val, min->val))
111         {
112             min = node;
113         }
114     }
115     heapSize++;
116 }
117
118 template <typename TYPE, typename COMP>
119 TYPE fib_heap<TYPE, COMP>::dequeue_min()
120 {
121     Node *z = min;
122     TYPE res = z->val;
123     while (z->child != NULL)
124     {
125         Node *p = z->child;
126         p->parent = NULL;
127         if (p == p->right)
128         {
129             z->child = NULL;
130         }
```

```
131         else
132         {
133             p->left->right = p->right;
134             p->right->left = p->left;
135             z->child = p->right;
136         }
137         p->right = min->right;
138         p->left = min;
139         min->right->left = p;
140         min->right = p;
141     }
142     heapSize--;
143     if (heapSize == 0)
144     {
145         min = NULL;
146     }
147     else
148     {
149         min = z->right;
150         z->left->right = z->right;
151         z->right->left = z->left;
152         consolidate();
153     }
154     delete z;
155     return res;
156 }
157
158 template<typename TYPE,typename COMP>
159 const TYPE &fib_heap<TYPE,COMP>::get_min() const
160 {
161     return min->val;
162 }
163
164 template<typename TYPE,typename COMP>
165 unsigned int fib_heap<TYPE,COMP>::size() const
166 {
167     return heapSize;
168 }
169
170 template<typename TYPE,typename COMP>
171 bool fib_heap<TYPE,COMP>::empty() const
172 {
173     return heapSize==0;
174 }
175
176 template <typename TYPE, typename COMP>
177 void fib_heap<TYPE, COMP>::link(Node *y, Node *x)
178 {
179     if (x->child == NULL)
180     {
```

```
181         y->left = y->right = y;
182         y->parent = x;
183         x->child = y;
184     }
185     else
186     {
187         y->left = x->child;
188         y->right = x->child->right;
189         x->child->right->left = y;
190         x->child->right = y;
191         y->parent = x;
192     }
193     x->degree++;
194 }
195
196 template <typename TYPE, typename COMP>
197 void fib_heap<TYPE, COMP>::consolidate()
198 {
199     using namespace std;
200     //int tmp_size = int(log(heapSize) / log((1 + sqrt(5)) / 2));
201     int tmp_size = int(size());
202     vector<Node *> array(tmp_size, NULL);
203     Node *itr = min;
204     while (1)
205     {
206         Node *x = itr;
207         itr = itr->right;
208         int d = x->degree;
209         while (array[d] != NULL)
210         {
211             Node *y = array[d];
212             if (compare(y->val,x->val))
213             {
214                 Node *tmp = x;
215                 x = y;
216                 y = tmp;
217             }
218             link(y, x);
219             array[d] = NULL;
220             d++;
221         }
222         array[d] = x;
223         if (itr == min)
224         {
225             break;
226         }
227     }
228     min = NULL;
229     for (auto &p : array)
230     {
```

```
231         if (p != NULL)
232         {
233             if (min == NULL)
234             {
235                 min = p;
236                 p->left = p->right = p;
237             }
238             else
239             {
240                 p->left = min;
241                 p->right = min->right;
242                 min->right->left = p;
243                 min->right = p;
244                 if (compare(p->val, min->val))
245                 {
246                     min = p;
247                 }
248             }
249         }
250     }
251 }
252
253 template <typename TYPE, typename COMP>
254 void fib_heap<TYPE, COMP>::clear(Node* x)
255 {
256     if(x == NULL)
257     {
258         return;
259     }
260     Node* p = x;
261
262     while(true)
263     {
264         Node* q = p->right;
265         clear(p->child);
266         delete p;
267         p = q;
268         if(p == x)
269         {
270             break;
271         }
272     }
273 }
274
275
276 #endif //FIB_HEAP_H
```

graph.h

```
1 #ifndef VE281LAB5_GRAPH_H
2 #define VE281LAB5_GRAPH_H
3 #include <vector>
4 #include "priority_queue.h"
5 #include "binary_heap.h"
6 #include "fib_heap.h"
7 #include "unsorted_heap.h"
8 #define FIB 1
9 #define BINARY 2
10 #define UNSORT 3
11 //#define TIME
12
13 struct myOption
14 {
15     bool verbose;
16     int implement;
17 };
18
19 struct Axis
20 {
21     int r;
22     int c;
23 };
24 class Vertex
25 {
26 public:
27     int cost;
28     int pathCost;
29     int row;
30     int column;
31     bool visited;
32     Axis prev;
33     Vertex(int _row, int _column, int _cost):row(_row),column(_column),
          cost(_cost)
34     {
35         pathCost = 0;
36         prev = {-1,-1};
37         visited = false;
38     }
39     struct compare_t
40     {
41         bool operator()(Vertex *a,Vertex *b) const
42         {
43             if (a->pathCost < b->pathCost)
44             {
45                 return true;
46             }
47             else if (a->pathCost == b->pathCost)
48             {
49                 if(a->column < b->column)
```

```
50                  {
51                      return true;
52                  }
53                  else if(a->column == b->column)
54                  {
55                      if(a->row < b->row)
56                      {
57                          return true;
58                      }
59                  }
60              }
61              return false;
62          }
63      };
64
65  };
66
67  class Graph
68  {
69  private:
70      std::vector<std::vector<Vertex>> points;
71      Axis start;
72      Axis end;
73      int shortest;
74      int mode;
75      int row_num;
76      int column_num;
77      bool verbose;
78      priority_queue<Vertex*,Vertex::compare_t> *pq;
79  public:
80      Graph(const myOption &option);
81      ~Graph();
82      void solve();
83  private:
84      void check();
85      Vertex& get_point(const Axis &a);
86      int set_neighbor(Vertex* v, const int& a, const int &b);
87      void track_path(const Axis &a);
88      void result_print();
89  };
90
91  #endif //VE281LAB5_GRAPH_H
```

graph.cpp

```
1  #include "graph.h"
2  #include <iostream>
3  std::ostream &operator<<(std::ostream &os, const Axis &axis)
4  {
```

```
5       return os << "(" << axis.c << ", " << axis.r << ")";
6   }
7
8   std::ostream &operator<<(std::ostream &os, const Vertex &v)
9   {
10      return os << "(" << v.column << ", " << v.row << ")" << " with
            accumulated length " << v.pathCost ;
11  }
12
13  Graph::Graph(const myOption &option){
14      using namespace std;
15      cin >> column_num;
16      cin >> row_num;
17      int st_r = 0, st_c = 0;
18      cin >> st_r;
19      cin >> st_c;
20      int ed_r = 0, ed_c = 0;
21      cin >> ed_r;
22      cin >> ed_c;
23      for(int i = 0 ; i < row_num;i++)
24      {
25          points.emplace_back();
26          for(int j = 0 ; j < column_num;j++)
27          {
28              int tmp = 0;
29              cin >> tmp;
30              points[i].emplace_back(i,j,tmp);
31          }
32      }
33      start = {st_c,st_r};
34      end = {ed_c,ed_r};
35      shortest = 0;
36      switch (option.implement)
37      {
38          case BINARY:
39          {
40              pq = new binary_heap<Vertex*,Vertex::compare_t>;
41              break;
42          }
43          case UNSORT:
44          {
45              pq = new unsorted_heap<Vertex*,Vertex::compare_t>;
46              break;
47          }
48          case FIB:
49          {
50              pq = new fib_heap<Vertex*,Vertex::compare_t>;
51              break;
52          }
53          default:break;
```

```
54      }
55      verbose = option.verbose;
56  #ifdef TIME
57      cout << row_num << ",";
58  #endif
59      //check();
60  }
61
62  Graph::~Graph()
63  {
64      delete pq;
65  }
66
67  void Graph::solve()
68  {
69      using namespace std;
70      int counter = 0;
71      Vertex &s = get_point(start);
72      s.pathCost = s.cost;
73      s.visited = true;
74      pq->enqueue(&s);
75      std::string output;
76      stringstream ss;
77      while (!pq->empty())
78      {
79          if(verbose)
80          {
81              ss << "Step " << counter++ << "\n";
82          }
83          Vertex* minimum = pq->dequeue_min();
84          if(verbose)
85          {
86              ss << "Choose cell " << *minimum << ".\n";
87          }
88          if(set_neighbor(minimum,0,1,ss) == 1)
89              break;
90          if(set_neighbor(minimum,1,0,ss) == 1)
91              break;
92          if(set_neighbor(minimum,0,-1,ss) == 1)
93              break;
94          if(set_neighbor(minimum,-1,0,ss) == 1)
95              break;
96          //std::cerr << "??" << "\n";
97      }
98  #ifndef TIME
99      result_print(ss);
100     output = ss.str();
101     cout << output;
102 #endif
103 }
```

```
104
105 void Graph::check()
106 {
107     for(auto &p:points)
108     {
109         for(auto &q : p)
110         {
111             std::cout << q.cost << " ";
112         }
113         std::cout << std::endl;
114     }
115 }
116
117 Vertex& Graph::get_point(const Axis &a)
118 {
119     return points[a.r][a.c];
120 }
121
122 void Graph::result_print(std::stringstream &ss)
123 {
124     ss << "The shortest path from "
125             << start << " to " << end
126             << " is " << shortest << ".\n"
127             << "Path:" << "\n";
128     track_path(end,ss);
129 }
130
131 void Graph::track_path(const Axis &a, std::stringstream &ss)
132 {
133     Vertex &v = get_point(a);
134     if(v.prev.r >= 0)
135     {
136         track_path(v.prev, ss);
137     }
138     ss << a << "\n";
139 }
140
141 int Graph::set_neighbor(Vertex* v, const int& r, const int &c,std::
    stringstream &ss)
142 {
143     if(v->row + r >= 0 && v->row + r < row_num &&
144         v->column + c >= 0 && v->column + c < column_num)
145     {
146         Vertex &neighbor = points[v->row+r][v->column+c];
147         if(!neighbor.visited)
148         {
149             neighbor.visited = true;
150             neighbor.pathCost = neighbor.cost + v->pathCost;
151             neighbor.prev = {v->row,v->column};
152             if(v->row+r == end.r && v->column + c == end.c)
```

```
153             {
154                 if(verbose)
155                     ss << "Cell " << neighbor << " is the ending point
                            .\n";
156                 shortest = neighbor.pathCost;
157                 return 1;
158             }
159             else
160             {
161                 pq->enqueue(&neighbor);
162                 if(verbose)
163                     ss << "Cell " << neighbor << " is added into the
                            queue.\n";
164             }
165         }
166 
167     }
168     return 0;
169 
170 }
```

interface.h

```
1 #ifndef VE281LAB5_INTERFACE_H
2 #define VE281LAB5_INTERFACE_H
3 
4 #include "priority_queue.h"
5 #include "unsorted_heap.h"
6 #include "fib_heap.h"
7 #include "binary_heap.h"
8 #include "graph.h"
9 
10 myOption parseArgs(int argc, char** argv);
11 
12 #endif //VE281LAB5_INTERFACE_H
```

interface.cpp

```
1 #include "interface.h"
2 #include <cstdlib>
3 #include <cstdio>
4 #include <cstring>
5 #include <unistd.h>
6 #include <getopt.h>
7 #include <iostream>
8 //#define  DEBUG
9 myOption parseArgs(int argc, char** argv)
10 {
```

```
11      using namespace std;
12      int inputChar = 0;
13      struct option opts[] = {
14              {"implementation", 1, NULL, 'i'},
15              {"verbose", 0, NULL, 'v'}
16      };
17      myOption res;
18      res.verbose = false;
19      res.implement = -1;
20      while((inputChar = getopt_long(argc,argv,"i:v",opts,NULL)) != -1)
21      {
22          switch(inputChar)
23          {
24              case 'i':
25              {
26  #ifdef DEBUG
27                  cout << optarg << endl;
28  #endif
29                  if(!strcmp(optarg,"BINARY"))
30                  {
31                      res.implement = BINARY;
32                  }
33                  else if(!strcmp(optarg,"UNSORTED"))
34                  {
35                      res.implement = UNSORT;
36                  }
37                  else if(!strcmp(optarg,"FIBONACCI"))
38                  {
39                      res.implement = FIB;
40                  }
41                  break;
42              }
43              case 'v':
44              {
45                  res.verbose = true;
46                  break;
47              }
48          }
49      }
50      return res;
51  }
```

main.cpp

```
1 #include "interface.h"
2 #include "graph.h"
3 #include <iostream>
4 #include<time.h>
5 int main(int argc,char** argv)
```

```
6  {
7      using namespace std;
8      ios::sync_with_stdio(false);
9      cin.tie(0);
10     myOption option = parseArgs(argc,argv);
11     Graph graph(option);
12     clock_t start = clock();
13     graph.solve();
14 #ifdef TIME
15     cout << float(clock() - start)/CLOCKS_PER_SEC << "\n";
16 #else
17     cerr << "Time: " << float(clock() - start)/CLOCKS_PER_SEC << "\n";
18 #endif
19 }
```