Worksheet 11

Overview

The w orkshop for Week 12 w ill start w ith a tutorial on All-pairs shortest paths, and w ill optionally implement it.

Tutorial Questions

![Floyd Warshall Graph](Week 12 - Workshop - FWGraph.png)
Question 11.1

1. Draw the matrix representation of the graph above.

						From						
		0	1	2	3	4	5	6	7	8	9	
	0											
	1											
	2											
	3											
To	4											
	5											
	6											
	7											
	8											
	9											

- Trace the operation of Floyd-Warshall algorithm on this graph, to get the shortest paths betw een all pairs of vertices. (I'd recommend you copy your answ er from the previous question to w ork w ith.)

- Does the order in w hich the vertices are examined affect the operation of the algorithm? Change the numbering of the vertices (but not the w eights) as indicated below and retrace the algorithm. You should end up w ith the same result. Have any of the intermediate steps changed? (I'd recommend you copy your answ er from part 1 and simply rename the vertices to see how this performs).

- Old vertex number 3, rename to 5
- Old vertex number 4, rename to 3
- Old vertex number 5, rename to 4
- Old vertex number 6, rename to 7
- Old vertex number 7, rename to 9
- Old vertex number 9, rename to 6

1. Does the Floyd-Warshall algorithm w ork on graphs w here there are negative w eights? Justify your answ er.

2. Given a sparse graph represented as an adjacency list, how would you approach the all pairs shortest paths problem? Does this differ from the approach you would take for a dense graph represented as a matrix? Compare the computational complexity of the two approaches.

Programming Exercises

Programming 11.1 You can use the laboratory time to implement the Floyd-Warshall algorithm and test it on the graph you have just worked on in the tutorial. Or work through the past exam paper.

```
/* adjmatrix.h */
/* NOTE: this could be made a lot more robust,
    but it's easier not to. */
#include <limits.h>
#define NOPATH INT_MAX/2
/* Allocates space for a size * size array
    and sets all values to the NOPATH value. */
int **newMatrix(int size);

/* Adds an edge between one given index and another. */
void addEdge(int **adjMatrix, int src, int dst, int weight);

/* Frees a matrix allocated with the newMatrix function. */
void freeMatrix(int **matrix, int size);

/* adjmatrix.c */
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
/* #include "adjmatrix.h" */

int **newMatrix(int size){
    int **retMatrix;
    int i, j;
    retMatrix = (int **) malloc(sizeof(int *)*size);
    assert(retMatrix);
    for(i = 0; i < size; i++){
        retMatrix[i] = (int *) malloc(sizeof(int)*size);
        assert(retMatrix[i]);
        for(j = 0; j < size; j++){
            retMatrix[i][j] = NOPATH;
        }
    }
    return retMatrix;
}

void addEdge(int **adjMatrix, int src, int dst, int weight){
    adjMatrix[src][dst] = weight;
}

void freeMatrix(int **matrix, int size){
    if(!matrix){
        return;
    }
    int i;
    for(i = 0; i < size; i++){
        free(matrix[i]);
    }
    free(matrix);
}

/* fw.h */
/* Finds shortest paths between all nodes and all other nodes.
    Allocates space for path matrix and stores this in the space pointed
    to by pathMatrix.
    size is the dimension of the matrix. */
void fw(int **adjMatrix, int ***pathMatrix, int size);

/* Prints the path from the source node given to the destination node
    given using the given path matrix. The path printed is new line
```

```
    terminated. */
void printPath(int **pathMatrix, int src, int dest);

/* Frees a path matrix and all the values within it. */
void freePathMatrix(int **pathMatrix, int size);

/* fw.c */
void fw(int **adjMatrix, int ***pathMatrix, int size){
    /* FILL IN */
}

void printPath(int **pathMatrix, int src, int dest){
    /* FILL IN */
}

void freePathMatrix(int **pathMatrix, int size){
    /* Fill in */
}

/* main.c */
#include <stdio.h>
#include <stdlib.h>
/* #include "adjmatrix.h" */
/* #include "fw.h" */

#define SIZE 10

int main(int argc, char **argv){

    int **graph = newMatrix(SIZE);
    int **pathMatrix = NULL;
    int i, j;

    addEdge(graph, 0, 4, 2);
    addEdge(graph, 1, 2, 2);
    addEdge(graph, 1, 3, 5);
    addEdge(graph, 1, 4, 10);
    addEdge(graph, 2, 3, 2);
    addEdge(graph, 3, 4, 2);
    addEdge(graph, 3, 5, 6);
    addEdge(graph, 4, 5, 2);
    addEdge(graph, 6, 7, 5);
    addEdge(graph, 6, 8, 9);
    addEdge(graph, 7, 8, 6);
    addEdge(graph, 8, 9, 3);

    printf("Running Floyd-Warshall\n");
    fw(graph, &pathMatrix, SIZE);

    printf("Shortest path distances:\n");
    for(i = 0; i < SIZE; i++){
        for(j = 0; j < SIZE; j++){
            if(graph[i][j] == NOPATH){
                printf("N/A ");
            } else {
                printf("%3d ",graph[i][j]);
            }
        }
        printf("\n");
    }
    for(i = 0; i < SIZE; i++){
        printf("Shortest path from %d:\n",i);
        for(j = 0; j < SIZE; j++){
            printf("to %d: ",j);
            printPath(pathMatrix, i, j);
        }
    }

    freeMatrix(graph, SIZE);
    freePathMatrix(pathMatrix, SIZE);

    return 0;
}
```

Programming Challenge 11.1 For this week's problem, you're helping out your buddy, Truckdriver Danni, who is contractually obligated to take the shortest path between her last job and the destination of her next job, but is paid by the kilometre. Danni asked you to help find the longest path, starting from the 0th node (home) and visiting the 1st node, 6th node and 9th node and then returning to the 0th node, only using the shortest paths between each source and destination so she can earn the most money from her job.

By pure chance, the graph is *exactly* like the one you saw in the workshop, except it is now undirected.

NOTE: For various reasons, this problem may be *very* hard to solve efficiently, so aim for something reasonable and simple that works instead.

```
/* adjmatrix.h */
/* NOTE: this could be made a lot more robust,
    but it's easier not to. */
#include <limits.h>
#define NOPATH INT_MAX/2
/* Allocates space for a size * size array
    and sets all values to the NOPATH value. */
int **newMatrix(int size);

/* Adds an edge between one given index and another. */
void addEdge(int **adjMatrix, int src, int dst, int weight);

/* Frees a matrix allocated with the newMatrix function. */
void freeMatrix(int **matrix, int size);

/* adjmatrix.c */
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
/* #include "adjmatrix.h" */

int **newMatrix(int size){
    int **retMatrix;
    int i, j;
    retMatrix = (int **) malloc(sizeof(int *)*size);
    assert(retMatrix);
    for(i = 0; i < size; i++){
        retMatrix[i] = (int *) malloc(sizeof(int)*size);
        assert(retMatrix[i]);
        for(j = 0; j < size; j++){
            retMatrix[i][j] = NOPATH;
        }
    }
    return retMatrix;
}

void addEdge(int **adjMatrix, int src, int dst, int weight){
    adjMatrix[src][dst] = weight;
}

void freeMatrix(int **matrix, int size){
    if(!matrix){
        return;
    }
    int i;
    for(i = 0; i < size; i++){
        free(matrix[i]);
    }
    free(matrix);
}

/* main.c */
#include <stdio.h>
#include <stdlib.h>
/* #include "adjmatrix.h" */

#define SIZE 10

int main(int argc, char **argv){
```

```
    int **graph = newMatrix(SIZE);
    int **pathMatrix = NULL;
    int home        = 0;
    int firstPoint  = 1;
    int secondPoint = 6;
    int thirdPoint  = 9;
    int i, j, k;

    addEdge(graph, 0, 4, 2);
    addEdge(graph, 1, 2, 2);
    addEdge(graph, 1, 3, 5);
    addEdge(graph, 1, 4, 10);
    addEdge(graph, 2, 3, 2);
    addEdge(graph, 3, 4, 2);
    addEdge(graph, 3, 5, 6);
    addEdge(graph, 4, 5, 2);
    addEdge(graph, 6, 7, 5);
    addEdge(graph, 6, 8, 9);
    addEdge(graph, 7, 8, 6);
    addEdge(graph, 8, 9, 3);

    /* Add reversed edges. */
    addEdge(graph, 4, 0, 2);
    addEdge(graph, 2, 1, 2);
    addEdge(graph, 3, 1, 5);
    addEdge(graph, 4, 1, 10);
    addEdge(graph, 3, 2, 2);
    addEdge(graph, 4, 3, 2);
    addEdge(graph, 5, 3, 6);
    addEdge(graph, 5, 4, 2);
    addEdge(graph, 7, 6, 5);
    addEdge(graph, 8, 6, 9);
    addEdge(graph, 8, 7, 6);
    addEdge(graph, 9, 8, 3);

    /*
        Find the ordering of points that
        gives the maximum distance using the
        shortest paths between each sequential
        point.
    */


    printf("The maximum ordering is %d (home) to #%d to #%d to #%d and back to %d\n",
        home, firstPoint, secondPoint, thirdPoint, home);

    freeMatrix(graph,SIZE);

    return 0;
}
```