VE281 Priority Queues Comparison Report

Xinhao Liao 516370910037

November 2018

1 Introduction Background

Given a rectangular grid of cells. Each cell has a number indicating its weight, which is the cost of passing through the cell. We can use priority queue to find the shortest path from the source cell to the ending cell. In this project, the problem is solved with 3 different types of priority queue, the binary heap, the unsorted heap, and the Fibonacci heap. The performances are compared with different size of the grid.

The binary heap, the unsorted heap, and the Fibonacci heap are respectively implemented in *binary_heap.h*, *unsorted_heap.h*, and *fib_heap.h* as shown in Appendix A.

The algorithm used to solve the problem is pretty similar to the Lee's wire routing algorithm. And it's implemented in the function *implement* in *compare.cpp* shown in Appendix B. Notice that the procedure of tracing back to obtain the path is omitted for the comparison. That is, only the time cost for finding the endpoint is measured.

For the ease of comparison, we generate grids with the width equal to the height. The start points are ste to be the at the left-top corner, and the end points are set to be at the right-bottom corner. We respectively check the time cost for the size of 4×4, 16×16, 64×64, 256×256, and 1024×1024. For each size, 5 grids with different random weight distribution and generated, and the average time cost is measured. The implementation is shown in *compare.cpp* in Appendix B.

2 Result

Size	4×4	16×16	64×64	256×256	1024×1024
BINARY	0.0274ms	0.0302ms	0.4006ms	7.0772ms	148.243ms
UNSORTED	0.0112ms	0.0354ms	1.5446ms	82.7834ms	5402.45ms
FIBONACCI	0.0164ms	0.2066ms	3.254ms	57.0634ms	1086.62ms

Table 1: Average time cost for different heaps with various sizes.

The measured average time for finding a shortest path for the grids with different size is shown in Table 1. The weights are integers randomly generated in the range $[1, 99]$. And we can plot all three curves corresponding to the three types of priority queues in the same figure in Figure 1. The curves are plotted in log scales.

The results are porduced in Linux Ubuntu18.04 running in VMware Workstation 14.x virtual machine distributed with 4GB memory and 3 processors, with Intel CORE i9.

According to the results shown above, when the size is small,the three types of priority queues show similar performances. However, when the size is big enough, the binary heap shows the best performance, and the unsorted heap shows the worst performance.

3 Discussion and analysis

When the size is small, the three types of priority queues have similar performance. But as the size increases, the time needed for the unsorted heap increases much faster than the other 2 types of priority queues.

Analyzing the time complexity for the tree types of priority queues, we respectively consider inserting an element and extracting the minimum element. Assume there are n elements in the heap. For inserting, binary

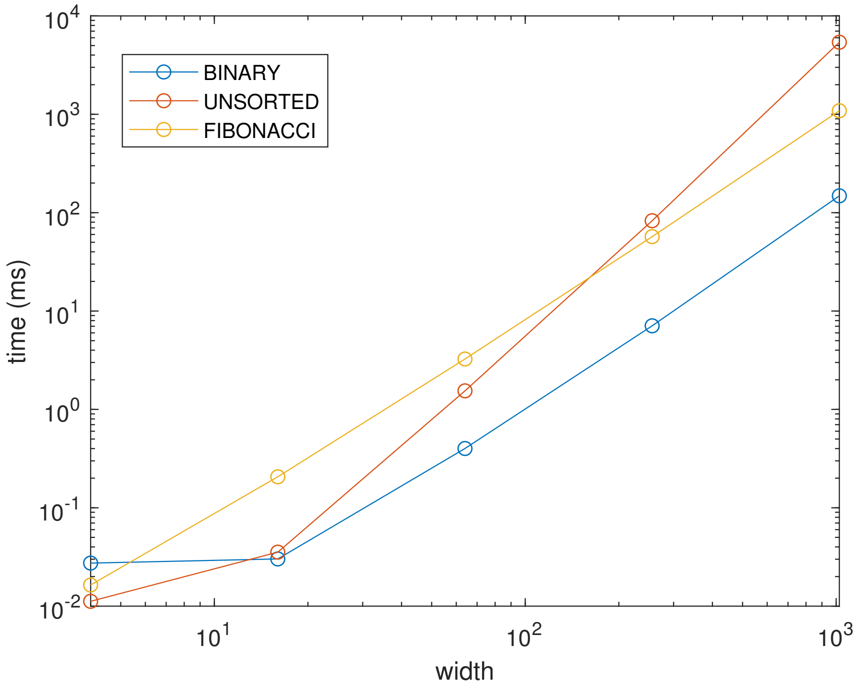

Figure 1: Curves corresponding to the 3 selecting algorithms.

heap requires $O(logn)$ time for the worst case, and the unsorted heap and Fibonacci heap both require $O(1)$ time. To extract the minimum element, binary heap requires $O(logn)$ time for the worst case, and that's $O(n)$ time for the unsorted heap for the worst case. For Fibonacci heap, it needs $O(logn)$ time on average.

Then for the whole algorithm, assume that altogether there are N steps. For each step, we need to insert no more than 4 elements and extract 1 minimum element. Then in step m, there are $O(m)$ elements in the heap. Inserting and extracting require $O(logm)$ time for binary heap and Fibonacci heap, and $O(m)+O(1) = O(m)$ time for unsorted heap. Overall, binary heap and Fibonacci heap need $O(NlogN)$ time, and unsorted heap need $O(N^2)$ time. That conforms to the result that unsorted heap shows the worst performance.

For the comparison of Binary heap and Fibonacci heap, Fibonacci heap will have better performance if there are much more procedures of inserting than extracting and deleting elements. But for this particular problem, for every step, there are no more than 4 elements to insert and 1 minimum element to extract. The number of procedures of inserting and deleting are close to each other. And due to the more complex implementation of Fibonacci heap, it has a large constant factor omitted. That's why binary heap shows a better performance than Fibonacci heap in this problem.

4 Appendix A. Three types of priority queues implementation

4.1 Priority queue

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
```

```cpp
//            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;
  // EFFECTS: Get the number of elements in the heap.
  // RUNTIME: O(1)
  virtual size_type size() const = 0;
  // EFFECTS: Return true if the heap is empty.
  // RUNTIME: O(1)
  virtual bool empty() const = 0;
};

#endif //PRIORITY_QUEUE_H
```

4.2 Binary heap

binary_heap.h

```cpp

#ifndef BINARY_HEAP_H
#define BINARY_HEAP_H

#include <algorithm>
#include "priority_queue.h"



// OVERVIEW: A specialized version of the 'heap' ADT implemented as a binary
//            heap.
template<typename TYPE, typename COMP = std::less<TYPE> >
class binary_heap: public priority_queue<TYPE, COMP> {
public:
  typedef unsigned size_type;
  // EFFECTS: Construct an empty heap with an optional comparison functor.
  //           See test_heap.cpp for more details on functor.
  // MODIFIES: this
  // RUNTIME: O(1)
  binary_heap(COMP comp = COMP());
  // EFFECTS: Add a new element to the heap.
  // MODIFIES: this
  // RUNTIME: O(log(n))
  virtual void enqueue(const TYPE &val);
  // EFFECTS: Remove and return the smallest element from the heap.
  // REQUIRES: The heap is not empty.
  // MODIFIES: this
  // RUNTIME: O(log(n))
  virtual TYPE dequeue_min();
```

```
  // EFFECTS: Return the smallest element of the heap.
  // REQUIRES: The heap is not empty.
  // RUNTIME: O(1)
  virtual const TYPE &get_min() const;
  // EFFECTS: Get the number of elements in the heap.
  // RUNTIME: O(1)
  virtual size_type size() const;
  // EFFECTS: Return true if the heap is empty.
  // RUNTIME: O(1)
  virtual bool empty() const;
private:
  // Note: This vector *must* be used in your heap implementation.
  std::vector<TYPE> data;
  // Note: compare is a functor object
  COMP compare;
private:
  // EFFECTS: Swap 2 elements with the given index.
  void swap(size_type id1, size_type id2);
  void percolateUp(size_type id);
  void percolateDown(size_type id);
};

template<typename TYPE, typename COMP>
void binary_heap<TYPE, COMP> :: swap(size_type id1, size_type id2) {
    TYPE temp = data[id1];
    data[id1] = data[id2];
    data[id2] = temp;
}


template<typename TYPE, typename COMP>
binary_heap<TYPE, COMP> :: binary_heap(COMP comp) {
    compare = comp;
    std::vector<TYPE> v;
    data = v;
}


template<typename TYPE, typename COMP>
void binary_heap<TYPE, COMP> :: percolateUp(size_type id) {
    while (id>0 && compare(data[id],data[(id-1)/2])){
    swap(id, (id-1)/2);
    id = (id-1)/2;
  }
}


template<typename TYPE, typename COMP>
void binary_heap<TYPE, COMP> :: percolateDown(size_type id) {
    size_type j;
    for(j=2*id+1; j<data.size(); j=2*id+1){
    if(j < (data.size()-1) && compare(data[j+1],data[j])) j++;
    if(compare(data[id],data[j])) break;
    swap(id,j);
    id = j;
  }
}


template<typename TYPE, typename COMP>
void binary_heap<TYPE, COMP> :: enqueue(const TYPE &val) {
    data.push_back(val);
    percolateUp(data.size()-1);

}


template<typename TYPE, typename COMP>
TYPE binary_heap<TYPE, COMP> :: dequeue_min() {
    TYPE result = data[0];
    data[0] = data[data.size()-1];
```

```
    data.pop_back();
    percolateDown(0);
    return result;
}


template<typename TYPE, typename COMP>
const TYPE &binary_heap<TYPE, COMP> :: get_min() const {
    return data[0];
}


template<typename TYPE, typename COMP>
bool binary_heap<TYPE, COMP> :: empty() const {
    return data.size() == 0;
}


template<typename TYPE, typename COMP>
unsigned binary_heap<TYPE, COMP> :: size() const {
    return data.size();
}

#endif //BINARY_HEAP_H
```

4.3 Unsorted heap

unsorted_heap.h

```

#ifndef UNSORTED_HEAP_H
#define UNSORTED_HEAP_H

#include <algorithm>
#include "priority_queue.h"


// OVERVIEW: A specialized version of the 'heap' ADT that is implemented with
//           an underlying unordered array-based container. Every time a min
//           is required, a linear search is performed.
template<typename TYPE, typename COMP = std::less<TYPE> >
class unsorted_heap: public priority_queue<TYPE, COMP> {
public:
  typedef unsigned size_type;
  // EFFECTS: Construct an empty heap with an optional comparison functor.
  //          See test_heap.cpp for more details on functor.
  // MODIFIES: this
  // RUNTIME: O(1)
  unsorted_heap(COMP comp = COMP());
  // EFFECTS: Add a new element to the heap.
  // MODIFIES: this
  // RUNTIME: O(1)
  virtual void enqueue(const TYPE &val);
  // EFFECTS: Remove and return the smallest element from the heap.
  // REQUIRES: The heap is not empty.
  // MODIFIES: this
  // RUNTIME: O(n)
  virtual TYPE dequeue_min();
  // EFFECTS: Return the smallest element of the heap.
  // REQUIRES: The heap is not empty.
  // RUNTIME: O(n)
  virtual const TYPE &get_min() const;
  // EFFECTS: Get the number of elements in the heap.
  // RUNTIME: O(1)
  virtual size_type size() const;
  // EFFECTS: Return true if the heap is empty.
  // RUNTIME: O(1)
  virtual bool empty() const;

```

```cpp

private:
  // Note: This vector *must* be used in your heap implementation.
  std::vector<TYPE> data;
  // Note: compare is a functor object
  COMP compare;

private:
  // EFFECTS: Find the minimum element
  size_type findMin() const;
};



template<typename TYPE, typename COMP>
unsigned unsorted_heap<TYPE, COMP> :: findMin() const {
    unsigned min_index = 0;
    for(size_type i = 0; i<data.size(); i++){
    if(compare(data[i], data[min_index])){
      min_index = i;
    }
  }
  return min_index;
}


template<typename TYPE, typename COMP>
unsorted_heap<TYPE, COMP> :: unsorted_heap(COMP comp) {
    compare = comp;
    std::vector<TYPE> v;
    data = v;
}


template<typename TYPE, typename COMP>
void unsorted_heap<TYPE, COMP> :: enqueue(const TYPE &val) {
    data.push_back(val);
}


template<typename TYPE, typename COMP>
TYPE unsorted_heap<TYPE, COMP> :: dequeue_min() {
    unsigned min_index = findMin();
    TYPE min = data[min_index];
  data.erase(data.begin()+min_index);
  return min;
}


template<typename TYPE, typename COMP>
const TYPE &unsorted_heap<TYPE, COMP> :: get_min() const {
    unsigned min_index = findMin();
    return data[min_index];
}


template<typename TYPE, typename COMP>
bool unsorted_heap<TYPE, COMP> :: empty() const {
    return data.size()==0;
}


template<typename TYPE, typename COMP>
unsigned unsorted_heap<TYPE, COMP> :: size() const {
    return data.size();
}

#endif //UNSORTED_HEAP_H
```

4.4 Fibonacci heap

fib_heap.h

```

#ifndef FIB_HEAP_H
#define FIB_HEAP_H

#include <algorithm>
#include <cmath>
#include "priority_queue.h"

// OVERVIEW: A specialized version of the 'heap' ADT implemented as a
//           Fibonacci heap.
template<typename TYPE, typename COMP = std::less<TYPE> >
class fib_heap: public priority_queue<TYPE, COMP> {
public:
  typedef unsigned size_type;
  // EFFECTS: Construct an empty heap with an optional comparison functor.
  //          See test_heap.cpp for more details on functor.
  // MODIFIES: this
  // RUNTIME: O(1)
  fib_heap(COMP comp = COMP());
  // EFFECTS: Deconstruct the heap with no memory leak.
  // MODIFIES: this
  // RUNTIME: O(n)
  ~fib_heap();
  // EFFECTS: Add a new element to the heap.
  // MODIFIES: this
  // RUNTIME: O(1)
  virtual void enqueue(const TYPE &val);
  // EFFECTS: Remove and return the smallest element from the heap.
  // REQUIRES: The heap is not empty.
  // MODIFIES: this
  // RUNTIME: Amortized O(log(n))
  virtual TYPE dequeue_min();
  // EFFECTS: Return the smallest element of the heap.
  // REQUIRES: The heap is not empty.
  // RUNTIME: O(1)
  virtual const TYPE &get_min() const;
  // EFFECTS: Get the number of elements in the heap.
  // RUNTIME: O(1)
  virtual size_type size() const;
  // EFFECTS: Return true if the heap is empty.
  // RUNTIME: O(1)
  virtual bool empty() const;
private:
  // Note: compare is a functor object
  COMP compare;
private:
  //A class to represent nodes in the heap
  class FibNode{
  public:
  TYPE key;
  int degree;
  FibNode * left;
  FibNode * right;
  FibNode * child;
  FibNode * parent;
  FibNode(const TYPE & val = NULL):key(val), degree(0),
             left(this),right(this),child(NULL),parent(NULL){}
  };
  int n;
  FibNode *min;
  //EFFECT: calculate the allowed max degree D(n)
  int maxdegree();
  void consolidate();
  //EFFECT: add node x to be n's sibling, that is, n's parent's child
  void addnode(FibNode* x,FibNode* n);
  //EFFECT: add node x to be y's child
  void fibHeapLink(FibNode* y,FibNode* x);
```

```
  //EFFECT: check whether x is in the array A with size given
  bool belong(FibNode* x, FibNode** A, int size);
};

template<typename TYPE, typename COMP>
fib_heap<TYPE, COMP> :: fib_heap(COMP comp) {
    compare = comp;
    n = 0;
    min = NULL;
}

template<typename TYPE, typename COMP>
fib_heap<TYPE, COMP> :: ~fib_heap() {
    while(n>0){
    dequeue_min(); // delete all the nodes in the heap
  }
}

template<typename TYPE, typename COMP>
const TYPE &fib_heap<TYPE, COMP> :: get_min() const {
    return min->key;
}

template<typename TYPE, typename COMP>
void fib_heap<TYPE, COMP> :: addnode(FibNode* x,FibNode* y) {
  x->left->right = x->right; //make the siblings has correct right and left
  x->right->left = x->left;
    x->left = y->left;
  y->left->right = x;
  x->right = y;
  y->left = x;
}

template<typename TYPE, typename COMP>
void fib_heap<TYPE, COMP> :: enqueue(const TYPE &val) {
    FibNode* x=new FibNode(val);
    if(min==NULL)
    min = x;
  else{
    addnode(x,min);
    if(compare(x->key, min->key))
      min = x;
  }
  n++;
}

template<typename TYPE, typename COMP>
TYPE fib_heap<TYPE, COMP> :: dequeue_min() {
    FibNode* z = min;
    FibNode* x = NULL;
    TYPE result = z-> key;
    //link every child of min to the root list
    while(z->child!=NULL){
    x = z->child;
    if(x->left == x)
      z->child = NULL;
    else
      z->child = x->left;
    addnode(x,min);
    x->parent = NULL;
  }
    z->left->right = z->right;
    z->right->left = z->left;
    if(z->left == z)
    min = NULL;
  else {
    min = z->left;
    consolidate();
  }
    delete z;
    n--;
```

```
  return result;
}

template<typename TYPE, typename COMP>
int fib_heap<TYPE, COMP> :: maxdegree() {
    double phi = (1.0 + sqrt(5.0))/2.0;
    int result = log(n)/log(phi);
    return result;
}

template<typename TYPE, typename COMP>
bool fib_heap<TYPE, COMP> :: belong(FibNode* x, FibNode** A, int size){
  for(int i=0; i<size; i++)
    if(x==A[i]) return 1;
  return 0;
}

template<typename TYPE, typename COMP>
void fib_heap<TYPE, COMP> :: consolidate() {
    FibNode** A = new FibNode*[maxdegree()+1];
    for(int i=0; i<=maxdegree(); i++)
    A[i] = NULL;
    FibNode* x = min;
    if(x==NULL) return;
  do{
    int d = x->degree;
    while(A[d]!=NULL){
      //if there exists a sub heap of degree d, connect them together to make a subheap of degree d+1
      FibNode* z = A[d];
      if(compare(z->key,x->key)){
        FibNode* temp = z;
        z =x;
        x = temp;
      }
      fibHeapLink(z,x);
      A[d]=NULL;
      d++;
    }
    A[d] = x;
    x = x->left;
  }while(!belong(x, A, maxdegree()+1));//visit every member in the list
  min = NULL;
  for(int i=0; i<=maxdegree();i++){
    if(A[i]!=NULL){
      if(min==NULL)
        min = A[i];
      else{
        if(compare(A[i]->key,min->key))
          min = A[i];
      }
    }
  }
  delete[] A;
}

template<typename TYPE, typename COMP>
void fib_heap<TYPE, COMP> :: fibHeapLink(FibNode* y, FibNode* x) {
  if(x->child==NULL){
    y->left->right = y->right;
    y->right->left = y->left;
    x->child = y;
    y->left = y;
    y->right = y;
  }
  else{
    addnode(y,x->child);
  }
  y->parent = x;
  x->degree++;
}
```

```
template<typename TYPE, typename COMP>
unsigned fib_heap<TYPE, COMP> :: size() const {
    return n;
}

template<typename TYPE, typename COMP>
bool fib_heap<TYPE, COMP> :: empty() const {
    return (min == NULL);
}

#endif //FIB_HEAP_H
```

5 Appendix B. Algorithm and comparing implementation

compare.cpp

```

#include <fstream>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <string>
#include "priority_queue.h"
#include "binary_heap.h"
#include "unsorted_heap.h"
#include "fib_heap.h"
using namespace std;

const int MAXSIZE=8;  //the maximum size of the arrays to be tested
const int MINSIZE=4;    //the minimum size of the arrays to be tested
const int STEP=2;
const int METHODSNUMBER=3;
const int INDEXNUMBER=5;
const string METHOSNAMES[3]={"BINARY", "UNSORTED","FIB"};

struct point{
  int y;
  int x;
  int weight;
  int pathcost;
  point* predecessor = NULL;
};


struct compare_t
{   //compare pathcost first, then x, then y
    bool operator()(point* a, point* b) const
    {
        if(a->pathcost == b->pathcost && a->x == b-> x)
      return a->y < b->y;
    else if(a->pathcost == b->pathcost)
      return a->x < b->x;
    else
      return a->pathcost < b->pathcost;
    }
};


void implement(priority_queue<point*, compare_t> * PQ, int end, point** points, bool**
    reached);

int main(){
  ios::sync_with_stdio(false);
    cin.tie(0);
  ofstream output;
  output.open("CompareResult.txt");
  clock_t start, end;
```

```
  double time[METHODSNUMBER]={0,0,0};
  bool **reached = new bool*[MAXSIZE]; //bool map to check whether reached
  point **points = new point*[MAXSIZE];
  int **grid = new int*[MAXSIZE]; //store the weight distribution, used to refresh points
  for(int j = 0; j < MAXSIZE; j++){
    reached[j] = new bool[MAXSIZE];
    points[j] = new point[MAXSIZE];
    grid[j] = new int[MAXSIZE];
  }
  for(int i=MINSIZE; i<=MAXSIZE; i*=STEP){

    for(int num=0; num<INDEXNUMBER; num++){
      for(int j=0;j<i;j++){
        for(int k=0; k<i; k++){
          while((grid[j][k] = lrand48()%100) <= 0){}//the weight is set to be in [1,99]
        }
      }

      for(int choice=0; choice<METHODSNUMBER; choice++){
          //refresh points
        for(int j=0;j<i;j++){
          for(int k=0; k<i; k++){
            reached[j][k] = 0;
            points[j][k].y = j;
            points[j][k].x = k;
            points[j][k].weight = grid[j][k];
            points[j][k].pathcost = 0;
            points[j][k].predecessor = NULL;
          }
        }

        if (choice == 0){
          priority_queue<point*, compare_t> * PQ = new binary_heap<point*, compare_t>;
          start = clock();
          implement(PQ, i-1, points, reached);
          end = clock();
          time[choice] += (double) (( end - start )* 1000.0 / CLOCKS_PER_SEC);
          output<<"Method "<<METHOSNAMES[choice]<<" after Trial "<<num<<" accumulates: "<<time[choice]<<"ms."<<endl;
          delete PQ;
        }

        else if (choice ==1){
          priority_queue<point*, compare_t> * PQ = new unsorted_heap<point*, compare_t>;
          start = clock();
          implement(PQ, i-1, points, reached);
          end = clock();
          time[choice] += (double) (( end - start )* 1000.0 / CLOCKS_PER_SEC);
          output<<"Method "<<METHOSNAMES[choice]<<" after Trial "<<num<<" accumulates: "<<time[choice]<<"ms."<<endl;
          delete PQ;
        }
        else{
          priority_queue<point*, compare_t> * PQ = new fib_heap<point*, compare_t>;
          start = clock();
          implement(PQ, i-1, points, reached);
          end = clock();
          time[choice] += (double) (( end - start )* 1000.0 / CLOCKS_PER_SEC);
          output<<"Method "<<METHOSNAMES[choice]<<" after Trial "<<num<<" accumulates: "<<time[choice]<<"ms."<<endl;
          delete PQ;
        }
      }

    }
    for(int choice = 0; choice<METHODSNUMBER; choice++){
      time[choice] /= INDEXNUMBER;
      output<<METHOSNAMES[choice]<<" on average "<<time[choice]<<"ms with size of "<<i<<endl;
      time[choice] = 0;
    }

```

```
  }
  for(int j = 0; j < MAXSIZE; j++){
      delete[] reached[j];
      delete[] points[j];
      delete[] grid[j];
    }
    delete[] reached;
    delete[] points;
    delete[] grid;
  output.close();
}


void implement(priority_queue<point*, compare_t> * PQ, int end, point** points, bool** reached){
  points[0][0].pathcost = points[0][0].weight;
  reached[0][0] = 1;
  PQ -> enqueue(&points[0][0]);
  while(!PQ->empty()){
    point* C = PQ->dequeue_min();
    point* N;
    //check the 4 neighbors
    for(int i=0; i<4; i++){
      N = C;
      if (i==0 && (C->x + 1) <= end)
        N = &points[C->y][C->x+1];
      if (i==1 && (C->y + 1) <= end)
        N = &points[C->y+1][C->x];
      if (i==2 && (C->x - 1) >= 0)
        N = &points[C->y][C->x-1];
      if (i==3 && (C->y - 1) >= 0)
        N = &points[C->y-1][C->x];
      //if already reached then skip it
      if (reached[N->y][N->x]) continue;
      N->pathcost = C->pathcost + N->weight;
      reached[N->y][N->x] = 1;
      N->predecessor = C;
      if(N->y == end && N->x == end){
        cout<<"pathcost:"<<N->pathcost<<endl;
        return;
      }
      else{
        PQ->enqueue(N);
      }
    }
  }
}
```