Listing 1: Graph.hpp

```
1  #pragma once
2
3  #include <algorithm>
4  #include <cmath>
5  #include <doctest.h>
6  #include <iostream>
7  #include <limits>
8  #include <list>
9  #include <queue>
10 #include <stdexcept>
11 #include <sstream>
12 #include <string>
13 #include <vector>
14
15 /*-----------------------------------------------------------------------------
16  * class definition
17  *----------------------------------------------------------------------------*/
18
19 /**
20  * @brief CMP 246 Module 7 non-templated class representing a weighted graph
21  * G=(V, E).
22  *
23  * This class uses an adjacency-list representation for the graph. The code is
24  * based on the graph algorithms discussed in
25  *
26  * Cormen, T. H., &; Leiserson, C. E. (2009). Introduction to algorithms, 3rd Ed.
27  */
28 class Graph {
29 private:
30     /**
31      * @brief (vertex, weight) pair stored in adjacency lists.
32      *
33      * This class represents the (vertex, weight) pairs stored in the adjacency
34      * list representation of the graph. Stream insertion is overloaded for
35      * printing, and equality operation allows us to search a adjacency list
36      * by vertex number.
37      */
38     class AdjEntry {
39     public:
40         /**
41          * @brief Construct a new Adj Entry object
42          *
43          * @param vertex Vertex v in a (u, v) edge
44          * @param weight Weight of the edge u, v
45          */
46         AdjEntry(size_t vertex, double weight) : v(vertex), wUV(weight) { }
47
48         /**
49          * @brief Vertex v in a (u, v) edge
50          */
51         size_t v;
52
53         /**
54          * @brief Weight of the edge (u, v)
55          */
56         double wUV;
57
58         /**
59          * @brief Override of equality operator.
60          *
61          * @param vertex vertex number to compare to
62          * @return true if this vertex number is equal to the parameter
63          */
64         bool operator==(size_t vertex) { return v == vertex; }
65
66         /**
67          * @brief Override of stream insertion operator.
68          *
69          * @param out std::ostream to write to
70          * @param a AdjacencyEntry object to output
71          * @return the out ostream object
72          */
73         friend std::ostream &operator<<(std::ostream &out, const AdjEntry &a) {
74             out << "(" << a.v << ",␣" << a.wUV << ")";
```

```
75              return out;
76          }
77      };
78
79  public:
80      /**
81       * @brief Construct a new Graph object.
82       *
83       * @param numVertices Number of vertices in the graph. Defaults to 100.
84       */
85      Graph(size_t numVertices = 100);
86
87      /**
88       * @brief Destroy the Graph object
89       *
90       */
91      ~Graph();
92
93      /**
94       * @brief Add an edge (u, v) to the graph.
95       *
96       * @param u "From" vertex of the edge.
97       *
98       * @param v "To" vertex of the edge.
99       *
100      * @param weight Weight associated with edge (u, v).
101      *
102      * @throws std::out_of_range if either u or v is < 1 or > |V|.
103      */
104     void addEdge(size_t u, size_t v, double weight);
105
106     /**
107      * @brief Perform breadth-first search on this graph
108      *
109      * This method performs a breadth-first search to look find the shortest
110      * path, in terms of edge count, from a vertex s to any other vertex in
111      * the graph.
112      *
113      * @param s Vertex to search from
114      *
115      * @param pDist Array of doubles with size |V| + 1. Upon return, array
116      * elements [1, ..., |V|] will contain the distance from vertex s to all
117      * the other vertices, in terms of number of edges.
118      *
119      * @param pPred Array of size_t values with size |V| + 1. Upon return,
120      * array elements [1, ..., |V|] will contain predecessor elements. These
121      * allow us to work backwards to create the minimum hop count path from s
122      * to any other element in the graph.
123      *
124      * @throws std::out_of_range if s is < 1 or > |V|.
125      */
126     void BFS(size_t s, double *pDist, size_t *pPred) const;
127
128     /**
129      * @brief Perform Dijkstra's algorithm on this graph
130      *
131      * This method performs Dikstra's algorithm to look find the shortest
132      * path, in terms of edge weights, from a vertex s to any other vertex in
133      * the graph.
134      *
135      * @param s Vertex to search from
136      *
137      * @param pDist Array of doubles with size |V| + 1. Upon return, array
138      * elements [1, ..., |V|] will contain the distance from vertex s to all
139      * the other vertices, in terms of path edge weights.
140      *
141      * @param pPred Array of size_t values with size |V| + 1. Upon return,
142      * array elements [1, ..., |V|] will contain predecessor elements. These
143      * allow us to work backwards to create the minimum hop count path from s
144      * to any other element in the graph.
145      *
146      * @throws std::out_of_range if s is < 1 or > |V|.
147      */
148     void Dijkstra(size_t s, double *pDist, size_t *pPred) const;
149
150     /**
```

```
151         * @brief Get the weight of the edge (u, v).
152         *
153         * @param u "From" vertex of the edge.
154         *
155         * @param v "To" vertex of the edge.
156         *
157         * @return double Weight of the edge (u, v), NaN if there is no edge
158         * between u and v.
159         *
160         * @throws std::out_of_range if either u or v is < 1 or > |V|.
161         */
162        double getEdge(size_t u, size_t v) const;
163
164        /**
165         * @brief Get number of edges in the graph.
166         *
167         * @return size_t value holding the number of edges in the graph.
168         */
169        size_t numEdges() const;
170
171        /**
172         * @brief Get number of vertices in the graph.
173         *
174         * @return size_t value holding the number of vertices in the graph.
175         */
176        size_t numVertices() const { return numV; }
177
178        /**
179         * @brief Stream insertion override.
180         *
181         * @param out std::ostream to write to
182         *
183         * @param g Graph to print
184         *
185         * @return std::ostream& object written to
186         */
187        friend std::ostream& operator<<(std::ostream &out, const Graph &g) {
188            for(size_t i = 1; i <= g.numV; i++) {
189                out << i << ":␣[";
190                auto itr = g.pAdjacents[i]->begin();
191                while(itr != g.pAdjacents[i]->end()) {
192                    out << *itr << ",␣";
193                    itr++;
194                }
195                out << "]" << std::endl;
196            }
197            return out;
198        }
199
200    private:
201        /**
202         * @brief Number of vertices in the graph.
203         *
204         */
205        size_t numV;
206
207        /**
208         * @brief Pointer to an array of pointers to STL lists, representing the
209         * adjacency lists for this graph.
210         *
211         */
212        std::list<AdjEntry> **pAdjacents;
213    };
214
215    //------------------------------------------------------------------------------
216    // function implementations
217    //------------------------------------------------------------------------------
218
219    /*
220     * constructor implementation.
221     */
222    Graph::Graph(size_t numVertices) : numV(numVertices) {
223        pAdjacents = new std::list<AdjEntry>*[numV + 1];
224        for(size_t i = 1; i <= numV; i++) {
225            pAdjacents[i] = new std::list<AdjEntry>();
226        }
```

```
227 }
228
229 /*
230  * destructor implementation.
231  */
232 Graph::~Graph() {
233     for(size_t i = 1; i <= numV; i++) {
234         delete pAdjacents[i];
235     }
236     delete [] pAdjacents;
237 }
238
239 /*
240  * addEdge method implementation.
241  */
242 void Graph::addEdge(size_t u, size_t v, double weight) {
243     if(u == 0u ||  v == 0u || u > numV || v > numV) {
244         throw std::out_of_range("Illegal␣vertex␣in␣Graph::addEdge()");
245     }
246     pAdjacents[u]->push_back(AdjEntry(v, weight));
247 }
248
249 /*
250  * Breadth-first search implementation.
251  */
252 void Graph::BFS(size_t s, double *pDist, size_t *pPred) const {
253     // make sure s is a valid index number
254     if(s == 0u || s > numV) {
255         throw std::out_of_range("Illegal␣vertex␣in␣Graph::BFS()");
256     }
257
258     // coloring constants to keep track of which vertices have been
259     // explored
260     const int WHITE = 0;
261     const int GRAY = 1;
262     const int BLACK = 2;
263
264     // allocate tracking attribute memory
265     int *pColor = new int[numV + 1];
266
267     // initialize tracking attributes for each non-s vertex
268     for(size_t u = 1; u <= numV; u++) {
269         if(u != s) {
270             pColor[u] = WHITE;
271             pDist[u] = std::numeric_limits<double>::infinity();
272             pPred[u] = 0u;
273         }
274     }
275
276     // initialize tracking attributes for node s
277     pColor[s] = GRAY;
278     pDist[s] = 0;
279     pPred[s] = 0u;
280
281     // use STL queue to keep track of edges we need to explore
282     std::queue<size_t> Q;
283     // start at node s
284     Q.push(s);
285
286     // keep going until no more vertices are reachable
287     while(!Q.empty()) {
288         // dequeue first element
289         size_t u = Q.front();
290         Q.pop();
291
292         // examine all vertices adjacent to u
293         for(AdjEntry v : *(pAdjacents[u])) {
294             // add unexplored vertices to the queue
295             if(pColor[v.v] == WHITE) {
296                 // color as "discovered but incomplete"
297                 pColor[v.v] = GRAY;
298                 // track distance from v to u so far
299                 pDist[v.v] = pDist[u] + 1;
300                 // v's predecessor is u
301                 pPred[v.v] = u;
302
```

```
                Q.push(v.v);
            }
        }
        // this vertex is completely explored
        pColor[u] = BLACK;
    }

    // free color tracking memory
    delete [] pColor;
}

// doctest unit tests for BFS
TEST_CASE("testing Graph::BFS()") {
    Graph g(8);

    g.addEdge(1, 2, 1);
    g.addEdge(1, 5, 1);
    g.addEdge(2, 1, 1);
    g.addEdge(2, 6, 1);
    g.addEdge(3, 4, 1);
    g.addEdge(3, 6, 1);
    g.addEdge(3, 7, 1);
    g.addEdge(4, 7, 1);
    g.addEdge(4, 8, 1);
    g.addEdge(5, 1, 1);
    g.addEdge(1, 5, 1);
    g.addEdge(6, 2, 1);
    g.addEdge(6, 3, 1);
    g.addEdge(6, 7, 1);
    g.addEdge(7, 6, 1);
    g.addEdge(7, 3, 1);
    g.addEdge(7, 4, 1);
    g.addEdge(7, 8, 1);
    g.addEdge(8, 7, 1);
    g.addEdge(8, 4, 1);

    double pDist[9];
    size_t pPred[9];

    g.BFS(2, pDist, pPred);

    double pDistActual[9] = {0, 1, 0, 2, 3, 2, 1, 2, 3};
    size_t pPredActual[9] = {0, 2, 0, 6, 3, 1, 2, 6, 7};

    for(size_t i = 1; i <= 8; i++) {
        CHECK(pDistActual[i] == pDist[i]);
        CHECK(pPredActual[i] == pPred[i]);
    }

    // check exceptions
    bool flag = true;
    try {
        g.BFS(0, pDist, pPred);
        flag = false;
    } catch(std::out_of_range oor) {
    }
    CHECK(flag);

    flag = true;
    try {
        g.BFS(9, pDist, pPred);
        flag = false;
    } catch(std::out_of_range oor) {
    }
    CHECK(flag);
}

/*
 * Dijkstra method implementation.
 */
void Graph::Dijkstra(size_t s, double *pDist, size_t *pPred) const {
    // sanity check on vertex number s
    if(s < 1 || s > numV) {
        throw std::out_of_range("Illegal vertex number in Graph::Dijkstra()");
    }

```

```
379     // initialize distances and predecessors
380     for(size_t i = 1; i <= numV; i++) {
381         pDist[i] = std::numeric_limits<double>::infinity();
382         pPred[i] = 0;
383     }
384     pPred[s] = 0;
385     pDist[s] = 0;
386
387     // populate the "priority queue" with all vertices in the graph
388     std::list<size_t> Q;
389     for(size_t v = 1; v <= numV; v++) {
390         Q.push_back(v);
391     }
392
393     // continue until we have examined all vertices
394     while(!Q.empty()) {
395         // examine the node closest to the set of nodes we've looked at so far
396         auto uItr = Q.begin();
397         double minValue = std::numeric_limits<double>::infinity();
398         for(auto i = Q.begin(); i != Q.end(); i++) {
399             if(pDist[*i] < minValue) {
400                 minValue = pDist[*i];
401                 uItr = i;
402             }
403         }
404         // by now, uItr is an iterator on the smallest value in the queue,
405         // so save its value as vertex u
406         size_t u = *uItr;
407         // then use the std::list erase method to remove the value at the
408         // iterator location
409         Q.erase(uItr);
410
411         // update distances and predecessors for the vertex we just removed
412         // from the "priority queue"
413         for(AdjEntry v : *(pAdjacents[u])) {
414             if(pDist[v.v] > pDist[u] + v.wUV) {
415                 pDist[v.v] = pDist[u] + v.wUV;
416                 pPred[v.v] = u;
417             } // if
418         } // for
419     } // while
420 }
421
422 // doctest unit test for Dijkstra's algorithm method
423 TEST_CASE("testing Graph::Dijkstra()") {
424     Graph g(5);
425
426     g.addEdge(1, 2, 10);
427     g.addEdge(1, 4, 5);
428     g.addEdge(2, 3, 1);
429     g.addEdge(2, 4, 2);
430     g.addEdge(3, 5, 4);
431     g.addEdge(4, 2, 3);
432     g.addEdge(4, 3, 9);
433     g.addEdge(4, 5, 1);
434     g.addEdge(5, 1, 7);
435     g.addEdge(5, 3, 6);
436
437     double pDist[6];
438     size_t pPred[6];
439
440     g.Dijkstra(1, pDist, pPred);
441
442     double pDistPredicted[6] = {0, 0, 8, 9, 5, 6};
443     size_t pPredPredicted[6] = {0, 0, 4, 2, 1, 4};
444
445     for(size_t i = 1; i <= 5; i++) {
446         CHECK(pDist[i] == pDistPredicted[i]);
447         CHECK(pPred[i] == pPredPredicted[i]);
448     }
449
450     // check exception handling
451     bool flag = true;
452     try {
453         g.Dijkstra(0, pDist, pPred);
454         flag = false;
```

```
455     } catch(std::out_of_range oor) {
456
457     }
458     CHECK(flag);
459     flag = true;
460     try {
461         g.Dijkstra(6, pDist, pPred);
462         flag = false;
463     } catch(std::out_of_range oor) {
464
465     }
466     CHECK(flag);
467 }
468
469 /*
470  * getEdge method implementation.
471  */
472 double Graph::getEdge(size_t u, size_t v) const {
473     if(u == 0u || v == 0u || u > numV || v > numV) {
474         throw std::out_of_range("Illegal vertex in Graph::getEdge()");
475     }
476
477     auto idx = std::find(pAdjacents[u]->begin(), pAdjacents[u]->end(), v);
478     if(idx == pAdjacents[u]->end()) {
479         return std::numeric_limits<double>::quiet_NaN();
480     } else {
481         return (*idx).wUV;
482     }
483 }
484
485 // doctest unit tests for addEdge and getEdge
486 TEST_CASE("testing Graph::addEdge() and getEdge()") {
487     Graph g(5);
488
489     g.addEdge(1, 2, 1);
490     CHECK(g.getEdge(1, 2) == 1.0);
491     g.addEdge(1, 5, 2);
492     CHECK(g.getEdge(1, 5) == 2.0);
493     g.addEdge(2, 1, 3);
494     CHECK(g.getEdge(2, 1) == 3.0);
495     g.addEdge(2, 5, 4);
496     CHECK(g.getEdge(2, 5) == 4.0);
497     g.addEdge(2, 4, 5);
498     CHECK(g.getEdge(2, 4) == 5.0);
499     g.addEdge(2, 3, 6);
500     CHECK(g.getEdge(2, 3) == 6.0);
501     g.addEdge(3, 2, 7);
502     CHECK(g.getEdge(3, 2) == 7.0);
503     g.addEdge(3, 4, 8);
504     CHECK(g.getEdge(3, 4) == 8.0);
505     g.addEdge(4, 2, 9);
506     CHECK(g.getEdge(4, 2) == 9.0);
507     g.addEdge(4, 3, 10);
508     CHECK(g.getEdge(4, 3) == 10.0);
509     g.addEdge(4, 5, 11);
510     CHECK(g.getEdge(4, 5) == 11.0);
511     g.addEdge(5, 1, 12);
512     CHECK(g.getEdge(5, 1) == 12.0);
513     g.addEdge(5, 2, 13);
514     CHECK(g.getEdge(5, 2) == 13.0);
515     g.addEdge(5, 4, 14);
516     CHECK(g.getEdge(5, 4) == 14.0);
517
518     // test edges that don't exist
519     double w = g.getEdge(5, 3);
520     CHECK(std::isnan(w));
521     w = g.getEdge(1, 4);
522     CHECK(std::isnan(w));
523
524     // test exception handling of addEdge
525     bool flag = true;
526     try {
527         g.addEdge(0, 5, 13);
528         flag = false;
529     } catch(std::out_of_range oor) {
530     }
```

```
531     CHECK(flag);
532     flag = true;
533     try {
534         g.addEdge(5, 0, 13);
535         flag = false;
536     } catch(std::out_of_range oor) {
537     }
538     CHECK(flag);
539     flag = true;
540     try {
541         g.addEdge(6, 5, 13);
542         flag = false;
543     } catch(std::out_of_range oor) {
544     }
545     CHECK(flag);
546     flag = true;
547     try {
548         g.addEdge(5, 6, 13);
549         flag = false;
550     } catch(std::out_of_range oor) {
551     }
552     CHECK(flag);
553 
554     // check exception handling for getEdge
555     flag = true;
556     try {
557         g.getEdge(0, 5);
558         flag = false;
559     } catch(std::out_of_range oor) {
560     }
561     CHECK(flag);
562     flag = true;
563     try {
564         g.getEdge(5, 0);
565         flag = false;
566     } catch(std::out_of_range oor) {
567     }
568     CHECK(flag);
569     flag = true;
570     try {
571         g.getEdge(6, 5);
572         flag = false;
573     } catch(std::out_of_range oor) {
574     }
575     CHECK(flag);
576     flag = true;
577     try {
578         g.getEdge(5, 6);
579         flag = false;
580     } catch(std::out_of_range oor) {
581     }
582     CHECK(flag);
583 }
584 
585 /*
586  * numEdges method implementation.
587  */
588 size_t Graph::numEdges() const {
589     size_t sum = 0u;
590     for(size_t i = 1u; i <= numV; i++) {
591         sum += pAdjacents[i]->size();
592     }
593 
594     return sum;
595 }
596 
597 // doctest unit test for numEdges method
598 TEST_CASE("testing␣Graph::numEdges()") {
599     Graph g(5);
600     CHECK(0 == g.numEdges());
601 
602     g.addEdge(1, 2, 1);
603     CHECK(1 == g.numEdges());
604     g.addEdge(1, 5, 1);
605     CHECK(2 == g.numEdges());
606     g.addEdge(2, 1, 1);
```

```
    CHECK(3 == g.numEdges());
    g.addEdge(2, 5, 1);
    CHECK(4 == g.numEdges());
    g.addEdge(2, 4, 1);
    CHECK(5 == g.numEdges());
    g.addEdge(2, 3, 1);
    CHECK(6 == g.numEdges());
    g.addEdge(3, 2, 1);
    CHECK(7 == g.numEdges());
    g.addEdge(3, 4, 1);
    CHECK(8 == g.numEdges());
    g.addEdge(4, 2, 1);
    CHECK(9 == g.numEdges());
    g.addEdge(4, 3, 1);
    CHECK(10 == g.numEdges());
    g.addEdge(4, 5, 1);
    CHECK(11 == g.numEdges());
    g.addEdge(5, 1, 1);
    CHECK(12 == g.numEdges());
    g.addEdge(5, 2, 1);
    CHECK(13 == g.numEdges());
    g.addEdge(5, 4, 1);
    CHECK(14 == g.numEdges());
}

// doctest unit test for numVertices method
TEST_CASE("testing␣Graph::numVertices()") {
    Graph g;
    CHECK(100 == g.numVertices());
    Graph h(5);
    CHECK(5 == h.numVertices());
}

// doctest unit test for operator<<
TEST_CASE("testing␣operator<<") {
    Graph g(5);

    g.addEdge(1, 2, 1);
    g.addEdge(1, 5, 1);
    g.addEdge(2, 1, 1);
    g.addEdge(2, 5, 1);
    g.addEdge(2, 4, 1);
    g.addEdge(2, 3, 1);
    g.addEdge(3, 2, 1);
    g.addEdge(3, 4, 1);
    g.addEdge(4, 2, 1);
    g.addEdge(4, 3, 1);
    g.addEdge(4, 5, 1);
    g.addEdge(5, 1, 1);
    g.addEdge(5, 2, 1);
    g.addEdge(5, 4, 1);

    std::ostringstream oss;
    oss << g;
    std::string expected("1:␣[(2,␣1),␣(5,␣1),␣]\n");
    expected += "2:␣[(1,␣1),␣(5,␣1),␣(4,␣1),␣(3,␣1),␣]\n";
    expected += "3:␣[(2,␣1),␣(4,␣1),␣]\n";
    expected += "4:␣[(2,␣1),␣(3,␣1),␣(5,␣1),␣]\n";
    expected += "5:␣[(1,␣1),␣(2,␣1),␣(4,␣1),␣]\n";
    CHECK(oss.str() == expected);
}
```

Listing 2: PathFinding.cpp

```
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>
#include "../1-Graph/Graph.hpp"

/**
 * @brief Convert a map tile character to a weight.
 *
 * @param c Character representing a map tile.
 * @return double Weight associated with moving to that tile.
 */
```

```
13 double charToWeight(char c) {
14     switch(c) {
15         case 'c': return 1;
16         case 's': return 1.5;
17         case 'w': return 3;
18         case 'x': return -1; // flag - do not add an edge here
19         case 'b': return 1;
20         case 'e': return 1;
21     }
22     return -1;
23 }
24
25 /**
26  * @brief Convert a (row, col) coordinate to a vertex number.
27  *
28  * @param numR Number of rows in the map.
29  * @param r Row of the coordinate
30  * @param c Column of the coordinate
31  * @return size_t Vertex number in [1, numR * numC]
32  */
33 size_t rowColToVertex(size_t numR, size_t r, size_t c) {
34     return r * numR + c + 1;
35 }
36
37 /**
38  * @brief Application entry point.
39  *
40  */
41 int main() {
42
43     // read map file into character array; first, get size
44     std::ifstream inFile("map.txt");
45     size_t numR, numC;
46     inFile >> numR >> numC;
47
48     // allocate 2D array for map storage
49     char **pMap = new char*[numR];
50     for(size_t r = 0; r < numR; r++) {
51         pMap[r] = new char[numC];
52     }
53
54     // read remaining lines from the file, place characters into the array
55     std::string line;
56     for(size_t r = 0; r < numR; r++) {
57         inFile >> line;
58         for(size_t c = 0; c < numC; c++) {
59             pMap[r][c] = line[c];
60         }
61     }
62     inFile.close();
63
64     // build a graph representation of the map
65     Graph g(numR * numC);
66     for(size_t r = 0; r < numR; r++) {
67         for(size_t c = 0; c < numC; c++) {
68             // source vertex number
69             size_t vertex = rowColToVertex(numR, r, c);
70             double weight;
71
72             // examine all valid neighbors and enter edges w/ weights
73             if(r > 0) {
74                 // north neighbor
75                 weight = charToWeight(pMap[r - 1][c]);
76                 if(weight > 0) {
77                     g.addEdge(vertex, rowColToVertex(numR, r - 1, c), weight);
78                 }
79             }
80             if(r < numR - 1) {
81                 // south neighbor
82                 weight = charToWeight(pMap[r + 1][c]);
83                 if(weight > 0) {
84                     g.addEdge(vertex, rowColToVertex(numR, r + 1, c), weight);
85                 }
86             }
87             if(c < numC - 1) {
88                 // east neighbor
```

```
89                  weight = charToWeight(pMap[r][c + 1]);
90                  if(weight > 0) {
91                      g.addEdge(vertex, rowColToVertex(numR, r, c + 1), weight);
92                  }
93              }
94              if(c > 0) {
95                  // west neighbor
96                  weight = charToWeight(pMap[r][c - 1]);
97                  if(weight > 0) {
98                      g.addEdge(vertex, rowColToVertex(numR, r, c - 1), weight);
99                  }
100             }
101         } // for c
102     } // for r
103
104     // execute Dijkstra's algorithm to find a path to the end point
105     double pDist[numR * numC + 1];
106     size_t pPred[numR * numC + 1];
107     g.Dijkstra(1, pDist, pPred);
108
109     // construct path from start (assumed to be vertex 1) to the end
110     std::list<size_t> path;
111     path.push_front(numR * numC);
112     size_t p = numR * numC;
113     while(pPred[p] != 1) {
114         path.push_front(pPred[p]);
115         p = pPred[p];
116     }
117     path.push_front(1);
118
119     // output path
120     std::cout << "Path␣from␣start␣to␣finish:␣";
121     for(size_t p1 : path) {
122         std::cout << p1 << "␣";
123     }
124     std::cout << std::endl;
125
126     // free memory allocated for the map
127     for(size_t r = 0; r < numR; r++) {
128         delete [] pMap[r];
129     }
130     delete [] pMap;
131
132     return EXIT_SUCCESS;
133 }
```