GB21802 - Programming Challenges

Week 4 - Graph Problems (Part I)

Claus Aranha
caranha@cs.tsukuba.ac.jp

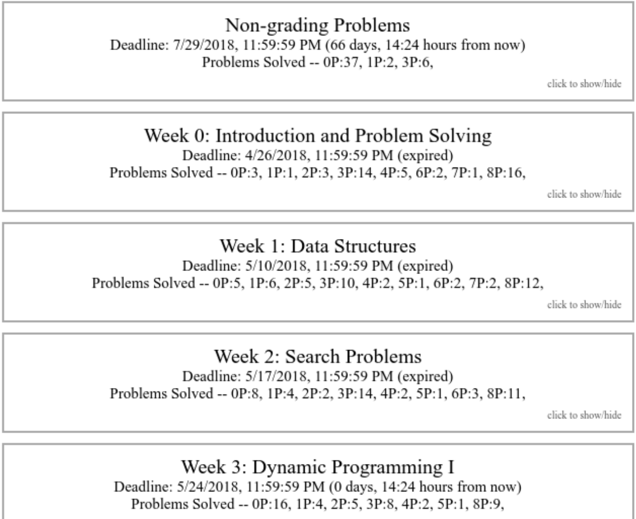
College of Information Science

2018-05-25,28

Last updated May 24, 2018

Results for the Previous Week

Here are the results for last week:



Week 4 and 5 – Outline

This Week - Graph I

- Graph Basics review: Concepts and Data Structure;
- Depth First Search and Breadth First Search;
- Problems you solve with DFS and BFS;
- Minimum Spanning Tree: Kruskal and Prim Algorithms;

Next Week - Graph II

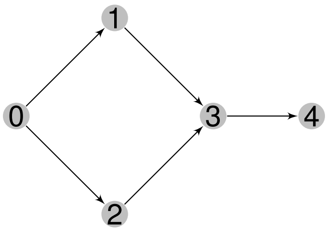
- Single Sourse Shortest Path (Djikstra);
- All Pairs Shortest Path (Floyd Warshall);
- Network Flow and related Problems;
- Bipartite Graph Matching and related Problems;

Motivation Problem – UVA 11902: Dominator

Problem Summary

We define that vertex X dominates vertex Y if every path from a start vertex 0 to Y must go through X.

For all Vertices, determine which nodes dominate which other.



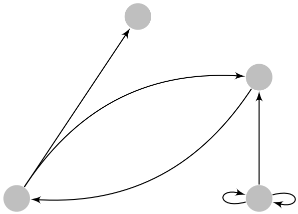
- 0 dominates all nodes;
- 3 dominates 4;
- 1 does not dominate 3;

How do you solve it?

Base definition

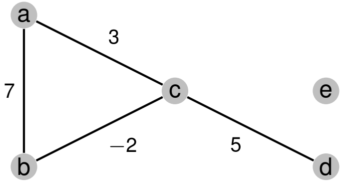
A graph $G = V, E$ is defined as a set of vertices V, and edges E. Each edge connects exactly two vertices.

- Edges can be directed or undirected;
- Edges and can be weighted (and sometimes vertices too!);
- Edges can be self-edges, and/or multiple edges



Graph Problems:

- Find a vertex or edge with a certain characteristic;
- Find a relationship between vertices or edges;



How do we implement a graph?

Adjacency Matrix - Stores connection between Vertices

```
int adj[100][100];
// adj[i][j] is 0 or -1 if no edge between i,j
// adj[i][j] is A if edge of weight A links i,j
```

- Pro: Very simple to program and use;
- Con:
  - Cannot store multigraph;
  - Waste space for sparse graphs;
  - Time $O(V)$ to calculate number of neighbors;

How do we implement a graph?

Vector-Edge List – Stores Edges list for each Vertex

```
typedef pair<int,int> edge; // <destination, weight> pair
typedef vector<edge> neighb;// all the neighbors of one V
vector<neighb> AdjList;     // all vertices

AdjList[2];                 // Adjency list of vertex 2

vector<double> Vcost;       // Array with Vertex weights
```

- Pro: $O(V + E)$ space, efficient if graph is sparse; Can store multigraph;
- Con: Code is more complex than adjacency list; $O(\log V)$ to test if two vertices are adjacent.

If you need special Vertex data, you can save it in a separate array.

Data Structures for Graphs (2)

Edge List – stores only the edges

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

This is a very unusual structure! It is used by Kruskal's Algorithm

- Some algorithms require specialized data structures.
- Use arrays to save data about vertices.

Searching in a Graph: BFS and DFS

- Almost every Graph algorithm uses BFS or DFS;
- Learn to do this with your eyes closed;
- Of course, some problems require special implementation;

Depth First Search – DFS

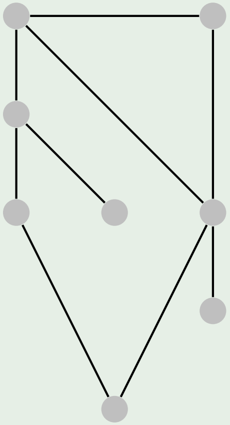
- Easy to implement with Recursion;
- Visit first edge of each vertice until you find a loop;

Breadth First Search – BFS

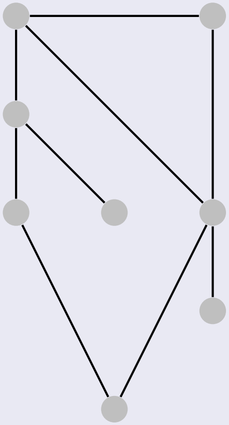
- Easy to implement on a loop;
- Put every edge on a FIFO queue, then visit next vertex;

BFS/DFS: Visualization

DFS

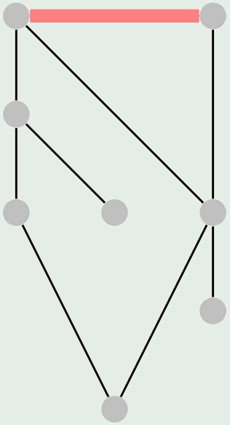


BFS

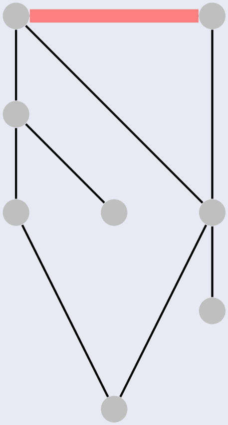


BFS/DFS: Visualization

DFS

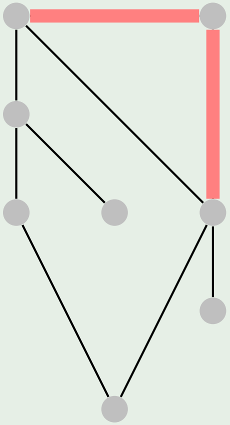


BFS

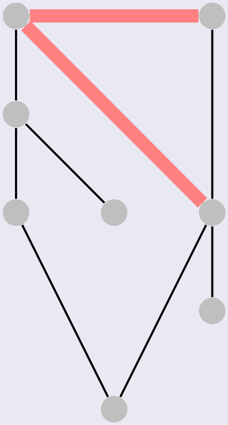


BFS/DFS: Visualization

DFS

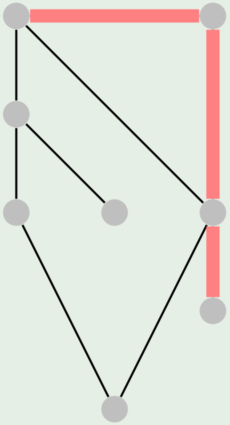


BFS

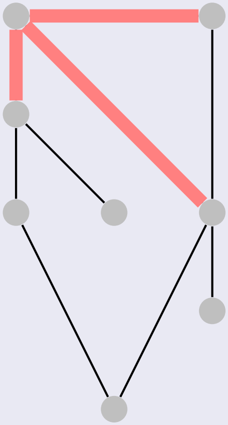


BFS/DFS: Visualization

DFS

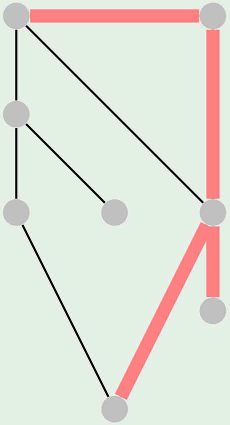


BFS

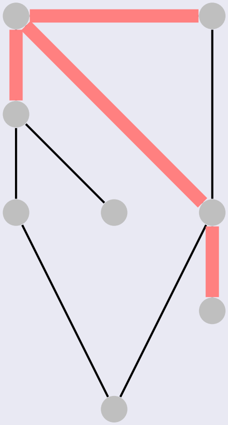


BFS/DFS: Visualization

DFS

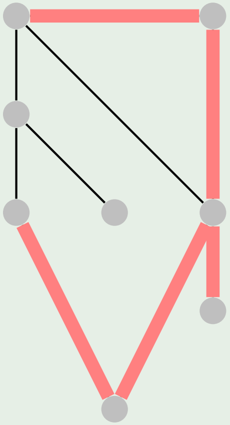


BFS

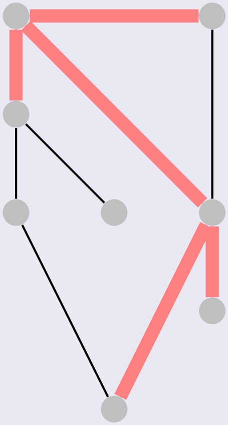


BFS/DFS: Visualization

DFS

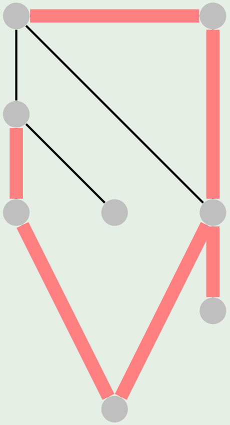


BFS

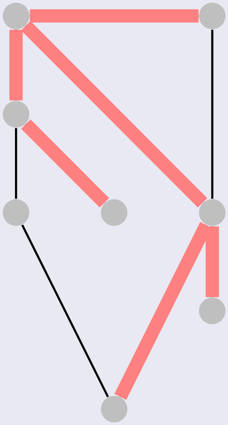


BFS/DFS: Visualization

DFS

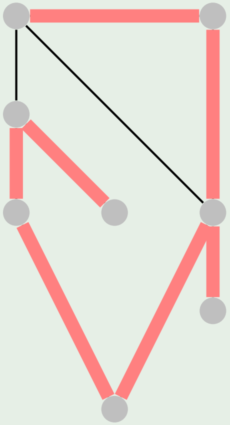


BFS

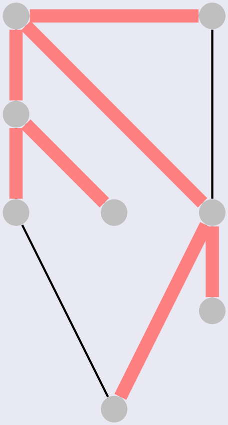


BFS/DFS: Visualization

DFS



BFS



BFS/DFS: Implementation

Implementing DFS with an adjacency list:

DFS

```
#define UNVISITED 0
#define VISITED 1

vector<int> dfs_vis; // list visited nodes

void dfs(int v) {
   dfs_vis[v] = VISITED;
   for (int i; i < (int) Adj_list[v].size(); i++) {
      pair <int,int> u = Adj_list[v][i];
      if (dfs_vis[u.first] == UNVISITED)
         dfs(v.first);
      // else -- Found Loop!
   }
}

dfs(1);
```

BFS/DFS: Implementation

Implementing BFS with an adjacency matrix:
Note that this implementation is $O(N^2)$!

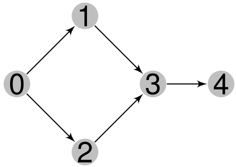
BFS

```
#define INF 1000000
int s = 1;                              // Start Vertice

vector<int> d(N,INF); d[s] = 0;   // Distance array
queue<int> q; q.push(s);

while(!q.empty()) {
  int u = q.front(); q.pop();
  for (int i=0; i < N; i++) {
    if (adj[u][i] && d[i] == INF) {
      d[i] = d[u] + 1;              // Visiting node
      q.push(i);
    }
  }
}
```

Back to the Dominator Problem

Thinking about the problem again

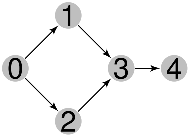
X is Dominated by Y if all paths from the root to Y go through X.



Back to the Dominator Problem

N^2 solution: DFS N times

```
def DFS2(S,P): ...  // Modified DFS: never visits P;

DFS2(0,-1)
Mark every VISITED vertex as DOMINATED by 0

for i in (1:N):
   reset VISITED array
   DFS2(0,i)
   For every NOT-VISITED vertex j:
      if j is DOMINATED by 0:
         j is DOMINATED by i
```



Common Algorithms Using DFS/BFS

With small modifications to BFS/DFS, we can solve many simple problems

Problem Example: Extra cables

You own a network of N computers;

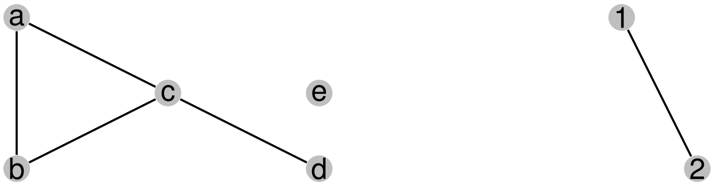
You have a list of every pair of computers connected by cable (i, j)

You need to buy M extra cables to connect ALL computers. How many cables do you need?

Common Algorithms Using DFS/BFS

Connected Components

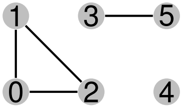
This problem is defined in graph theory as finding Connected Components.



Common Algorithms Using DFS/BFS
Connected Components

One way to find all connected components, is to repeat a BFS (or DFS) until all nodes are visited.

```
int cables = 0;
// dfs(x) from slide 11

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



Common Algorithms Using DFS/BFS

Connected Components

We can also find the connected components using UFDS.

How would you implement it?

More Common Algorithms Using DFS/BFS

Is this a graph problem?

Problem: The Biggest Island

You want to build a new castle in the game of CraftMine. For this, you need to find the biggest island in the world map.

Input: The following 2D map of the map:

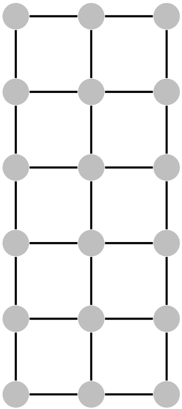
```
....................................
.###.......###.....#.....###.####...
.#####....#####.##.#####.##....#....
.###........###..#...##..#....###...
......###.......###...####...##.....
....####.............######.....###.
....####.......#.......###......###.
....................................
```

Is this even a graph???

More Common Algorithms Using DFS/BFS

Implicit Graphs and Floodfill

- Implict Graphs suggest graph organization;
- But it is not necessary to store the vertices and edges;
- Example: Game boards; Distances; Grids;
- No graph Structure, but same graph algorithms;



Flood Fill

We can solve the "Biggest Island" problem with Flood Fill, which is just a variation of BFS/DFS for grids.

```
int dr[] = {1,1,0,-1,-1,-1,0,1}; // trick to explore an
int dc[] = {0,1,1,1,0,-1,-1,-1}; // implicit NESW graph

int floodfill(int y, int x) {
  if (y < 0 || y >= R || x < 0 || x >= C) return 0;
  if (grid[y][x] != '#') return 0;
  int ans = 1;
  grid[y][x] = '.';                    // CHANGE the map
  for (int d = 0; d < 8; d++)
     ans += floodfill(y+dr[d], x+dc[d]);
  return ans;
}
```

Another Classical Problem

Preparing a Course Curriculum

Problem Outline

You are a teacher and you have to prepare a course curriculum. You have a list of topics, but some topics are pre-requisite to others.

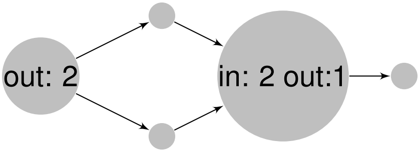
Input: M, N, The list of M topics, followed by N ordered pairs of topics.
Output: A sorted list with all topics.

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


=====================
Result: Programming Search DP Graph Flow
```

Another Classical Problem

Topological sort

- A directed graph has a topological sort if it has no cycles;
- We can use the in-degrees and out-degrees to calculate the topological sort;



Topological Sort (Directed Acyclic Graphs)

Khan's algorithm for Topological sort (modified edge-BFS)

```
Q = queue(); toposort = list();
for j in edge:
   in_degree[j.destination] += 1
for i in node:                         // Start nodes to queue
   if in_degree[i] == 0: Q.add(i);
while (Q.size() > 0):
   u = Q.dequeue(); toposort.add(u);
   for i in u.out_edges():             // reduce in-degree
       v = i.destination
       in_degree[v] =- 1
       if in_degree[v] == 0:           // new start node
          Q.add(v);
```

Topological Sort and Bottom-Up Dynamic Programming

What is the relationship between Topological Sort and Bottom-up DP?

Bottom-up DP are Topological sorts on Tables!

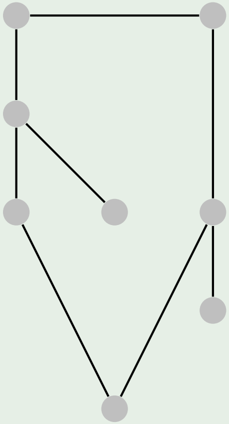
Bipartite Check

To check whether a graph is bipartite, we perform a BFS or DFS on the graph, and set the color of every node to black or white, alternatively. Pay attention to collision conditions.

```
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 (int j=0; j < adj_list[u].size(); j++) {
      pair<int,int> v = adj_list[u][j];
      if (color[v] == INF) {
         color[v.first] = 1 - color[i];
         q.push(v.first);}
      else if (color[v.first] == color[u]) {
         isBipartite = False;
}}}
```

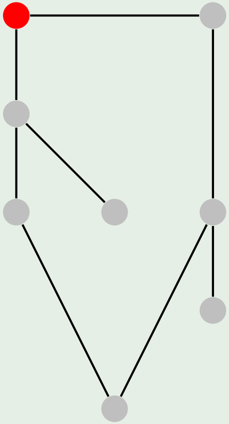
Bipartite Check – Visualization

Testing Bipartite property



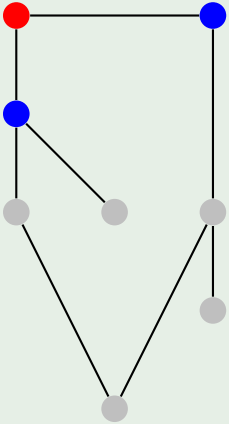
Bipartite Check – Visualization

Testing Bipartite property



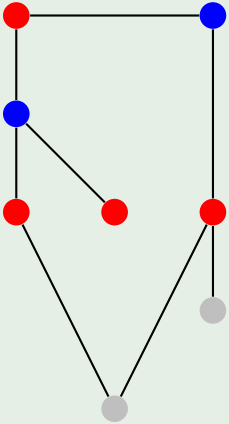
Bipartite Check – Visualization

Testing Bipartite property



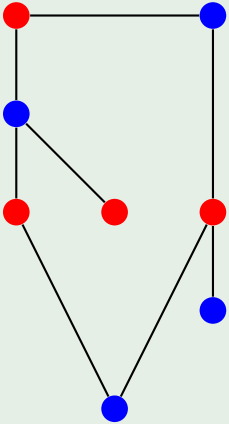
Bipartite Check – Visualization

Testing Bipartite property



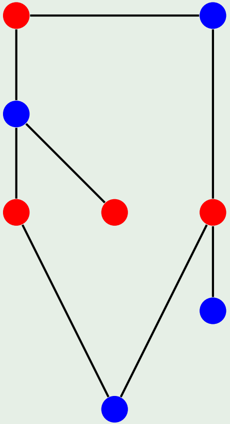
Bipartite Check – Visualization

Testing Bipartite property

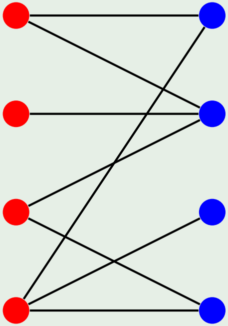


Bipartite Check – Visualization

Testing Bipartite property



Rearranging the nodes

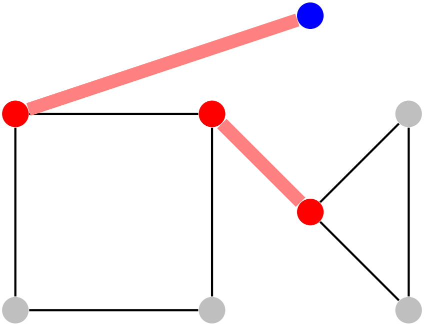


Articulation Points and Bridges

Problem Description

In an undirected graph G:

- A verted V is an Articulation Point if removing V would make G disconnected.
- An edge E is a Bridge if removing E would make G disconnected.



Articulation Points and Bridges: Algorithm

Complete Search algorithm for Articulation Points

1. Run DFS/BFS, and count the number of CC in the graph;
2. For each vertex v, remove v and run DFS/BFS again;
3. If the number of CC increases, v is a connection point;

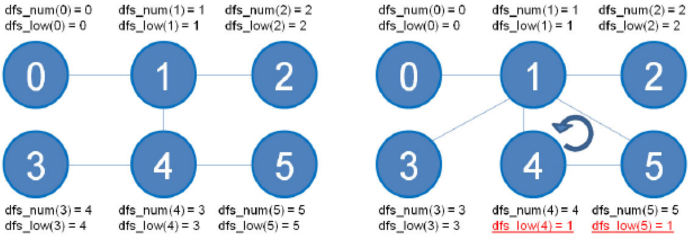
Since DFS/BFS is $O(V + E)$, this algorithm runs in $O(V^2 + EV)$.

... but we can do better!

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

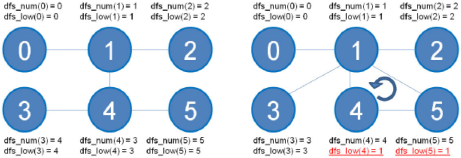
Tarjan Variant: $O(V + E)$

Main idea: Add extra data to the DFS to detect articulations.



- dfs_num[]: Recieves the number of the iteration when this node was reached for the first time;
- dfs_low[]: Recieves the lowest dfs_num[] which can be reached if we start the DFS from here;
- For any neighbors u, v, if dfs_low[v] >= dfs_num[u], then u is an articulation node.

Tarjan's DFS variant for Articulation point (2)



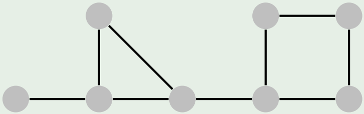

```
void dfs_a(u){
   dfs_num[u] = dfs_low[u] = IterationCounter++;   // dfs_num[u] is a simple counter
   for (int i = 0; i < AdjList[u].size(); i++){
      v = AdjList[u][i];
      if (dfs_num[v] == UNVISITED) {
         dfs_parent[v] = u;                        // store parent
         if (u == 0) rootChildren++:               // special case for root node

         dfs_a(v);
         if (dfs_low[a] >= dfs_num[u])
            articulation_vertex[u] = true;
         dfs_low[u] = min(dfs_low[u],dfs_low[v])

      else if (v != dfs_parent[u])                 // found a cycle edge
         dfs_low[u] = min(dfs_low[u],dfs_num[v])
}}
```

Strongly Connected Components (Directed Graph)

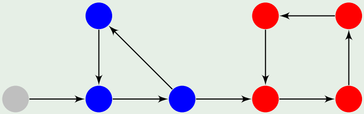
Problem Description

On a directed graph G, a Strongly Connected Component (SCC) is a subset G' where for every pair of nodes $a, b \in G'$, there is both a path $a \to b$ and a path $b \to a$.

One CC



Three SCC



Strongly Connected Components – Algorithm

We can use a simple modification of the algorithm for bridges and articulation points:

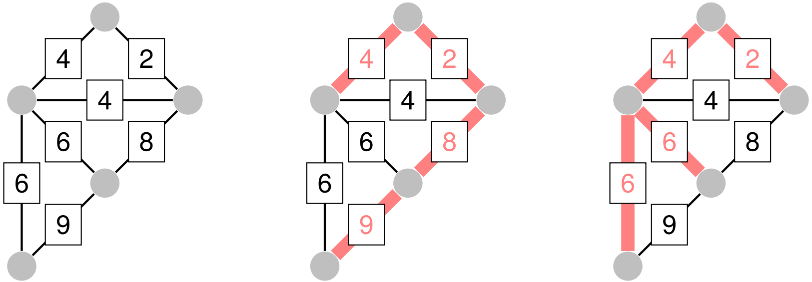
- Every time we visit a new node, put that node in a stack S;
- When we finish visiting a node i, test if dfs_num[i] == dfs_min[i].
- If the above condition is true, i is the root of the SCC. Pop all vertices in the stack as part of the SCC.

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.



MST – Use cases and Algorithms

Problems using MST

Problems using MST usually involve calculating the minimum costs of infrastructure such as roads or networks.

Some variations may require you to find the maximum spanning tree, or define some edges that must be taken in advance.

Algorithms for MST

The two main algorithms for calculating the MST are the Kruskal's algorithms and the Prim's algorithms.

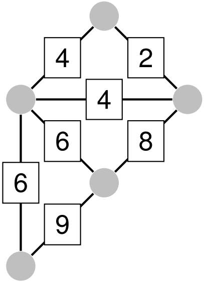
Both are greedy algorithms that add edges to the MST in weight order.

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;

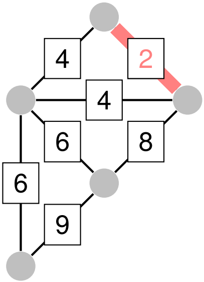


Kruskal's Algorithm

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;

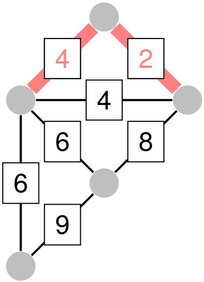


Kruskal's Algorithm

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;

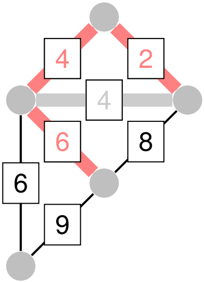


Kruskal's Algorithm

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;

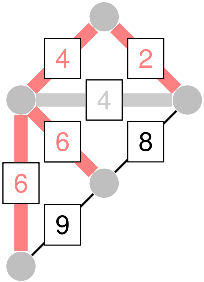


Kruskal's Algorithm

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

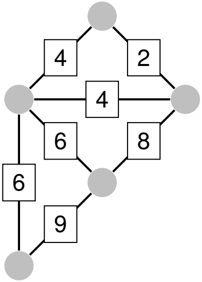
```
vector<pair<int, pair<int,int>> Edgelist;
sort(Edgelist.begin(),Edgelist.end());
int mst_cost = 0;
UnionFind UF(V);
  // note 1: Pair object has built-in comparison;
  // note 2: Need to implement UnionSet class;

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;

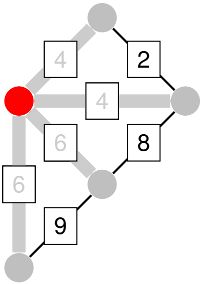


Prim's Algorithm

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;

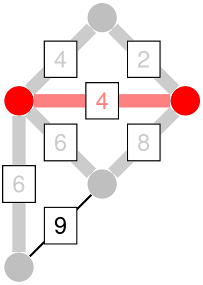


Prim's Algorithm

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;

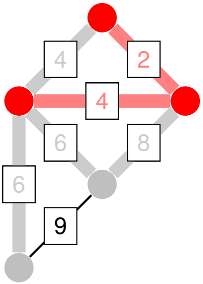


Prim's Algorithm

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;

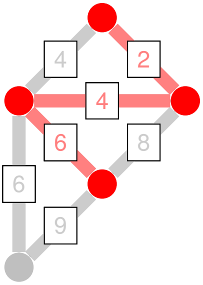


Prim's Algorithm

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;

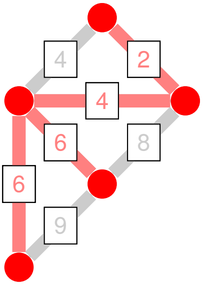


Prim's Algorithm

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;



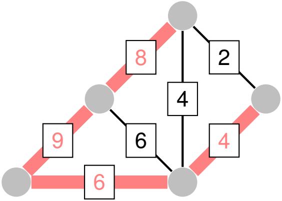
Prim's Algorithm – Implementation

```
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.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.

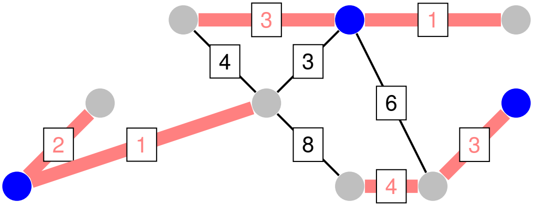
It is very easy to implement the Maximum MST by reversing the sort order of the edges (Kruskal), or the weighting of the priority Queue (Prim).



MST variant 2 – Minimum Spanning Subgraph, Forest

In one importante variant of the MST, 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;
- What if you are given a number of Connected Components?



MST Variant 3 – *n*th Best MST

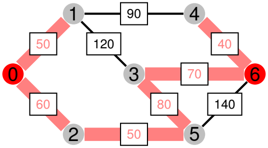
Problem Definition

Consider that you can order MST by their costs: $G_1, G_2, \ldots, G_n$. This variant asks you to calculate the n^{th} best spanning tree.

Basic Idea:

- Calculate the MST (using Kruskal or Prim);
- For every edge in the MST, remove that edge from the graph and calculate a new MST;
- The new MST with minimum weight is the 2^{nd} best MST;

MST Variant 4 – Min-max (or Max-min)



Problem Definition

Given two vertices i, j, find a path $i \to j$ so that the cost of the most expensive edge is minimized;

Another way to write this problem is: Find the cheapest path where the cost of the path is the cost of the most expensive edge.

How to solve

The MST finds the path that connects all nodes while keeping the cost of individual edges minimal.

To solve the minimax problem, we calculate the MST for G, and then find the path from i to j in the MST.

Class Summary

What did we learn?

- Graphs come in a wide variety of types;
- Graph problems also have many different types;
- Most problems involve small modifications of DFS and BFS;

Hints on solving Problems

Keep a code Library

Graph problems/algorithms use a lot of common code. Store a template, and copy/paste as necessary!

- Structures for keeping visited flags and vertex distances;
- Structures for keeping parent and children lists;
- Adjacency lists;

Common tricky cases – Be careful!

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

Next Week

More Graphs! Specially path finding

- Shortest Paths (Single Source and All Pairs);
- Network Flow (and related problems);
- Graph Matching (bipartite matching, etc) (and related problems);

This Week's Problems

- Dominator
- Forwarding Emails
- Ordering
- Place the Guards
- Doves and Bombs
- Come and Go
- ACM Contest and Blackout
- Ancient Messages

Problem Hints (1)

Dominator

- If All paths from 0 to node B pass through node A, then node A **dominates** node B;
- For all pair of nodes i, j, output "Y" if i **dominates** j, or "N" if not;

The idea of this problem is one of "reachability" – can I reach node j if I remove node i from the graph?

Note: if j is not connected to "0", then *no one dominates j*

Problem Hints (2)

Forwarding Emails

Every person *i* sends e-mail only to person *j*.

What is the longest email chain?

Where does it start?

- How do you deal with loops?
- Time limit is not very large, Try to find an O(n) solution!

Problem Hints (3)

Ordering

Print all possible Orderings of a Direct Acyclic Graph

Generalize the DAG ordering algorithm which we discussed in class.

Palace Guards

- How do you represent the roads and junctions as a Graph?
- Find a “guard-no guard” assignment to vertices of the graph.
- First test if a solution is possible!

Problem Hints (4)

Doves and Bombs

This problem is about finding “critical vertices” in a graph. But how do you calculate the “pigeon value” of a vertex?

Come and Go

Straight implementation of “Strong Connected Components”. Be careful with tricky graphs!

Problem Hints (5)

ACM Contest and Blackout

Goal: Find the **First** minimum spanning Tree and the **Second** minimum spanning Tree

- In this class we discussed how to find the Minimum Spanning Tree
- How would we find the **second minimum?**
- Idea: Maybe if we remove some edges from the graph?

Problem Hints (6)

Ancient Message – Challenge problem!

Count the symbols inside an image – order does not matter!

What is the **Main** difference between the symbols?

- The shape and size of the symbols is actually not important!
- Before you begin programming, discover what is the real difference between the symbols.
- Hint: The numbers “1”, “0”, “8” have the same difference.