Worksheet 10

Overview

The workshop for week 11 will start with a tutorial on Graph representation, and you will implement your first Graph algorithm.

Tutorial Questions

Question 10.1 In this question, you are asked to construct weighted graphs from records whose format is *vertex, vertex', weight*. For example, in a weighted, directed graph, the record 1,3,500 means that there is an edge from vertex 1 to vertex 3 with weight 500.

The following questions all refer to the input data below.

1 2 200

1 3 100

1 4 500

2 3 150

2 4 300

4 5 100

1. Draw a weighted directed graph that reflects these edges and weights (logical representation).
2. Construct an adjacency matrix for the weighted digraph you have just drawn, including the weights. Be explicit about how you are going to handle matrix cells for which there is no information in the data, and self-loops (edges between a node and itself).
3. If you are told that these inputs refer to a weighted *undirected* graph, how could your adjacency matrix change.
4. Which representation, adjacency matrix or adjacency list, would be most suitable for this graph?

Programming exercises

Programming 10.1 Given the weighted digraph G from the previous question where the edge weights represent distances between two vertices, use Dijkstra's Algorithm to find the shortest path from a selected vertex s to a target vertex t in G. You can use the priority queue from last week's workshop.

There's a fair bit involved in making the priority queue do what you want it to in a proper way, so something just a little hackier will be fine (such as casting it to a particular exposed data type and pulling the bit from it you need or passing a function to apply to return an integer representing the priority of the item).

```
/* digraph.h */
struct digraph;

/* Return type for edges.
    Linked list concrete data type. */
struct weightedEdge {
    int destIndex;
    int weight;
    struct weightedEdge *next;
};

/* Create a new weighted directed graph containing no edges. */
```

```
struct digraph *newDigraph();

/*
    Add an edge from the source to the destination with the given weight
    to the given graph.
*/
void addEdge(struct digraph *graph, int source, int destination, int weight);

/*
    Returns a list of edges which occur in the given graph, will be in order.
    Nodes without an edge to or from them will not be returned. This list should
    be freed by the caller. Value sent into size will be set to the size of the
    returned list. The final value in the returned list will also be given a value
    of graph->lowIndex - 1, so can be used as a sentinel (the size will exclude
    this value).
*/
int *getNodes(struct digraph *graph, int *size);

/*
    Returns the list of edges from a particular node in the graph as indices
    into the provided list. (ie a graph with nodes -1, 3, 4 and a list [-1,3,4],
    the edge list [[-1,50],[3,30],[4,20]] will be transformed to
    [[0,50],[4,30],[5,20]], this is for the purposes of abstracting out some of
    the internal behaviour of the digraph).
    The source index should be the value at the index in the nodeList given (so 0
    would be the index -1).
    The returned array is allocated as a list of weighted edges. This should be
    freed by the calling function.
    The original list is expected to obey the same constraints as that in the getNodes
    function.
*/
struct weightedEdge *getAdjacent(struct digraph *graph, int *nodeList, int sourceIndex);

/*
    Converts a given index to an index in the given node list (linear search). Could be
    binary search, but keeping it simple it should still be fast enough in this case.
*/
int convertIndex(int *list, int originalIndex);

/*
    Frees all the memory used by a particular graph.
*/
void freeDigraph(struct digraph *graph);

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

struct digraph{
    int lowIndex;
    int highIndex;
    struct weightedEdge **adjacencyList;
    int allocated;
    int used;
};

struct digraph *newDigraph(){
    struct digraph *retDigraph = (struct digraph *) malloc(sizeof(struct digraph));
    assert(retDigraph);

    /* Don't initialise until we have at least one edge so we don't go over-allocate. */
    retDigraph->adjacencyList = NULL;
    retDigraph->allocated = 0;
    retDigraph->used = 0;

    return retDigraph;
}

void addEdge(struct digraph *graph, int source, int destination, int weight){
    int initialSize;
    int newLow;
    int newHigh;
```

```
    int i;
    struct weightedEdge *newEdge;
    if(! graph->allocated){
        initialSize = source - destination;
        if(initialSize < 0){
            initialSize = -initialSize;
        }
        /* Assume initialSize positive now, theoretically it may not be, but
            we needn't worry about that for now. */
        initialSize = initialSize + 1;
        graph->adjacencyList = (struct weightedEdge **)
            malloc(sizeof(struct weightedEdge *)*initialSize);
        assert(graph->adjacencyList);
        graph->allocated = initialSize;
        /* Initialise edge list. */
        for(i = 0; i < initialSize; i++){
            (graph->adjacencyList)[i] = NULL;
        }
        /* Initialise low and high indices. */
        if(source < destination){
            graph->lowIndex = source;
            graph->highIndex = destination;
        } else {
            graph->lowIndex = destination;
            graph->highIndex = source;
        }
        graph->used = initialSize;
    } else {
        /* Ensure we have space for the new edge */
        if(graph->lowIndex > source || graph->lowIndex > destination){
            if(source < destination){
                newLow = source;
            } else {
                newLow = destination;
            }
        } else {
            newLow = graph->lowIndex;
        }
        if(graph->highIndex < source || graph->highIndex < destination){
            if(source > destination){
                newHigh = source;
            } else {
                newHigh = destination;
            }
        } else {
            newHigh = graph->highIndex;
        }
        if(newHigh != graph->highIndex || newLow != graph->lowIndex){
            /* Wasn't space for edge in current adjacency graph,
                realloc space, move all edge lists along. */
            graph->allocated = newHigh - newLow + 1;
            graph->adjacencyList = (struct weightedEdge **)
                realloc(graph->adjacencyList, sizeof(struct weightedEdge *)*graph->allocated);
            assert(graph->adjacencyList);
            /* Move from end of list to start of list to ensure none are overwritten. */
            for(i = graph->highIndex - graph->lowIndex; i >= 0; i--){
                graph->adjacencyList[i - newLow + graph->lowIndex] = graph->adjacencyList[i];
            }
            /* Fill in new low values. */
            for(i = 0; i < (graph->lowIndex - newLow); i++){
                graph->adjacencyList[i] = NULL;
            }
            /* Fill in new high values. */
            for(i = graph->highIndex - newLow + 1; i < (newHigh - newLow + 1); i++){
                graph->adjacencyList[i] = NULL;
            }

            graph->lowIndex = newLow;
            graph->highIndex = newHigh;
            graph->used = graph->allocated;
        }
    }

    /* Add the edge to the relevant place [prepend the list]. */
```

```
    newEdge = (struct weightedEdge *) malloc(sizeof(struct weightedEdge));
    assert(newEdge);
    newEdge->destIndex = destination;
    newEdge->weight = weight;
    newEdge->next = graph->adjacencyList[source - graph->lowIndex];
    graph->adjacencyList[source - graph->lowIndex] = newEdge;
}

int *getNodes(struct digraph *graph, int *size){
    /* A list of active edges, edges will be flipped on as traversed,
        the bits will then be counted and the exact amount of space needed
        will be allocated. */
    char *bitmask;
    int i, j;
    int count;
    int *returnList;
    struct weightedEdge *current;
    bitmask = (char *) malloc(sizeof(char)*graph->used);
    assert(bitmask);
    for(i = 0; i < graph->used; i++){
        /* This conversion is for clarity, '\0' would have done the trick
            too. */
        bitmask[i] = (char) 0;
    }
    /* Fill in all the bits from the adjacency list. */
    for(i = 0; i < graph->used; i++){
        if(graph->adjacencyList[i]){
            bitmask[i] = (char) 1;
            current = graph->adjacencyList[i];
            while(current){
                bitmask[current->destIndex - graph->lowIndex] = (char) 1;
                current = current->next;
            }
        }
    }

    count = 0;
    for(i = 0; i < graph->used; i++){
        if(bitmask[i]){
            count++;
        }
    }

    returnList = (int *) malloc(sizeof(int)*(count + 1));

    j = 0;
    for(i = 0; i < graph->used; i++){
        if(bitmask[i]){
            returnList[j] = i + graph->lowIndex;
            j++;
        }
    }
    /* Add sentinel value. */
    returnList[j] = graph->lowIndex - 1;

    if(size){
        *size = j;
    }

    free(bitmask);

    return returnList;
}

struct weightedEdge *getAdjacent(struct digraph *graph, int *nodeList, int sourceIndex){
    /* Find length of provided node list. */
    int length = 0;
    while(nodeList[length] >= graph->lowIndex){
        length++;
    }

    struct weightedEdge *current = NULL;
    struct weightedEdge *returnList = NULL;
    struct weightedEdge *currentGraph = NULL;
```

```
    currentGraph = graph->adjacencyList[nodeList[sourceIndex] - graph->lowIndex];
    if(currentGraph){
        while(currentGraph){
            /* Move current to its next value. */
            if(! current){
                returnList = (struct weightedEdge *) malloc(sizeof(struct weightedEdge));
                assert(returnList);
                current = returnList;
            } else {
                current->next = (struct weightedEdge *) malloc(sizeof(struct weightedEdge));
                assert(current->next);
                current = current->next;
            }
            /* Append modified value. */
            current->next = NULL;
            current->weight = currentGraph->weight;
            current->destIndex = convertIndex(nodeList, currentGraph->destIndex);
            currentGraph = currentGraph->next;
        }
        return returnList;
    } else {
        /* No edges for source index. */
        return NULL;
    }
}

int convertIndex(int *list, int originalIndex){
    int i = 0;
    while(list[i] != originalIndex){
        i++;
    }
    return i;
}

void freeDigraph(struct digraph *graph){
    int i;
    struct weightedEdge *current;
    struct weightedEdge *next;
    if(!graph){
        return;
    }
    for(i = 0; i < graph->allocated; i++){
        if(graph->adjacencyList[i]){
            current = graph->adjacencyList[i];
            graph->adjacencyList[i] = NULL;
            while(current){
                next = current->next;
                free(current);
                current = next;
            }
        }
    }
    if(graph->allocated > 0){
        free(graph->adjacencyList);
    }
    free(graph);
}

/* dijkstra.h */
/* #include "digraph.h" */
#include <limits.h>
#define NOPATH INT_MAX
struct dijkstraRes {
    /* The node that all paths originate from, index is into
        indices list. */
    int sourceNode;
    /* The shortest paths from the source node to
        the destination.  */
    int *shortestPaths;
    /* The list of destination indices that the shortest
        paths correspond to. */
    int *indices;
    /* If you like, you can also add an item for
```

```
        predecessors, but the actual path isn't
        being printed, so this isn't too important
        for the moment. Remember to update this in
        the free dijkstra struct function if you
        make memory-related changes to this struct. */

    /* The number of items in the index list. */
    int nodeCount;
};

/* Finds the shortest paths from the given source node to all
    other nodes in the graph (which have at least one edge in either
    direction) and returns a structure of these nodes.
    Assumes that the given source node is actually in the graph. */
struct dijkstraRes *dijkstra(struct digraph *graph, int source);

/*
    Prints all the shortest paths.
*/
void printDijkstra(struct dijkstraRes *res);

/*
    Frees a dijkstraRes struct and all the memory it
    is responsible for. (In the provided code, this is
    assumed to be: .shortestPaths and .indices, as well
    as the struct sent).
*/
void freeDijkstraRes(struct dijkstraRes *res);

/* dijkstra.c */
/* #include "digraph.h" */
/* #include "dijkstra.h" */
#include <stdlib.h>
#include <stdio.h>
/* #include "pqueue.h" */
/* NOTE: If you didn't write pqueue.h & pqueue.c last week, consider doing
    so before this exercise. */

struct dijkstraRes *dijkstra(struct digraph *graph, int source){
    /* FILL IN HERE */









}

void printDijkstra(struct dijkstraRes *res){
    int i;
    printf("Shortest paths from %d\n", res->indices[res->sourceNode]);
    for(i = 0; i < res->nodeCount; i++){
        if ((res->shortestPaths[i]) == NOPATH){
            printf("to %d: NO PATH;\n", res->indices[i]);
        } else {
            printf("to %d: %d;\n", res->indices[i], res->shortestPaths[i]);
        }
    }
}

void freeDijkstraRes(struct dijkstraRes *res){
    if(! res){
        return;
    }
    if(res->shortestPaths){
        free(res->shortestPaths);
    }
    if(res->indices){
        free(res->indices);
    }
    free(res);
}

/* main.c */
#include <stdio.h>
```

```
#include <stdlib.h>
/* #include "digraph.h"*/
/* #include "dijkstra.h"*/
#define EDGES 6

int main(int argc, char **argv){
    struct digraph *graph = NULL;
    struct dijkstraRes *res = NULL;
    int src[EDGES];
    int dest[EDGES];
    int weight[EDGES];
    int i = 0;
    src[i] = 1; dest[i] = 2; weight[i] = 200; i++;
    src[i] = 1; dest[i] = 3; weight[i] = 100; i++;
    src[i] = 1; dest[i] = 4; weight[i] = 500; i++;
    src[i] = 2; dest[i] = 3; weight[i] = 150; i++;
    src[i] = 2; dest[i] = 4; weight[i] = 300; i++;
    src[i] = 4; dest[i] = 5; weight[i] = 100;

    graph = newDigraph();

    for(i = 0; i < EDGES; i++){
        addEdge(graph, src[i], dest[i], weight[i]);
    }

    printf("Running Dijkstra's Algorithm on graph\n");
    res = dijkstra(graph, 1);

    printf("Results:\n");
    printDijkstra(res);

    freeDigraph(graph);
    freeDijkstraRes(res);

    return 0;
}
```

Programming Challenge 10.1 In this challenge question, you're involved in a city planning project which will be evaluating whether the estimated road capacity is likely to support the city goers who use them. A few simplifying assumptions have been made:

- All citizens will only take the shortest paths from one of the three major residential districts,
- All traffic occurs at the same time (that is, proper capacity is required at its worst),
- Citizens only work at one of ten workplaces (all other traffic is marginal),
- Micro-scale agent behaviour is not important (the main focus is simply capacity) and
- All roads are two-way and each road has separate, but equal, capacity in either direction.

The question you need to be able to answer from your results is whether all roads along the shortest paths from residential districts to get to their workplaces support the populations that need to traverse them. Where multiple shortest paths are present which could split traffic isn't too important, so you needn't worry about trying to deal with that.

The digraph in this question has been modified to add an additional property:

1. Road capacity, the maximum population which can travel along the edge

The data structure has otherwise remained mostly the same (though you are free to add additional properties if they make the task easier). An additional method *getAdjReference* gives you the actual list of weighted edges, this isn't necessarily the cleanest way to do it, but it should be fairly fast.

```
/* digraph.h */
struct digraph;

/* Return type for edges.
    Linked list concrete data type. */
struct weightedEdge {
```

```
    int destIndex;
    int weight;
    int capacity;
    struct weightedEdge *next;
};

/* Create a new weighted directed graph containing no edges. */
struct digraph *newDigraph();

/*
    Add an edge from the source to the destination with the given weight
    to the given graph.
*/
void addEdge(struct digraph *graph, int source, int destination, int weight, int capacity);

/*
    Returns a list of edges which occur in the given graph, will be in order.
    Nodes without an edge to or from them will not be returned. This list should
    be freed by the caller. Value sent into size will be set to the size of the
    returned list. The final value in the returned list will also be given a value
    of graph->lowIndex - 1, so can be used as a sentinel (the size will exclude
    this value).
*/
int *getNodes(struct digraph *graph, int *size);

/*
    Returns the list of edges from a particular node in the graph as indices
    into the provided list. (ie a graph with nodes -1, 3, 4 and a list [-1,3,4],
    the edge list [[-1,50],[3,30],[4,20]] will be transformed to
    [[0,50],[4,30],[5,20]], this is for the purposes of abstracting out some of
    the internal behaviour of the digraph).
    The source index should be the value at the index in the nodeList given (so 0
    would be the index -1).
    The returned array is allocated as a list of weighted edges. This should be
    freed by the calling function.
    The original list is expected to obey the same constraints as that in the getNodes
    function.
*/
struct weightedEdge *getAdjacent(struct digraph *graph, int *nodeList, int sourceIndex);

/*
    Returns the weighted edge list from the graph for the given index. This is a reference
    instead of a copy, so changes will be reflected in the graph itself.
*/
struct weightedEdge *getAdjReference(struct digraph *graph, int sourceIndex);

/*
    Converts a given index to an index in the given node list (linear search). Could be
    binary search, but keeping it simple it should still be fast enough in this case.
*/
int convertIndex(int *list, int originalIndex);

/*
    Frees all the memory used by a particular graph. All references to internal graph
    structures become invalid after this function is is called for that graph.
*/
void freeDigraph(struct digraph *graph);

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

struct digraph{
    int lowIndex;
    int highIndex;
    struct weightedEdge **adjacencyList;
    int allocated;
    int used;
};

struct digraph *newDigraph(){
    struct digraph *retDigraph = (struct digraph *) malloc(sizeof(struct digraph));
```

```
    assert(retDigraph);

    /* Don't initialise until we have at least one edge so we don't go over-allocate. */
    retDigraph->adjacencyList = NULL;
    retDigraph->allocated = 0;
    retDigraph->used = 0;

    return retDigraph;
}

void addEdge(struct digraph *graph, int source, int destination, int weight, int capacity){
    int initialSize;
    int newLow;
    int newHigh;
    int i;
    struct weightedEdge *newEdge;
    if(! graph->allocated){
        initialSize = source - destination;
        if(initialSize < 0){
            initialSize = -initialSize;
        }
        /* Assume initialSize positive now, theoretically it may not be, but
            we needn't worry about that for now. */
        initialSize = initialSize + 1;
        graph->adjacencyList = (struct weightedEdge **)
            malloc(sizeof(struct weightedEdge *)*initialSize);
        assert(graph->adjacencyList);
        graph->allocated = initialSize;
        /* Initialise edge list. */
        for(i = 0; i < initialSize; i++){
            (graph->adjacencyList)[i] = NULL;
        }
        /* Initialise low and high indices. */
        if(source < destination){
            graph->lowIndex = source;
            graph->highIndex = destination;
        } else {
            graph->lowIndex = destination;
            graph->highIndex = source;
        }
        graph->used = initialSize;
    } else {
        /* Ensure we have space for the new edge */
        if(graph->lowIndex > source || graph->lowIndex > destination){
            if(source < destination){
                newLow = source;
            } else {
                newLow = destination;
            }
        } else {
            newLow = graph->lowIndex;
        }
        if(graph->highIndex < source || graph->highIndex < destination){
            if(source > destination){
                newHigh = source;
            } else {
                newHigh = destination;
            }
        } else {
            newHigh = graph->highIndex;
        }
        if(newHigh != graph->highIndex || newLow != graph->lowIndex){
            /* Wasn't space for edge in current adjacency graph,
                realloc space, move all edge lists along. */
            graph->allocated = newHigh - newLow + 1;
            graph->adjacencyList = (struct weightedEdge **)
                realloc(graph->adjacencyList, sizeof(struct weightedEdge *)*graph->allocated);
            assert(graph->adjacencyList);
            /* Move from end of list to start of list to ensure none are overwritten. */
            for(i = graph->highIndex - graph->lowIndex; i >= 0; i--){
                graph->adjacencyList[i - newLow + graph->lowIndex] = graph->adjacencyList[i];
            }
            /* Fill in new low values. */
            for(i = 0; i < (graph->lowIndex - newLow); i++){
```

```
                graph->adjacencyList[i] = NULL;
            }
            /* Fill in new high values. */
            for(i = graph->highIndex - newLow + 1; i < (newHigh - newLow + 1); i++){
                graph->adjacencyList[i] = NULL;
            }

            graph->lowIndex = newLow;
            graph->highIndex = newHigh;
            graph->used = graph->allocated;
        }
    }

    /* Add the edge to the relevant place [prepend the list]. */
    newEdge = (struct weightedEdge *) malloc(sizeof(struct weightedEdge));
    assert(newEdge);
    newEdge->destIndex = destination;
    newEdge->weight = weight;
    newEdge->next = graph->adjacencyList[source - graph->lowIndex];
    newEdge->capacity = capacity;
    graph->adjacencyList[source - graph->lowIndex] = newEdge;
}

int *getNodes(struct digraph *graph, int *size){
    /* A list of active edges, edges will be flipped on as traversed,
        the bits will then be counted and the exact amount of space needed
        will be allocated. */
    char *bitmask;
    int i, j;
    int count;
    int *returnList;
    struct weightedEdge *current;
    bitmask = (char *) malloc(sizeof(char)*graph->used);
    assert(bitmask);
    for(i = 0; i < graph->used; i++){
        /* This conversion is for clarity, '\0' would have done the trick
            too. */
        bitmask[i] = (char) 0;
    }
    /* Fill in all the bits from the adjacency list. */
    for(i = 0; i < graph->used; i++){
        if(graph->adjacencyList[i]){
            bitmask[i] = (char) 1;
            current = graph->adjacencyList[i];
            while(current){
                bitmask[current->destIndex - graph->lowIndex] = (char) 1;
                current = current->next;
            }
        }
    }

    count = 0;
    for(i = 0; i < graph->used; i++){
        if(bitmask[i]){
            count++;
        }
    }

    returnList = (int *) malloc(sizeof(int)*(count + 1));

    j = 0;
    for(i = 0; i < graph->used; i++){
        if(bitmask[i]){
            returnList[j] = i + graph->lowIndex;
            j++;
        }
    }
    /* Add sentinel value. */
    returnList[j] = graph->lowIndex - 1;

    if(size){
        *size = j;
    }
```

```
    free(bitmask);

    return returnList;
}

struct weightedEdge *getAdjacent(struct digraph *graph, int *nodeList, int sourceIndex){
    /* Find length of provided node list. */
    int length = 0;
    while(nodeList[length] >= graph->lowIndex){
        length++;
    }

    struct weightedEdge *current = NULL;
    struct weightedEdge *returnList = NULL;
    struct weightedEdge *currentGraph = NULL;

    currentGraph = graph->adjacencyList[nodeList[sourceIndex] - graph->lowIndex];
    if(currentGraph){
        while(currentGraph){
            /* Move current to its next value. */
            if(! current){
                returnList = (struct weightedEdge *) malloc(sizeof(struct weightedEdge));
                assert(returnList);
                current = returnList;
            } else {
                current->next = (struct weightedEdge *) malloc(sizeof(struct weightedEdge));
                assert(current->next);
                current = current->next;
            }
            /* Append modified value. */
            current->next = NULL;
            current->weight = currentGraph->weight;
            current->destIndex = convertIndex(nodeList, currentGraph->destIndex);
            currentGraph = currentGraph->next;
        }
        return returnList;
    } else {
        /* No edges for source index. */
        return NULL;
    }
}

struct weightedEdge *getAdjReference(struct digraph *graph, int sourceIndex){
    if(!graph || sourceIndex > graph->highIndex || sourceIndex < graph->lowIndex){
        return NULL;
    }
    return (graph->adjacencyList)[sourceIndex - graph->lowIndex];
}

int convertIndex(int *list, int originalIndex){
    int i = 0;
    while(list[i] != originalIndex){
        i++;
    }
    return i;
}

void freeDigraph(struct digraph *graph){
    int i;
    struct weightedEdge *current;
    struct weightedEdge *next;
    if(!graph){
        return;
    }
    for(i = 0; i < graph->allocated; i++){
        if(graph->adjacencyList[i]){
            current = graph->adjacencyList[i];
            graph->adjacencyList[i] = NULL;
            while(current){
                next = current->next;
                free(current);
                current = next;
            }
        }
```

```
    }
    if(graph->allocated > 0){
        free(graph->adjacencyList);
    }
    free(graph);
}


/* main.c */
#include <stdio.h>
#include <stdlib.h>
/* #include "digraph.h" */
#define EDGES 27
#define RESIDENTIAL 3
#define WORKPLACES 10

int main(int argc, char **argv){
    /*
        For this problem, nodes with index 1 - 10 are workplaces 1 - 10 and
        nodes with index 11 - 13 are the residential districts 1 - 3
    */
    int src[EDGES];
    int dest[EDGES];
    int weight[EDGES];
    int capacity[EDGES];
    /*
        The number of workers from each residential district that work
        at each workplace.
    */
    int workers[RESIDENTIAL][WORKPLACES];
    struct digraph *graph = newDigraph();
    int i = 0, j;
    /*
        As edges are symmetric, only the first occurrence of each edge
        is included.
    */
    src[i] = 1 ; dest[i] = 5 ; weight[i] = 95 ; capacity[i] = 600; i++;
    src[i] = 1 ; dest[i] = 10; weight[i] = 81 ; capacity[i] = 200; i++;
    src[i] = 1 ; dest[i] = 11; weight[i] = 100; capacity[i] = 900; i++;
    src[i] = 1 ; dest[i] = 21; weight[i] = 50 ; capacity[i] = 500; i++;
    src[i] = 2 ; dest[i] = 12; weight[i] = 100; capacity[i] = 500; i++;

    src[i] = 2 ; dest[i] = 14; weight[i] = 50 ; capacity[i] = 500; i++;
    src[i] = 3 ; dest[i] = 12; weight[i] = 100; capacity[i] = 800; i++;
    src[i] = 3 ; dest[i] = 15; weight[i] = 60 ; capacity[i] = 400; i++;
    src[i] = 3 ; dest[i] = 17; weight[i] = 20 ; capacity[i] = 700; i++;
    src[i] = 4 ; dest[i] = 6 ; weight[i] = 109; capacity[i] = 800; i++;

    src[i] = 4 ; dest[i] = 12; weight[i] = 32 ; capacity[i] = 500; i++;
    src[i] = 4 ; dest[i] = 13; weight[i] = 16 ; capacity[i] = 600; i++;
    src[i] = 5 ; dest[i] = 1 ; weight[i] = 95 ; capacity[i] = 600; i++;
    src[i] = 5 ; dest[i] = 6 ; weight[i] = 37 ; capacity[i] = 900; i++;
    src[i] = 7 ; dest[i] = 18; weight[i] = 42 ; capacity[i] = 800; i++;

    src[i] = 7 ; dest[i] = 20; weight[i] = 50 ; capacity[i] = 900; i++;
    src[i] = 8 ; dest[i] = 14; weight[i] = 50 ; capacity[i] = 500; i++;
    src[i] = 8 ; dest[i] = 15; weight[i] = 40 ; capacity[i] = 400; i++;
    src[i] = 9 ; dest[i] = 22; weight[i] = 47 ; capacity[i] = 300; i++;
    src[i] = 10; dest[i] = 22; weight[i] = 55 ; capacity[i] = 400; i++;

    src[i] = 11; dest[i] = 20; weight[i] = 50 ; capacity[i] = 900; i++;
    src[i] = 12; dest[i] = 16; weight[i] = 135; capacity[i] = 900; i++;
    src[i] = 12; dest[i] = 13; weight[i] = 204; capacity[i] = 230; i++;
    src[i] = 16; dest[i] = 17; weight[i] = 168; capacity[i] = 900; i++;
    src[i] = 17; dest[i] = 19; weight[i] = 15 ; capacity[i] = 700; i++;

    src[i] = 18; dest[i] = 19; weight[i] = 61 ; capacity[i] = 800; i++;
    src[i] = 21; dest[i] = 22; weight[i] = 50 ; capacity[i] = 500;

    for(i = 0; i < EDGES; i++){
        addEdge(graph, src[i], dest[i], weight[i], capacity[i]);
        addEdge(graph, dest[i], src[i], weight[i], capacity[i]);
    }
```

```
    /* Workers from residential district 1. */
    workers[0][0] = 300; workers[0][1] = 200; workers[0][2] = 50 ;
    workers[0][3] = 40 ; workers[0][4] = 60 ; workers[0][5] = 80 ;
    workers[0][6] = 90 ; workers[0][7] = 70 ; workers[0][8] = 90 ;
    workers[0][9] = 20 ;

    /* Workers from residential district 2. */
    workers[1][0] = 80 ; workers[1][1] = 300; workers[1][2] = 200;
    workers[1][3] = 30 ; workers[1][4] = 35 ; workers[1][5] = 80 ;
    workers[1][6] = 100; workers[1][7] = 150; workers[1][8] = 10 ;
    workers[1][9] = 15 ;

    /* Workers from residential district 3. */
    workers[2][0] = 20 ; workers[2][1] = 10 ; workers[2][2] = 50 ;
    workers[2][3] = 500; workers[2][4] = 100; workers[2][5] = 200;
    workers[2][6] = 30 ; workers[2][7] = 5  ; workers[2][8] = 75 ;
    workers[2][9] = 10 ;

    /*
        Work out whether the sum of traffic over all workers
        and workplaces is greater than the capacity of any road
        which is taken to get to any workplaces from any of the
        residential districts.
    */


    freeDigraph(graph);

    return 0;
}
```