GB21802 - Programming Challenges

Week 5 - Graph Problems (Part I)

Claus Aranha
caranha@cs.tsukuba.ac.jp

College of Information Science

2015-05-27,30

Last updated May 27, 2016

Last Week Results

Week 3 - DP I

- Jill Rides Again - 15/32
- Maximum Sum - 4/32
- SDI (rockets) - 8/32
- Is Bigger Smarter - 12/32
- Ferry Loading - 2/32
- Unidirectional TSP - 5/32
- Flight Planner 3/32
- e-Coins 3/32

Week 4 - DP II (At Deadline)

- Collecting Beepers - 9/32
- Shopping Trip - 1/32
- Bar Codes - 7/32
- Cutting Sticks - 4/32
- String Popping - 2/32
- Divisibility - 3/32
- Marks Distribution - 8/32
- Squares - 3/32

Special Notes

Week 5 and 6 – 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 (Monday);

Next Week - Graph II

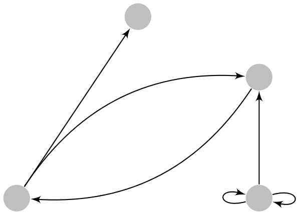
- Single Sourse Shortest Path (Djikstra);
- All Pairs Shortest Path (Floyd Warshall);
- Network Flow and related Problems;
- Bipartite Graph Matching and related Problems;

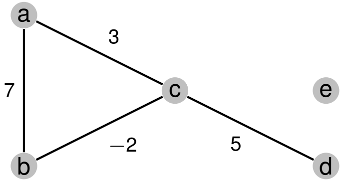
Many variations in graph problems!

Quick Review of Graph Terms (1)

You probably know all of these. If not, ask questions!

- A Graph G is made of a set of vertices V and edges E.
- Edges can be directed (has source and destination vertices);
- Edges can be weighted or not (all weigths = 1);
- Sets of nodes can be connected or disconnected
- Directed Graphs can be Strongly Connected
- Edges can be self-edges, and/or multiple edges

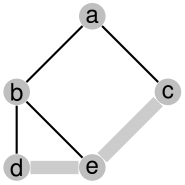


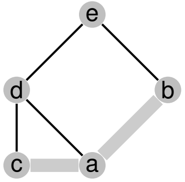


Quick Review of Graph Terms (2)

You probably know all of these. If not, ask questions!

- A path is a set of vertices connected by edges;
- A cycle is a path with first and last vertices identical;
- Labelled graphs and Isomorphic graphs;
- A tree is a acyclical, undirected graph;
- A spanning tree is a subset of edges from E' that form a tree, connecting all nodes $V \in G$;
- A spamming tree houses very noisy insects in summer;

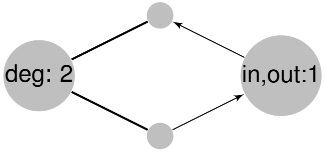


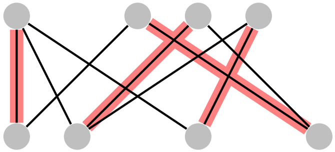


Quick Review of Graph Terms (3)

You probably know all of these. If not, ask questions!

- The degree of a node is the number of edges connected to it;
- Directed graphs have in-degrees and out-degrees;
- A bipartite graph can be divided in two sets of unconnected vertices;
- A Match or Pairing is a set of edges that connects the nodes in the bipartite graph;





Data Structures for Graphs (1)

Adjacency Matrix - Stores connection between Vertices

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

- Pro: Very simple to program, manipulate;
- Con: Cannot store multigraph; Wastes space for sparse graphs; Requires time $O(V)$ to calculate number of neighbors;

Edge List – Stores Edges list for each Vertex

```
typedef pair<int,int> ii;
typedef vector<ii> vii;
vector<vii> AdjList;
```

- Pro: $O(V + E)$ space, efficient if graph is sparse; Can store multigraph;
- Con: A (bit) more code than Adjacency Matrix

Data Structures for Graphs (2)

Edge List

```
vector< pair <int,ii>> Edgelist;
```

Stores a list of all the edges in the graph. Vertices are implicit from the edge list. This is useful for Kruskal's algorithm (which we will see later), but otherwise complicates things.

Implicit Graph

Some graphs do not need to be stored in a special structure if they have very clear rules about when two vertices connect.

Examples:

- A square grid;
- Knight's chess moves;
- Two vertices i, j connect if $i + j$ is prime;

Searching in a Graph: BFS and DFS

Almost all graph problems involve visiting each of its vertices in some form. There are two approaches for visiting the nodes in a graph:

Depth First Search – DFS

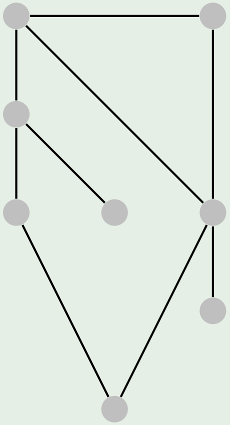
DFS is commonly implemented as a recursive search. For every node visited, immediately visit the first edge in it, backtracking when a loop is reached, or no more edges can be followed.

Breadth First Search – BFS

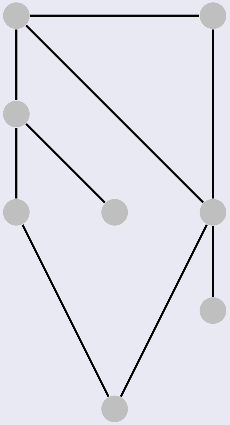
BFS is commonly implemented iterating over a FIFO queue. For every node visited, all new edges are put on the back of the queue. Visit the next edge at the top of the queue.

BFS/DFS: Visualization

DFS

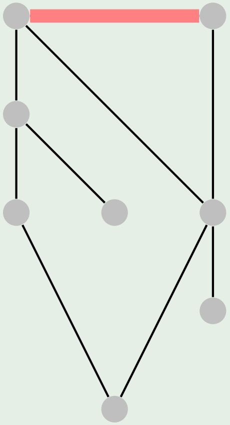


BFS

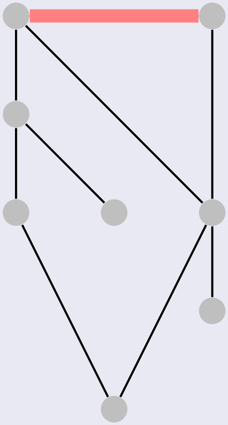


BFS/DFS: Visualization

DFS

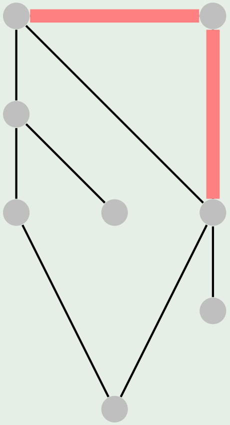


BFS

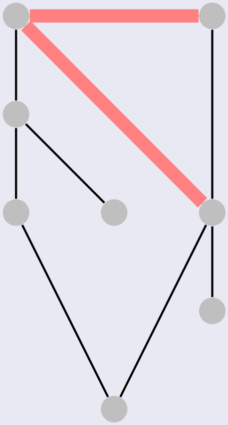


BFS/DFS: Visualization

DFS

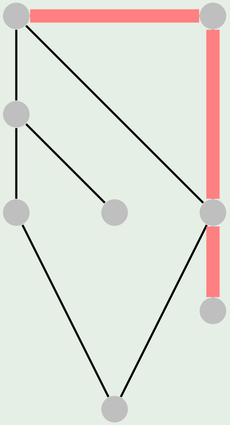


BFS

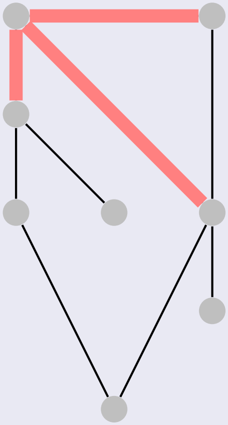


BFS/DFS: Visualization

DFS

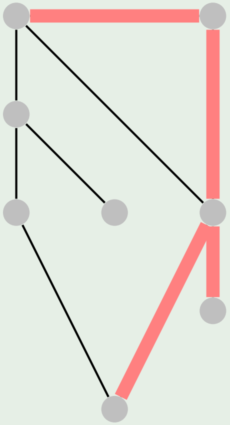


BFS

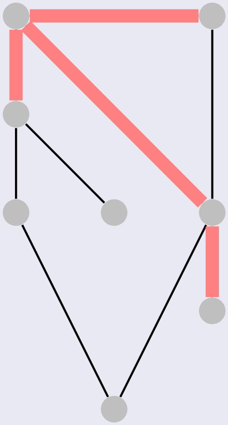


BFS/DFS: Visualization

DFS

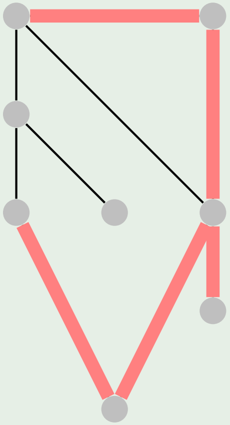


BFS

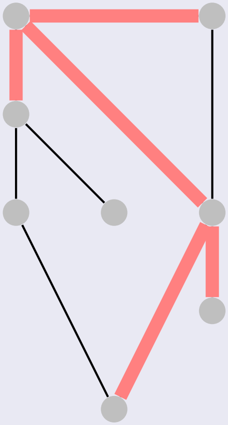


BFS/DFS: Visualization

DFS

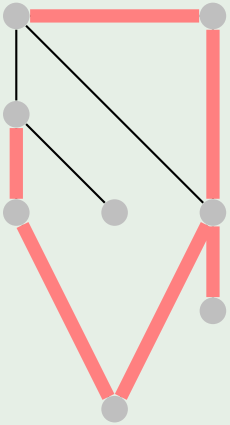


BFS

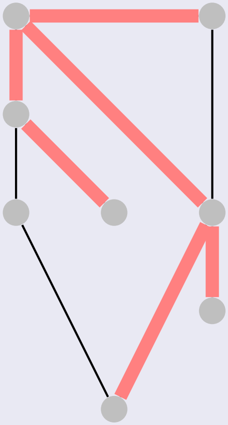


BFS/DFS: Visualization

DFS

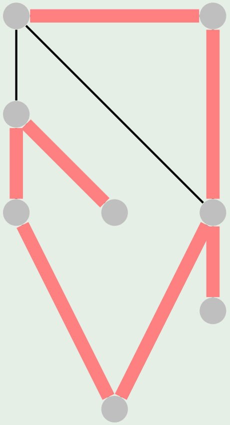


BFS

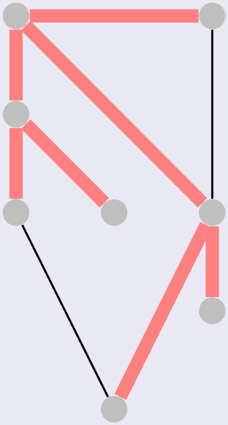


BFS/DFS: Visualization

DFS



BFS



BFS/DFS: Implementation

There are many ways to implement BFS/DFS, here is a suggestion.

DFS

```
vector<int> dfs_vis; // initially all set to UNVISITED
void dfs(int v) {
   dfs_vis = VISITED;
   for (int i; i < (int)Adj_list[v].size(); i++) {
      pair <int,int> u = Adj_list[u][i];
      if (dfs_vis[u.first] == UNVISITED) dfs(v.first)
}}
```

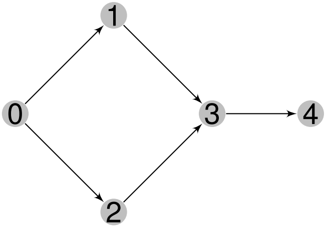
BFS

```
vector<int> d(V,INF); d[s] = 0; queue<int> q; q.push(s);
while(!q.empty()) {
   u = q.front(); q.pop();
   for (int i=0; i < (int)Adj_list[q].size(); i++) {
   pair <int,int> v = Adj_list[u][i]; //same as dfs
   if (d[v.first] == INF) {
      d[v.first] = d[u] + 1; q.push(v.first);
}}}
```

Simple BFS/DFS – UVA 11902: Dominator

Problem Summary

Vertex X dominates vertex Y if every path from a start vertex 0 to Y must go through X. Determine which nodes dominate which other.



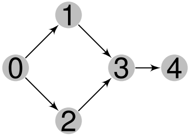
- 0 dominates all nodes;
- 3 dominates 4;
- 1 does not dominate 3;

How do you solve it?

Simple BFS/DFS – UVA 11902: Dominator

Solution

```
DFS(0);
for i in (0:N):
   if i is reached: dominate[0][i] = 1;

for i in (1:N):
   remove i from graph;
   DFS(0)
   for j in (1:N):
       if (j is not reached) and (dominate[0][j] == 1):
           dominate[i][j] = 1
   return i to graph
```

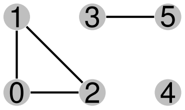


Common Algorithms: Connected Components

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

Since a single run of DFS/BFS finds all connected nodes, we can use it to find (and count) all the connected components (CC) of an undirected graph.

```
numCC = 0;
dfs_num.assign(V,UNVISITED);
for (int = 0; i < V; i++)
   if (dfs_num[i] == UNVISITED)
      cout << "\nCC " << ++numCC << ":"; dfs(i);
      // modify dfs() to print every node it visits
```




```
CC 1: 0 1 2
CC 2: 3 5
CC 3: 4
```

Flood Fill

A simple twear of the BFS (or DFS) can be used to label/color and count the size of each CC.

“flood fill“ is often used in problems involving implicit 2D grids.

```
####..#
#.###.#
#..@.##
##d.###
#..####
```

```
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, char c1, char c2) {
  if (y < 0 || y >= R || x < 0 || x >= C) return 0;
  if (grid[y][x] != c1) return 0;
  int ans = 1;
  grid[y][x] = c2;
  for (int d = 0; d < 8; d++)
     ans += floodfill(y+dr[d], x+dc[d], c1, c2);
  return ans;
}
```

Topological Sort (Directed Acyclic Graphs)

A Topological sort is a linear ordering of vertices of a DAG so that vertex u comes before vertex v if edge $u \rightarrow v$ exits in the DAG. Topological Sorts are useful for problems involving the ordering of pre-requisites.

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:
   if in_degree[i] == 0: Q.add(i);
while (Q.size() > 0):
   u = Q.dequeue(); toposort.add(u);
   for i in u.out_edges():
       v = i.destination
       in_degree[v] =- 1
       if in_degree[v] == 0:
          Q.add(v);
```

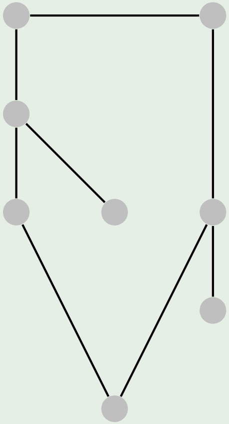
Bipartite Check

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;
}}}
```

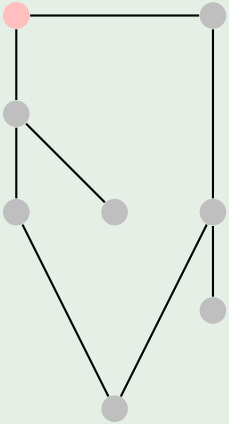
Bipartite Check – Visualization

Testing Bipartite property



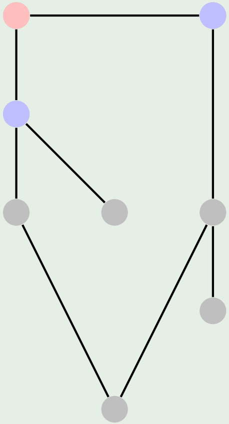
Bipartite Check – Visualization

Testing Bipartite property



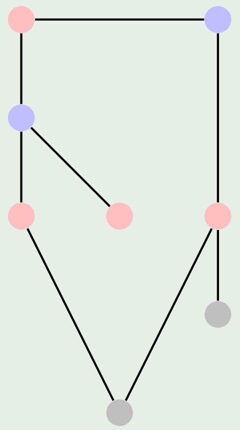
Bipartite Check – Visualization

Testing Bipartite property



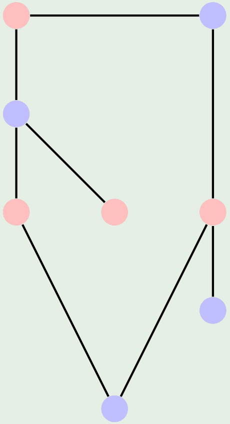
Bipartite Check – Visualization

Testing Bipartite property



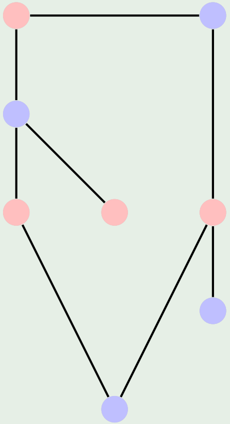
Bipartite Check – Visualization

Testing Bipartite property

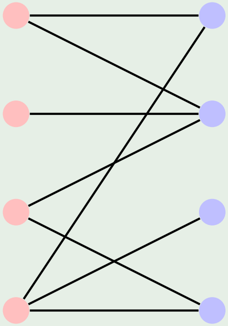


Bipartite Check – Visualization

Testing Bipartite property



Rearranging the nodes

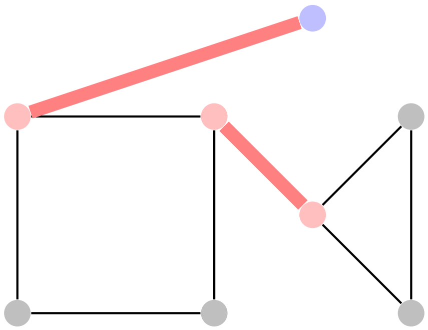


Articulation Points and Bridges

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;

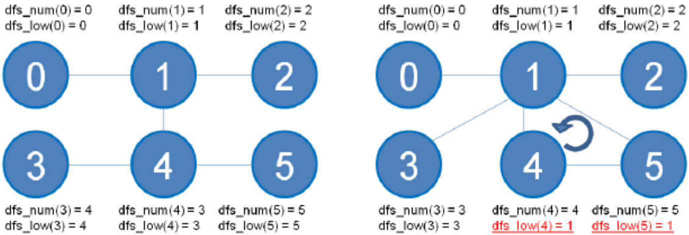
Since DFS/BFS is $O(V + E)$, this algorithm runs in $O(V^2 + EV)$.

... but we can do better!

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

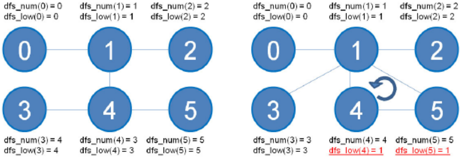
Tarjan Variant: $O(V+E)$

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)



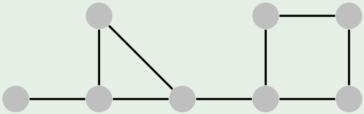

```
void dfs_a(u){
   dfs_num[u] = dfs_low[u] = IterationCounter++;   // dfs_num[u] is a simple counter
   for (int i = 0; i < AdjList[u].size(); i++){
      v = AdjList[u][i];
      if (dfs_num[v] == UNVISITED) {
         dfs_parent[v] = u;                        // store parent
         if (u == 0)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

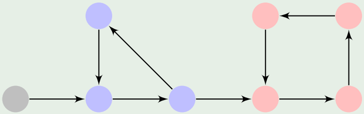
Problem Description

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 $a \rightarrow b$ and a path $b \rightarrow a$.

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.

Spanning Trees

On Monday Class:

- Spanning Trees
  (DO NOT CONFUSE WITH SPAMMING TREES)
- Problem discussion and hints
- Questions (?)

Summary

This Week's Problems

- Dominator;
- Knight in a War grid;
- Wetlands in Florida;
- Battleships;
- Pick up Sticks;
- Place the Guards;
- Street Directions;
- Dominos;
- Freckles;
- Artic Network;

Next Week

More Graphs!

- Network Flow (and related problems);
- Graph Matching (bipartite matching, etc) (and related problems);