HEURISTICS GUILD

Application

Karthik Kashyap

EE23B030™

All of the codes referenced in this application can be found on my GitHub. No help of ChatGPT was taken to write the codes of this Application. Slight use was done for the Contest Write-up, which is mentioned in that respective document.

§0 About Myself :)

1. *(The present?)* How's life? Also why the Heuristics Guild?

   Life is good! Having nice family time right now.

   About the Heuristics Guild, I was initially a little hesitant to join the Heuristics guild because I didnt really know much about it then. It was after being part of the contingent that I realised how fun it is. Looking for research papers, reading them, combining logics from several papers and then implementing them were all very interesting. Though the implementation that I worked on, being mostly on Genetic Algorithms, did not yield the best results and was not part of the final implementation, I leant a lot from it and it was fun to see something work as well as it did.

   I wish to be able to have that experience again, and hence I am applying for Heuristics Guild.

2. *(The future)* What are your various commitments for the upcoming year?

   I am currently the Head of the Mathematics Club. Apart from this, I am only applying to be a part of the Heuristics Guild, nothing more.

3. *(The past)* Briefly describe your knowledge/experience with programming in general and maybe even about your experience with Heuristics or other domains of Software Development.

   a) I have been a coordinator at Programming Club in the last tenure, where I conducted events for both Dev and CP verticals, and also headed DCs in both verticals.

   b) I have worked on making several small games for fun, nothing too serious here.

   c) I have been part of the last year's Networks and Heuristics Guild.

   d) As part of this, I was even part of the last year's IITM Contingent team working on the PS by FedEx about bin packing of priority and economy packages. Here, I mostly worked with different variations of Genetic Algorithms to get the best fit: mainly the Biased Random Key Genetic Algorithms.

4. *(The past but fun)* Tell us a fun fact about yourself.

   Here goes ~

   a) I have been playing Minecraft since 2019, specifically $v1.13$. It has been the game of my life, so I appreciate all references of Minecraft in the app. I thrive on Bedwars final kills.

   

   b) I have never been in a flight.

   c) I know almost all songs in my regional language, i.e., Kannada. Had challenged some of my friends to show me a song I havent known and they have never succeeded so far.

5. *(The past but... zzz)* Tell us a boring fact about yourself.

   a) ~~Ragging~~ Pulling leg of all juniors I see.

   b) Can't work productively without my second monitor and my music.

§1 Programming

1. *(Favourite Data Structure)* Name your favourite data structure and a good reason why.

   I genuinely took some time to think about what data structure might be my favourite. Then I realised, that the one I use the most is my favourite: vectors. Being like a dynamic array, I have stopped using arrays since I discovered it's existence. Not just as an array, it can also used to denote directed and undirected graphs. Might be a bit of a lame answer, and not what you expected, but I seem to have no other answer: every other data structure is as is and not as fun as this.

2. *(Favourite Algorithm)* Name your favourite algorithm and a good reason why.

   Again, not a very special but quite a useful one, Binary Search is my favourite algorithm. The simplicity, yet the possibility of using it is just so nice. I remember in my very early days of learning programming I was solving a question on Codeforces and implemented a linear search and got TLE. Was confused why this happened and left it as it is. A month or two later when I knew Bin Search, I happened to come back to this question and just replace the search method to Bin Search. The satisfaction of getting accepted has not left me yet, even after more than a year of this incident. So henceforth I call Binary Search as my favourite algorithm.

3. *(Graph Algorithms)* Implementation of code is very important for this guild.

   Note: Only the required amount of code is shown here, for the entire code, refer to this GitHub Repo. The codes have been written with help from concepts of take U forward.
   Also, some parts of the code have been shortened to fit within the page.

   Assumption made: The input graph is a positive weighted directed graph.

   a) Cycle Detection:

Cycle detection in a Directed Cyclic Graph can be done using either DFS or BFS.

DFS:

We implement dfs at each node and check if in the subseqeuent dfs calls we come across a vertex which is in the same path followed through these dfs calls. If so, cycle exists, otherwise it does not.

```
bool dfs(int i, vector<vector<pair<int, int>>> &graph, vector<int>
    &visited, vector<int> &path_visited)
{
    visited[i] = 1;
    path_visited[i] = 1;

    for (auto j : graph[i])
    {
        if (!visited[j.first])
        {
            if (dfs(j.first, graph, visited, path_visited))
                return true;
        }
        else if (path_visited[j.first])
            return true;
    }

    path_visited[i] = 0;
    return false;
}

bool check_cycle(int n, vector<vector<pair<int, int>>> &graph)
{
    vector<int> visited(n), path_visited(n);

    for (int i = 0; i < n; i++)
    {
        if (!visited[i])
        {
            if (dfs(i, graph, visited, path_visited))
                return true;
        }
    }

    return false;
}
```

The Time Complexity of the algorithm is $\mathcal{O}(V + E)$.

BFS:

We implement Kahn's algorithm for Topological Sorting. Topological sorting can only be done in acyclic graphs, which implies that the length of topo sort is equal to number of nodes only when the graph is acyclic. Hence, if we obtain length to be less than this, we can conclude that the graph is acyclic.

```
bool check_cycle(int n, vector<vector<pair<int, int>>> &graph)
{
    vector<int> in_degree(n);

    for (int i = 0; i < n; i++)
    {
        for (auto j : graph[i])
        {
            in_degree[j.first]++;
        }
    }

    queue<int> q;
    for (int i = 0; i < n; i++)
    {
        if (!in_degree[i])
            q.push(i);
    }

    int topo_sort_count = 0, node;
    while (!q.empty())
    {
        node = q.front();
        q.pop();
        topo_sort_count++;

        for (auto i : graph[node])
        {
            in_degree[i.first]--;
            if (!in_degree[i.first])
                q.push(i.first);
        }
    }

    if (topo_sort_count < n)
        return true;
    else
        return false;
}
```

The Time Complexity of the algorithm is $\mathcal{O}(V + E)$.

b) Shortest path from a given node to another:

Note: I misunderstood the question to be shortest *distance* instead of shortest *path*. I had already written the codes by the time I noticed, hence I have just added them here

Shortest *Distance* using Topological Sort:

If the Graph is a Directed Acyclic Graph, then we can use Topological sorting to obtain a linear ordering of the nodes. After this, we go from one end of the ordering to the other, and obtain minimum distances wrt the input node.

```
4  void dfs(int node, vector<vector<pair<int, int>>> &graph, stack<int>
       &topo, vector<int> &visited)
5  {
6      visited[node] = 1;
7
8      for (auto j : graph[node])
9      {
10         if (!visited[j.first])
11             dfs(j.first, graph, topo, visited);
12     }
13
14     topo.push(node);
15 }
16
17 int shortest_path(int n, vector<vector<pair<int, int>>> &graph, int
       node1, int node2)
18 {
19     stack<int> topo;
20     vector<int> visited(n);
21     for (int i = 0; i < n; i++)
22     {
23         if (!visited[i])
24             dfs(i, graph, topo, visited);
25     }
26
27     vector<int> distance(n, INT32_MAX);
28     distance[node1] = 0;
29     while (!topo.empty())
30     {
31         int node = topo.top();
32         topo.pop();
33
34         for (auto i : graph[node])
35         {
36             distance[i.first] = min(distance[i.first], distance[node] +
                   i.second);
37         }
38     }
39
40     return distance[node2];
41 }
```

The Time Complexity of the algorithm is $\mathcal{O}(V + E)$.

Shortest *Distance* using Dijkstra's algorithm:

On the other hand, if the graph is Directed Cyclic Graph, then we cannot use Topological sorting. Here, we have to resort to Dijkstra's algorithm.

```
// Function to implement Dijkstra's algorithm
vector<int> dijkstra(int n, vector<vector<pair<int, int>>> &graph, int
    node1)
{
    priority_queue<pair<int, int>, vector<pair<int, int>>,
        greater<pair<int, int>>> pq;
    vector<int> distance(n, INT32_MAX);

    distance[node1] = 0;
    pq.push(make_pair(0, node1));

    while (!pq.empty())
    {
        int dist = pq.top().first; // Choosing minimum distance pair
        int node = pq.top().second;
        pq.pop();

        for (auto i : graph[node])
        {
            if (dist + i.second < distance[i.first])
            {
                distance[i.first] = dist + i.second;
                pq.push(make_pair(distance[i.first], i.first));
            }
        }
    }
    return distance;
}
```

The Time Complexity of the algorithm is $\mathcal{O}(E \log V)$.

And this is the point where I realised that I have to find the shortest path and not the shortest distance. Oops. Correct code on the next page.

Shortest *Path* using Dijkstra's algorithm:

Very similar to the shortest distance using Dijkstra's algorithm, only added a parent vector to track where the minimum distance occurs from, and finally backtrack from *node*2 to *node*1 using the parent vector to obtain the path.

```
// Function to implement Dijkstra's algorithm to find path
vector<int> dijkstra(int n, vector<vector<pair<int, int>>> &graph, int
    node1, int node2)
{
    priority_queue<pair<int, int>, vector<pair<int, int>>,
        greater<pair<int, int>>> pq;
    vector<int> distance(n, INT32_MAX);
    vector<int> parent(n);
    for (int i = 0; i < n; i++)
        parent[i] = i;

    distance[node1] = 0;
    pq.push(make_pair(0, node1));

    while (!pq.empty())
    {
        int dist = pq.top().first; // Choosing minimum distance pair
        int node = pq.top().second;
        pq.pop();

        for (auto i : graph[node])
        {
            if (dist + i.second < distance[i.first])
            {
                distance[i.first] = dist + i.second;
                pq.push(make_pair(distance[i.first], i.first));
                parent[i.first] = node;
            }
        }
    }

    if (distance[node2] == INT32_MAX)
        return {-1};

    vector<int> path;
    int node = node2;
    while (parent[node] != node)
    {
        path.push_back(node);
        node = parent[node];
    }
    path.push_back(node1);
    reverse(path.begin(), path.end());
    return path;
}
```

The Time Complexity of the algorithm is $\mathcal{O}(V + E \log V)$.

c) Finding the value of the Minimum spanning tree using Kruskal's Algorithm:

Minimum Spanning Tree:

To find the minimum spanning tree, we need to only consider the edges which are necessary to have all nodes in the same structure, and with the minimum weight. To do this, let us sort the edges in ascending order of weights. Now we start from the beginning and only consider the edges in which the two nodes are not part of the same structure.

To identify if the two nodes are in the same structure, we can brute force this by applying DFS, but a better and optimal solution which runs in near constant time is by using Disjoint Set. Here's the implementation of Disjoint Set that I have used for these purposes, where I have considered union based on the rank of the parent nodes:

```
class DisjointSet {
private:
    vector<int> rank, parent;
public:
    DisjointSet(int n)
    {
        rank.resize(n, 0);
        parent.resize(n);
        for (int i = 0; i < n; i++)
            parent[i] = i;
    }

    int find_Uparent(int node)
    {
        if (parent[node] == node)
            return node;

        return parent[node] = find_Uparent(parent[node]);
    }

    void union_rank(int u, int v)
    {
        int u_Uparent = find_Uparent(u);
        int v_Uparent = find_Uparent(v);

        if (u_Uparent == v_Uparent)
            return;

        if (rank[u_Uparent] < rank[v_Uparent])
            parent[u_Uparent] = v_Uparent;
        else if (rank[u_Uparent] > rank[v_Uparent])
            parent[v_Uparent] = u_Uparent;
        else
        {
            parent[v_Uparent] = u_Uparent;
            rank[u_Uparent]++;
        }
    }
};
```

The Time Complexity of the Disjoint Set functions is $\mathcal{O}(\alpha(n))$, where where $\alpha(n)$ is the inverse Ackermann function. This is nearly constant time, as it's very slowly increasing.

Now that we have the Disjoint set, the implementation should be straight forward:

```cpp
int mst_kruskal(int n, vector<vector<pair<int, int>>> &graph)
{
    // Vector to store the edges to sort based on weight
    vector<pair<int, pair<int, int>>> edges; // {weight, {u, v}}
    for (int i = 0; i < n; i++)
    {
        for (auto j : graph[i])
        {
            edges.push_back(make_pair(j.second, make_pair(i, j.first)));
        }
    }
    sort(edges.begin(), edges.end());

    // Defining disjoing set for the nodes.
    int weight_mst = 0;
    DisjointSet ds(n);
    for (auto i : edges)
    {
        int w = i.first, u = i.second.first, v = i.second.second;

        if (ds.find_Uparent(u) != ds.find_Uparent(v)) // If Part of
            different structure
        {
            ds.union_rank(u, v);
            weight_mst += w;
        }
    }

    return weight_mst;
}
```

The Time Complexity of the function is $\mathcal{O}(E \times \alpha(n))$, which can be practically taken to be $\mathcal{O}(E)$.

d) Forming an SCC DAG and also returning its topological sorting:

I have chosen Kosaraju's algorithm to find SCCs:

```
void dfs_ft(int node, vector<vector<pair<int, int>>> &graph, stack<int>
    &finishing_time, vector<int> &visited){
    visited[node] = 1;
    for (auto j : graph[node]){
        if (!visited[j.first])
            dfs_ft(j.first, graph, finishing_time, visited);
    }
    finishing_time.push(node);
} // DFS to find the finishing time of the nodes

void dfs_scc(int node, vector<vector<pair<int, int>>> &graph,
    vector<int> &visited, vector<int> &single_scc){
    visited[node] = 1;
    for (auto j : graph[node]){
        if (!visited[j.first])
            dfs_scc(j.first, graph, visited, single_scc);
    }
    single_scc.push_back(node);
} // DFS on the reversed edges graph

// Function to Find the SCCs of the graph
vector<vector<int>> scc(int n, vector<vector<pair<int, int>>> &graph){
    vector<int> visited(n);
    stack<int> finishing_time;
    for (int i = 0; i < n; i++){ // DFS to find the finishing time

        if (!visited[i])
            dfs_ft(i, graph, finishing_time, visited);
    }

    vector<vector<pair<int, int>>> reversed_graph(n);
    for (int i = 0; i < n; i++){ // Reversing the edges of the graph
        for (auto j : graph[i])
            reversed_graph[j.first].push_back(make_pair(i, j.second));
    }

    visited = vector<int>(n, 0);
    vector<vector<int>> vect_scc;
    // Finding the SCCs with DFS on the reversed edge graph
    while (!finishing_time.empty()){
        int node = finishing_time.top();
        finishing_time.pop();
        if (!visited[node]){
            vector<int> single_scc;
            dfs_scc(node, reversed_graph, visited, single_scc);
            vect_scc.push_back(single_scc);
        }
    }
    return vect_scc;
}
```

The core logic behind this algorithm revolves around reversing the edges. All the edges inside an SCC would point to all other vertices inside it, be it direct or indirect. This is not the case for edge connecting an SCC to another. Hence when we reverse the edges, the nodes inside an SCC previously still remain in the same SCC, whereas the edge connecting SCC is reversed, and traverses differently. DFS is hence used here twice: once before the reversal and one after.

To conduct Topo Sorting on these SCCs, we take these SCCs to be equivalent to super nodes, and apply the same topological sorting technique normally applied to graphs. I have used the same topo sort function that I have used in 3a Cycle Detection BFS.

Code for the topo sort function is as below:

```
// Function to convert graph into SCCs for topo sorting
vector<int> topo_scc(int n, vector<vector<pair<int, int>>> &graph,
    vector<vector<int>> &vector_scc)
{
    int n_scc = vector_scc.size();

    vector<int> node_scc(n); // Keys for the SCCs
    for (int i = 0; i < n_scc; i++)
    {
        for (auto j : vector_scc[i])
        {
            node_scc[j] = i;
        }
    }

    vector<unordered_set<int>> adj_scc_set(n_scc); // Set to hold edges
        between SCCs
    for (int i = 0; i < n; i++)
    {
        for (auto j : graph[i])
        {
            if (node_scc[j.first] != node_scc[i])
                adj_scc_set[node_scc[i]].insert(node_scc[j.first]);
        }
    }

    vector<vector<int>> adj_scc(n_scc);
    for (int i = 0; i < n_scc; i++)
    {
        for (auto j : adj_scc_set[i])
            adj_scc[i].push_back(j);
    }

    return topo_sort(n_scc, adj_scc);
}
```

The Time Complexity of the algorithm is $\mathcal{O}(V + E)$.

§2 Heuristics

4. *(Welcome to Heuristics)* Solve the problem sheet and explain how each question is different and what approaches you could apply for each question.

All of the three questions require finding a hamiltonian cycle in a given undirected graph. To do this, we can take the naive approach, which is to start at a starting vertex, which in this case is given to be 0 and then try to backtrack through every available path looking for unvisited nodes.

Here is the code implementation for this approach:

```cpp
#include <bits/stdc++.h>
using namespace std;

bool node_available(vector<vector<int>> &roads, vector<int> &path, int pos,
    int next, int n){
    if (!roads[path[pos - 1]][next])
        return false;

    for (int i = 0; i < pos; i++){
        if (path[i] == next){
            return false;
        }
    }
    return true;
}

bool check_cycle(vector<vector<int>> &roads, vector<int> &path, int pos,
    int n){
    if (pos == n){
        return roads[path[n - 1]][0];
    }

    for (int i = 1; i < n; i++){
        if (node_available(roads, path, pos, i, n)){
            path[pos] = i;
            if (check_cycle(roads, path, pos + 1, n)){
                return true;
            }
            path[pos] = -1;
        }
    }
    return false;
}

vector<int> cycle(vector<vector<int>> &roads, int n){
    vector<int> path(n, -1);
    path[0] = 0;
    if (check_cycle(roads, path, 1, n)){
        return path;
    }
    return vector<int>(n, -1);
}
```

```cpp
int main(){
    int n, m;
    cin >> n >> m;

    vector<vector<int>> roads(n, vector<int>(n, 0));

    int u, v;
    for (int i = 0; i < m; i++){
        cin >> u >> v;

        roads[u][v] = 1;
        roads[v][u] = 1;
    }

    vector<int> path(n, -1);
    path = cycle(roads, n);

    if (path[0] == -1){
        cout << -1 << endl;
        return 0;
    }

    cout << 1 << endl;
    for (auto i : path){
        cout << i << ' ';
    }
    cout << 0 << endl;

    return 0;
}
```

Code can also be found on my GitHub.

As this approach backtracks through all possible paths of nodes, the worst case time complexity of this approach is $\mathcal{O}(n!)$, which is pretty bad and changes drastically for each increment of n.

a) For the A problem, n lies between 2 and 10. The worst case number of iterations is thus 10!, which is approximately 3×10^6, well in the range of number of iterations to run in less than 2 seconds.

b) For the B problem, n lies between 2 and 20. The worst case number of iterations is now 20!, which is approximately 2×10^{18}?? I am not entirely sure why this algorithm was able to pass the test cases. After giving some thought, I think it's because it's very unlikely to have the worst case scenario, which is the case where the graph is mostly complete connected, and it so happens that the cycle only occurs in the last iteration of the code (which I don't see how it can) considering the code discards all paths where there isnt any connectivity. If that were a case, or a close one then I think my code would have given TLE. This is considering the fact that for $\mathcal{O}(n!)$, $n \leq 12$ to run in less than $1s$. So $n = 20$ happens to be close enough to the safe limit that for most general cases, it runs in a comfortable amount of time.

c) For the C problem, n is restricted to be 100. This gives number of iterations as nearly 9×10^{157} lmao. This, in no way passes the test cases in $2s$, and this is expected of as the outcome for an NP-complete problem. This is exactly why we need to approach the problem with Heuristic solutions.