Solutions of Algorithims 420 Final Project

John Wesley Hayhurst

April 29, 2015

1 Problem 1: Who's Got Game

1.1 Sample Input and Output

For each test case, print a single line of output containing 'Infeasible game' if the proposed gameplay sequences are impossible, 'Linear gameplay' if exactly one sequence is possible, or 'Nonlinear gameplay possible' if multiple arrangements are possible (no matter how many arrangements there are). The input of the user at start up can have multiple games within it.

Sample Input:

5 4
1 5
5 2
3 2
4 3
5 4
3 1
4 2
1 5
5 4
2 2
1 2
2 1
0 0

Sample Output:

Nonlinear gameplay possible
Linear gameplay possible
Infeasible game

1.2 Discussion

The best approach in my opinion for this problem would be to imply a depth first search to check for if it is linear or if it is cyclic. Using this approach we do not need to worry about finding out if the graph passed in would be non-linear we would only have to worry about the first two cases.

The main method of the program takes command line arguments and then checks if the next two values are not 0 and 0. It will then check if the values of N and M are within their bounds. The program will then continue to run until the starting values of the next "game" are 0 and 0. The code will then set the first two integers as n and m respectively. It will then create an adjacency matrix of size $n + 1$ since there will not be a 0 node and the matrix is to represent nodes $1 \rightarrow n$. The next loop will then go through the next nodes adding them to the adjacency matrix with the matrix at position d holding the value u. It will check to make sure d and u fall within their bounds. The main method will then check if the "game" is linear, non-linear, or the game is infeasible. The main method invokes isCyclic and isLinear to check the game type.

The isCyclic method checks if there are cycles in the graph, the method also has a recursive method that helps detect the cycles. The non-recursive method then initializes a visited and recursive stack boolean array of size $V+1$ (Where V is the total number of nodes in the graph) since the adjacency matrix is shifted to make sure each node numerically corresponds to the adjacency matrix. Then for all of the nodes in the graph it will use the recursive call to check if there is a way to get back to the initial node from traversing the nodes linked to it. To detect this the code keeps track of the nodes currently in the recursive stack while doing a depth first search. This is achieved by checking if the node has not been visited and then marks it visited in the visited and recursive stack boolean array. It will then construct an iterator that will grab the child nodes from the adjacency matrix. If the child is the same as the parent, then the graph contains a self cycle and will return true. If it does not, it will then call itself recursively to make sure that it has not visited the child and continue the depth first search. Then if the recursive stack of the child is marked as true then it will return true as the child has been visited before. If it still has not detected a cycle it will then set the node corresponding to the recursive stack to false and then return false. This method is a modified DFS and thus has a complexity of $\mathcal{O}(V + E)$.

The isLinear method is similar to the isCyclic method in that it also has a recursive helper and uses a modified depth first search to check if the graph

is linear. The method will first check if any of the nodes have more than one edge. If a node has more than one edge then it will then have a branching pathway making non-linear gameplay possible thus making linear gameplay impossible. After the check the method will create a visited boolean array that will store if that node has been visited. Then it will invoke the recursive helper if the starting node has not been visited. The recursive helper has a base case if the number of visited nodes using the recursive call is the same size as the stack then the game must be linear. If the size of the graph using depth first search is not equal the size of the stack then it must either be a disjointed graph or the starting node is not the head of the graph and thus will return false. The function will then check the remaining unvisited nodes as the starting point of the depth first search. Please take note of the for loop in the recursive call. This might be a cause for concern to say the method is not optimal. The for loop exists to safely check for the child node of the parent. It does this by moving the iterator before executing the for loop to make sure then child is not null thus having a null pointer exception. Since this method uses a modified depth first search without adding complexity the complexity of this method would be $\mathcal{O}(V + E)$.

The program overall has a worst case cost of $\mathcal{O}(V+E)$ with constants being ignored. Without constants being ignored the worst case cost of program is $\mathcal{O}(2V+2E)$ because calling isCyclic costs $\mathcal{O}(V+E)$ and isLinear $\mathcal{O}(V+E)$. Adding these two costs together would yield $\mathcal{O}((V + E) + (V + E)) \Rightarrow \mathcal{O}(2V + 2E)$.

1.3 Code

```
#include <iostream>
#include <string>
#include <stdlib.h>
#include <vector>
#include <list>

/**
 * Recursively calls the function to check if the node has been not
    visited.
 * If it has not been visited then sets visited in the bool stack
    to be true and the recStack to be true as well to check later
    in recursive calls.
 * then loops through each node to check for the next node using
    DFS. If the node loops onto itself i.e. you need 4 to unlock 4
    then it would be cyclic. If the child node is not visited and
```

```
     the child node using the recursive call is able to make it
     back to the child then it will be cyclic.
 */
bool isCyclicRecursive(std::vector<int> *pMatrix, int V, bool
    visited[], bool *recStack){
    if (visited[V] == false){
        visited[V] = true;
        recstack[V] = true;

        for(std::vector<int>::const_iterator i = pMatrix[V].begin();
            i != pMatrix[V].end(); ++i){
            if (*i == V) return true;
            if (!visited[*i] && isCyclicRecursive(pMatrix,*i,
                visited, recStack))
                return true;
            else if (recStack[*i])
                return true;
        }
    }
    recStack [V] = false;
    return false;
}

/**
 * This method will check if the adjMatrix is cyclic or not
 * It inits the visited and recStack matrixes to false as they have
     not visited any node
 */
bool isCyclic(std::vector<int> *pMatrix, int V){

    bool *visited = new bool[V];
    bool *recStack = new bool[V];
    for (int i = 1; i < V; i ++){
        visited[i] = false;
        recStack[i] = false;
    }
    for (int i = 1; i < V; i++)
        if (isCyclicRecursive(pMatrix,i, visited, recStack))
            return true;
    return false;
}


/**
 * This is the recursive call for the Linear check
```

```
 * The for loop is to safely check for the end of the node and does
     not contribute to the function cost as the node would not have
     more than one child.
 * It increases the visited count until it reaches the last node
     witch will not exicute the for loop because of the
     pMatrix.begin() == pMatrix.end()
 * If it reaches the end of the recursive call then it will return
     false
 *
 * The recursive call has a cost of O(V+E)
 */
bool isLinearRecursive(std::vector<int> *pMatrix, bool
    (&visited)[], int node,int visitedCount){
    visited[node] = true;
    visitedCount++;
    //This is the base case it which the call will return true
    if(visitedCount == visited.size()-1) return true;

    for(std::vector<int>::const_iterator i = pMatrix[V].begin(); i
        != pMatrix[V].end(); ++i){
        //returns true for the call if at the end of the nodes the
            visited count will = the stackSize
        if(isLinearRecursive(pMatrix, visited, *i, visitedCount,
            stackSize))
            return true;
    }
    return false;
}

/**
 * This method will check id the adjMatrix is linear or not.
 * If the node has more than one child then it would be non linear
     gameplay thus would not be linear.
 * This would use DFS to check the children to see if it is the
     same as the stack size.
 * Would have an visited stack to check the next starting node.
 */
bool isLinear(std::vector<int> *pMatrix, int V){
    //This loop checks if the matrix has multiple children
        connecting it
    //if the loop has multiple children then the gameplay is non
        linear
    for (int k = 1; k < V; k++)
        if (pMatrix[k].size() > 1)
            return false;
```

```
    bool *visited = new bool[V];
    for (int i = 1; i < V; i++)
        visited[i] = false;

    for (int j = 1; j < V; j++){
        //V-1 is passed in because V is n+1 as to create a loop that
            will check pMatrix[1-n]
        //v-1 is also the size of the stack in this case
        if ( visited[j] = false){
            if (isLinearRecursive(pMatrix, visited, j, 0)) return
                true;
        }
    }
    return false;
}

/**
 * Main method for determining gameplay type
 */
int main(int argc,char* argv[]){

    //This initalizes the variables for use
    int d = 0; //D allows access to item or action U
    int u = 0; //U is the item or action
    int n = 0; //N is the number of items or tasks
    int m = 0; //M is the number of relations between items
    //Ends initilization of the variables

    int i = 1;
    while (atoi(argv[i]) != 0 || atoi(argv[i+1]) != 0){
        //Imagine that their care check in this method to see if N,
            M, D, U fall within their correct bounds

        //This loop inits values N and M and then incriments
        n = atoi(argv[i]);
        m = atoi(argv[i+1]);

        std::vector<int> *adjMatrix = new std::vector<int> [n+1];
        int k = i +2;
        for (int j = 0; j < m*2; j+=2){
            //This loop goes through the D and U variables and does
                something with them
            d = atoi(argv[k+j]);
            u = atoi(argv[k+1+j]);
```

```
            adjMatrix[d].push_back(u);

        }
        //printVector(adjMatrix, n+1);

        //This checks for what type of gameplay is possible
        if (isCyclic(adjMatrix, n+1)){
            std::cout << "Infeasable Game" << std::endl;
        }
        else if (isLinear(adjMatrix, n+1)){
            std::cout<< "Linear Gameplay" << std::endl;
        }
        else {
            std::cout << "Non-Linear Gameplay Possible" << std::endl;
        }

        //Incriments I to go to the next inputed value
        i += (atoi(argv[i+1])*2) + 2;
    }
}
```

2 Fun, Fun, Fun "Auf der Autobahn"

2.1 Sample Input and Output

For each route request, your program should output the starting point and the destination point as well as the total travel distance in km and the total distance traveled on the Autobahn (also in km).

Sample Input:

6
A B C D E F
7
A B 10 a
B C 10 a
D A 1 b
E B 5 b
F C 1 b
D E 10 b
E F 10 b

3
A B
D E
F D

Sample Output:
A B 10 10
D E 10 0
F D 22 20

2.2 Discussion

The best approach in my opinion would be to use Dijkstra's algorithm to find the shortest path between the start and the end node specified for this graph. The approach is modified to fit the problem so that the path will favor autobahns more than back roads and then will return the distance traveled on the autobahn. The way I approached this is to check when adding in the edge weights into the adjacency matrix I also make the weights for autobahn roads negative. This way when checking a backroad vs an autobahn the algorithm will favor the autobahn when picking its path. Then when adding the path values check a corresponding boolean adjacency matrix when adding the cost of the path values to add that cost to the distance travelled on the autobahn.

The main method of my code will read in the number of "destinations" or nodes within the graph. It will then put the "names" of the road represented by chars into an array of chars. This will give the chars a corresponding numerical value to be used later on in determining the shortest path. Then the method takes in the number of edges that it needs to loop through. The method initializes two 2D arrays that will build the adjacency matrix that will hold the cost of the edge weights between the two nodes and will build a corresponding boolean adjacency matrix that will tell if the edge weights are an autobahn road. The main method will then take in the starting node, the ending node, the edge weight (in this case distance) and if the road is autobahn or not. It will then use cipherCharToInt() to convert the corresponding "name" of the node to the numerical representation to be used in the adjacency matrix. If the edge weight passed in is an autobahn road then the distance will be negative in the adjacency matrix to give preference to that edge weight when calculating the shortest path. At the same time it will set the autobahn boolean adjacency matrix to true. The method will then take in the total number of shortest paths it will have to calculate.

Then it will calculate the shortest paths for inputted values. The main method has 3 for loops and another for loop that initializes the autobahn index to be false for all values. The first loop would have a cost of $\mathcal{O}(N)$ where N is the number of nodes in the graph. The next for loop would just cost the total number of edges but also calls cipherCharToInt() which has a loop of the size of the total number of nodes. The cipherCharToInt() is called twice in the loop so then the cost is $\mathcal{O}(2NE)$ with E being the number of edge weights. The last loop is the number of shortest paths and also calls the cipherCharToInt() two times as well. Ignoring the cost of invoking the dijkstra() method, the cost is $\mathcal{O}(2NS)$ with S being the number of shortest paths to calculate. The cost of the entire main method excluding the cost of calculating the shortest path would be $\mathcal{O}(N + 2NE + 2NS)$.

The dijkstra's algorithm is used to calculate shortest path to all of the nodes from the starting node. It has a helper method minDistance which will determine the next node to build the path by checking the edge weight giving preference to nodes that will have negative edges because they are autobahn. The minDistance also makes sure that the negative weights' absolute value is taken. the dijkstra method then creates and initializes a distance array and a shortest path array which are used to help calculate the best node to be used in the minDistance method. Since the distance of the source node will always be zero the method will set that distance to zero. Then for the size of the nodes in the graph. It will then use the minDistance function to get the index with the least edge weight. It will then set the shortest path array corresponding to the index just gotten set to true. Then the method invokes another loop inside the first one to determine and populate the values of distance array. It makes sure to check and use absolute value when calculating the distance to avoid adding negative weights to the distance array. It also checks if the corresponding distance being added is an autobahn distance and then adds it to the autobahn traveled distance. It will then print the result of the distance traveled by calling the printResult method witch does not invoke any significant cost as it is $\mathcal{O}(1)$. There was no major changes to dijkstra's algorithm just slight modifications to make it work to fit the problem. The cost of the algorithm is $\mathcal{O}(V^2)$ with V being the number of nodes within the graph.

The worst case scenario for this program to run would then be $\mathcal{O}(V^2)$ with dijkstra's algorithm outweighing any cost of invoking the main method.

2.3 Code

```
#include <iostream>
#include <string>
#include <stdlib.h>
#include <cmath>
/**
 * finds the index of the minimum distance value of the verticies
     not yet in the shortest path tree.
 * Will check if the edge weight is negative to make sure that an
     autobahn is chosen over a normal road
 */
int minDistance(int distance[], bool shortestPath[],int
    numberOfNodes){
    int min = INT_MAX;
    int minIndex;
    for (int i = 0; i < numberOfNodes, i++){
        if (shortestPath[i] == false && distance[i] <= min &&
            distance[i] > 0){
            min = distance[i];
            minIndex = i;
        }
        if (shortestPath[i] == false && distance[i] <= min &&
            distance[i] < 0){
            if (std::abs(distance) <=min){
                min = distance[i];
                minIndex = i;
            }
        }
    }
    return minIndex;
}

/**
 * This is the cipher to change the char values to the
     corrisponding integer values to be used in dijkstra's
     algorithim
 */
int cipherCharToInt(char charMatrix[], char to){
    for (int i = 0; i< charMatrix.size(); i++){
        if (to == charMatrix[i]) return i;
    }
}

/**
 * This method prints the results of the dijkstra algorithim
 */
```

```
void printResult(int start, int end, int distance, int autoBahnKM,
    char charMatrix[]){
    char charStart = charMatrix[start];
    char charEnd = charMatrix[end];
    std::cout << charStart << " " << charEnd << " " << distance <<
        " " << autoBahnKM << std::endl;
}

/**
 * Uses dijkstra's algorithim to check for all the distances from a
    starting part and then has an array
 * containing all the distances to the nodes.
 * Calls the printResults() method to print the result of the
    shortest path from the src to the end
 */
void dijkstra(int adjMatrix[][], int src, int end, bool
    autoBahnIndex[][], int numberOfNodes, char charMatrix[]){
    int distance[numberOfNodes];
    bool shortestPath[numberOfNodes];
    for (int i = 0; i < numberOfNodes; i++){
        distance[i] = INT_MAX;
        shortestPath[i] = false;
    }
    //the distance from a node to itself will always be zero
    distance[src] = 0;

    for (int k= 0; count < numberOfNodes; k++){
        int mindex = minDistance(distance, shortestPath,
            numberOfNodes);
        //will mark the index as being used in the shortest path
        shortestPath[mindex] = true;

        for (int j = 0; j < numberOfNodes; j++){
            if (!shortestPath[j] && std::(adjMatrix[mindex][j]) &&
                std::abs(distance[mindex]) != INT_MAX &&
                std::abs(distance[mindex]) +
                std::abs(adjMatrix[mindex][j]) <
                std::abs(distance[j])){
                //makes sure that the distance that is added is not
                    negative
                distance[j] = std::abs(distance[mindex]) +
                    std::abs(adjMatrix[mindex][j]);
                if (autoBahnIndex[mindex][j]){
                    //Adds the traveled miles from the adjMatrix to
                        the total autoBahn miles if their is an
```

```
                    autoBahn path at that location
                autoBahnKM += std::abs(adjMatrix[mindex][j]);
            }
        }

    }
}
printResult(start, end, distance[end], autoBahnKM, charMatrix);
}



/**
 * Implements the adjMatrix to have negative edges to make checking
    for auto Bahn easier.
 * A cipher will be implemented that will make the use of
    dijkstra's algo using an array of ints rather than chars to
    make implementation easier.
 *
 */
int main(){
    int numberOfNodes;
    std::cin >> numberOfNodes;
    char charMatrix[numberOfNodes];
    for (int i = 0, i < numberOfNodes; i++)
        std::cin <<charMatrix[i];

    int numberOfEdges;
    cin >> numberOfEdges;
    int adjMatrix[numberOfNodes][numberOfNodes];
    bool autoBahnIndex[numberOfNodes][numberOfNodes];

    //Inits the autoBahnIndex to be false
    for (int k = 0; k < numberofNodes; k++)
        for (int l = 0; l < numberOfNodes; l++)
            autoBahnIndex[k][l] = false;

    for (int j = 0; j < numberOfEdges; j++){

        char charStart;
        char charEnd;
        int distance;
        int start;
        int end;
```

```
        char autoBahn;

        std::cin << charStart;
        std::cin << charEnd;
        std::cin << distance;
        std::cin << autoBahn;

        start = cipherCharToInt(charMatrix, charStart);
        end = cipherCharToInt(charMatrix, charEnd);

        if (autoBahn == a){
            //If the road is autobahn then the edge weight for the
                distance will be negative
            adjMatrix[start][end] = distance *-1;
            adjMatrix[end][start] = distance *-1;
            autoBahnIndex[start][end] = true;
            autoBahnIndex[end][start] = true;
        }
        else{
            adjMatrix[start][end] = distance;
            adjMatrix[end][start] = distance;
        }
    }
    int numberOfShortestPaths;
    std::cin << numberOfShortestPaths;

    for (int m = 0; m < numberOfShortestPaths; m++){
        char charStart;
        char charEnd;
        int start;
        int end;

        std::cin << start;
        std::cin << end;

        start = cipherCharToInt(charMatrix, charStart);
        end = cipherCharToInt(charMatrix, charEnd);

        dijkstra(adjMatrix, start, end, autoBahnIndex,
            numberOfNodes, charMatrix);
    }
}
```

3 Closing Remarks

This project was interesting and fun to work on. I feel that if given more time to do the project and without the stress of finals week I would be better able to implement some of my methods and fix any bugs that could / would arise.