

Lab5 Priority Queues

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

* Name: Jin Zhejian Student ID: 517370910167 Email: jinzhejian@outlook.com

1 Preparation

In order to show the relationship between runtime and grid size better, I choose the grid width 10, 20, 50, 100, 200. I generate random numbers and weight (assuming the maximum possible value of weight is 1000), each grid size for 5 different data in order to calculate the average runtime, and then save them in .in file. The source code is shown below.

```
1 #include <iostream>
2 #include <fstream>
3 #include <stdlib.h>
4
5 using namespace std;
6
7 int main() {
8     srand((int)time(0));
9     fstream fso("data_200_1.in", ios::out);
10    int size = 200;
11    int weight = 1000;
12
13    fso << size << endl;
14    fso << size << endl;
15    fso << 0 << " " << 0 << endl;
16    fso << size-1 << " " << size-1 << endl;
17    int i;
18    for (i = 0; i < size*size; ++i)
19        fso << rand()%weight + 1 << ' ';
20    fso.close();
21
22    return 0;
23 }
24
25
```

Listing 1: generate.cpp

Then, I calculate the average runtime, the data is shown below:

size	BINARY	UNSORTED	FIBONACCI
10	179.2	254.6	232.4
20	799.4	951.4	969.2
50	2540.2	3423.6	2439.0
100	5260.4	14270.4	6680.2
200	21713	108144	31526

2 Binary Heaps

In this lab, I implemented 3 heaps, they are: Binary Heap, Unsorted Heap, and Fibonacci Heap. Then, I use the three heaps I implemented to solve a finding shortest path problem. I choose the grid width 10, 20, 50, 100, 200, assuming the weight of each point is positive number and doesn't exceed 1000, and I record each of the run times of the 3 heaps for each of the grid size.

I choose clocks as the time of unit.

The graph are shown as follows:

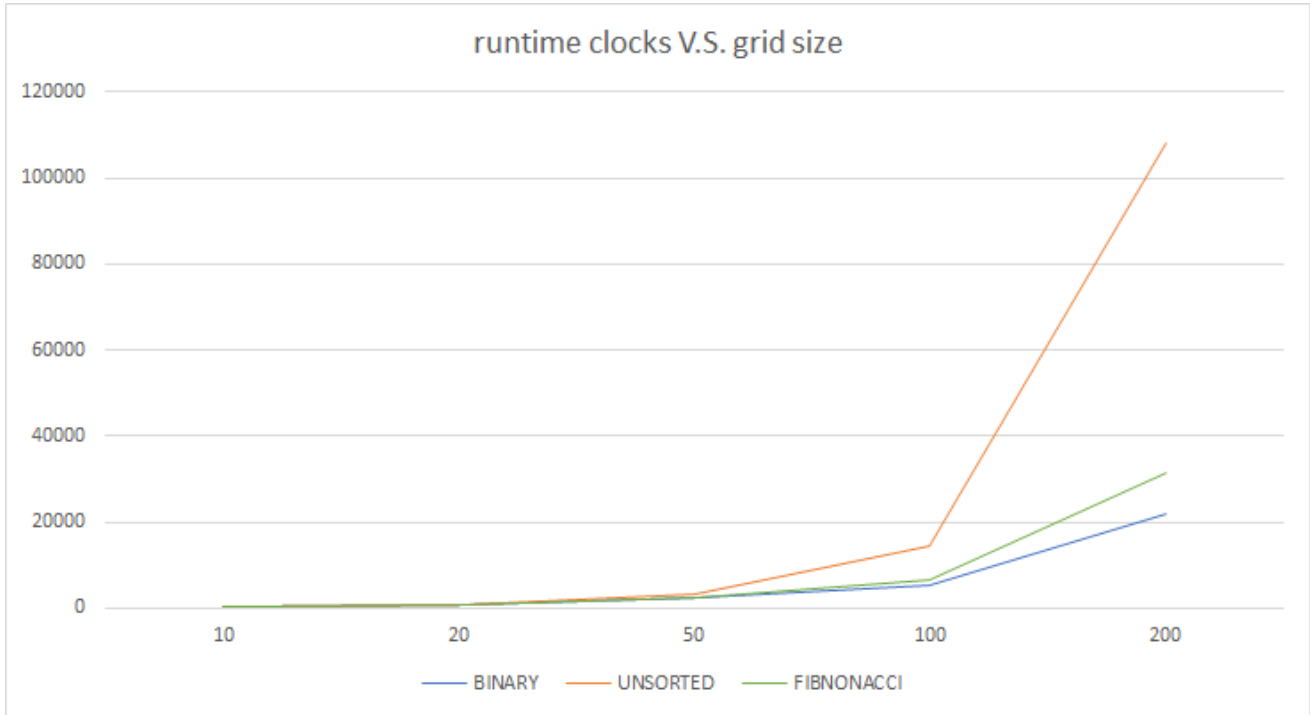


Figure 1: runtime (clocks) of 3 heaps versus the array size

From the figure, we can find that the runtime of Unsorted Heap is slower than that of Binary Heap and Fibonacci Heap. However, the run time of Fibonacci Heap is a little slower than that of Binary Heap, which contradicts to the theoretical knowledge.

Theoretically, the time complexity of **enqueue** and **dequeue_min** of Fibonacci Heap is: $O(1)$ and amortized $O(\log n)$, while the time complexity of **enqueue** and **dequeue_min** of Binary Heap is $O(\log n)$ and $O(\log n)$.

One possible reason is that: the number of my Fibonacci Heap is too complex, for example, the class **node** in my implementation contains 7 private members (`node* prev`; `node* next`; `node* child`; `node* parent`; `TYPE value`; `int degree`; `bool marked`;), which cost much when doing assignment work. Also, the **std::vector** type of private member in my Binary Heap provides an efficient way to assign / delete values.

3 Appendix

priority_queue.h:

```
1 #ifndef PRIORITY_QUEUE_H
2 #define PRIORITY_QUEUE_H
3
4 #include <functional>
5 #include <vector>
6
7 // OVERVIEW: A simple interface that implements a generic heap.
8 //           Runtime specifications assume constant time comparison and
9 //           copying. TYPE is the type of the elements stored in the priority
10 //           queue. COMP is a functor, which returns the comparison result of
11 //           two elements of the type TYPE. See test_heap.cpp for more details
12 //           on functor.
13 template<typename TYPE, typename COMP = std::less<TYPE> >
14 class priority_queue {
15 public:
16     typedef unsigned size_type;
17
18     virtual ~priority_queue() {}
19
20     // EFFECTS: Add a new element to the heap.
21     // MODIFIES: this
22     // RUNTIME: O(n) – some implementations *must* have tighter bounds (see
23     //           specialized headers).
24     virtual void enqueue(const TYPE &val) = 0;
25
26     // EFFECTS: Remove and return the smallest element from the heap.
27     // REQUIRES: The heap is not empty.
28     //           Note: We will not run tests on your code that would require it
29     //           to dequeue an element when the heap is empty.
30     // MODIFIES: this
31     // RUNTIME: O(n) – some implementations *must* have tighter bounds (see
32     //           specialized headers).
33     virtual TYPE dequeue_min() = 0;
34
35     // EFFECTS: Return the smallest element of the heap.
36     // REQUIRES: The heap is not empty.
37     // RUNTIME: O(n) – some implementations *must* have tighter bounds (see
38     //           specialized headers).
39     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 #endif //PRIORITY_QUEUE_H
51
52
53
```

Listing 2: priority_queue.h

binary_heap.h:

```
1 #ifndef BINARY_HEAP_H
2 #define BINARY_HEAP_H
3
4 #include <algorithm>
5 #include <cmath>
6 #include "priority_queue.h"
7 using namespace std;
8 // OVERVIEW: A specialized version of the 'heap' ADT implemented as a binary
9 //           heap.
10 template<typename TYPE, typename COMP = std::less<TYPE> >
11 class binary_heap: public priority_queue<TYPE, COMP> {
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     // virtual void print() const;
46
47 private:
48     // Note: This vector *must* be used in your heap implementation.
49     std::vector<TYPE> data;
50     COMP compare;
51 private:
52     size_type Size;
53     void percolateUp(int id);
54     void percolateDown(int id);
55 };
56
57
58 template<typename TYPE, typename COMP>
59 binary_heap<TYPE, COMP> :: binary_heap(COMP comp) : Size(0) {
```

```

60 compare = comp;
61 data.push_back(TYPE());
62 }
63
64
65 template<typename TYPE, typename COMP>
66 void binary_heap<TYPE, COMP> :: percolateUp(int id){
67 while (id > 1 && compare(data[id], data[id/2]) ){
68 swap(data[id], data[id/2]);
69 id = id/2;
70 }
71 }
72
73 template<typename TYPE, typename COMP>
74 void binary_heap<TYPE, COMP> :: percolateDown(int id){
75 for (size_type j = 2*id; j <= Size ; j = 2*id) {
76 if (j < Size && compare(data[j+1] , data[j]) ) j++;
77 if (!compare(data[j], data[id])) break;
78 swap(data[id], data[j]);
79 id = j;
80 }
81 }
82
83 template<typename TYPE, typename COMP>
84 void binary_heap<TYPE, COMP> :: enqueue(const TYPE &val) {
85 Size = Size + 1;
86 data.push_back(val);
87 percolateUp(Size);
88 }
89
90 template<typename TYPE, typename COMP>
91 TYPE binary_heap<TYPE, COMP> :: dequeue_min() {
92 swap(data[1], data[Size--]);
93 percolateDown(1);
94 TYPE tmp = data[Size+1];
95 data.pop_back();
96 return tmp;
97 }
98
99 template<typename TYPE, typename COMP>
100 const TYPE &binary_heap<TYPE, COMP> :: get_min() const {
101 return data[1];
102 }
103
104 template<typename TYPE, typename COMP>
105 bool binary_heap<TYPE, COMP> :: empty() const {
106 return (this->Size == 0);
107 }
108
109 template<typename TYPE, typename COMP>
110 unsigned binary_heap<TYPE, COMP> :: size() const {
111 return this->Size;
112 }
113
114 #endif //BINARY_HEAP_H
115

```

Listing 3: binary_heap.h

unsorted_heap.h:

```
1  #ifndef UNSORTED_HEAP_H
2  #define UNSORTED_HEAP_H
3
4  #include <algorithm>
5  #include "priority_queue.h"
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 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     unsorted_heap(COMP comp = COMP());
20
21     // ~unsorted_heap(){
22     //     while(!data.empty()){
23     //         data.pop_back();
24     //     }
25     // }
26     // EFFECTS: Add a new element to the heap.
27     // MODIFIES: this
28     // RUNTIME: O(1)
29     virtual void enqueue(const TYPE &val);
30
31     // EFFECTS: Remove and return the smallest element from the heap.
32     // REQUIRES: The heap is not empty.
33     // MODIFIES: this
34     // RUNTIME: O(n)
35     virtual TYPE dequeue_min();
36
37     // EFFECTS: Return the smallest element of the heap.
38     // REQUIRES: The heap is not empty.
39     // RUNTIME: O(n)
40     virtual const TYPE &get_min() const;
41
42     // EFFECTS: Get the number of elements in the heap.
43     // RUNTIME: O(1)
44     virtual size_type size() const;
45
46     // EFFECTS: Return true if the heap is empty.
47     // RUNTIME: O(1)
48     virtual bool empty() const;
49
50 private:
51     // Note: This vector *must* be used in your heap implementation.
52     std::vector<TYPE> data;
53     // Note: compare is a functor object
54     COMP compare;
55 private:
56     // Add any additional member functions or data you require here.
57     size_type Size;
58 };
59
```

```

60  template<typename TYPE, typename COMP>
61  unsorted_heap<TYPE, COMP> :: unsorted_heap(COMP comp):Size(0) {
62  compare = comp;
63  // Fill in the remaining lines if you need.
64  data.push_back(TYPE());
65  }
66
67
68  template<typename TYPE, typename COMP>
69  void unsorted_heap<TYPE, COMP> :: enqueue(const TYPE &val) {
70  Size = Size +1;
71  data.push_back(val);
72  }
73
74  template<typename TYPE, typename COMP>
75  TYPE unsorted_heap<TYPE, COMP> :: dequeue_min() {
76  size_type minIndex = 1;
77  const TYPE *tmp = &data[1];
78  // find the index of the smallest one
79  for (size_type i = 2; i <= Size ; ++i) {
80  if (compare(data[i], *tmp)) {
81  minIndex = i;
82  tmp = &data[i];
83  }
84  }
85  TYPE min = data[minIndex];
86  // move the data after the smallest one forward
87  if (minIndex < Size) {
88  for (size_type j = minIndex; j < Size; ++j) {data[j] = data[j + 1];}
89  }
90  data.pop_back();
91  Size = Size -1;
92  return min;
93  }
94  template<typename TYPE, typename COMP>
95  const TYPE &unsorted_heap<TYPE, COMP> :: get_min() const {
96  const TYPE *tmp = &data[1];
97  for (size_type i = 2; i <= Size ; ++i) {
98  if (compare(data[i], *tmp)) tmp = &data[i];
99  }
100 return *tmp;
101 }
102 template<typename TYPE, typename COMP>
103 bool unsorted_heap<TYPE, COMP> :: empty() const {
104 return (this->Size == 0);
105 }
106
107 template<typename TYPE, typename COMP>
108 unsigned unsorted_heap<TYPE, COMP> :: size() const {
109 return this->Size;
110 }
111
112 #endif //UNSORTED_HEAP_H
113

```

Listing 4: unsorted_heap.h

fib_heap.h:

```
1  #ifndef FIB_HEAP_H
2  #define FIB_HEAP_H
3
4  #include <algorithm>
5  #include <cmath>
6  #include <list>
7  #include "priority_queue.h"
8  typedef unsigned size_type;
9  template<typename TYPE, typename COMP= std::less<TYPE>>
10 class fib_heap: public priority_queue<TYPE, COMP> {
11     class node {
12     private:
13     node* prev;
14     node* next;
15     node* child;
16     node* parent;
17     TYPE value;
18     int degree;
19     bool marked;
20     public:
21     friend class fib_heap<TYPE,COMP>;
22     };
23
24     protected:
25     node* heap;
26     public:
27     fib_heap(COMP comp = COMP()) {
28     compare = comp;
29     heap = nullptr;
30     }
31
32     ~fib_heap() {
33     if(heap) {
34     deleteAll(heap);
35     }
36     }
37
38     virtual void enqueue(const TYPE &val) {
39     node* ret=_singleton(val);
40     heap=_merge(heap,ret);
41     }
42
43     virtual TYPE dequeue_min() {
44     node* old=heap;
45     heap=_removeMinimum(heap);
46     TYPE ret=old->value;
47     delete old;
48     return ret;
49     }
50
51     virtual const TYPE &get_min() const {
52     return heap->value;
53     }
54     virtual size_type size() const {
55     return 0;
56     }
57     virtual bool empty() const {
58     return heap == nullptr;
59     }
```



```

60
61 private:
62 COMP compare;
63 node* __singleton(TYPE value) {
64     node* n=new node;
65     n->value=value;
66     n->prev=n->next=n;
67     n->degree=0;
68     n->marked=false;
69     n->child=NULL;
70     n->parent=NULL;
71     return n;
72 }
73
74 void deleteAll(node* n) {
75     if(n!=NULL) {
76         node* c=n;
77         do {
78             node* d=c;
79             c=c->next;
80             deleteAll(d->child);
81             delete d;
82         } while(c!=n);
83     }
84 }
85
86 node* __merge(node* a,node* b) {
87     if(a==NULL)return b;
88     if(b==NULL)return a;
89     if(compare(b->value, a->value)) {
90         node* temp=a;
91         a=b;
92         b=temp;
93     }
94     node* an=a->next;
95     node* bp=b->prev;
96     a->next=b;
97     b->prev=a;
98     an->prev=bp;
99     bp->next=an;
100    return a;
101 }
102
103 void __addChild(node* parent,node* child) {
104     child->prev=child->next=child;
105     child->parent=parent;
106     parent->degree++;
107     parent->child=__merge(parent->child, child);
108 }
109
110 void __unMarkAndUnParentAll(node* n) {
111     if(n==NULL)return;
112     node* c=n;
113     do {
114         c->marked=false;
115         c->parent=NULL;
116         c=c->next;
117     } while(c!=n);
118 }
119

```

```

120 node* _removeMinimum(node* n) {
121     _unMarkAndUnParentAll(n->child);
122     if(n->next==n) {
123         n=n->child;
124     } else {
125         n->next->prev=n->prev;
126         n->prev->next=n->next;
127         n=_merge(n->next, n->child);
128     }
129     if(n==NULL) return n;
130     node* trees[64]={NULL};
131
132     while(true) {
133         if(trees[n->degree]!=NULL) {
134             node* t=trees[n->degree];
135             if(t==n) break;
136             trees[n->degree]=NULL;
137             if(compare(n->value, t->value)) {
138                 t->prev->next=t->next;
139                 t->next->prev=t->prev;
140                 _addChild(n, t);
141             } else {
142                 t->prev->next=t->next;
143                 t->next->prev=t->prev;
144                 if(n->next==n) {
145                     t->next=t->prev=t;
146                     _addChild(t, n);
147                     n=t;
148                 } else {
149                     n->prev->next=t;
150                     n->next->prev=t;
151                     t->next=n->next;
152                     t->prev=n->prev;
153                     _addChild(t, n);
154                     n=t;
155                 }
156             }
157             continue;
158         } else {
159             trees[n->degree]=n;
160         }
161         n=n->next;
162     }
163     node* min=n;
164     node* start=n;
165     do {
166         if(compare(n->value, min->value)) min=n;
167         n=n->next;
168     } while(n!=start);
169     return min;
170 }
171 };
172
173 #endif //FIB_HEAP_H
174

```

Listing 5: fib_heap.h

main.cpp:

```
1  #include <iostream>
2  #include <fstream>
3  #include <cstring>
4  #include "priority_queue.h"
5  #include "binary_heap.h"
6  #include "fib_heap.h"
7  #include "unsorted_heap.h"
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <getopt.h>
11 #include <time.h>
12
13 using namespace std;
14
15 struct Cell{
16     int x = 0;
17     int y = 0;
18     int weight = 0;
19     int isReached = 0;
20     int index = 0;
21     int pathcost = 0;
22     int predecessor_index = 0;
23 };
24
25 struct World{
26     int width;
27     int height;
28     Cell Start; //only used for storing the start point of x and y, will not be
                //modified after input
29     Cell End;
30     vector <Cell> grid;
31 };
32
33 struct compare_t
34 {
35     bool operator()(Cell a, Cell b) const
36     {
37         if (a.pathcost != b.pathcost) return a.pathcost < b.pathcost;
38         else{
39             if (a.x != b.x) return a.x < b.x;
40             else return a.y < b.y;
41         }
42     }
43 };
44
45
46 void argAnalysis(int argc, char** argv, int &impl_index, bool &vFlag){
47     int opt = 0, option_index = 0;
48     const char* iArg = ""; //store the implementation argument
49     vector<const char*> iName {"BINARY", "UNSORTED", "FIBONACCI"}; //implementation
                    //name tag
50
51     const char *optstring = "i:v";
52     static struct option long_options[] = {
53         {"implementation", required_argument, nullptr, 'i'},
54         {"verbose", no_argument, nullptr, 'v'}
55     };
56
57     while ((opt = getopt_long(argc, argv, optstring, long_options, &option_index)) !=
```

```

-1) {
58 switch (opt){
59 case 'i':
60 if (optarg) iArg = optarg;
61 break;
62 case 'v':
63 vFlag = true;
64 break;
65 case '?':
66 cout << "error optopt: " << optopt << endl;
67 cout << "error opterr: " << opterr << endl;
68 break;
69 default:
70 break;
71 }
72 }
73 for(int i = 0; i < 3; i++){
74 if(strcmp(iArg, iName[i]) == 0) impl_index = i;
75 }
76 }
77
78
79
80 int main(int argc, char* argv[]) {
81
82 ios::sync_with_stdio(false);
83 cin.tie(nullptr);
84
85 bool OutputMode = false; // 0 for brief, 1 for verbose
86 int ImplementWay = 0; // 0 for BINARY, 1 for UNSORTED, 2 for FIBONACCI
87
88 argAnalysis(argc, argv, ImplementWay, OutputMode);
89
90 //initialize our priority_queue and the world
91 priority_queue<Cell, compare_t> *PQ = nullptr;
92 switch (ImplementWay){
93 case 0 :
94 PQ = new binary_heap<Cell, compare_t>;
95 break;
96 case 1 :
97 PQ = new unsorted_heap<Cell, compare_t>;
98 break;
99 case 2:
100 PQ = new fib_heap<Cell, compare_t>;
101 break;
102 default :
103 cout << "Wrong input for implementation way."<< endl;
104 return 0;
105 }
106
107 vector <Cell> Path;
108 World world = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
109
110 // input & initialization
111 cin >> world.width >> world.height >> world.Start.x >> world.Start.y >> world.End
    .x >> world.End.y;
112
113 world.Start.index = world.Start.x + world.width*world.Start.y;
114 world.End.index = world.End.x + world.width*world.End.y;
115

```

```

116 int size = world.width * world.height;
117 Cell cellTmp = {0,0,0,0,0,0,0};
118
119 for (int j = 0; j < world.height; ++j) {
120     for (int i = 0; i < world.width; ++i) {
121         cellTmp.x = i;
122         cellTmp.y = j;
123         cellTmp.index = cellTmp.x + world.width*cellTmp.y;
124         cin >> cellTmp.weight;
125         world.grid.push_back(cellTmp);
126     }
127 }
128
129 // find the shortest path
130 Cell &start_point = world.grid[world.Start.index];
131 Cell &end_point = world.grid[world.End.index];
132
133 start_point.pathcost = start_point.weight;
134 start_point.isReached = 1;
135 PQ->enqueue(world.grid[world.Start.index]);
136 int step = 0, isEnd = 0;
137 while ((!PQ->empty()) && ! isEnd) {
138     Cell C = PQ->dequeue_min();
139     Cell *N = nullptr;
140     if (OutputMode == 1) {
141         cout << "Step " << step << endl;
142         cout << "Choose cell (" << C.x << ", " << C.y << ") with accumulated length " << C.
143             pathcost << "." << endl;
144     }
145     // for each neighbor N of C that has not been reached
146     for (int i = 0; i < 4; ++i) {
147         // right
148         if (! isEnd && i == 0 && (C.index < size) && (world.grid[C.index + 1].isReached ==
149             0)) {
150             N = &world.grid[C.index + 1];
151         }
152         // down
153         else if (! isEnd && i == 1 && (C.y != (world.height - 1)) && (world.grid[C.index +
154             world.width].isReached == 0)) {
155             N = &world.grid[C.index + world.width];
156         }
157         // left
158         else if (! isEnd && i == 2 && (C.x != 0) && world.grid[C.index - 1].isReached ==
159             0) {
160             N = &world.grid[C.index - 1];
161         }
162         // up
163         else if (! isEnd && i == 3 && (C.y != 0) && (world.grid[C.index - world.width].
164             isReached == 0)) {
165             N = &world.grid[C.index - world.width];
166         } else N = nullptr;
167
168         if (N != nullptr) {
169             N->pathcost = C.pathcost + N->weight;
170             N->isReached = 1;
171             N->predecessor_index = C.index;
172
173             if ( end_point.x == N->x && end_point.y == N->y) {
174                 isEnd = 1;
175                 // trace_back_path(): save the path into the Path vector

```

```

171 while ((N->x != start_point.x) || (N->y != start_point.y)){
172     Path.push_back(*N);
173     N = &world.grid[N->predecessor_index];
174 }
175 }
176 //for the output information
177 if (OutputMode == 1&& !isEnd && ((N->x != start_point.x) || (N->y != start_point.y
178 ))) {
179     cout << "Cell (" << N->x << ", " << N->y << ") with accumulated length " << N->
180         pathcost << " is added into the queue." << endl;
181 }
182 else if (OutputMode == 1&& isEnd){
183     cout << "Cell (" << world.End.x << ", " << world.End.y << ") with accumulated
184         length " << Path[0].pathcost << " is the ending point." << endl;
185 }
186 if( !isEnd) PQ->enqueue(*N);
187 }
188 }
189 //print
190 cout << "The shortest path from (" << start_point.x << ", " << start_point.y << ")
191     to ("
192 << end_point.x << ", " << end_point.y << ") is " << Path[0].pathcost << "." << endl;
193 cout << "Path:" << endl;
194 cout << "(" << start_point.x << ", " << start_point.y << ")" << endl;
195
196 for (unsigned long j = Path.size() - 1; j >= 1; --j) {
197     cout << "(" << Path[j].x << ", " << Path[j].y << ")" << endl;
198 }
199 cout << "(" << Path[0].x << ", " << Path[0].y << ")" << endl;
200
201 //free the memory
202 delete PQ;
203 vector<Cell>().swap(Path);
204 {
205     vector <Cell> tmp;
206     Path.swap(tmp);
207 }
208
209 return 0;
210 }
211
212

```

Listing 6: main.cpp