CSE 4304:Data Structure Lab-08
Graph Basic

Namisa Najah Raisa 210042112

November 2023

1 Problem A:Back to underworld

This problem is asking to find out the maximum number of whichever raced individual that participated in a duel. As input it takes information of the two individuals but not their races.So for each test case I have to find out the maximum count of an individual.

```cpp
#include <iostream>
#include <vector>
using namespace std;

vector<int> g[20009];
bool vis[20009];
int xx, yy;

void dfs(int source, bool p)
{
    if (vis[source]) return;
    vis[source] = true;
    if (p)
        xx++;
    else
        yy++;
    int sz = g[source].size();
    for (int i = 0; i < sz; i++)
        dfs(g[source][i], !p);
}

int main()
{
    int t, n, a, b;
    cin >> t;
    for (int i = 1; i <= t; i++)
    {
        for (int j = 0; j < 20009; j++)
            g[j].clear();

        for (int j = 0; j < 20009; j++)
            vis[j] = false;

        cin >> n;

        for (int j = 1; j <= n; j++)
        {
            cin >> a >> b;
            g[a].push_back(b);
            g[b].push_back(a);
        }

        int ans = 0;
        for (int j = 1; j < 20009; j++)
        {
            if (!vis[j] && !g[j].empty())
            {
                xx = yy = 0;
                dfs(j, true);
                ans += max(xx, yy);
            }
        }
        cout << "Case " << i << ": " << ans << endl;
    }
    return 0;
}
```

I solved this using Depth-First Search to traverse the graph.

Here g[] is an array of vectors which is used to represent an adjacency list of a graph.Each g[i] is a vector containing neighbours of vertex 'i'. bool vis[] declares an array booleans.It marks whether a vertex was visited during DFS por not.The 'xx' and "yy' variables are for counting the number of members for the two different races.

The **'void dfs(int source,bool p)'** takes two parameters.'source'—>The vertex from which the traversal will start.'p'—>A boolean variable to keep track of the current race(vampire or lykan)being assigned to the members during the traversal.

'if(vis[source]) return'—>if vis[source] is true,it means that vertex has been visited and it returns early.This is to avoid infinite loops in the DFS.

'vis[source]=true'—>this marks the current 'source' vertex as visited so that it's not visited again during DFS.

'if(p)xx++;else yy++' —> refers that if p is true then it is one of the races and it increments that member. If it's not true then the other member gets incremented.

'int sz=g[source].size()'—>counts the number of neighbours of the current 'source' vertex by finding out the number of elements of 'g[source]' which contains the neighbours of the 'source' vertex.

The FOR loop after that iterates through all the neighbours of the current 'source' vertex.For each neighbour it calls the dfs() function recursively,passing the neighbour as the new 'source' and switching the 'p' by '!p'.It means that if the current vertex is associated with one race (vampire), the neighbor will be associated with the other race (lykan), ensuring that the two races alternate as the DFS progresses through the graph.

In the main function t,n,a,b refers to the number of test cases,number of dual fights,and the pairs of individuals involved in dual fights respectively.

The first FOR loop iterates for the number of test cases.

'for (int j =0; j < 20009; j++) g[j].clear();'—> This loop clears the adjacency list represented by the g array for the current test case. It ensures that the graph is reset before working for each new test case.

'for (int j = 0; j < 20009; j++) vis[j] = false;'—> This loop initializes the vis array, marking all vertices as unvisited for the current test case. This is important to ensure that the DFS traversal starts from an unvisited vertex.

'for (int j = 1; j <= n; j++)'—> This is a loop that iterates through each dual fight for the current test case. It starts from j = 1 and continues until j reaches the value of n.Takes a,b inputs.

'g[a].`push_back(b)`;g[b].push`push_back(a)`'→These lines add the individuals a and b to each other's list of neighbors in the adjacency list g. This is done because dual fights are represented as edges in an undirected graph, and these lines ensure that the graph reflects the connections between individuals involved in the fights.

'int ans = 0;'→ This line initializes the ans variable to zero.This will store the maximum possible number of members for one of the races within a connected component of the graph.

'for (int j = 1; j < 20009; j++)'→ This is a loop that iterates through each vertex in the range from 1 to the size of the array. It aims to find connected components and calculate the maximum possible number of members for one of the races in each connected component.

'if (!vis[j] && !g[j].empty())'→ This conditional checks if the vertex j has not been visited (!vis[j]) and if it has at least one neighbor, meaning it is part of a connected component (!g[j].empty()). If these conditions are met, it means a new connected component is encountered.

In that IF condition **'xx = yy = 0;dfs(j, true);ans+=max(xx, yy);'**→initializes xx(vampire) and yy(lykan) to zero.Then calls the dfs() function to start the traversal from 'j'. The 'true' indicates that the initial race is vampire.After the DFS traversal 'ans' calculates the maximum possible number of members.

At last it just prints the output.

________________X________________

2 Problem B:Kefa and Park

In this problem I had to find the number of distinct leaves in the tree where Kefa can go to a restaurant without encountering more than 'm' consecutive vertices with cats on the path from house to the restaurant.

```cpp
#include <iostream>
#include <vector>
using namespace std;
const int MAX = 100001;
vector<int> tree[MAX];
int cats[MAX];
int consecutive_cats[MAX];
int n, m, ans;
void dfs(int v, int parent, int total_cats)
{
    if (cats[v] == 1)
    {
        total_cats++;
        if (total_cats > m)
        {
            return;
        }
    }
    else
    {
        total_cats = 0;
    }
    if (tree[v].size() == 1 && v != 1)
    {
        ans++;
        return;
    }
    for (int u : tree[v])
    {
        if (u != parent)
        {
            dfs(u, v, total_cats);
        }
    }
}
int main()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
    {
        cin >> cats[i];
    }
    for (int i = 1; i < n; i++)
    {
        int x, y;
        cin >> x >> y;
        tree[x].push_back(y);
        tree[y].push_back(x);
    }
    ans = 0;
    dfs(1, 0, 0);
    cout << ans << endl;
    return 0;
}
```

I solved this using DFS to traverse the tree.

'const int MAX=100001' refers to the maximum number of vertices in the tree.

'vector<int> tree[MAX]' refers to the tree structure.tree[i] contains the neighbours of vertex i.

'int cats[MAX]' refers to the information of the cats present at each vertex of the tree.

'int consecutive_cats[MAX];' is intended to keep tack of the consecutive cats encountered in the path from the root to the current vertex.

'int n,m,ans' refers to the vertices of the tree,maximum number of consecutive vertices with cats that Kefa is okay with,the final count of the the leaves where Kefa can go to a restaurant without encountering too many consecutive cats.

'void dfs(int v, int parent, int total_cats())'→ takes 3 parameters.'v' is the current vertex being visited,'parent' is the parent of the current vertex,'total_cats is the number of consecutive cats encountered so far on the path from the root to the current vertex.

'if (cats[v]==1)' checks if the current vertex has a cat(represented by 1).If it has then the total_cats is incremented(indicates one more consecutive cat has been encountered).If it exceeds the number of allowed consecutive cats(>m) it returns from the current DFS branch.Kefa can't go on this path.

'else'→If the current vertex does not have a cat it resets the total_cats count to zero(indicating consecutive cats are only counted when encountered and are reset to zero when a non-cat vertex is visited.

'if (tree[v].size() == 1 && v != 1)'→This checks if the current vertex is a leaf node and if it's not the root.If these conditions are met,it means that 'v' is a restaurant Kefa can go to and 'ans' is incremented.Then it returns from the current DFS branch.No need to go further.

'for (int u : tree[v])'→This FOR loop iterates over all the neighbours 'u' of the current vertex 'v'.It checks if 'u' is the parent of 'v'.So the function doesn't backtrack to the vertex it came from in the DFS traversal.If 'u' is not the parent then the function is called recursively on 'u'.

The main function takes input of 'n' and 'm'.Then takes input of cats[i] from 1 to n(if the vertex has a cat or not).

'for (int i = 1; i < n; i++)'→This loops iterates from 1 to (n-1).It reads the tree's edges.Each edge is represented by a pair of integers 'x' and 'y'. 'tree[x].push_back(y); and tree[y].push_back(x);' add vertices 'x' and 'y' to the adjacency list of each other.This constructs the tree structure by creating the edges between the vertices.

]'ans=0' initializes ans to zero.

'dfs(1,0,0)'→It starts from the root(1),sets parent of the root to zero and initializes the total_cats to zero.

Finally the prints the ans as in the number of eligible restaurants for Kefa.

________________X________________

3 Problem C:Oil Deposits

In this problem I had to find out how many oil deposits a grid contains.In the grid each plot is represented as either '*'(absence of oil) or '@'(oil pocket).Oil deposits are considered connected,and two pockets are part of the same deposit if they are adjacent horizontally, vertically,or diagonally.

```cpp
#include <iostream>
#include <vector>
using namespace std;

void dfs(vector<vector<char>>& grid, int row, int col)
{
    int m = grid.size();
    int n = grid[0].size();
    if (row < 0 || row >= m || col < 0 || col >= n ||
        grid[row][col] != '@')
    {
        return;
    }
    grid[row][col] = '*';
    dfs(grid, row - 1, col);
    dfs(grid, row + 1, col);
    dfs(grid, row, col - 1);
    dfs(grid, row, col + 1);
    dfs(grid, row - 1, col - 1);
    dfs(grid, row - 1, col + 1);
    dfs(grid, row + 1, col - 1);
    dfs(grid, row + 1, col + 1);
}
int countOilDeposits(vector<vector<char>>& grid)
{
    int m = grid.size();
    int n = grid[0].size();
    int count = 0;
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (grid[i][j] == '@')
            {
                count++;
                dfs(grid, i, j);
            }
        }
    }
    return count;
}
int main()
{
    int m, n;
    while (cin >> m >> n && m != 0)
    {
        vector<vector<char>> grid(m, vector<char>(n));
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
            {
                cin >> grid[i][j];
            }
        }
        int numDeposits = countOilDeposits(grid);
        cout << numDeposits << endl;
    }
    return 0;
}
```

I solved it using DFS.

'void dfs()' function takes 3 parameters.It takes a grid represented as a vector of vectors of characters('grid'), and two integers 'row' and 'col' referring to the current position in the grid.

'int m=grid.size()' calculates the number of rows in the grid.It determines the maximum valid row index. 'int n=grid[0].size()' calculates the number of columns in the grid.Determines the maximum valid column index.Assuming that all rows have the same number of columns,it's good to access grid[0] to get the number of columns.

'if (row < 0 || row >= m || col < 0 || col >= n || grid[row][col] != '@')'→This line checks several conditions to determine whether the current position is a valid position to explore and whether it contains an oil pocket '@' or no.It does the following checks:

- 'row < 0':the position is outside the grid's upper boundary.
- 'row >= m':position is outside the grid's lower boundary.
- 'col >= n':position is outside the grid's right boundary.

- 'grid[row][col] != '@' ':checks if the current position does not contain an oil pocket. If any of these conditions are met the function returns.

'grid[row][col] = '*';'→This line marks the current position as visited by changing the character at that position from '@' to '*'.This prevents revisiting this oil pocket and ensures it's counted as part of the current deposit.

The next 8 lines represent recursive calls to the dfs() function.They explore the 8 possible adjacent positions from the current position:

1. 'dfs(grid, row - 1, col);'→explores the position above the current one.
2. 'dfs(grid, row + 1, col);'→explores the position below the current one.
3. 'dfs(grid, row, col - 1);'→explores the position left to the current one.
4. 'dfs(grid, row, col + 1);'→explores the position right to the current one.
5. 'dfs(grid, row - 1, col - 1);'→explores the position diagonally up-left.
6. 'dfs(grid, row - 1, col + 1);'→explores the position diagonally up-right.
7. 'dfs(grid, row + 1, col - 1);'→explores the position diagonally down-left.
8. 'dfs(grid, row + 1, col + 1);'→explores the position diagonally down-right.

'int countOilDeposits(vector<vector<char»& grid)'→It takes a 2D grid represented as a reference to a vector of vectors of characters as an argument.

'int count=0' This variable will keep track of the number of distinct oil deposits found in the grid.

The next FOR loop iterates through the rows of the grid.'i' is the current row.

Within the outer loop,this FOR loop iterated through the columns of the grid. 'j' is the current column.

'if (grid[i][j] == '@')'→This line checks if the current position in the grid contains oil pocket ('@').If it does the count is incremented.Then calls the dfs(grid,i,j) which will mark the current deposit and explore all connected oil pockets in a depth-first manner.After the inner loop finishes it moves to the next row and continues the process until all rows and columns in the grid have been checked.The function returns the final count variable representing the total number of distinct oil deposits found in the grid.

In the main function I took m,n inputs(row,column)

The WHILE loop continues as long as it successfully read 'm' and 'n' and 'm' is not equal to zero.Inside the loop 'vector<vector<char» grid(m, vector<char>(n));' this line declares and initializes a 2D vector called 'grid'.It creates a vector with 'm' rows and 'n' columns.

In the FOR loop that iterates through the rows there's an inner FOR loop that iterates through the columns of the grid.Inside the inner loop I took input of the grid populating it with rows and columns.

After the two nested loops end **'int numDeposits = countOilDeposits(grid);'** this line calls the countOilDeposits() function to determine the number of distinct oil deposits in the current grid.In the end it prints the number of distinct oil deposits.

________________X________________

4 Problem D:Travelling Cost

The problem is asking to find the minimum cost for travelling from a given source location(U) to a set of destination locations(V) in a city with roads and fixed travel costs.For each query I had to determine the minimum cost to travel from U to V using the available roads.If there is no path that connects U to V, the output would be 'NO PATH'.

'const int MAX_LOCATIONS=501;' defines the maximum number of locations in the city,and this value is used to define the size of various data structures in the program.

'struct Edgeint to;int weight;;' this line defines a 'struct' named 'Edge'.It represents and edge in the graph and has two integer members 'to',which refers to the target location of the edge, and 'weight', which refers to the cost of travelling that edge.

'vector<vector<Edge» graph(MAX_LOCATIONS);'this line creates a 2D vector named 'graph'.Each element in the 'graph' vector represents a location in the city,and the inner vector contains the edges from that location to other locations.The size of the graph vector is set to 'MAX_LOCATIONS' which allows for a maximum of 501 locations.

'vector<int> dist(MAX_LOCATIONS, INT_MAX);' this line creates a vector named 'dist' which is used to store the minimum distances from the source location to all other locations in the city.The vector is initialized with 'INT_MAX' indicating that the distances are initially unknown or infinite.

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;
```

```cpp
const int MAX_LOCATIONS = 501;
struct Edge
{
    int to;
    int weight;
};
vector<vector<Edge>> graph(MAX_LOCATIONS);
vector<int> dist(MAX_LOCATIONS, INT_MAX);
void dijkstra(int start)
{
    priority_queue<pair<int, int>, vector<pair<int,
       int>>, greater<pair<int, int>>> pq;
    dist[start] = 0;
    pq.push({0, start});

    while (!pq.empty())
    {
        int u = pq.top().second;
        int u_dist = pq.top().first;
        pq.pop();
        if (u_dist != dist[u]) continue;
        for (const Edge& edge : graph[u])
        {
            int v = edge.to;
            int weight = edge.weight;

            if (dist[u] + weight < dist[v])
            {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }
}

int main()
{
    int N;
    cin >> N;
    for (int i = 0; i < N; i++)
    {
        int A, B, W;
        cin >> A >> B >> W;
        graph[A].push_back({B, W});
        graph[B].push_back({A, W});
    }
    int U;
    cin >> U;
    int Q;
    cin >> Q;
    for (int q = 0; q < Q; q++)
    {
        int V;
        cin >> V;
        dist.assign(MAX_LOCATIONS, INT_MAX);
        dijkstra(U);
        if (dist[V] != INT_MAX)
        {
            cout << dist[V] << endl;
        }
        else
        {
            cout << "NO PATH" << endl;
        }
    }
    return 0;
}
```

I used Dijkstra's algorithm to solve this problem.

'void dijkstra(int start)'→This function implements dijkstra's algorithm to find the minimum distances from a starting location('start') to all other locations in the city.

'priority_queue<pair<int, int>, vector<pair<int, int», greater<pair<int, int»> pq;'→This declares a priority queue 'pq' of pairs of integers.This is to keep track of locations to visit next during the algorithm.Each pair in the queue represents a location and its current distance from the source.The 'greater<pair<int ,int»'argument is used to ensure that the priority queue returns the smallest distance first.

'dist[start]=0'→This initializes the distance of the 'start' location to zero because the distance from the source to itself is zero.

'pq.push(0, start)'→This line pushes the starting location('start')and its distance(0) into the priority queue to begin the algorithm.

the WHILE loop that continues as long as there are locations in the priority queue to visit.

'int u = pq.top().second'→This retrieves the distance(represented by the second element of the pair)with the smallest distance from the priority queue and assigns it to 'u'.This will be the one currently being considered for updates.

'int u_dist = pq.top().first'$\rightarrow$This retrieves the distance (represented by the first element of the pair) associated with the location 'u' and assigns it to the variable 'u_dist'. This is the current minimum distance from the source to 'u'.

'pq.pop()' this removes the location 'u' from the priority queue since it's about to be processed.

'if (u_dist != dist[u]) continue'$\rightarrow$This checks if the distance I just retrieved (u_dist) is still the minimum distance for location 'u'. If it's not, it means that a shorter path to 'u' has already been found and processed, so it continues to the next location without further processing 'u'.

The FOR loop iterates through all the edges from location 'u' to other locations in the graph.

Inside the loop,I retrieve the target location of the current edge and assign it to the variable 'v'.

I also retrieve the weight(cost) associated with the edge and assign it to the variable 'weight'.

'if (dist[u] + weight < dist[v])'$\rightarrow$This checks if the distance to location 'v' through the current edge is shorter than the previously known distance to 'v'.If it is,then the distance of location 'v' is updated to be the new shorter distance.

'pq.push(dist[v], v)'$\rightarrow$If it is updated to location 'v', a pair representing the new distance and the location 'v' is pushed into the priority queue.This ensures that location 'v' and its updated distance will be later considered in the algorithm.

In the main function N is the variable to store the number of roads constructed in the city.

In the FOR loop that iterates N times.Inside the loop there are three integers.A,B and W.These are for storing the source location(A),destination location(B) and the cost of travelling that road(W).

'graph[A].push_back(B, W)'$\rightarrow$This line updates the graph data structure.It adds and edge from location 'A' to location 'B' with a weight of 'W'.This is done by pushing a new 'Edge' object into the vector associated with location

’A’.I also added a reverse edge from ’B’ to ’A’ because the roads are bidirectional.

’int U’ variable ’U’ stores the source location from which Rohit wants to travel to other locations.

’int Q’ variable ’Q’ stores the number of queries Rohit wants to perform.Each query will involve finding the minimu cost to travel from location ’U’ to a specific destination.

In the FOR loop that iterates ’Q’ times ’V’ is the variable that stores the destination location of the current query.

’dist.assign(MAX_LOCATIONS, INT_MAX)’→This line re-initializes the ’dist’ vector with all elements set to ’INT_MAX’.This is done before each query to ensure that the algorithm starts with unknown distances.

Then dijkstra(U) function passes the source location ’U’.This calculates the minimum distances from ’U’ to all other locations in the city.

’if (dist[V] != INT_MAX)’→This line checks if there is a valid path from ’U’ to ’V’ by looking at the value stored in the ’dist’ vector for location ’V’.If it’s not equal to ’INT_MAX’,it means there is a path and you can print the minimum distance.If a valid path exists,print the minimum distance to the destination ’V’.IS there’s no valid path print "NO PATH."

________________X________________