

# Programming Challenges (GB21802)

## Week 5 - Graph Part I: Basics

Claus Aranha

caranha@cs.tsukuba.ac.jp

University of Tsukuba, Department of Computer Sciences

2020/5/26

(last updated: May 25, 2020)

Version 2020.1

# Graphs in Computer Science

Graph Data structures explain the relationships between data, and are used in several real world problems:

- Geography and Maps;
  - Pathing between locations;
  - Cycles and Tours;
- Human Networks;
  - Social Networks;
  - Citation Clusters;
- State Machines;
  - Program Pipelines;
  - Library Requirements;
- Natural Language;
  - Graph Grammars;

# Graph Algorithms: Week 4 and 5

## Graphs Part I (This Week)

- Graphs Data Structure;
- Depth First Search and Breadth First Search;
- Graph Search Problems (DFS and BFS);
- Minimum Spanning Tree: Kruskal and Prim Algorithms;

## Graphs Part II (Next Week)

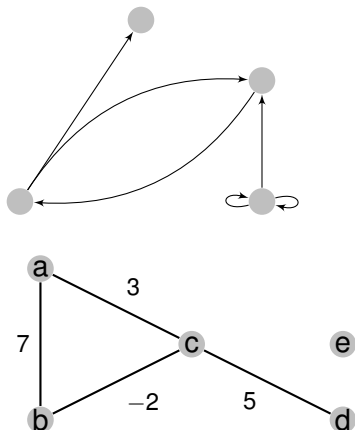
- Single Source Shortest Path (Dijkstra);
- All Pairs Shortest Path (Floyd-Warshall);
- Network Flow;
- Bipartite Graph Matching;

# What is a graph?

A graph  $G = \{V, E\}$  is composed of a set of **vertices**  $V$ , which are connected to a set of **edges**  $E$ . Each edge connects exactly two vertices.

Edge and vertices characteristics:

- An edge can be **directed** or **undirected**;
- An edge or a vertex can have **weights** or **labels**;
- **Self-edges** and **multi-edges**;
- A graph can be **connected** or **disconnected**;



# Common questions about graphs

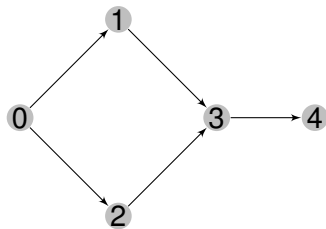
Some tasks are common across graph problems:

- Test if a path exist between vertice  $V_i$  and  $V_j$  (test if they are **connected**)
- Test the shortest path between vertice  $V_i$  and  $V_j$ 
  - With or without weights
  - Test if there is more than one path
- Add or remove vertices or edges from a graph;
- Test characteristics of a graph;
- etc.;

# UVA 11902 – Dominator

A vertex  $V_i$  **dominates**  $V_j$  if all paths  $V_0 \rightarrow V_j$  must include  $V_i$ .

- input:** A directed graph  $\{V, E\}$ ;
- output:** A table with which vertices dominate each other;



## Input

```

1
5
0 1 1 0 0
0 0 0 1 0
0 0 0 1 0
0 0 0 0 1
0 0 0 0 0
  
```

## Output

Case 1:

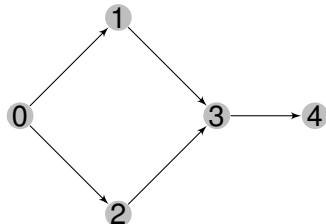
```

+-----+
|Y|Y|Y|Y|Y|
+-----+
|N|Y|N|N|N|
+-----+
|N|N|Y|N|N|
+-----+
|N|N|N|Y|Y|
+-----+
|N|N|N|N|Y|
+-----+
  
```

# UVA 11902 – Dominator

## Quiz: How do we solve this problem?

- How do we store the graph from the input?
- How do we determine "dominator" status?



### Input

```

1
5
0 1 1 0 0
0 0 0 1 0
0 0 0 1 0
0 0 0 0 1
0 0 0 0 0
  
```

### Output

Case 1:

```

+-----+
|Y|Y|Y|Y|Y|
+-----+
|N|Y|N|N|N|
+-----+
|N|N|Y|N|N|
+-----+
|N|N|N|Y|Y|
+-----+
|N|N|N|N|Y|
+-----+
  
```

# Storing the Dominator Graph

Adjacency Matrix: stores the connection between vertices

```
int adj[100][100];  
  
for (int i = 0; i < n; i++)  
    for (int j = 0; j < n; j++)  
        cin >> adj[i][j]; // 0 if no edge, 1 if edge
```

- Pros:

- Easy to program;
- Quickly access any edge;

- Cons:

- Cannot store multigraph;
- Wastes memory with sparse graphs;
- Time  $O(V)$  to calculate number of neighbors;



# Storing the Dominator Graph

## Adjacency List: stores edge list for each Vertex

```
typedef pair<int,int> edge; // pair: <neighbor, weight>
typedef vector<edge> neighb; // all neighbors of V_i
vector<neighb> AdjList;      // all V_i
```

```
int e;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        cin >> e;
        if (e == 1) { AdjList[i].push_back(pair(j,1)); }
```

- **Pro:**

- Memory efficient if the graph is sparse;
- Can store multigraph;

- **Cons:**

- $O(\log(V))$  to test if two vertices are adjacent;
- Code can be more complex;

# Storing the Dominator Graph

## Edge List

```
pair <int,int> edge; // Edge between i and j
vector<pair <int,edge>> Elist; // All edges;

int e;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        cin >> e;
        if (e == 1)
            Elist.push_back(pair(1, pair(i,j)));
```

- Not very common;
- Has to sort edges to find the neighbors of a Vertice;
- Used for specialized algorithms, such as Minimum Spanning Tree;

# Graph Search: BFS and DFS

- Basic algorithms to visit every vertex in a graph.
- Many graph algorithms require BFS or DFS;
- So learn to do these with your eyes closed;

## Depth First Search – DFS

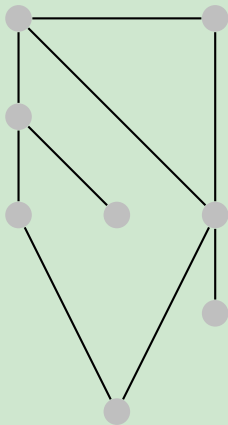
- Visit the first edge in the next vertex;
- No guarantee of order, tends to go away from start;
- Easy to implement with recursion;

## Breadth First Search – BFS

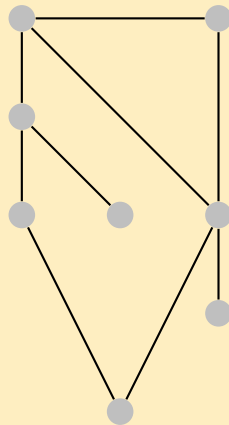
- First visit the vertices closer to starting point;
- Place new edges on a FIFO, and search with a loop;

## BFS/DFS: Visualization

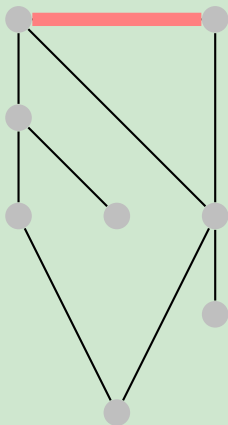
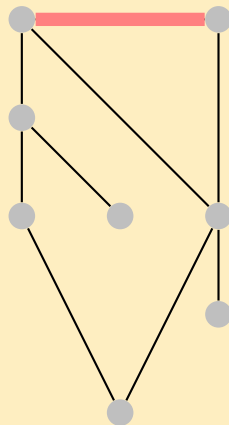
DFS



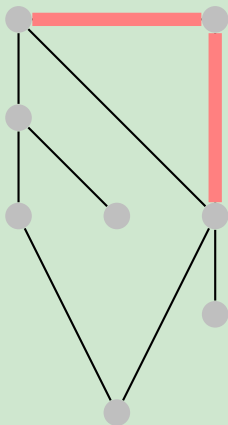
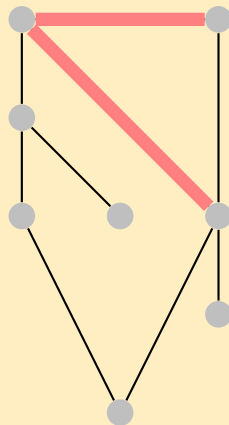
**BFS**



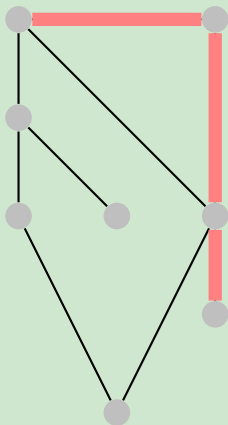
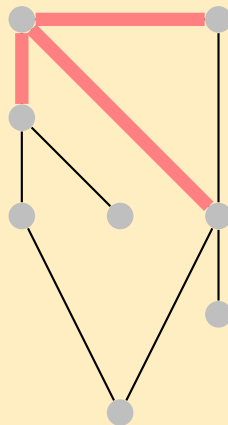
# BFS/DFS: Visualization

**DFS****BFS**

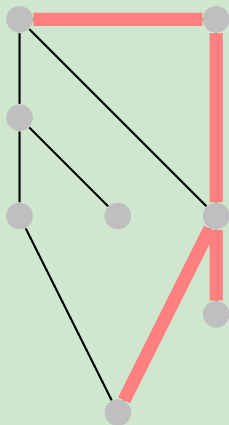
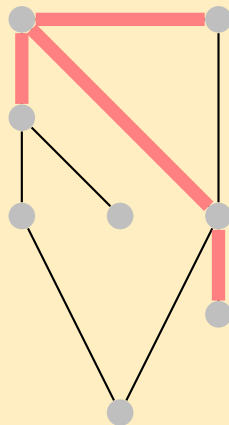
# BFS/DFS: Visualization

**DFS****BFS**

# BFS/DFS: Visualization

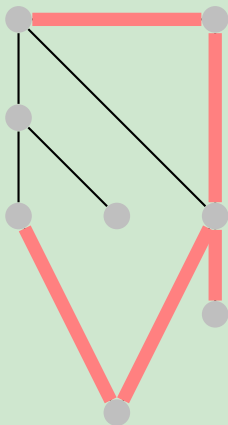
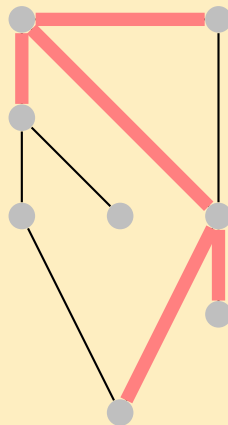
**DFS****BFS**

# BFS/DFS: Visualization

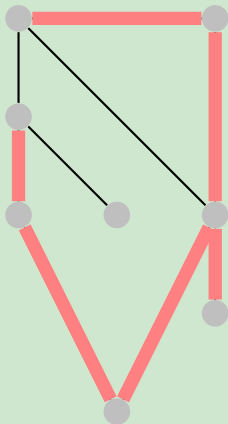
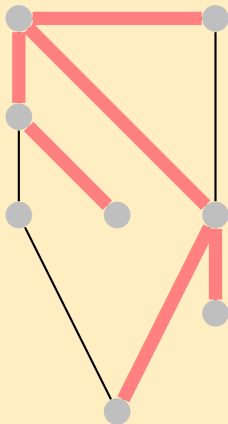
**DFS****BFS**



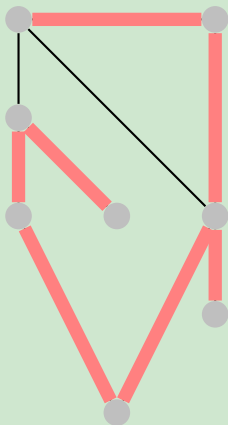
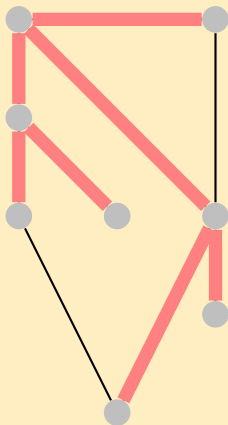
# BFS/DFS: Visualization

**DFS****BFS**

# BFS/DFS: Visualization

**DFS****BFS**

# BFS/DFS: Visualization

**DFS****BFS**

# DFS Implementation

## DFS (Using Adjacency List)

```
vector<int> dfs_vis; // visited nodes, init to 0

void dfs(int v) {
    dfs_vis[v] = 1;
    for (int i; i < AdjList[v].size(); i++)
    {
        edge u = AdjList[v][i]; // u = neighb, weight
        // do something...
        if (dfs_vis[u.first] == 0)
            dfs(v.first);
    }
}

dfs(start_vertice);
```

# BFS Implementation

## BFS (Using adjacency List)

```
vector<int> bfs_vis;    // visited nodes; init to 0
queue<int> q;           // list of vertices to visit;
q.push(start_vertice); // Start BFS

while(!q.empty()) {
    int u = q.front(); q.pop();
    bfs_vis[u] = 1;
    // Do something...
    for (int i = 0; i < AdjList[v].size(); i++) {
        edge e = AdjList[v][i];
        if (bfs_vis[e.first] == 0)
            q.push(e.first);
    }
}
```

# BFS and DFS

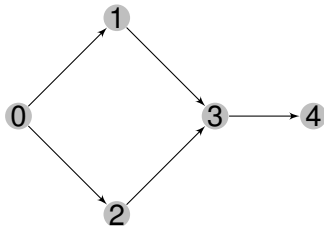
## Computational Cost

In the full BFS and DFS, you need to check every vertex and every edge in the graph:

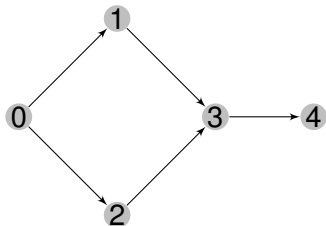
- A BFS/DFS implemented with **Adjacency List**, costs  $O(V + E)$ .
- A BFS/DFS implemented with **Adjacency Matrix**, costs  $O(V^2)$ .
  - That's because to visit every edge of a vertex in an Adjacency Matrix, it costs  $O(V)$ .

# Solving the Dominator Problem with DFS

- $V_j$  is dominated by  $V_i$ , if all paths from  $S$  to  $V_j$  pass through  $V_i$ ;
- In other words, you cannot access  $V_j$  from  $S$ , if  $V_i$  is not available;
- **Algorithm:** Remove  $V_i$ , and test if you can access  $V_j$ ;



# Solving the Dominator Problem with DFS



## Solution: DFS $N$ Times (BFS OK too)

```
// Modified DFS: never visits V_i;  
boolean DFS2(S,i);  
  
// init: Processing Root node  
DFS2(0,-1);  
for (int j = 0; j < N; j++)  
    if (VISITED[j])  
        DOMINATED[0][j] == 1;  
  
for (int i = 1; i < N; i++) {  
    memset(VISITED,0,sizeof(VISITED));  
    DFS2(0,i);  
    for (int j = 0; j < N; j++)  
        if (!VISITED[j] && DOMINATED[0][j])  
            DOMINATED[i][j] == 1;  
}
```



# Common Graph Problems

In this section, we will review several common problems involving graphs in Programming Challenges.

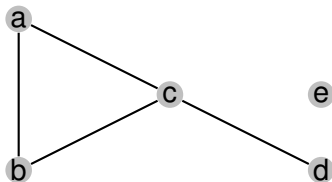
Most of these can be solved with small modifications to DFS or BFS.

- Connected Components;
- Flood Fill;
- Topological Sort;
- Bipartite Checking;
- Articulation Vertices;
- Strongly Connected Components;

# Connected Components

## Definition

A **connected component** of a (undirected) graph is a subset of vertices  $C^k$  where every  $V_i \in C^k$  is reachable.



# Connected Components

## Example Problem

### Problem Example: Extra cables

There is a network of  $N$  computers. Some of the computers are connected by cables. Computers connected by cables, even if indirectly, are said to be on the **same network**.

What is the minimum number of cables that you need to make sure that all  $N$  computers are part of the same network?

**Solution:** Count the number of Connected Components ( $C$ ), the answer is  $C - 1$ .

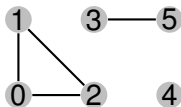
**Quiz:** How do you implement this?

# Connected Components

## Algorithm

We can find all connected components by looping through all vertices, and running BFS/DFS on each unvisited vertex;

```
int cables = 0;
for (int i = 0; i < N; i++)
    if (dfs_vis[i] == 0) // New component!
    {
        dfs(i);           // Visit vertices
        cables += 1;
    }
cout << "Need " << cables - 1 << ".\n";
```



# Connected Components

## UFDS Variant

You could also count Connected Components using the **UFDS** data structure from lecture 2.

It costs  $O(E)$  to build the UFDS, and  $O(V)$  to count the number of components.

If your problem is dynamic and includes several additions to the graph edges, this might be a good choice, because it is cheaper to recalculate the CCs.

# Flood Fill

## Problem: The Biggest Island

You want to build a large in Minecraft. For this, you need to find the biggest island in the map.

**Input:** A 2D representation of the map:

```

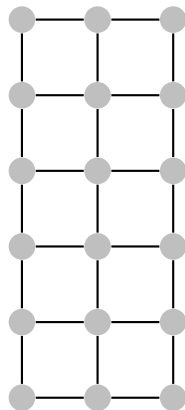
. . . . .
. ### . . . . . ### . . . . . # . . . . . ### . ### . . .
. ##### . . . ##### . ## . ##### . ## . . . . # . . .
. ### . . . . . ### . # . . . ## . # . . . ### . . .
. . . . . ### . . . . . ### . . . ### . . . ## . . . .
. . . . ### . . . . . . . . ##### . . . . ### .
. . . . ### . . . . . # . . . . . ### . . . . ### .
. . . . .

```

Is this a graph problem?

# Implicit Graphs

- **Implicit Graphs** are data that suggest graph organization. Examples:
  - grids (NSWE connections)
  - maps (distance = weights)
- In some problems, it is not necessary to store the entire graph from the beginning;
- **Grid Floodfill**: Painting images, Walkable tiles in videogames, etc;
- Algorithm is just BFS/DFS with vertex labels;



# Flood Fill

## Implementation

"Biggest Island" can be solved with BFS/DFS modifying labels;

```
int dr[] = {1,1,0,-1,-1,-1,0,1}; // neighbors for a grid
int dc[] = {0,1,1,1,0,-1,-1,-1}; // with diagonals;

int floodfill(int y, int x) { // size of one position
    if (y < 0 || y >= R || x < 0 || x >= C) return 0;
    if (grid[y][x] != '#') return 0;
    int size = 1;
    grid[y][x] = '.'; // CHANGE THE MAP (**key**)
    for (int d = 0; d < 8; d++)
        size += floodfill(y+dr[d], x+dc[d]);
    return ans;
}

biggest = 0;
for (int i = 0; i < C; i++)
    for (int j = 0; j < R; j++)
        biggest = max(biggest, floodfill(i,j));
```



# Topological Sort

## Problem: Preparing a Curriculum

You have a list of courses and requisites. Choose an **ordering** of topics that respect all requisites.

**Input:** list M topics, and N pairs of topics;

**Output:** Sorted list of all topics;

**\*\* Example Input:**

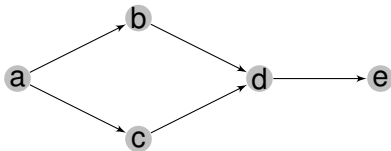
```
5 4 Graphs DP Search Flow Programming
Programming Search
Search DP
Graph Flow
Search Graph
```

**\*\* Example Output:**

```
Course: Programming Search DP Graph Flow
```

# Topological Sort Definition

In a **directed graph**, a topological sort is an ordering of vertices where  $V_i \prec V_j$  **iff** there is no path  $V_j \rightarrow V_i$ .



For this graph, a topological sort would be  $a \prec b \prec c \prec d \prec e$ .

- Toposorts are **not unique**:
  - $a \prec c \prec b \prec d \prec e$  is also a toposort.
- A graph only has a toposort if it has **no cycles**.
- To find the toposort, we use **in-degrees and out-degrees**:
  - $a$  – In-deg: 0; Out-deg: 2;
  - $d$  – In-deg: 2; Out-deg: 1;
  - $e$  – In-deg: 1; Out-deg: 0;

# Topological Sort Algorithm

**Khan's algorithm** can be used to find the toposort of a graph. It is composed of a **BFS** where we keep track of the in-degrees of a vertex before adding it to the visit queue.

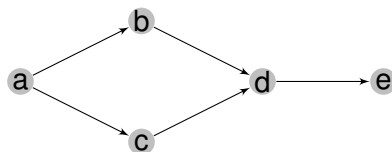
```
queue<int> q; vector<int> toposort;
vector<int> in-deg; // initialize to 0 for all N;

for (int i = 0; i < EdgeList.size(); i++)
    in-deg[EdgeList[i].second]++;
for (int i = 0; i < N; i++)
    if (in-deg[i] == 0) q.push(i); // queue in-deg = 0;

while (!q.empty()) {
    u = q.front(); q.pop(); toposort.push_back(u); // Do something
    for (int i = 0; i < EdgeList[u].size(); i++) {
        d = EdgeList[u][i].first; in-deg[d]--;
        if (in-deg[d] == 0) q.push(d); // queue in-deg = 0;
    }
}
```

# Khan's Algorithm

## Simulation



### In-deg list:

- a:
- b:
- c:
- d:
- e:

### Toposort:

# Bipartite Graphs

## Definition

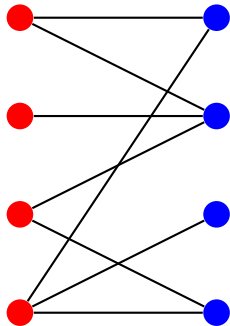
Intuitively, a **Bipartite Graph** is one that we can separate between a "left" side and a "right" side.

More generally, a graph  $(V, E)$  is bipartite if you can completely partition its vertices in two subsets:  $V_1$  and  $V_2$ , so that **there are no edges** connecting two vertices in the same subset.

Bipartite graphs appear in a large number of algorithms. In particular, **flow graphs** (next week) are bipartite graphs.

Most neural networks are bipartite graphs too!

**Quiz:** How do you test if a graph is bipartite?



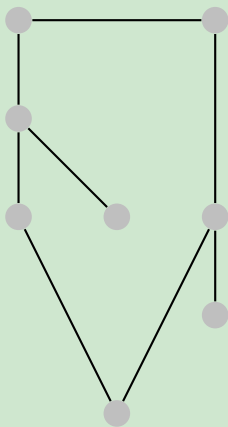
# Bipartite Check Algorithm

We can check if a graph is bipartite using a BFS/DFS. Every time we visit a vertice, we mark it "0" or "1". If two adjacent vertices are of the same colors, the graph is not bipartite.

```
queue<int> q; q.push(s);
vector<int> color(V, -1); color[s] = 0; // Starting vertex
bool isBipartite = True;

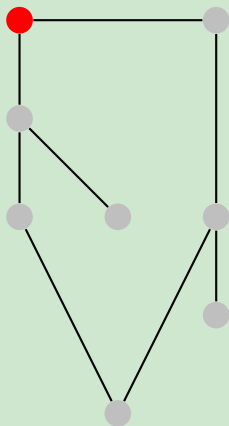
while (!q.empty() && isBipartite) {
    int u = q.front(); q.pop();
    for (int j=0; j < adj_list[u].size(); j++) {
        v = adj_list[u][j].first;
        if (color[v] == -1) {
            color[v] = 1 - color[u];           // Coloring new vertex
            q.push(v.first);
        }
        else if (color[v.first] == color[u]) {
            isBipartite = False;                // Bipartite collision
        }
    }
}
```

## Testing Bipartite property



# Bipartite Check – Visualization

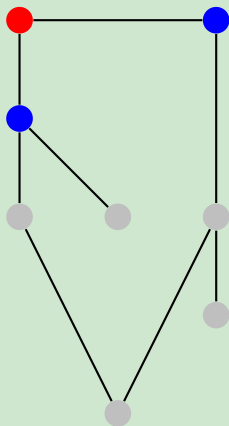
## Testing Bipartite property





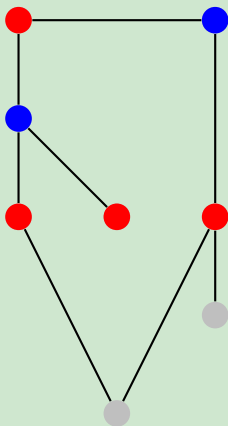
# Bipartite Check – Visualization

## Testing Bipartite property



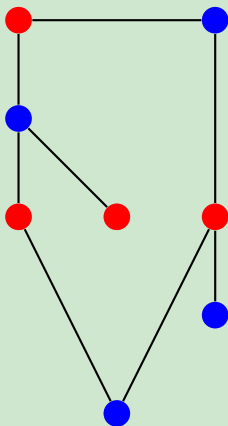
# Bipartite Check – Visualization

## Testing Bipartite property



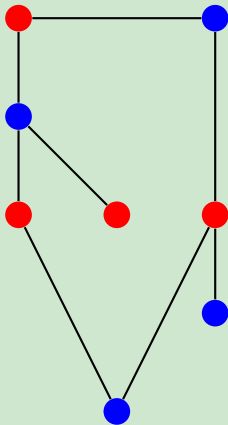
# Bipartite Check – Visualization

## Testing Bipartite property

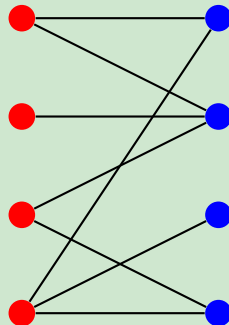


# Bipartite Check – Visualization

## Testing Bipartite property



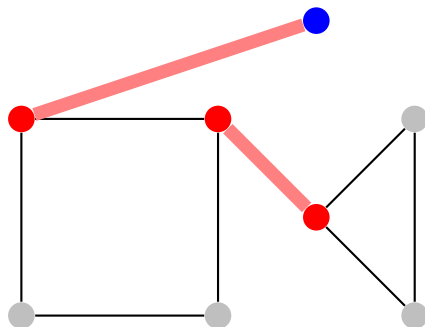
## Rearranging the nodes



# Articulation Points and Bridges

Definition: In a graph  $G$

- Vertex  $v_i$  is an **Articulation Point** if removing  $v_i$  makes  $G$  disconnected.
- Edge  $e_{i,j}$  is a **Bridge** if removing  $e_{i,j}$  makes  $G$  disconnected.



# Problems and Naive Algorithm

## Example Problems

- Find nodes that can be/can not removed from a graph;
- Add extra nodes to "reinforce" a graph;
- Measure the reliability of a network;
- etc;

## Complete Search algorithm for Articulation Points $O(V^2 + VE)$

- 1 Run DFS/BFS, and count the number of CC in the graph;
- 2 For each vertex  $v_i$ , remove  $v_i$  and run DFS/BFS again;
- 3 If the number of CC increases,  $v$  is an articulation point;

# Tarjan's DFS variant for Articulation point ( $O(V+E)$ )

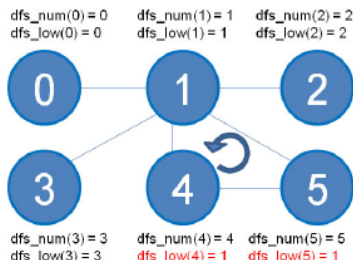
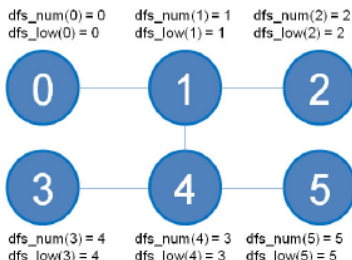
Find Articulation Points/Bridges in a single DFS pass:  $O(V + E)$

Main idea: Track loops to detect articulations:

- **dfs\_num[i]**: iteration number (DFS order)
- **dfs\_low[i]**: lowest dfs\_num reachable from  $i$ ;
  - (not counting direct parent);

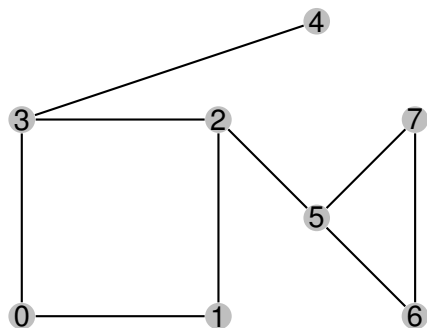
For any  $u, v$ , if  $\text{low}[v] \geq \text{num}[u]$ ,  $u$  is an articulation node (except root)

For any  $u, v$ , if  $\text{low}[v] > \text{num}[u]$ ,  $e_{u,v}$  is a bridge;



# Tarjan's Algorithm for Articulation Point

## Simulation



DFS\_NUM

DFS\_LOW

0

1

2

3

4

5

6

7



# Tarjan's Algorithm for Articulation Point Implementation

```
void articulation(u){
    dfs_num[u] = dfs_low[u] = IterationCounter++; // update num[u], init low[u]
    for (int i = 0; i < AdjList[u].size(); i++){
        v = AdjList[u][i];
        if (dfs_num[v.first] == UNVISITED) {           // DFS tree edge
            dfs_parent[v.first] = u;                 // store parent
            if (u == 0) rootTreeEdge++;              // special case for root node

            articulation(v.first);                    // visit next

            // This code happens AFTER we finish the DFS from u;
            if (dfs_low[v.first] >= dfs_num[u])
                articulation_vertex[u] = true;        // do something
            if (dfs_low[v.first] > dfs_num[u])
                bridge[u][v.first] = true;           // do something
            dfs_low[u] = min(dfs_low[u], dfs_low[v.first])
        }
        else if (v.first != dfs_parent[u])           // found a cycle edge
            dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);
    } // for j;
} // articulation(u);

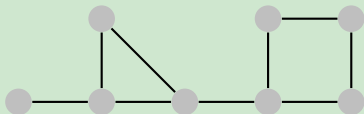
// Starting the algorithm from the main program:
articulation(0);
if (rootTreeEdge > 1)
    articulation_vertex[0] = true;                  // special case for root
```

# Strongly Connected Components

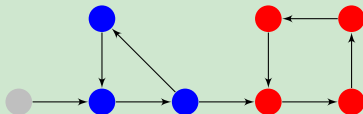
## Definition

Given a **directed** graph  $G(V, E)$ , a **Strongly Connected Component (SCC)** is a subset of vertices  $V_1$  where for every pair of vertices  $v_i, v_j \in V_1$ , there is both a path  $v_i \rightarrow v_j$  and a path  $v_j \rightarrow v_i$ .

## One Connected Component (undirected)



## Three Strongly Connected Components (directed)



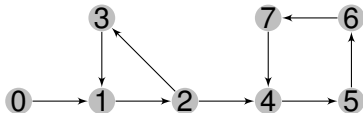
# Algorithm for Finding SCCs

We can modify Tarjan's algorithm (for articulation points and bridges) to find Strongly Connected Components:

- Every time we visit a new vertex  $u$ , we put  $u$  in a stack  $S$ ;
- Only update `dfs_low` for vertices with the "visited" flag = 1;
- After visiting all edges of  $u$ , check if `dfs_num[u] == dfs_min[u]`;
- If the condition is true,  $u$  is the root of a new SCC.
- Pop all vertices in  $S$  until (and including)  $u$ ;
- Add all popped vertices to the SCC.

# Algorithm for Finding SCCs

## Simulation



## SCC Stack:

0    1    2    3    4    5    6    7

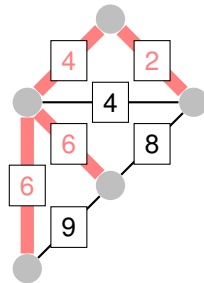
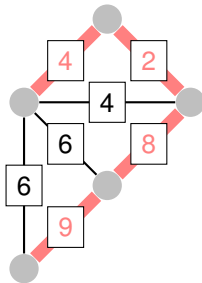
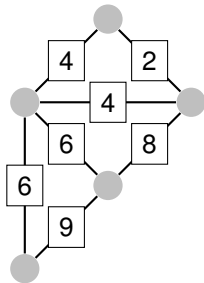
dfs\_low

dfs\_sum

# Minimum Spanning Trees (MST) – Definition

A **Spanning Tree** is a subset  $E'$  from graph  $G$  so that all vertices are connected without cycles.

A **Minimum Spanning Tree** is a spanning tree where the sum of edge's weights is minimal.



# Usage Cases for Minimum Spanning Trees

- Problems with MST often ask for a minimal cost to connect all elements in a graph (e.g. minimal infrastructure cost).
- **Variations:** Maximum Spanning Tree, Spanning Forest, Force some edges in advance;

## Main algorithms for MST

Two greedy algorithms that add edges to MST:

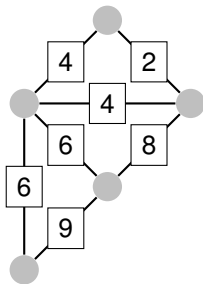
- **Kruskal Algorithm:** based on edge list;
- **Prim's Algorithm:** based on vertex list;

# Kruskal's Algorithm

## Outline

Kruskal's algorithm sorts all edges by their weight, and try to add each edge to the MST, checking whether adding that edge would create a cycle.

- 1 Sort all edges;
- 2 If smallest edge does not create a cycle, add to MST;
- 3 If smallest edge creates a cycle, remove it from list;
- 4 Go to 2;

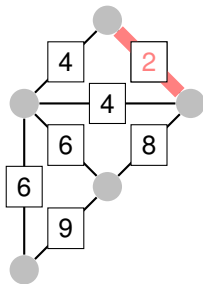


# Kruskal's Algorithm

## Outline

Kruskal's algorithm sorts all edges by their weight, and try to add each edge to the MST, checking whether adding that edge would create a cycle.

- 1 Sort all edges;
- 2 If smallest edge does not create a cycle, add to MST;
- 3 If smallest edge creates a cycle, remove it from list;
- 4 Go to 2;



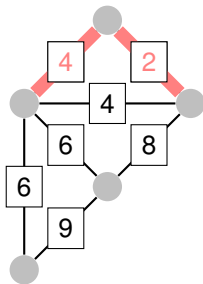


# Kruskal's Algorithm

## Outline

Kruskal's algorithm sorts all edges by their weight, and try to add each edge to the MST, checking whether adding that edge would create a cycle.

- 1 Sort all edges;
- 2 If smallest edge does not create a cycle, add to MST;
- 3 If smallest edge creates a cycle, remove it from list;
- 4 Go to 2;

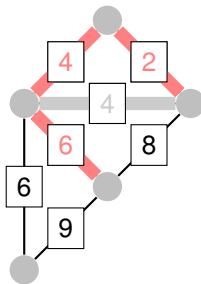


# Kruskal's Algorithm

## Outline

Kruskal's algorithm sorts all edges by their weight, and try to add each edge to the MST, checking whether adding that edge would create a cycle.

- 1 Sort all edges;
- 2 If smallest edge does not create a cycle, add to MST;
- 3 If smallest edge creates a cycle, remove it from list;
- 4 Go to 2;

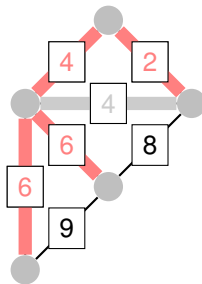


# Kruskal's Algorithm

## Outline

Kruskal's algorithm sorts all edges by their weight, and try to add each edge to the MST, checking whether adding that edge would create a cycle.

- 1 Sort all edges;
- 2 If smallest edge does not create a cycle, add to MST;
- 3 If smallest edge creates a cycle, remove it from list;
- 4 Go to 2;



# Kruskal's Algorithm – Implementation

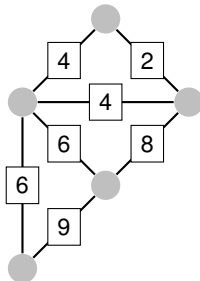
```
vector<pair<int, pair<int,int>> Edgelist;  
sort(Edgelist.begin(),Edgelist.end());  
int mst_cost = 0;  
UnionFind UF(V);  
    // note 1: Pair object has built-in comparison;  
    // note 2: Need to implement UnionSet class;  
  
for (int i = 0; i < Edgelist.size(); i++) {  
    pair <int, pair <int,int>> front = Edgelist[i];  
    if (!UF.isSameSet(front.second.first,  
                      front.second.second)) {  
        mst_cost += front.first;  
        UF.unionSet(front.second.first,front.second.second)  
    }  
}  
  
cout << "MST Cost: " << mst_cost << "\n"
```

# Prim's Algorithm

## Outline

Prim's algorithm adds nodes to the MST one at a time, and keeps the edges connected to those nodes in a **priority queue**. It then tests each edge in the priority queue to add more nodes to the MST, avoiding cycles.

- 1 Add node 0 to MST;
- 2 Add all edges from new node to Priority Queue;
- 3 Visit smallest edge in Queue;
- 4 If the edge leads to a new node, add it to MST;
- 5 Add new edges to Queue;
- 6 Go to 3;

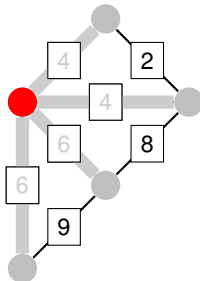


# Prim's Algorithm

## Outline

Prim's algorithm adds nodes to the MST one at a time, and keeps the edges connected to those nodes in a **priority queue**. It then tests each edge in the priority queue to add more nodes to the MST, avoiding cycles.

- 1 Add node 0 to MST;
- 2 Add all edges from new node to Priority Queue;
- 3 Visit smallest edge in Queue;
- 4 If the edge leads to a new node, add it to MST;
- 5 Add new edges to Queue;
- 6 Go to 3;

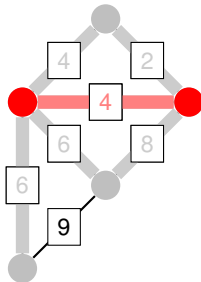


# Prim's Algorithm

## Outline

Prim's algorithm adds nodes to the MST one at a time, and keeps the edges connected to those nodes in a **priority queue**. It then tests each edge in the priority queue to add more nodes to the MST, avoiding cycles.

- 1 Add node 0 to MST;
- 2 Add all edges from new node to Priority Queue;
- 3 Visit smallest edge in Queue;
- 4 If the edge leads to a new node, add it to MST;
- 5 Add new edges to Queue;
- 6 Go to 3;

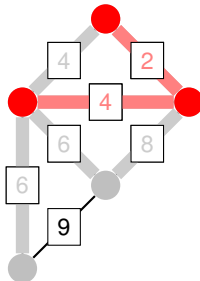


# Prim's Algorithm

## Outline

Prim's algorithm adds nodes to the MST one at a time, and keeps the edges connected to those nodes in a **priority queue**. It then tests each edge in the priority queue to add more nodes to the MST, avoiding cycles.

- 1 Add node 0 to MST;
- 2 Add all edges from new node to Priority Queue;
- 3 Visit smallest edge in Queue;
- 4 If the edge leads to a new node, add it to MST;
- 5 Add new edges to Queue;
- 6 Go to 3;



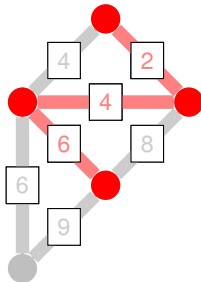


# Prim's Algorithm

## Outline

Prim's algorithm adds nodes to the MST one at a time, and keeps the edges connected to those nodes in a **priority queue**. It then tests each edge in the priority queue to add more nodes to the MST, avoiding cycles.

- 1 Add node 0 to MST;
- 2 Add all edges from new node to Priority Queue;
- 3 Visit smallest edge in Queue;
- 4 If the edge leads to a new node, add it to MST;
- 5 Add new edges to Queue;
- 6 Go to 3;

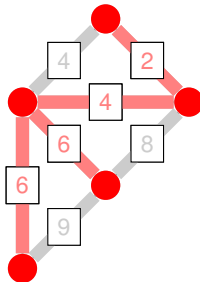


# Prim's Algorithm

## Outline

Prim's algorithm adds nodes to the MST one at a time, and keeps the edges connected to those nodes in a **priority queue**. It then tests each edge in the priority queue to add more nodes to the MST, avoiding cycles.

- 1 Add node 0 to MST;
- 2 Add all edges from new node to Priority Queue;
- 3 Visit smallest edge in Queue;
- 4 If the edge leads to a new node, add it to MST;
- 5 Add new edges to Queue;
- 6 Go to 3;



# Prim's Algorithm – Implementation

```
vector<int> taken;
priority_queue<pair<int,int>> pq;

void process (int v) {
    taken[v] = 1;
    for (int j = 0; j < (int)AdjList[v].size(); j++) {
        pair<int,int> ve = AdjList[v][j];
        if (!taken[ve.first])
            pq.push(pair<int,int> (-ve.second,-ve.second))
    }
}

taken.assign(V,0);
process(0);
mst_cost = 0;

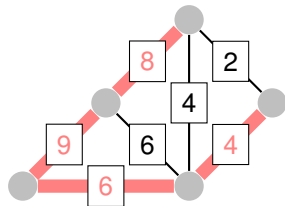
while (!pq.empty()) {
    vector<int,int> pq.top(); pq.pop();
    u = -front.secont, w = -front.first;
    if (!taken[u]) mst_cost += w, process(u);
}
```

# MST variant 1 – Maximum Spanning tree

The **Maximum Spanning Tree** variant requires the spanning tree to have maximum possible weight.

It is very easy to implement the Maximum MST:

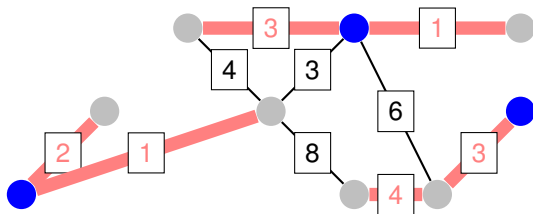
- **Kruskal**: Reverse the sort of the edge list;
- **Prim**: Invert the weight of the priority queue;



# MST variant 2 – Minimum Spanning Subgraph, Forest

In this variant, a subset of edges or vertices are pre-selected.

- In the case of pre-selected vertices, add them to the “taken” list in Kruskal’s algorithm before starting;
- In the case of edges, add the end vertices to the “taken” list;



# MST Variant 3 – Second Best MST

## Problem Definition

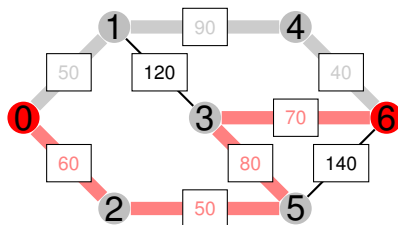
Suppose that you are required to calculate an alternative solution to an MST problem. In this case, you need to find the second cheapest spanning tree.

Simple Algorithm:

- Calculate the MST (using Kruskal or Prim);
- For every edge  $e_i$  in the MST:
  - Remove  $e_i$  from  $E$ ;
  - Calculate a new MST;
- Choose the best among the new MSTs as the second-best MST.

**QUIZ:** How to generalize this algorithm for the  $n$ -th best spanning tree?

# MST Variant 4 – Minmax path cost



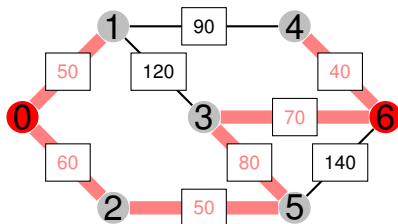
## Problem Definition

**Regular Cost** for a path is the sum of weights of all edges in the path.

**Minmax Cost** for a path is the maximum weight among all its edges.

Find the path  $v_i \rightarrow v_j$  with the smallest **minmax cost**

# Finding the Minmax path with MST



## Algorithm

- Generate the MST for the graph  $G$ .
- Find the path  $v_i \rightarrow v_j$  inside the MST.

That's it!



# Class Summary

## Graph Basics

- Graph Problems come in a large variety of types;
- But Many Algorithms are variations on BFS and DFS;
  - Connected Components and Flood Fill;
  - Topological Sort;
  - Articulation Points and Bridges;
  - Minimum Spanning Trees;
- There are several special cases for graph problems:

### Some common special cases

- Graphs with 0 or 1 Vertices; Graphs with 0 nodes;
- Unconnected Graphs;
- Self loops;
- Double edges;

# Class Summary

## Theme for Next Week

### Graph Path Search and Weighted Graphs:

- Shortest Paths (Single Source and All Pairs);
- Network Flow;
- Graph Matching;

### Graph Code Library

Graph problems share a lot of common code. I recommend that you prepare a code library as you learn new algorithms.

- Visited node flags and adjacency lists;
- Parent and children lists;
- Different algorithms;
- etc;

# This Week's Problems

- Dominator – Discussed in class;
- Forwarding Emails
- Ordering
- Place the Guards
- Doves and Bombs
- Come and Go
- ACM Contest and Blackout
- Ancient Messages

# Problem Hints

## Forwarding E-mails

### Problem Summary

- In a list of  $N$  people, every person  $i$  forwards e-mails to a single person  $j$ .
  - If you want to form the longest e-mail chain, where do you start?
  - **Output:** The person  $k$  that starts the longest chain;
- 
- The time limit is not very large, you should find an  $O(n)$  solution;
  - How do you deal with loops?

# Problem Hints

## Ordering and Palace Guards

### Ordering

- **Outline:** List all possible orderings of a given Direct Acyclic Graph;
- **Hint:** In the lecture, we saw an algorithm for a single ordering – for this problem, you need to generalize it for all orderings;

### Palace Guards

You need to find a guard assignment for the city that obey the rules:

- All roads need **one guard**.
- No road can have **two guards**.

Note that for some inputs a solution is impossible.

# Problem Hints

## Doves and Bombs / Come and Go

### Doves and Bombs

- **Outline:** You must choose one station (vertex) to remove, so that the number of unconnected subgraphs is maximum;
- **Hint:** Do you need to calculate the CCs every time you test a vertex, or is there a better way?

### Come and Go

- **Outline:** For a city road map, you must check if that city is strongly connected (you can go from every node  $i$  to every node  $j$ ) or not;
- **Hint:** Can you use the algorithm of strongly connected components for this?

# Problem Hints

## ACM Contest and Blackout

### Outline

Given a graph of a powerplant and several schools, you have to find the **two** cheapest graphs that connect all the nodes.

**Output:** Cost of cheapest graph, and cost of 2nd cheapest graph.

**Hint:** The cheapest graph could be found using the MST algorithm. But how do we find the **second** cheapest graph?

# Problem Hints

## Ancient Message

### Outline

- **Input:** A pixel image containing one or more "egyptian letters";
- **Output:** Identify which letters appear in the image;

### Hints:

- Note that the input is in hexadecimal format;
- First: Think about how to count the number of characters in an image. This will help you think a way to identify the characters.
- Pay attention to the "Rules" in the problem description.

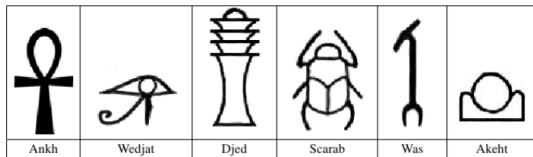


Figure C.2: AKW



# About these Slides

These slides were made by Claus Aranha, 2020. You are welcome to copy, re-use and modify this material.

Individual images in some slides might have been made by other authors. Please see the references in each slide for those cases.

# Image Credits I