Graph Theory Crash Course II

2015
Graph Representation Review

• Edge List
• Adjacency Matrix
• Adjacency List
Edge list

• Simply make a list (or vector) of pairwise relations.

```cpp
struct Edge {
    int a, b;
};

vector<Edge> EdgeList;

// Edge from 0 to 1
Edge e1;
e1.a = 0; e1.b = 1;

// Edge from 1 to 2
Edge e2;
e2.a = 1; e2.b = 2;

EdgeList.push_back(e1);
EdgeList.push_back(e2);
```
Adjacency Matrix

- Make a table. Rows correspond to the source node, columns to the destination node.
- A 1 in row R and column C means that the edge R->C exists.

![Adjacency Matrix Diagram]

\[
\begin{pmatrix}
1 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
\end{pmatrix}
\]

Coordinates are 1-6.
Adjacency Matrix

- **Undirected graph**: if $a \rightarrow b$ exists, so does $b \rightarrow a$
  - Therefore, matrix symmetric.
- **Weighted graph**
  - May replace 1 with the edge weight.
Adjacency Matrix

```c
int Graph[10][10]; // 10 nodes, 0-9
Graph[2][1] = 1;    // Create edge 2->1
Graph[3][2] = 1;    // Create edge 3->2
```
Adjacency Lists

• Each node stores the edges that extend from that node.

• Example:
  – Node 1 stores:
    • Edge to 5
    • Edge to 1
    • Edge to 2

• For undirected graphs, we need to be careful to add the reverse edges too.
Adjacency Lists

```cpp
struct Node {
    // Store endpoint of edges from this node
    vector<int> Edges;
};

vector<Node> Graph(10); // 10 nodes, 0-9

// Insert an undirected edge between nodes 0 and 1
Graph[0].Edges.push_back(1); // Edge from 0 to 1
Graph[1].Edges.push_back(0); // Edge from 1 to 0
```
Which to Use?

• Depends on the algorithm
  – Some algorithms are more naturally implemented on a particular representation.
  – Some queries are inefficient on edge lists, e.g. is there an edge between two given nodes?

• Depends on the graph
  – Adjacency matrix inefficient for sparse graphs, always takes $O(V^2)$ space.
Graph Algorithms

- Depth-first search (DFS)
- Breadth-first search (BFS)
- DFS & BFS Variants
- Minimum spanning tree (MST)
  - Kruskal’s algorithm & Prim’s algorithm
- Single-source shortest path (SSSP)
  - BFS, Djikstra’s algorithm, Bellman-ford
- All-pairs shortest path (APSP)
  - Floyd-Warshall
- Bipartite Graph
  - Bipartite graph check (BFS), Maximum matching
- Flow algorithms
Graph Algorithms: DFS & BFS

• Graph Search
• Single-source shortest path, unweighted (BFS)
• Strongly-connected components
  – Undirected: DFS / BFS
  – Directed: Tarjan’s algorithm
• Topological sort (DFS)
• Finding articulation points and bridges
Depth-First Search

- DFS on vertex U:
  - Mark U as visited.
  - For each neighbour V of U that has not been visited:
    - DFS on vertex V
Depth-First search

```c++
// Adjacency list representation.
// Graph[u][i] is the i'th neighbour of vertex u
vector<vector<int>> Graph;

vector<bool> visited;

void DFS(int u) {
    visited[u] = true;
    for (int i = 0; i < Graph[u].size(); ++i) {
        int v = Graph[u][i];
        if (!visited[v]) DFS(v);
    }
}
```
Breadth-First Search

• Use Queue to decide which node to visit next.
• BFS:
  Loop:
    – If Q empty, done!
    – Get and remove vertex U at front of Q
    – For each neighbour V of U:
      • If V has not been visited:
        – Set V to visited
        – Enqueue V
    – Goto: Loop
Breadth-First Search

```cpp
// Adjacency list representation.
// Graph[u][i] is the i'th neighbour of vertex u
vector<vector<int>> Graph;
vector<bool> visited;
queue<int> Q;

void BFS()
{
    while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        for (int i = 0; i < Graph[u].size(); ++i) {
            int v = Graph[u][i];
            if (visited[v]) continue;
            visited[v] = true; Q.push(v);
        }
    }
}
```
SSSP with BFS

- Works for unweighted graph, or where edges all have weight 1.
- Keep around a vector of “parents”
- Whenever you visit a node, record the node you came from as the parent
- Follow the parents to find the shortest path
// Adjacency list representation.
// Graph[u][i] is the i'th neighbour of vertex u
vector<vector<int>> Graph;
vector<bool> visited;
vector<int> parent;
queue<int> Q;

void BFS()
{
    while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        for (int i = 0; i < Graph[u].size(); ++i) {
            int v = Graph[u][i];
            if (visited[v]) continue;
            parent[v] = u;
            visited[v] = true; Q.push(v);
        }
    }
}
Strongly-Connected Components

- SCC is a subset of nodes where there exists a path between any pair of nodes in the subset.
- For an undirected graph, can use DFS or BFS to find them.
- For a directed graph, use Tarjan’s algorithm (variant of DFS)
SCC with DFS

• All nodes we reach during a single run of DFS are in the same SCC
• Simply run DFS on an unvisited node. All nodes the DFS visits are members of the newly found SCC
SCC with DFS

• For each node u:
  – If u has not been visited:
    • Report new SCC
    • DFS(u)
SCC with Tarjan’s Algorithm

- Finds SCCs in directed graphs
- Variant of DFS, we’ll get back to this one later.
Articulation points & Bridges

- Articulation point: Node that, if removed, disconnects the graph.
- Bridge: Edge that, if removed, disconnects the graph.
- Of strategic importance (cut off enemy supply lines, etc.)
- How to find these?
Articulation Points & Bridges

• Simple Method:
  – First, run DFS to verify graph connected.
  – For each node:
    • Remove the node.
    • Run DFS to see if the graph has been disconnected.

• $O(V(V+E))$
Articulation Points & Bridges

• More efficient method: Modified DFS
• Introduce two node labels: DFS_num and DFS_low
• DFS_num: Iteration on which we first saw this node.
• DFS_low: Smallest DFS_num we can reach in the DFS subtree below this node.