GB20602 - Programming Challenges

Week 5 - Graph Part I: Basics

Claus Aranha
caranha@cs.tsukuba.ac.jp

University of Tsukuba, Department of Computer Sciences

(last updated: May 15, 2022)

Version 2022

Part I – Graph Introduction

Graph Algorithms: Week 5 and 6

Graphs Part I (This Week)

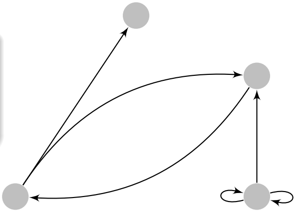
- Graphs Data Structure;
- Depth First Search and Breadth First Search;
- Graph Search Problems (DFS and BFS);
- Minimum Spanning Tree: Kruskal and Prim Algorithms;

Graphs Part II (Next Week)

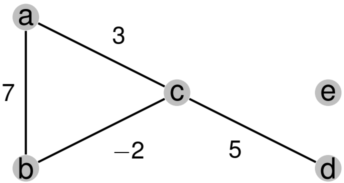
- Single Sourse Shortest Path (Djikstra);
- All Pairs Shortest Path (Floyd-Warshall);
- Network Flow;
- Bipartite Graph Matching;

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.



- An edge can be **directed** or **undirected**;
- An edge or a vertice can have **weights** or **labels**;
- **Self-edge**: edge between v_i and v_i;
- **Multi-edge**: two edges with same end-vertices;
- A graph can be **connected** or **disconnected**;



Graphs in Computer Science

Graph Data structures show relationships between data;
They are used in many 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;

Common graph tasks in an algorithm

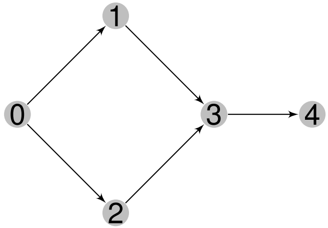
- Test if a path exist between vertice V_i and V_j (test if they are **connected**)
- Test the shortest path between vertice V_i and V_j
    - With or without weights
    - Test if there is more than one path
- Add or remove vertices or edges from a graph;
- Test some characteristics of a graph;
    - Longest path? Shortest path?
    - Does it have a **Cycle**?
    - Vertice with maximum number of vertices?
    - etc...

Programming Challenge Example

Dominator

Definition: A vertice 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 the DOMINATE relationship



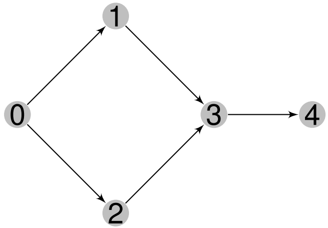

```
Input:
5
0 1 1 0 0
0 0 0 1 0
0 0 0 1 0
0 0 0 0 1
0 0 0 0 0
```

```
Output:
0 -> 0, 1, 2, 3, 4
1 -> 1
2 -> 2
3 -> 3, 4
4 -> 4
```

Programming Challenge Example

Dominator

- Which data structure should be used?
- How to calculate the "DOMINATE" status of a vertice?




```
Input:
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:
0 -> 0, 1, 2, 3, 4
1 -> 1
2 -> 2
3 -> 3, 4
4 -> 4
```

Data Structure for Graph 1

Adjacency Matrix: stores the connection between vertices

```
int adj[100][100];

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

- Pros:
  - Easy to program;
  - Access to edge e_{ij} is quick;
- Cons:
  - Cannot store multigraph;
  - Wastes memory with sparse graphs;
  - Time $O(V)$ to calculate number of neighbors of vertice v_i;

Data Structure for Graph 2

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; (QUIZ: Why log(V)?)

Data Structure for Graph 3

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, used in specialized algorithms (ex:MST);
- To find if two vertices are neighbors, list must be sorted;

Graph Search: BFS and DFS

- Graph Search Question: from vertice v_s, can we reach v_e?
- Many graph algorithms start from a graph search;
- Two basic algorithms for search: BFS, DFS;

Depth First Search – DFS

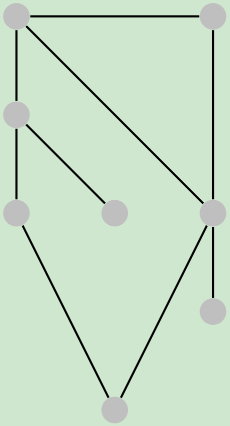
- Visit the first edge available;
- Vertice order is not guaranteed;
- Easy to implement with recursion or stack;

Breadth First Search – BFS

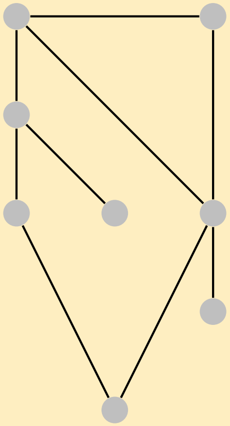
- First visit the vertices close to the starting point;
- Place new vertices on a list, and visit them with a loop;

BFS and DFS: Visualize the difference

DFS

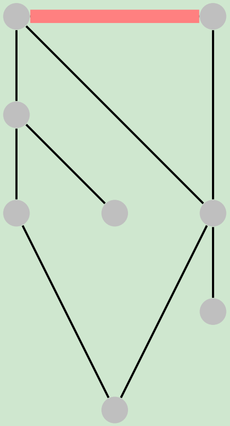


BFS

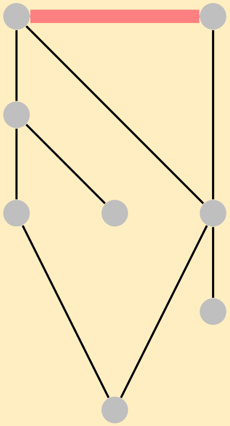


BFS and DFS: Visualize the difference

DFS

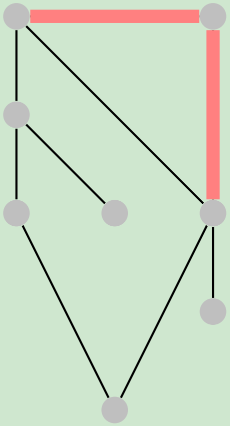


BFS

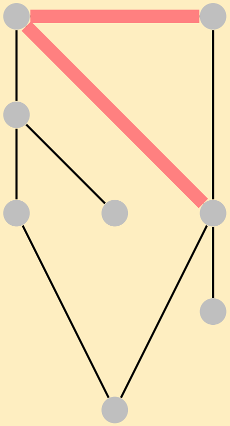


BFS and DFS: Visualize the difference

DFS

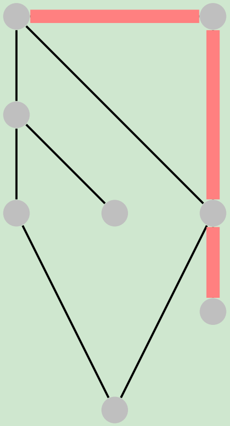


BFS

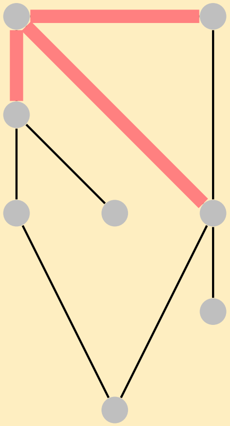


BFS and DFS: Visualize the difference

DFS

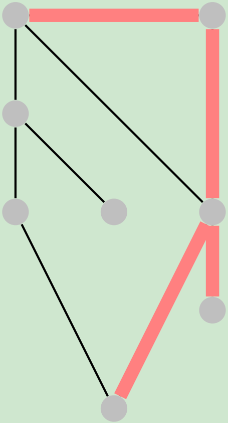


BFS

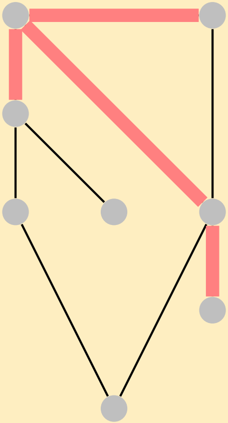


BFS and DFS: Visualize the difference

DFS

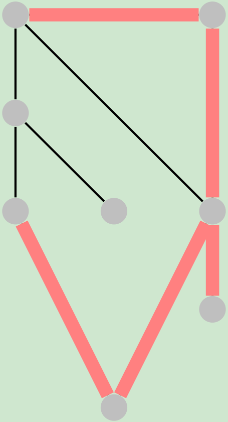


BFS

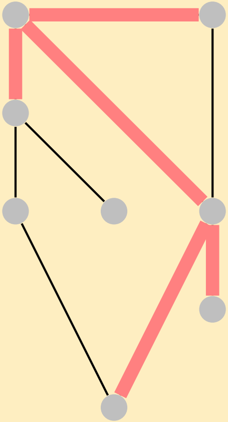


BFS and DFS: Visualize the difference

DFS

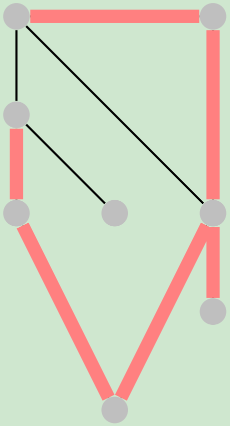


BFS

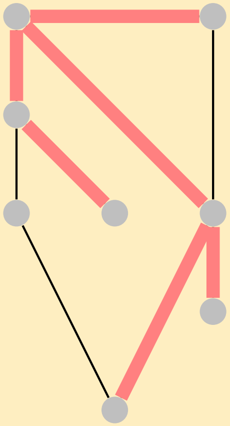


BFS and DFS: Visualize the difference

DFS

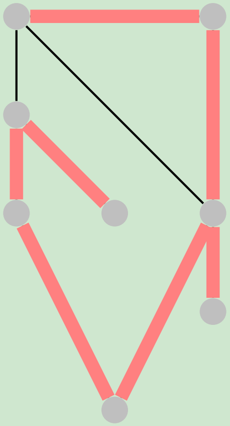


BFS

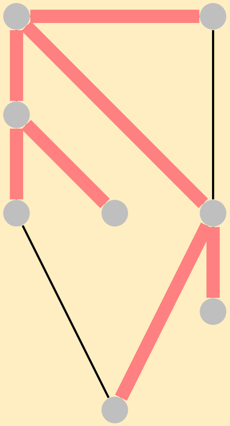


BFS and DFS: Visualize the difference

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)   // Check if node is visited
      q.push(e.first);
  }
}
```

BFS and DFS

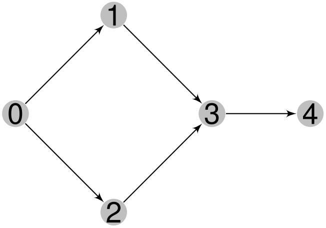
Computational Cost

In the full BFS and DFS, you need to check every vertice 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 vertice in an Adjacency Matrix, it costs $O(V)$.
- Adjacency List is faster, **if the graph is sparse (has few edges)**

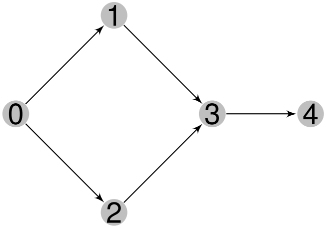
Solving the Dominator Problem with DFS

- v_j is dominated by v_i, if all paths from v_0 to v_j pass through v_i;
- In other words, you cannot access v_j from v_0, if v_i is not available;
- **Algorithm:** Remove v_i, and test if you can access v_j from v_0;



Solving the Dominator Problem with DFS

Use DFS/BFS N times




```
// Modified DFS: does not visit vertex v_i;
boolean DFS2(S,i) {...};

// initialization: which nodes v_0 can reach?
DFS2(0,-1);
for (int j = 0; j < N; j++)
  if (VISITED[j]) { DOMINATED[0][j] = 1; }

// check DOMINATED relationship of each v_i
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;
}
```

Part II: Common Graph Problems

Common Graph Problems in Competitive Programming

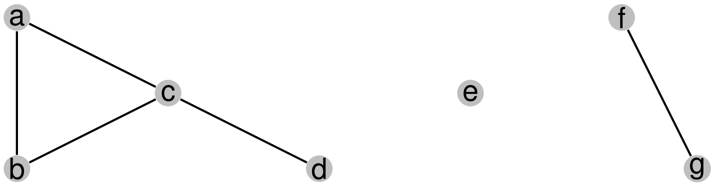
Let's see some common problems that can be solved using DFS or BFS.

- Connected Components;
- Flood Fill;
- Topological Sort;
- Bipartite Checking;

Connected Components (undirected graph)

A **connected component** of a graph is a subset of vertices $C \subset V$ where every pair of vertices $v_i, v_j \in C$ is connected.

The graph below has 3 connected components (abcd, e, fg)



Connected Components

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 $C - 1$.

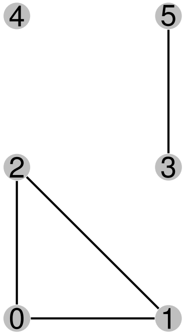
Quiz: How do you implement this?

Connected Components

Finding Connected Components using BFS/DFS

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

```
int dfs_vis[];              // visited vertices

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



Flood Fill

Problem: Find The Biggest Island

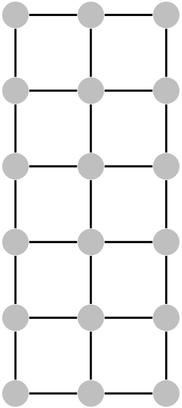
You want to find the biggest island in a game map to build a castle.
Input: A 2D representation of the map:

```
....................................
.###.......###.....#.....###.####...
.#####....#####.##.#####.##....#....
.###........###..#...##..#....###...
......###.......###...####...##.....
....####.............######.....###.
....####.......#.......###......###.
....................................
```

Can we solve this as a graph problem?

Implicit Graphs

- **Implict 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

Finding the "Biggest Island" with BFS/DFS and 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 to mark visited nodes
  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

Example Problem: Preparing a Curriculum

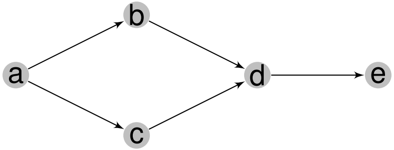
You have a list of courses and requisites.
Choose an **ordering** of topics that respect all requisites.

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

A **topological sort** is an ordering of vertices where $v_i \prec v_j$ only if there is no path $v_j \rightarrow v_i$.



For this graph, one possible topological sort is $a \prec b \prec c \prec d \prec e$.

- 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** of each vertex:
  - a – In-deg: 0; Out-deg: 2;
  - d – In-deg: 2; Out-deg: 1;
  - e – In-deg: 1; Out-deg: 0;

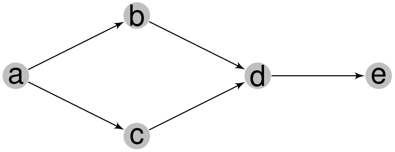
Finding Topological Sort – Khan's Algorithm

Modified BFS: Vertices are only added to the queue if they in-degree is 0.

```
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]++;       // calculate in-degrees based on edge list.
for (int i = 0; i < N; i++)
  if (in-deg[i] == 0) q.push(i);      // add vertices with in-deg = 0 to queue

while (!q.empty()) {
  u = q.front(); q.pop(); toposort.push_back(u); // Add top of queue to toposort
  for (int i = 0; i < EdgeList[u].size(); i++) {
    d = EdgeList[u][i].first; in-deg[d]--;       // remove edges from visited.
    if (in-deg[d] == 0) q.push(d); // queue in-deg = 0;
  }
}
```

Khan's Algorithm

Simulation

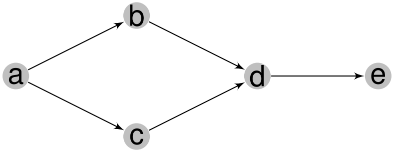


In-deg list:

Toposort:

Khan's Algorithm

Simulation



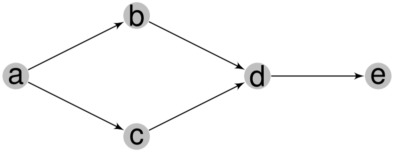
In-deg list:

- iteration 1: (a,0), (b,1), (c,1), (d,2), (e,1) visit a

Toposort: a,

Khan's Algorithm

Simulation



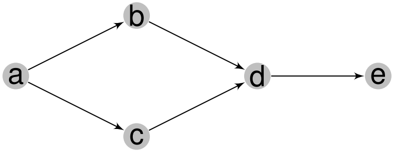
In-deg list:

- iteration 1: (a,0), (b,1), (c,1), (d,2), (e,1) visit a
- iteration 2: (b,0), (c,0), (d,2), (e,1) visit b

Toposort: a, b,

Khan's Algorithm

Simulation



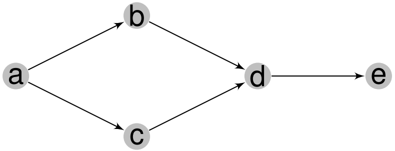
In-deg list:

- iteration 1: (a,0), (b,1), (c,1), (d,2), (e,1) visit a
- iteration 2: (b,0), (c,0), (d,2), (e,1) visit b
- iteration 3: (c,0), (d,1), (e,1), visit c

Toposort: a, b, c,

Khan's Algorithm

Simulation



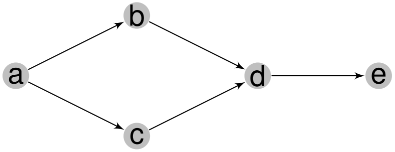
In-deg list:

- iteration 1: (a,0), (b,1), (c,1), (d,2), (e,1) visit a
- iteration 2: (b,0), (c,0), (d,2), (e,1) visit b
- iteration 3: (c,0), (d,1), (e,1), visit c
- iteration 4: (d,0), (e,1) visit d

Toposort: a, b, c, d,

Khan's Algorithm

Simulation



In-deg list:

- iteration 1: (a,0), (b,1), (c,1), (d,2), (e,1) visit a
- iteration 2: (b,0), (c,0), (d,2), (e,1) visit b
- iteration 3: (c,0), (d,1), (e,1), visit c
- iteration 4: (d,0), (e,1) visit d
- iteration 5: (e,0) visit e

Toposort: a, b, c, d, e

Bipartite Graphs

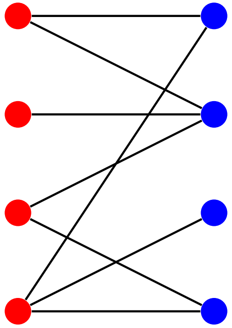
Definition

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?



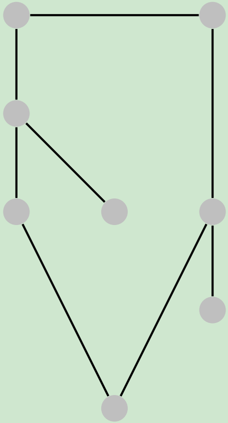
Bipartite Check Algorithm

Visit all vertices using 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[i];        // Coloring new vertex
         q.push(v.first);}
      else if (color[v.first] == color[u]) {
         isBipartite = False;            // Bipartite collision
}}}
```

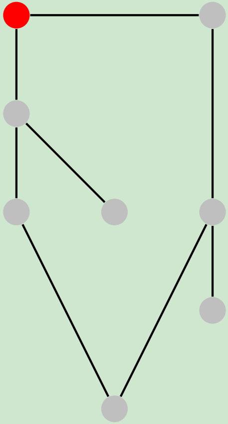
Bipartite Check – Visualization

Testing Bipartite property



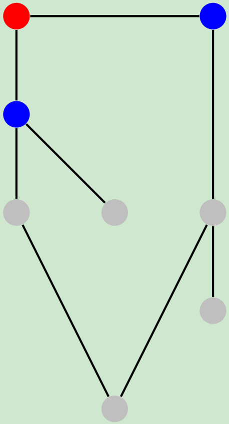
Bipartite Check – Visualization

Testing Bipartite property



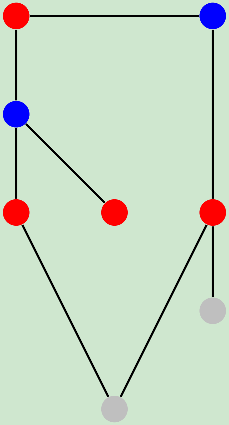
Bipartite Check – Visualization

Testing Bipartite property



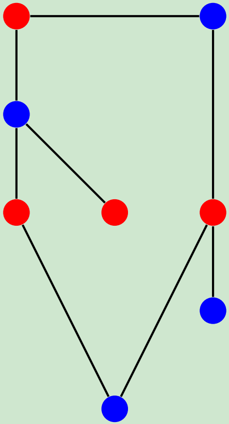
Bipartite Check – Visualization

Testing Bipartite property



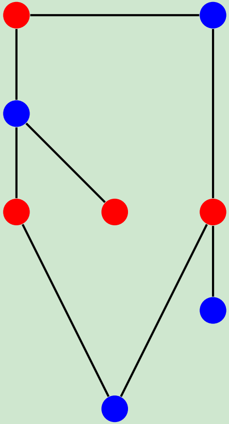
Bipartite Check – Visualization

Testing Bipartite property

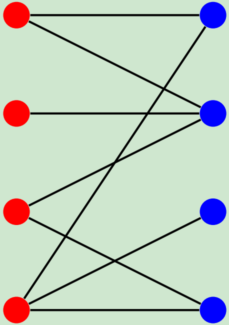


Bipartite Check – Visualization

Testing Bipartite property



Rearranging the nodes

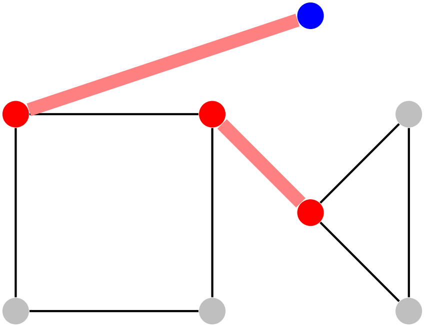


Part III – Articulation Vertices and Edges

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 vertices that can be removed from a graph to "break" it;
- Add extra edges to "reinforce" a graph;
- Measure the reliability of a network, etc;

Complete Search algorithm to find Articulation Points: $O(V \times (V + E)) = O(V^2 + VE)$

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_i 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:

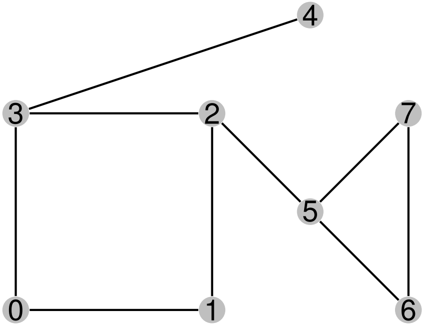
- **dfs_num[i]**: visitation order from DFS;
- **dfs_low[i]**: lowest dfs_num reachable from v_i;

For neighbors u, v, if low[v] >= num[u], then u is an articulation node (except root)

For neighbors u, v, if low[v] > num[u], $e_{u,v}$ is a bridge; (articulation edge)

Tarjan's Algorithm for Articulation Point

Simulation



First, use DFS to calculate dfs_num and dfs_low
Then compare neighbors to check articulation node/edge.

- dfs_num: 0; dfs_low: 0
- dfs_num: 1; dfs_low: 0
- dfs_num: 2; dfs_low: 0
- dfs_num: 3; dfs_low: 0
- dfs_num: 4; dfs_low: 4
- dfs_num: 5; dfs_low: 5
- dfs_num: 6; dfs_low: 5
- dfs_num: 7; dfs_low: 5

Tarjan's Algorithm for Articulation Point

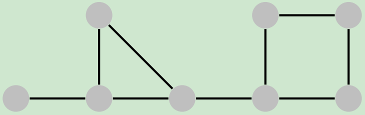
```
void articulation(u){
   dfs_num[u] = dfs_low[u] = IterationCounter++; // update num[u], init low[u]
   for (int i = 0; i < AdjList[u].size(); i++){  // Do DFS on each edge from u
      v = AdjList[u][i];
      if (dfs_num[v.first] == UNVISITED) {        // DFS tree edge
         dfs_parent[v.first]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

         // After we finish the DFS from u, we check if u is articulation.
         if (dfs_low[v.first] >= dfs_num[u])
            articulation_vertex[u] = true;        // u is articulation
         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]);
}  }
```

Strongly Connected Components

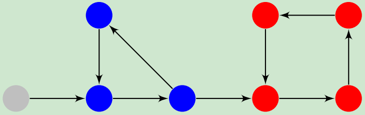
Definition

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 SCC (directed)



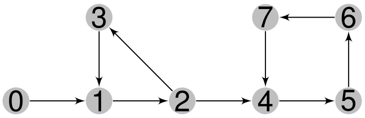
Algorithm for Finding SCCs

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_low[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

Do this simulation yourself!



SCC Stack:

	0	1	2	3	4	5	6	7
dfs_low								
dfs_num								

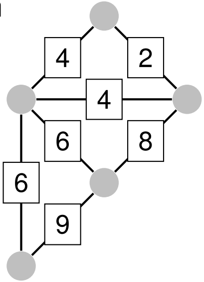
Part 4: Minimum Spanning Tree

Minimum Spanning Trees (MST) – Definition

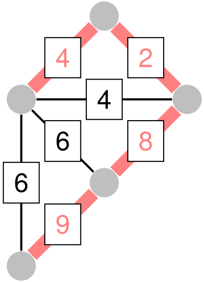
A **Spanning Tree** is a subset E' from graph G so that all vertices are connected without cycles.

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

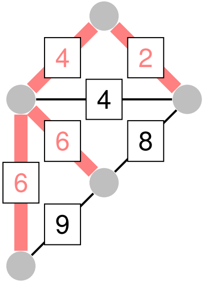
Graph



Spanning Tree



Minimum Spanning Tree



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:

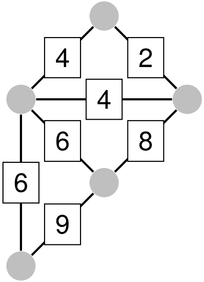
- **Kruskal Algorithm**: based on edge list;
- **Prim's Algorithm**: based on vertex list;

Kruskal's Algorithm

Outline

Kruskal's algorithms 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;

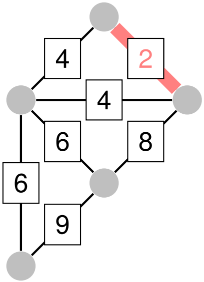


Kruskal's Algorithm

Outline

Kruskal's algorithms 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;

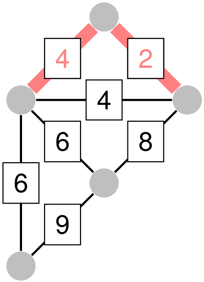


Kruskal's Algorithm

Outline

Kruskal's algorithms 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;

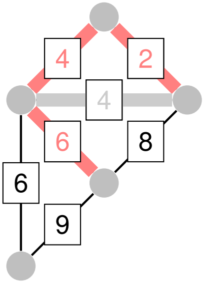


Kruskal's Algorithm

Outline

Kruskal's algorithms 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;

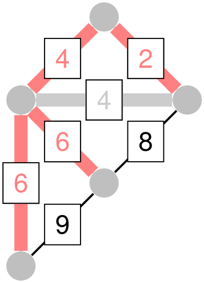


Kruskal's Algorithm

Outline

Kruskal's algorithms 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

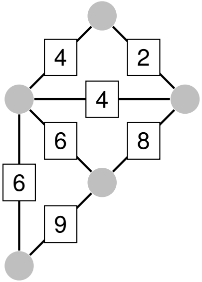
```
vector<pair<int, pair<int,int>> Edgelist;
sort(Edgelist.begin(),Edgelist.end());
int mst_cost = 0;
UnionFind UF(V);
  // note 1: Pair object has built-in comparison;
  // note 2: Need to implement UnionSet class;

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 algorith 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 leades to a new node, add it to MST;
5. Add new edges to Queue;
6. Go to 3;

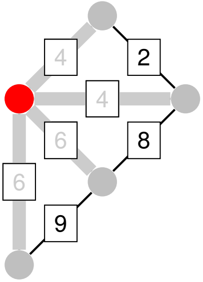


Prim's Algorithm

Outline

Prim's algorith 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 leades to a new node, add it to MST;
5. Add new edges to Queue;
6. Go to 3;

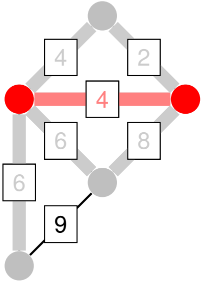


Prim's Algorithm

Outline

Prim's algorith 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 leades to a new node, add it to MST;
5. Add new edges to Queue;
6. Go to 3;

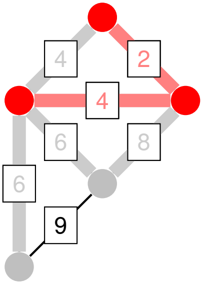


Prim's Algorithm

Outline

Prim's algorith 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 leades to a new node, add it to MST;
5. Add new edges to Queue;
6. Go to 3;

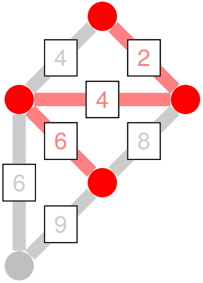


Prim's Algorithm

Outline

Prim's algorith 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 leades to a new node, add it to MST;
5. Add new edges to Queue;
6. Go to 3;

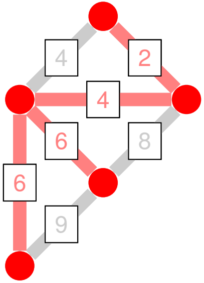


Prim's Algorithm

Outline

Prim's algorith 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 leades to a new node, add it to MST;
5. Add new edges to Queue;
6. Go to 3;



Prim's Algorithm – Implementation

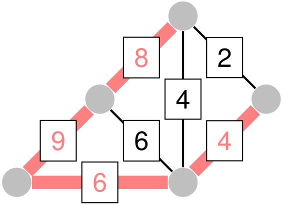
```
vector <int> taken; priority_queue <pair <int,int>> pq;

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.first, ve.second))
}}
taken.assign(V,0); process(0);
mst_cost = 0;

while (!pq.empty()) {
  vector <int,int> pq.top(); pq.pop();
  u = front.first, w = front.second;
  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:

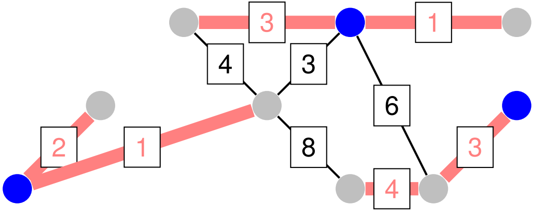
- **Kruskal**: Reverse the sort of the edge list;
- **Prim**: Invert the weight of the priority queue;



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

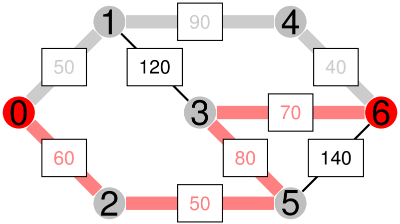
Suppose that you are required to calculate an alternative solution to an MST problem. In this case, you need to find the second cheapest spanning tree.

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



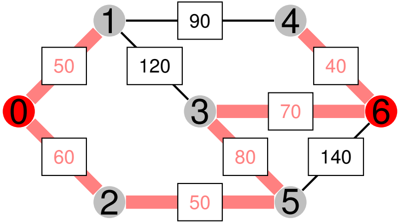
Problem Definition

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!

About these Slides

These slides were made by Claus Aranha, 2022. You are welcome to copy, distribute, re-use and modify this material. (CC-BY-4.0)

Individual images in some slides might have been made by other authors. Please see the following pages for details.

Image Credits I