ASS3

1. Introduction. This is a literate program which solves the problem set in assignment three—shortest path.

The specification for this assignment states:

> *This assignment involves extension to the single source-single destination shortest path problem.*
>
> *Your program should:*
>
> 1. *Read the name of a text file from the console. (Not the command line)*
> 2. *Read an undirected graph from the file.*
> 3. *Find the shortest path between the start and goal vertices specified in the file.*
> 4. *Print out the vertices on the path, in order from start to goal.*
> 5. *Print out the length of this path.*
> 6. *Find the second shortest path between the start and goal vertices specified in the file.*
> 7. *Print out the vertices on the path, in order from start to goal.*
> 8. *Print out the length of this path.*
>
> *The data file is constructed as follows:*
>
> - *Two integers: nVertices and nEdges, the number of vertices and edges in the graph.*
> - *nVertices triples consisting of the label and the x- and y-coordinates of each vertex. (An int followed by two doubles)*
> - *nEdges triples consisting of the labels of the start and end vertices of each edge, along with its weight. Note: the weight associated with an edge will be greater than or equal to the Euclidean distance between its start and end vertices as determined by their coordinates. (Two ints followed by a double)*
> - *Two labels identifying the start and goal vertices for which the paths are required. (Two ints)*

2. We will start with the outline of the program.

```
using namespace std;
⟨ Headers 9 ⟩
⟨ Global data 4 ⟩
⟨ Prototypes for functions 30 ⟩
⟨ The main program 7 ⟩
⟨ Implementation of functions 31 ⟩
```

3. Data structures. For this program we will start by declaring the main global data structures that will be needed.

First we will require definitions for vertices and edges.

4. Each vertex will require the following values to be stored: the x- and y-coordinates of the vertex, the shortest distance from the start vertex to this vertex found so far, the previous vertex in this best path and the heuristic (Euclidean) distance from this vertex to the goal.

These will be stored in the *vertices* array.

```
⟨ Global data 4 ⟩ ≡
  struct vertex {
    double xCoordinate;
    double yCoordinate;
    double length;
    int previous;
    double heuristic;
  };
  vertex *vertices;
```

See also sections 5, 6, 12, and 18.

This code is used in section 2.

5. The only information we need to store for an edge are the start and end vertices and the edge weight.

We will do this using an adjacency matrix, a square 2-D array in which the row and column indices represent the start and end verices for the edge and the contents represents the edge weight.

```
⟨ Global data 4 ⟩ +≡
  double **edgeWeight;
```

6. Vertices to which we have not yet determined the shortest path, candidate edges, are maintained in an array. This array needs only to contain the index of the vertex, all other information can be derived from the *vertices* array.

```
⟨ Global data 4 ⟩ +≡
  int *candidate;
```

7. Main. Now we can start writing *main*. The skeleton of the *main* program is as follows.

```
⟨ The main program 7 ⟩ ≡
  int main( ){ ⟨ Variables of main 8 ⟩
      ⟨ Initialize the data 10 ⟩
      ⟨ Find the shortest path 19 ⟩⟨ Find the second shortest path 23 ⟩
      ⟨ Finish up 29 ⟩
      return 0; }
```

This code is used in section 2.

8. The first thing we need to do is declare the variables we need to input words. Let's start with the character array *filename* and the input stream *fin*.

```
⟨ Variables of main 8 ⟩ ≡
  char filename[20];
  ifstream fin;
```

See also sections 13, 16, 20, 22, and 28.

This code is used in section 7.

9. We need a couple of header files *iostream* for stream-based input and *fstream* for managing files.

```
⟨ Headers 9 ⟩ ≡
#include <iostream>
#include <fstream>
```

See also section 15.

This code is used in section 2.

10. Now we can get the file opened and ready for input. We will prompt for the input file name using *cerr* so that we can redirect the output without getting the prompt in the output file and so that we can see the prompt even when we redirect standard output. We will then read in the file name and open an input stream. We should test for errors too, I guess.

```
⟨ Initialize the data 10 ⟩ ≡
  cerr ≪ "Please␣enter␣the␣name␣of␣the␣input␣file:␣";
  cin ≫ filename;
  fin.open(filename);
  if (¬fin) {
    cerr ≪ "Error␣opening␣file␣" ≪ filename ≪ ".␣Program␣will␣exit." ≪ endl;
    return 0;
  }
```

See also sections 11, 14, and 17.

This code is used in section 7.

11. We are now ready to start the program proper.

We start by reading in the vertex and edge counts.

Now that we know how many vertices are in the graph, we can create the *vertices* array. Then we can read the vertex information.

At this point, all we know are the coordinates of the vertex and its identity label. Because we are lazy, and because we know how the file is constructed, we will ignore the label value because it will always be equal to $i+1$. If this were not the case we would store the label an the vertex struct and set up a hash table to allow us to determine the index corresponding to a given label.

```
⟨ Initialize the data 10 ⟩ +≡
  fin ≫ nVertices ≫ nEdges;
  vertices = new vertex[nVertices];
  for (int i = 0; i < nVertices; i++) fin ≫ id ≫ vertices[i].xCoordinate ≫ vertices[i].yCoordinate;
```

12. We need to define some variables. One is global;

```
⟨ Global data 4 ⟩ +≡
  int nVertices;
```

13. and the rest are local.

```
⟨ Variables of main 8 ⟩ +≡
  int id;
  int nEdges;
```

14. Now we can get the edge data from the file.

The data will be subject to a check to ensure that we account for potential duplicate paths. In this case we will select the shortest such path. Because the graph is non-directed we will store both the $[i][j]$ and the $[j][i]$ weights.

Note: We decrement the input *row* and *col* values as we have zero-based arrays.

```
⟨ Initialize the data 10 ⟩ +≡
  edgeWeight = new double*[nVertices];
  for (int i = 0; i < nVertices; i++) edgeWeight[i] = new double[nVertices];
  for (int row = 0; row < nVertices; row++)
    for (int col = 0; col < nVertices; col++) edgeWeight[row][col] = HUGE_VAL;
  for (int i = 0; i < nEdges; i++) {
    fin ≫ row ≫ col ≫ edgeWeight[row − 1][col − 1];
    row−−;
    col−−;
    if (edgeWeight[row][col] < edgeWeight[col][row]) edgeWeight[col][row] = edgeWeight[row][col];
    else edgeWeight[row][col] = edgeWeight[col][row];
  }
```

15. We need a header file for `HUGE_VAL`.

```
⟨ Headers 9 ⟩ +≡
#include <cmath>
```

16. We also need some variables.

```
⟨ Variables of main 8 ⟩ +≡
  int row, col;
```

17. Finally, we read in the start and goal vertices and calculate the heuristic for each vertex. To do this we calculate the Euclidean distance between each vertex and the goal, using the x- and y-coordinates of the vertices $\sqrt{\delta x^2 + \delta y^2}$. Once again, we decrement the start and goal indices to account for zero-based arrays.

⟨ Initialize the data 10 ⟩ +≡
```
fin ≫ startVertex ≫ goalVertex;
startVertex −−;
goalVertex −−;
```

18. These variables should be declared globally.

⟨ Global data 4 ⟩ +≡
```
int startVertex, goalVertex;
```

19. Shortest path. To solve for the shortest path we need only call our A* function. Once it has found the path we can report the results.

```
⟨ Find the shortest path 19 ⟩ ≡
  status = astar( );
  ⟨ Report the shortest path information 21 ⟩
```

This code is used in section 7.

20. We should declare that.

```
⟨ Variables of main 8 ⟩ +≡
  int status;
```

21. To output the path in the correct order we need to create a stack onto which we will push the vertices, in order from goal back to start, using the parent indices.

We need only iterate backwards through this array to list the verices in start to goal order.

The *path* array and its size will also be required later for finding the second-shortest path.

```
⟨ Report the shortest path information 21 ⟩ ≡
  if (status ≡ 0) {
    cout ≪ "The␣shortest␣path␣has␣a␣length␣of␣" ≪ vertices[goalVertex].length ≪ endl;
    path = new int[nVertices];
    nPathVertices = 0;
    for (int i = goalVertex; i ≠ startVertex; i = vertices[i].previous) path[nPathVertices++] = i;
    path[nPathVertices++] = startVertex;
    cout ≪ "␣␣␣␣The␣vertices␣on␣this␣path␣are:␣";
    for (int i = nPathVertices; i > 0; i--) cout ≪ path[i − 1] + 1 ≪ "␣";
    cout ≪ endl;
  }
  else {
    cout ≪ "***␣No␣path␣found␣between␣vertices␣" ≪ startVertex ≪ "␣and␣" ≪ goalVertex ≪ endl;
    return 1;
  }
```

This code is used in section 19.

22. A few more local variables.

```
⟨ Variables of main 8 ⟩ +≡
  int *path;
  int nPathVertices;
```

23. Second shortest path. We have a copy of the shortest path saved in the *path* array. The edges in this array will now be removed from the graph, one-by-one, and a new shortest path found on the remaining graph. The shortest of these secondary solutions is the second shortest path.

```
⟨ Find the second shortest path 23 ⟩ ≡
  path2 = new int[nVertices];
  nPathEdges = nPathVertices − 1;
  bestLength = HUGE_VAL;
  for (int i = 0; i < nPathEdges; i++) {
    ⟨ Remove this edge 24 ⟩
    status = astar( );
    ⟨ Restore the removed edge 25 ⟩
    if (status ≡ 0 ∧ vertices[goalVertex].length < bestLength) ⟨ Update the best path so far 26 ⟩
  }
  if (nPath2Vertices > 0) {
    ⟨ Report the second-shortest path 27 ⟩
  }
  else
    cout ≪ "***␣No␣2nd␣path␣found␣between␣vertices␣" ≪ startVertex ≪ "␣and␣" ≪ goalVertex ≪
        endl;
```

This code is used in section 7.

24. First we modify the graph by removing an edge. We do this be setting the edge weights for this edge to infinity in both directions.

```
⟨ Remove this edge 24 ⟩ ≡
  from = path[i];
  to = path[i + 1];
  savedWeight = edgeWeight[from][to];
  edgeWeight[from][to] = edgeWeight[to][from] = HUGE_VAL;
```

This code is used in section 23.

25. Once we have run A* we restore the edges back to their original weight.

```
⟨ Restore the removed edge 25 ⟩ ≡
  edgeWeight[from][to] = edgeWeight[to][from] = savedWeight;
```

This code is used in section 23.

26. We will need to save the best 2nd best path.

```
⟨ Update the best path so far 26 ⟩ ≡
  {
    nPath2Vertices = 0;
    for (int i = goalVertex; i ≠ startVertex; i = vertices[i].previous) path2[nPath2Vertices++] = i;
    path2[nPath2Vertices++] = startVertex;
  }
```

This code is used in section 23.

27. We can now report on the 2nd shortest path.

```
⟨ Report the second-shortest path 27 ⟩ ≡
  cout ≪ "The␣2nd␣shortest␣path␣has␣a␣length␣of␣" ≪ vertices[goalVertex].length ≪ endl;
  cout ≪ "␣␣␣␣The␣vertices␣on␣this␣path␣are:␣";
  for (int i = nPath2Vertices; i > 0; i−−) cout ≪ path2[i − 1] + 1 ≪ "␣";
  cout ≪ endl;
```

This code is used in section 23.

28. Here are the variables needed for the second shortest path.

⟨ Variables of main 8 ⟩ +≡
```
int nPathEdges;
double bestLength;
double savedWeight;
int from, to;
int nPath2Vertices;
int *path2;
```

29. Cleaning up.

⟨ Finish up 29 ⟩ ≡

```
fin.close( );
delete[ ] vertices;
for (int i = 0; i < nVertices; i++) delete[ ] edgeWeight[i];
delete[ ] edgeWeight;
delete[ ] path;
delete[ ] path2;
```

This code is used in section 7.

30. The A* function.
The main function we need to consider is the A* shortest path function.

```
⟨ Prototypes for functions 30 ⟩ ≡
  int astar(void);
```

See also section 39.

This code is used in section 2.

31. The A* algorithm uses a set of candidate vertices from which it progressively removes members until the goal vertex is reached.

In each iteration we select a new vertex and update the distance from *startVertex* to each remaining candidate.

```
⟨ Implementation of functions 31 ⟩ ≡
  int astar( )
  {
    ⟨ Local variables for A* 32 ⟩
    ⟨ Prepare to run the A* algorithm 33 ⟩;
    while (⟨ We are not finished 35 ⟩) {
      ⟨ Remove the best candidate 37 ⟩
      for (int i = 0; i < nCandidates; i++) {
        current = vertices[candidate[i]].length;
        update = vertices[selected].length + edgeWeight[selected][candidate[i]];
        if (update < current) {
          ⟨ Update the candidates values and restore the heap 38 ⟩
        }
      }
    }
    ⟨ Return the appropriate completion status 36 ⟩
  }
```

See also sections 40, 41, and 42.

This code is used in section 2.

32. Some more local variables.

```
⟨ Local variables for A* 32 ⟩ ≡
  double current, update;
```

See also section 34.

This code is used in section 31.

33. Initially the *candidate* array contains every vertex except *startVertex*. We also need to initialize the *length* and *previous* components for each vertex.

The *candidate* array will be maintained as a min-heap ordered on the sum of the minimum distance from the *startVertex* to this vertex and the Euclidean distance from this vertex to the *goalVertex*;

```
⟨ Prepare to run the A* algorithm 33 ⟩ ≡
  candidate = new int[nVertices];
  for (int i = 0; i < nVertices; i++) candidate[i] = i;
  nCandidates = nVertices − 1;
  candidate[startVertex] = nCandidates;
  for (int i = 0; i < nVertices; i++) {
    vertices[i].length = edgeWeight[startVertex][i];
    vertices[i].previous = startVertex;
  }
  makeheap(candidate, nCandidates);
```

This code is used in section 31.

34. We should declare those variables.

```
⟨ Local variables for A* 32 ⟩ +≡
  int *candidate;
  int nCandidates;
```

35. The algorithm terminates in one of two ways: either we have arrived at the goal vertex or we have established that there is no path to it.

```
⟨ We are not finished 35 ⟩ ≡
  candidate[0] ≠ goalVertex ∧ vertices[candidate[0]].length ≠ HUGE_VAL
```

This code is used in section 31.

36. The way in which we leave the loop determines the status value we return. A *status* value of 0 is used to indicate a successful search, a value of 1 indicates that there is no path to be found.

```
⟨ Return the appropriate completion status 36 ⟩ ≡
  delete[] candidate;
  if (vertices[goalVertex].length ≠ HUGE_VAL) return 0;
  return 1;
```

This code is used in section 31.

37. At the start of each pass *candidate*[0] contains the index of vertex which we wish to select. We save this selection and then remove it from the top of the heap.

```
⟨ Remove the best candidate 37 ⟩ ≡
  int selected = candidate[0];
  nCandidates−−;
  candidate[0] = candidate[nCandidates];
  siftDown(candidate, nCandidates, 0);
```

This code is used in section 31.

38. The last step is to update the value of the current candidate and ensure that the min-heap property is maintained.

```
⟨ Update the candidates values and restore the heap 38 ⟩ ≡
  vertices[candidate[i]].length = update;
  vertices[candidate[i]].previous = selected;
  siftUp(candidate, i);
```

This code is used in section 31.

39. Heap functions. Finally, we need to provide the functions needed to manage the *candidate* heap.

```
⟨ Prototypes for functions 30 ⟩ +≡
  void makeheap(int *, int);
  void siftUp(int *, int);
  void siftDown(int *, int, int);
```

40. The *makeheap*( ) function does exactly what you would expect.

```
⟨ Implementation of functions 31 ⟩ +≡
  void makeheap(int *heap, int heapSize)
  {
    for (int i = heapSize/2; i ≥ 0; i--) siftDown(heap, heapSize, i);
  }
```

41. Next is *siftUp*( ). We recusively compare the current entry, *heap*[*i*] with its parent. *heap*[*p*] swapping the entries if the parent has a greater aggregate distance than the child.

```
⟨ Implementation of functions 31 ⟩ +≡
  void siftUp(int *heap, int i)
  {
    int temp;
    if (i ≡ 0) return;
    int p = (i − 1)/2;
    double iVal = vertices[heap[i]].length + vertices[heap[i]].heuristic;
    double pVal = vertices[heap[p]].length + vertices[heap[p]].heuristic;
    if (pVal < iVal) return;
    temp = heap[p];
    heap[p] = heap[i];
    heap[i] = temp;
    siftUp(heap, p);
    return;
  }
```

42. Finally we implement *siftDown*( ). This time we compare the current entry, *heap*[*i*], with the child, *heap*[*c*], having the smaller aggregate distance, again swapping and recursing as needed. This function is a little messier as we have somewhere between 0 and 2 children.

⟨ Implementation of functions 31 ⟩ +≡

```
void siftDown(int *heap, int heapSize, int i)
{
  int temp, c;
  double iVal, cVal, c1Val;
  c = 2 * i + 1;
  if (c ≥ heapSize) return;
  iVal = vertices[heap[i]].length + vertices[heap[i]].heuristic;
  cVal = vertices[heap[c]].length + vertices[heap[c]].heuristic;
  if (c + 1 < heapSize) {
    c1Val = vertices[heap[c + 1]].length + vertices[heap[c + 1]].heuristic;
    if (c1Val < cVal) {
      c++;
      cVal = c1Val;
    }
  }
  if (cVal > iVal) return;
  temp = heap[c];
  heap[c] = heap[i];
  heap[i] = temp;
  siftDown(heap, heapSize, c);
  return;
}
```

43. Index. This index is automatically created. It lists all the variables used in the program and the section(s) in which they are used. Underlined entries indicate where a variable is defined. The remaining sections of this document are also created automatically.