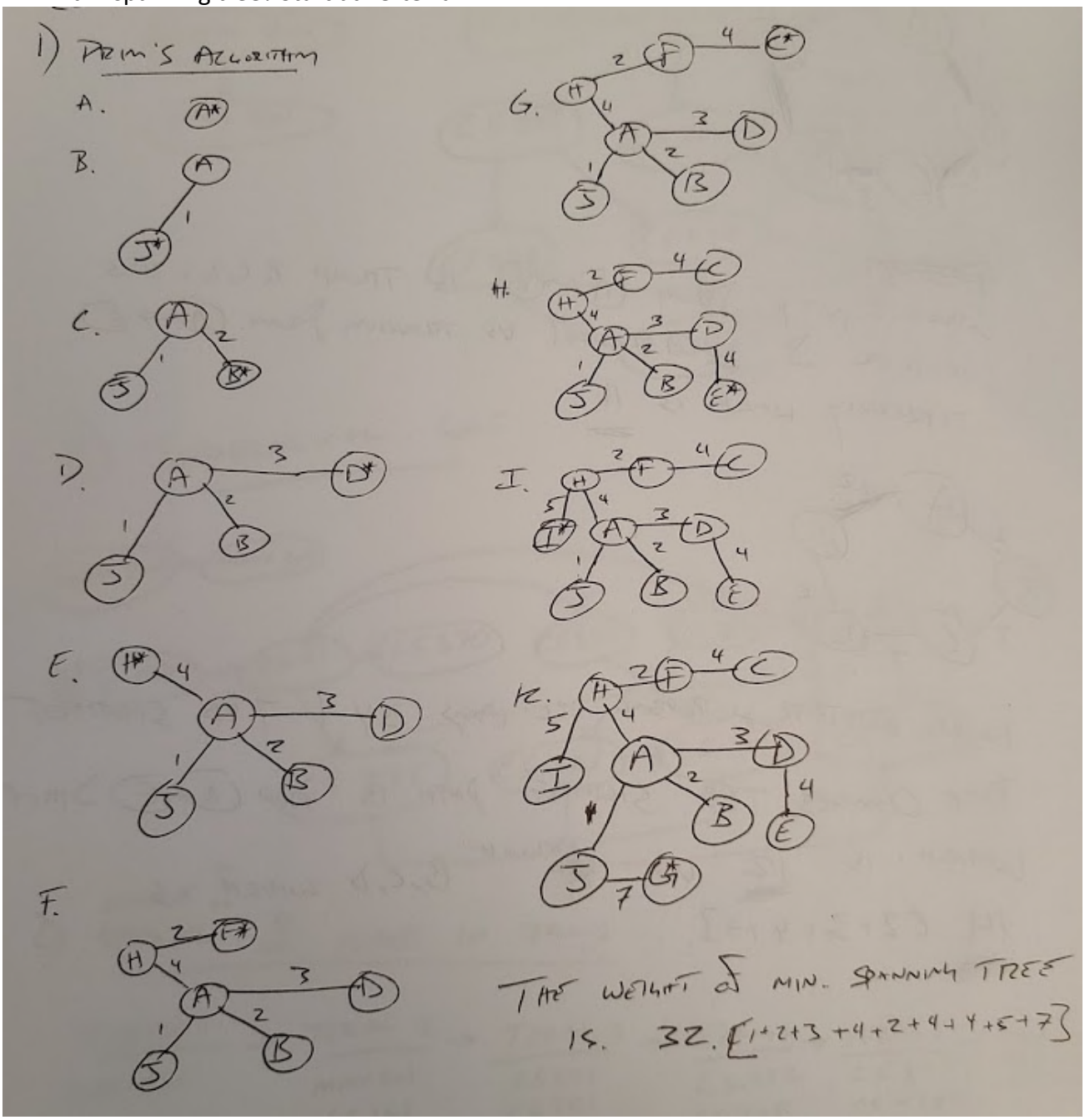
Zuhair Ahmed
CS325_400
Spring 2020

Homework #5

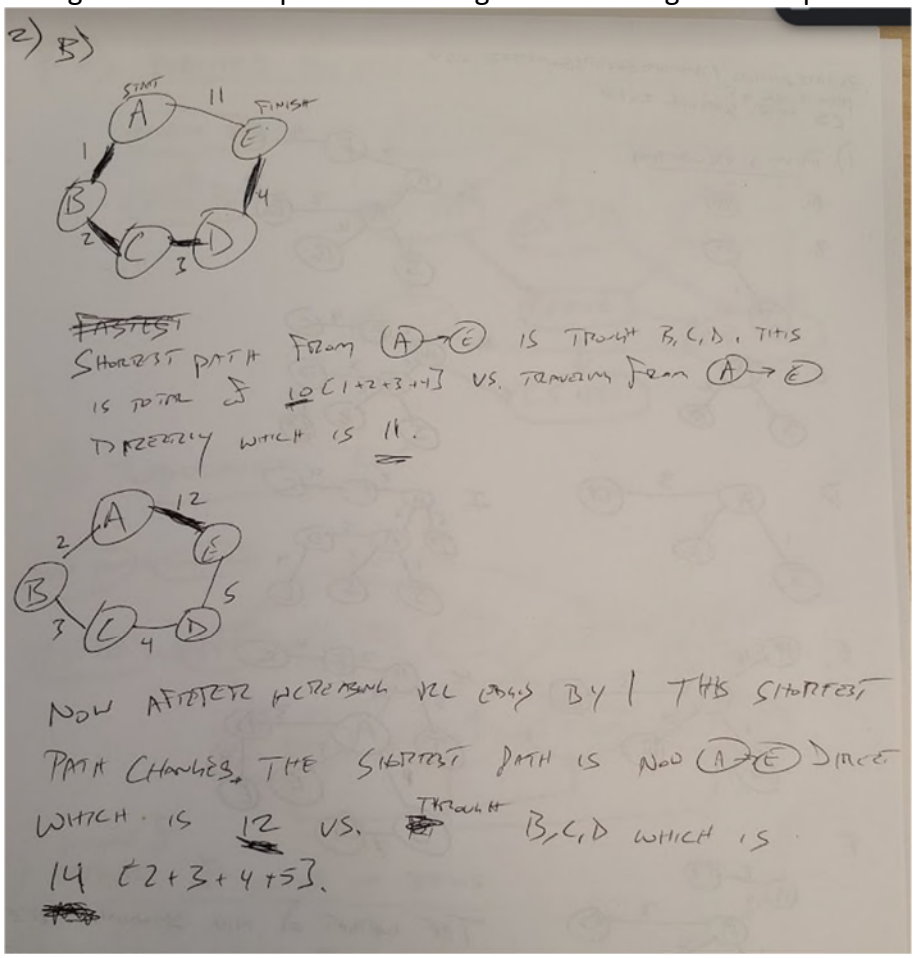
1. (3 points) Demonstrate Prim's algorithm on the graph below by showing the steps in subsequent graphs as shown in Figures 23.5 on page 635 of the text. What is the weight of the minimum spanning tree? Start at vertex a.



2. (6 points) Consider an undirected graph G=(V,E) with nonnegative edge weights w(u,v) > 0. Suppose that you have computed a minimum spanning tree G, and that you have also computed shortest paths to all vertices from vertex s exists in V. Now suppose each edge weight is increased by 1, the new weights w'(u,v) = w(u,v) + 1.

(a) Does the minimum spanning tree change? Give an example it changes or prove it cannot change.
No the MST should not change. This can be seen anecdotally in above diagram showing MST via Prim's algorithm. At each stage contains the optimal sub-problem (since this is a greedy algorithm). So if each of the edge weight is increased by 1 this will give us same solution at each stage as well as final result.
For simple proof by contradiction, assume that A(T) is the MST of a particular graph and B(T) is the MST of almost identical graph but each edge is increased by 1. If such a B(T) did exist that is unique / different from A(T), then A(T) could not have MST of prior graph as both graphs are identical with the exception of weight increase. This of course is a contradiction as A(T) is indeed the MST of original graph so it must also therefore be MST of graph with edges increased by 1 and no such unique B(T) can exist.

(b) Do the shortest paths change? Give an example where they change or prove they cannot change. Yes shortest paths can change. See drawing for example:



3. (4 points)In the bottleneck-path problem, you are given a graph G with edge weights, two vertices s and t and a particular weight W; your goal is to find a path from s to t in which every edge has at least weight W.

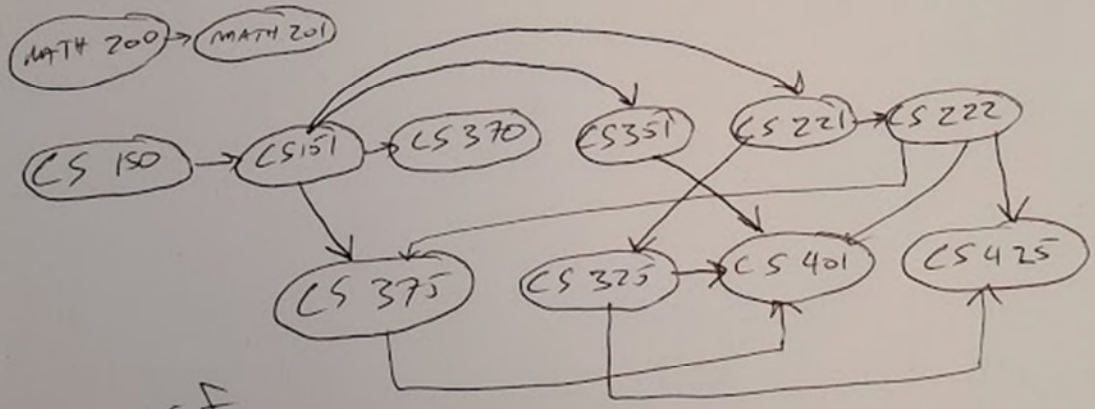
(a) Describe an efficient algorithm to solve this problem.
Example of an efficient algorithm could be a modified Breadth First Search (BFS) skipping over all edges with weight less than W. BFS is an traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the neighbor nodes (nodes which are directly connected to source node). This is analogous to "spilling ink on the page" and visiting all nodes with weight W or greater once. If t (finish node) is found, the algorithm will break and return all the parents that represent path from s (starting node) to t. If t is never found the algorithm will return false.

(b) What is the running time of your algorithm.
Run time should be O(V + E) or Big Oh of # of vertices and edges of the graph. This is because we must visit all vertices and only edges with weight W (in worst case) thus giving us a Minimum Spanning Tree (MST).

4. (5 points) Below is a list of courses and prerequisites for a factious CS degree.
(a) Draw a directed acyclic graph (DAG) that represents the precedence among the courses.
(b) Give a topological sort of the graph.
(c) If you are allowed to take multiple courses at one time as long as there is no prerequisite conflict, find an order in which all the classes can be taken in the fewest number of terms.
(d) Determine the length of the longest path in the DAG. How did you find it? What does this represent?
SEE NEXT PAGE

4) A) DIRECTED ACYCLIC GRAPH (DAG)

B) TOPOLOGICAL SORT.



C) EXAMPLE OF ORDER OF CLASSES IN TERMS

TERM 1	TERM 2	TERM 3	TERM 4	TERM 5	TERM 6
MATH 200	MATH 201	CS 221	CS 222	CS 375	CS 401
CS 150	CS 151	CS 351	CS 325	CS 425	
		CS 370			

D) LONGEST PATH IS 5 (N-1). SINCE IN PART C THERE 6 TERMS (N) (EDGES)
WE TAKE N-1 OR 5(6-1) TO FIND LONGEST PATH. THIS REPRESENTS THE EDGES BETWEEN TERMS.

5. (12 points) Suppose there are two types of professional wrestlers: “Baby faces” (“good guys”) and “Heels” (“bad guys”). Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n wrestlers and we have a list of r pairs of rivalries.

(a) Give pseudocode for an efficient algorithm that determines whether it is possible to designate some of the wrestlers as Baby faces and the remainder as Heels such that each rivalry is between a Baby face and a Heel. If it is possible to perform such a designation, your algorithm should produce it.

We should first draw the rivalries as a graph where as G(V, E). This means that each vertex represents a wrestler and each edge represents a rivalry. This is similar to the Two Coloring Problem described in the textbook which tries to color the vertices of graph of rivalries by two colors of "baby faces" and "heels". The graph will contain n vertices and r edges.

Pseudocode: We will complete BFS (Breath First Search) operations as required to hit all vertices

- Associate wrestlers whose distance is even to be “baby faces” and wrestlers whose distance is odd to be “heels”. This essentially means that “baby faces” are one color and “heels are another color.
- At each edge confirm whether the coloring is valid edge between two vertices of "baby faces" and "heels" then a rivalry can exists. If possible to find a designation if the coloring edge exists then this is valid. If the edge fails then it this is Not possible.

```
int wrestlers[999][999]; //double array of wrestlers and team
for (int i = 0; i < n; i++)
        wrestlers[i] = white; //if vertex exists then color white
for (int j = 0; j < k; i++)
{
        if (wresters[i] == white) //cycle through and if white then associate with rival
        {
                wrester[i][j] = “baby face”;
                wrester[i][k] = “heel”;
                addQueue(wrestler[i]); //add to end of queue
        }
        else //then it is not possible
                return Not Possible
}
```

(b) What is the running time of your algorithm?

The algorithm will take O(n + r) which is time for BFS. O(n) time to designate each wrestler and O(r) for each of the rivalries which is O(of number of wrestlers plus number of rivalries). So the overall running time of the algorithm is O(n + r).

(c) Implement: Babyfaces vs Heels.
Uploaded to both TEACH and CANVAS

```
/*******************************************************
 * Author: Zuhair Ahmed (ahmedz@oregonstate.edu)
 * Date Created: 5/10/2020
 * Filename: wrestling.cpp
 * Overview: Implemention of HW Problem #5c)
 *         Algorithm that determines whether it is possible to designate some
 *         of the wrestlers as Babyfaces and the remainder as Heels such that
 *         each rivalry is between a Babyface and a Heel. If it is possible
 *         to perform such a designation, your algorithm should produce it.
 * Input: .txt file whose name must be inputed at run time
 * Output: prints to terminal
 *******************************************************/

//included below necessary c++ std clases
#include<fstream>
#include<iostream>
#include<cstdio>
#include<vector>
#include<map>
#include<cstring>

//BreadthFirstSearch function to be called after main function
bool bfs(int position, int parentNode);

//Global int vector variable used in both main and bfs functions
#define MAXSIZE 999999
std::vector<int> adjMatrix[MAXSIZE]; //Vector of adjency matrix
bool visited[MAXSIZE];
int babyfaces[MAXSIZE];

//MUST include .txt file name after "./a.out" when running from terminal
int main(int argc, char* args[])
{
  int r; //number of rivals
  int n; //number of wrestlers
  int u; //index value of rival1
  int v; //index value of rival2

  //flag to check if possible to designate some wresterls as Babyface and
  //others as Heels
  bool possibleFlag = 1;
```

```
//create a map/hash table variable called data to store wresterls with counts
std::map<std::string, int> data1;

//string varibles to be used
std::string babyname;
std::string rival1;
std::string rival2;

 //**START OF FILE INPUT OPERATIONS**
   std::ifstream infile(args[1]); //input from user typing into terminal
   if (!infile) //error handling on file
   {
      std::cout << "ERROR CANNOT OPEN FILE!" << std::endl;
      return 0;
   }

   //input number of wrestlers from file
   infile >> n;

   //array of strings to store names of length n+1
   std::string wrestlerNames[n + 1];

   //read the names and store in map hash table
   for (int x = 0; x < n; x++)
   {
      infile >> babyname;
      data1[babyname] = x;
      wrestlerNames[x] = babyname;
   }

   //input number of rivals from file
   infile >> r;

   //input rivalries listed in pairs from file
   for (int x = 0; x < r; x++)
   {
      infile >> rival1 >> rival2;

      //index value of rival1
      u = data1[rival1];

      //index value of rival2
      v = data1[rival2];
```

```
      //including edges on to graph
      adjMatrix[u].push_back(v);
      adjMatrix[v].push_back(u);
   }

   infile.close();
//**END OF FILE INPUT OPERATIONS**

//memset all wrestlers as heels at first
memset(babyfaces, 0, sizeof babyfaces);

//breathFirstSearch nodes are all nodes declared unvisited at first
memset(visited, 0, sizeof visited);

//set babyfaces values to null
babyfaces[0] = 0;

//runs bfs function for connected components only
for (int x = 0; x < n; x++)
{
   if (visited[x])
      continue;

   //call the function bfs().
   possibleFlag = bfs(x, 0);

   //if not bipartite the partitioning is impossible
   if (!possibleFlag)
      break;
}

if (!possibleFlag)
   std::cout << "No. Not Possible to designate between Babyface and Heels \n";
else
{
   //intialize vector string of faces1 and heels1
   std::vector<std::string> faces1;
   std::vector<std::string> heels1;

   for (int x = 0; x < n; x++)
   {
      //push names into babyfaces if true
      if (babyfaces[x])
```

```
            faces1.push_back(wrestlerNames[x]);
        //else push into heels
        else
            heels1.push_back(wrestlerNames[x]);
    }

    std::cout << "Yes" << std::endl;
    std::cout << "Babyfaces: ";

    //output to console names of babyfaces
    for (int x = 0; x < faces1.size(); x++)
        std::cout << faces1[x] << " ";
    std::cout << std::endl;

    //output to console names of babyfaces
    std::cout << "Heels: ";
    for (int x = heels1.size() - 1; x >= 0; x--)
        std::cout << heels1[x] << " ";
    std::cout << std::endl;
  }

  return 0;
}

// bfs function is BOOL and confirms if graph of rivalries is bipartite
bool bfs(int position, int parentNode)
{
  //mark the position as visited
  visited[position] = 1;
  int value;

  //assign the node value different to its parent
  babyfaces[position] = 1 - babyfaces[parentNode];

  //iterate the adjacency matrix
  for (int x = 0; x < adjMatrix[position].size(); x++)
  {
    value = adjMatrix[position][x];

    //if the value is not visted call bfs recursively
    if (!visited[value])
      bfs(value, position);
    else
    {
```

```
        //if two adjacent nodes have same value, graph cannot be bipartite
        if (babyfaces[value] == babyfaces[position])
           return false;
      }
   }

   return true;
}
```