6: Graph I

CPCFI

UNAM's School of Engineering

2021

Based on: Halim S., Halim F.*Competitive Programming 3*. Handbook for ACM ICPC and IOI Contestants. 2013

Table of Contents

Graph Traversal - Motivation

- At least one graph problem in ICPC
- We should be familiar with the following concepts:

Vertices/Nodes	Edges	Set V; size $\|V\|$	Set E; size $\|E\|$	Graph $G(V, E)$
Un/Weighted	Un/Directed	Sparse	Dense	In/Out Degree
Path	Cycle	Isolated	Reachable	Connected
Self-Loop	Multiple Edges	Multigraph	Simple Graph	Sub-Graph
DAG	Tree/Forest	Eulerian	Bipartite	Complete

CPCFI

Graph Traversal

Topics to cover:

- Depth First Search (DFS)
- Breadth First Search (BFS)
- Finding Connected Components (Undirected graph)
- Flodd Fill - Labeling/Coloring the Connected Components
- Topological Sort (DAG)
- Bipartite Graph Check
- Graph Edges Property Check via DFS Spanning Tree
- Finding Articulation Points and Bridges
- Finding Strongly Connected Components

CPCFI

DFS - Depth First Search

- Depth First Search is an algorithm from traversing a graph
- DFS visits all the reachable nodes starting from a vertex v

Steps:

1. DFS starts from a source vertex v and starts going deeper into the graph until reaching a leaf node
2. Once DFS is on a leaf node, DFS will backtrack and explore other unvisited neighbors if any

CPCFI

DFS

```cpp
#define MAX_N 10000
vector<int> adjList[MAX_N];
vector<int> visited(MAX_N);

void dfs(int u) {
  visited[u] = 1;
  for(auto v : adjList[u]) {
    if (visited[v] == 0) {
      dfs(v);
    }
  }
}
```

CPCFI

DFS

- DFS runs in $O(V + E)$ time

BFS - Breadth First Search

- BFS will traverse the graph by expanding, first, all of the neighbors starting from vertex v
- BFS uses a queue to keep track of the vertices to visit next
- BFS also runs in $O(V + E)$ time

CPCFI

BFS

```
queue<int> q;

void bfs(int u) {
  q.push(u);
  while (!q.empty()) {
    int v = q.top(); q.pop();
    for (auto x : adjList[v]) {
      q.push(x);
    }
  }
}
```

Finding Connected Components - Undirected Graph

- Both DFS and BFS can also be applied to other problems
- A single call of DFS or BFS will visit vertices that are connected to a starting vertex u in an undirected graph

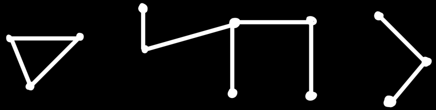

Figure: Undirected graph with 3 connected components

Finding Connected Components - Undirected Graph

- To find the number of connected components in an undirected graph iterate over all vertices
- For each vertex, run **dfs** if that vertex is unvisited
- Increase the count of connected components for each iteration over the vertices

Flood Fill - Labeling/Coloring the Connected Components

- Graph traversal can also be used to label a graph (color it)
- Or, counting the size of a connected component

CPCFI

Flood Fill - Labeling/Coloring the Connected Components

- Graph traversal can also be used to label a graph (color it)
- Or, counting the size of a connected component
- Counting the size of a connected component is referred as **flood fill**

Example: UVa 469 - Wetlands of Florida

Topological Sort - Directed Acyclic Graph

- The topological sort of a DAG is a linear ordering of the vertices so that vertex u comes before vertex v if edge $u \to v$ exists

Topological Sort - Directed Acyclic Graph

- The topological sort of a DAG is a linear ordering of the vertices so that vertex u comes before vertex v if edge $u \rightarrow v$ exists
- Every DAG has at least one topological sort

Topological Sort - Directed Acyclic Graph

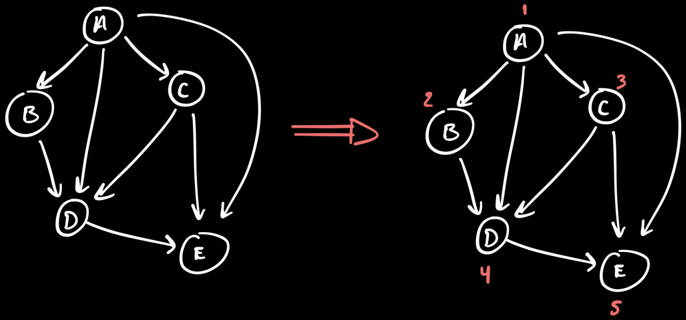

Figure: Example of a Topological Sort for a DAG

Topological Sort - Directed Acyclic Graph

- One common application of Topological Sort is to find a sort upcoming tasks in order of precedence
- For example, ordering the steps needed to cook pizza. Each step needs a previous step to be completed (mass, tomatoes, etc)

Topological Sort - Directed Acyclic Graph

- Topological Sort can be achieved by appending the current visited node in DFS -after visiting all the nodes in its subtree- to a list

Topological Sort - Directed Acyclic Graph

- Topological Sort can be achieved by appending the current visited node in DFS -after visiting all the nodes in its subtree- to a list

```
vector<int> topological_sort;

void dfs(int u) {
  visited[u] = 1;
  for (auto v : adjList[u]) {
    if (visited[v] == 0) {
      dfs(v);
    }
  }
  // Only change for TS
  topological_sort.push_back(u);
}
```

Bipartite Graph Check

- A graph is said to be bipartite if its 2-colorable
- Thus, we'll check if a graph is bipartite by attempting to color it only 2 colors
- We'll achieve this by modifying BFS

CPCFI

Bipartite Graph Check

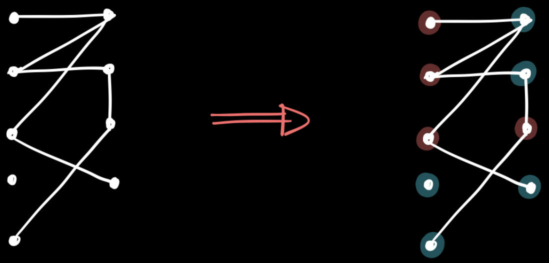

Figure: Bipartite graph coloring (2-colorable)

Bipartite Graph Check

```
int s; //initial vertex
queue<int> q; q.push(s);
vector<int> color(V, INF); color[s] = 0;
bool isBipartite = true;

while (!q.empty() && isBipartite) {
  int u = q.front(); q.pop();
  for (auto& v : adjList[u]) {
    if (color[v] == INF){
      color[v] = 1 - color[u]; //two colors {0,1}
      q.push(v);
    } else if (color[v] == color[u]) {
      // Coloring conflict
      isBipartite = false;
      break;
    }
  }
}
```

Bipartite Graph Check

Example: UVa 10004 - Bicoloring

Graph Edges Property Check via DFS Spanning Tree

Spanning Tree: given a connected graph G, its spanning tree is a tree that spans (covers) all vertices of G but only using a subset of the edges of G

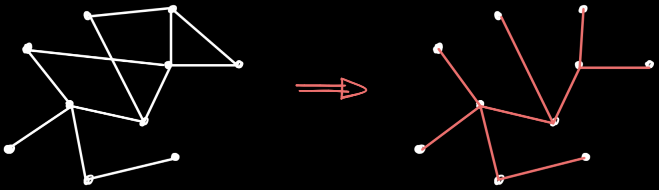

Figure: One possible spanning tree

Graph Edges Property Check via DFS Spanning Tree

- Running DFS on a connected graph generates a DFS spanning tree

[1] EXPLORED: visited but not yet completed
[2] VISITED: visited and completed

Graph Edges Property Check via DFS Spanning Tree

- Running DFS on a connected graph generates a DFS spanning tree
- We can classify edges into three types using one more vertex state (`EXPLORED` [1]) in addition to `VISITED` [2]:

[1] EXPLORED: visited but not yet completed
[2] VISITED: visited and completed

Graph Edges Property Check via DFS Spanning Tree

- Running DFS on a connected graph generates a DFS spanning tree
- We can classify edges into three types using one more vertex state (`EXPLORED` [1]) in addition to `VISITED` [2]:

  1. **Tree edge**: `EXPLORED` $\rightarrow$ `UNVISITED`. -Edge traversed by DFS-

[1] EXPLORED: visited but not yet completed
[2] VISITED: visited and completed

CPCFI

Graph Edges Property Check via DFS Spanning Tree

- Running DFS on a connected graph generates a DFS spanning tree
- We can classify edges into three types using one more vertex state (`EXPLORED` [1]) in addition to `VISITED` [2]:

  1. **Tree edge**: `EXPLORED` $\rightarrow$ `UNVISITED`. -Edge traversed by DFS-
  2. **Back edge**: `EXPLORED` $\rightarrow$ `EXPLORED`. -Edge that is part of a cycle-

[1] EXPLORED: visited but not yet completed
[2] VISITED: visited and completed

Graph Edges Property Check via DFS Spanning Tree

- Running DFS on a connected graph generates a DFS spanning tree
- We can classify edges into three types using one more vertex state (`EXPLORED` [1]) in addition to `VISITED` [2]:

  1. **Tree edge**: `EXPLORED` → `UNVISITED`. -Edge traversed by DFS-
  2. **Back edge**: `EXPLORED` → `EXPLORED`. -Edge that is part of a cycle-
  3. **Forward/Cross edges**: `EXPLORED` → `VISITED`

[1] EXPLORED: visited but not yet completed
[2] VISITED: visited and completed

CPCFI

Graph Edges Property Check via DFS Spanning Tree

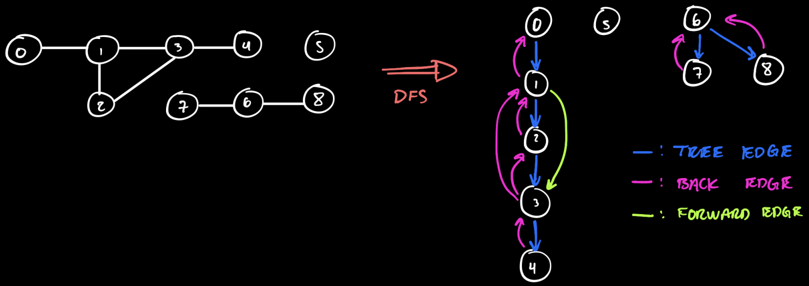

Figure: Example of property check [3]

[3] An undirected graph with multiple connected components generates a spanning forest

Finding Articulation Points and Bridges

Main problem: Given a road map (undirected graph) with sabotage costs associated to all intersections (vertices) and roads (edges), sabotage either a single intersection or a single road such that the road network breaks down (disconnected) and do so in the least cost way. - [Halim], p. 130

CPCFI

Finding Articulation Points and Bridges

- **Articulation Point**: vertex in G that its removal will cause G to be disconnected [4]
- **Bridge**: edge in G that its removal will cause G to become disconnected

[4] A graph without articulation point is called biconnected graph

CPCFI

Finding Articulation Points and Bridges

- **Articulation Point**: vertex in G that its removal will cause G to be disconnected [4]
- **Bridge**: edge in G that its removal will cause G to become disconnected

Note: these two problems are usually defined for undirected graphs. For directed graphs, they require different algorithms

[4] A graph without articulation point is called biconnected graph

Table of Contents

Minimum Spanning Tree

Table of Contents

CPCFI

Single-Source Shortest Paths

References

- Halim S., Halim F., *Competitive Programming 3*, Handbook for ACM ICPC and IOI Contestants. 2013
- Stroustrup B. *The C++ Programming Language*. Fourth ed.
- Skiena S. *The Algorithm Design Manual*. Springer. 2020

CPCFI