Graphs (part 1)

234901 Workshop in Competitive Programming

Graph Representation

	Adjacency List	Adjacency Matrix
Description	A list of neighbors for each vertex	$A_{i,j}$ is 1 if i and j are connected by an edge (or it holds the weight)
Implementation	An array of vectors	A two dimensional array
Space	$\|V\| + \|E\|$	$\|V\|^2$
Checking existence of an edge	$\deg(v)$	1
Iterating the neighbors of a node	$\deg(v)$	$\|V\|$
Iterating all edges of the graph	$\|V\| + \|E\|$	$\|V\|^2$

- We usually use an adjacency list:

```
typedef vector< vector<int> > vvi; // unweighted graph
typedef vector< vector< pair<int,int> > > vvii; // weighted graph
```

- Number the nodes 0…|V|-1 (use map if needed)

Graph Traversals

Your turn! Implement DFS.

- Graph traversal can be done depth or breadth first:

	DFS	BFS
Description	At each step handle the vertex and add its neighbors to the list of vertices to handle	
	First handle the last seen (and unhandled)	First handle the first seen (and unhandled)
Implementation	Stack (or recursion)	Queue
Applications	Finding strongly connected components	Finding shortest paths in unweighted graphs Identifying a bipartite graph (no cycles of odd length)
	Cycles detection Finding a spanning forest Finding connected components	

DFS: recursive vs. iterative

```
vii g;
vector<bool> visible;

void dfs(int s) {
   printf("%d\n", s);
   visible[s] = true;
   for(int u : g[s]) if(!visible[u]) dfs(u);
}

In main:
// load graph to g
visible.assign(g.size(),false);
dfs(0);
```

```
void dfs(const vvi& g, int s) {
   stack<int> q; q.push(s);
   vector<bool> visible (g.size(),false);
   visible[s]=true;
   while (!q.empty()) {
      int u = q.top(); q.pop();
      printf("%d\n", u);
      for (int v : g[u]) if (!visible[v]) {
         visible[v] = true;
         q.push(v);
}}}
```

BFS vs. DFS

```
void dfs(const vvi& g, int s) {
   stack<int> q; q.push(s);
   vector<bool> visible (g.size(),false);
   visible[s]=true;
   while (!q.empty()) {
      int u = q.top(); q.pop();
      printf("%d\n", u);
      for (int v : g[u]) if (!visible[v]) {
         visible[v] = true;
         q.push(v);
}}}
```

```
void bfs(const vvi& g, int s) {
   queue<int> q; q.push(s);
   vector<bool> visible (g.size(),false);
   visible[s]=true;
   while (!q.empty()) {
      int u = q.front(); q.pop();
      printf("%d\n", u);
      for (int v : g[u]) if (!visible[v]) {
         visible[v] = true;
         q.push(v);
}}}
```

Strongly Connected Components (SCC)

- Input: Directed graph
- Output: A list of strongly connected components
- Time: $O(|V| + |E|)$
- Description:
  1. DFS and push the vertices to a stack when done handling them.
  2. Generate a new graph with the edges reversed.
  3. Iteratively start a DFS on the reversed graph from each unseen vertex according to the order of step 1. Each DFS tree is a SCC.

SCC

```
void KosarajuDFS(const vvi& g, int u, int color,
                                    vi& S, vi& colorMap) {
  colorMap[u] = color;
  for (auto& v : g[u]) if (colorMap[v] == UNSEEN)
      KosarajuDFS(g, v, S, colorMap, color);
  S.push_back(u);}
```

```
int findSCC(const vvi& g, vi& components) {
  // first pass: record the `post-order' of original graph.
  vi order, seen;
  seen.assign(g.size(), UNSEEN);
  for (int i = 0; i < g.size(); ++i) if (seen[i] == UNSEEN)
      KosarajuDFS(g, i, SEEN, order, seen);
  // second pass: explore the SCCs based on first pass result.
  vvi reverse_g(g.size(),vi()); for (int u=0; u<g.size(); u++) for (int v : g[u]) reverse_g[v].push_back(u);
  int numSCC = 0; components.assign(g.size(), UNSEEN);
  vi dummy;
  for (int i = (int)g.size()-1; i >= 0; --i) if (components[order[i]] == UNSEEN)
      KosarajuDFS(reverse_g, order[i], numSCC++, dummy, components);
  return numSCC; }
```

Topological Sort

Your turn!
Implement topological sort.

- Definition: An ordering of the vertices, such that
  for each edge $u \to v$, u appears before v.
- A topological sort exists iff the graph is directed acyclic (DAG).

- Input: Directed graph
- Output: Topological sort or an error is the graph is not a DAG
- Time: $O(|V| + |E|)$
- Description:
  As long as there is a vertex with no incoming edges, set it to be next element in the ordering and remove it from the graph.
  If there are unprocessed vertices and no source, the graph is not a DAG.
- Implementation: only need to maintain the indegree of each vertex and a set of sources.

Topological Sort

```
bool topologicalSort(const vvi& g, vi& order) {
   // compute indegree of all nodes
   vi indegree (g.size(), 0);
   for (int v=0; v<g.size(); v++) for (int u : g[v]) indegree[u]++;
   // order sources first
   order = vector<int>();
   for (int v=0; v<g.size(); v++) if (indegree[v] == 0) order.push_back(v);
   // go over the ordered nodes and remove outgoing edges, add new sources to the ordering
   for (int i=0; i<order.size(); i++) for (int u : g[order[i]]) {
      indegree[u]--;
      if (indegree[u]==0) order.push_back(u);}
return order.size()==g.size();}
```

Shortest Paths

	BFS	Dijkstra	Bellman-Ford	Floyd-Warshall
Paths found	From a single source	From a single source	From a single source	Between every pair
Weights	Unweighted	Non-negative	All weights	All weights
Time	$O(\|E\| + \|V\|)$	$O(\|E\| \log \|V\|)$	$O(\|V\|\|E\|)$	$O(\|V\|^3)$
Description	Traverse the graph in layers by distance from the source	Start with the closest unvisited vertex, and relax* its outgoing edges	Relax* all edges $\|V\| - 1$ times	For every vertex k and for every pair (u, v) relax** $dist[u, v]$ according to k
More applications			Detecting a reachable negative cycle (If more relaxation can be done)	Detecting a negative cycle ($dist(u, u)$ will be negative)

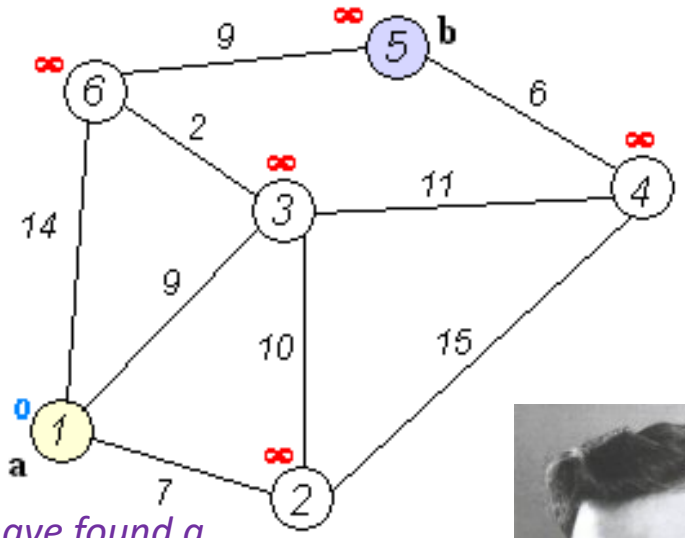
* Relaxing the edge $u \to v$ means assigning $dist[v] = \min(dist[v], dist[u] + w(u \to v))$
** Relaxing $dist[u, v]$ according to k means assigning $dist[u, v] = \min(dist[u, v], dist[u, k] + dist[k, v])$

BFS for shortest paths

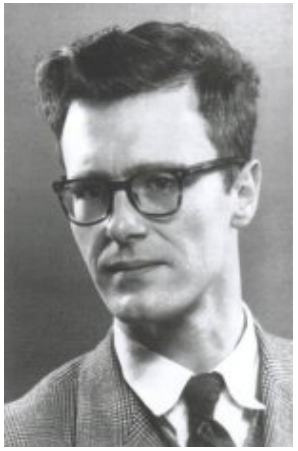
```
void bfs(const vvi& g, int s, vector<int>& d ) {
   queue<int> q; q.push(s);
   vector<bool> visible (g.size(),false);
   visible[s]=true;
   d.assign(g.size(),INF); d[s] = 0;
   while (!q.empty()) {
      int u = q.front(); q.pop();
      for (int v : g[u]) if (!visible[v]) {
         visible[v] = true;
         d[v] = d[u]+1;
         q.push(v);
}}}
```

```
void Dijkstra(const vvii& g, int s, vi& d) {
  d = vi(g.size(),INF); d[s] = 0;
  priority_queue<ii, vii, greater<ii>> q;
  q.push({0, s});
  while (!q.empty()) {
    ii front = q.top(); q.pop();
    int dist = front.first, u = front.second;
    if (dist > d[u]) continue;
    for (ii next : g[u]) {
      int v = next.first, w = next.second;
      if (d[u]+w < d[v]) {
        d[v] = d[u]+w;
        q.push({d[v], v});
      }}}}
```

Dijkstra (1956)



We may have found a shorter way to get to u after inserting it to q. In that case, we want to ignore the previous insertion to q.



```
bool BellmanFord(const vvii& g, int s, vi& d) {
   d.assign(g.size(), INF); d[s] = 0;
   bool changed = false;
   // V times
   for (int i = 0; i < g.size(); ++i) {
      changed = false;
      // go over all edges u->v with weight w
      for (int u = 0; u < g.size(); ++u) for (ii e : g[u]) {
         int v = e.first;
         int w = e.second;
         // relax the edge
         if (d[u] < INF && d[u]+w < d[v]) {
            d[v] = d[u]+w;
            changed = true;
         }}}
   // a negative cycle iff changes in the last iteration
   return changed;
}
```

Bellamn-Ford (1955)

Floyd-Warshall (1962)

```
// assume d is initialize with the distances according to the edges
for (int k=0; k<g.size(); ++k)
    for (int u=0; u<g.size(); ++u)
        for (int v=0; v<g.size(); ++v)
            d[u][v] = min(d[u][v], d[u][k]+d[k][v]);
```

Just can’t get enough?!

- Additional Resources:
  - Algorithms 1: http://webcourse.cs.technion.ac.il/234247
  - Wikipedia.
  - Auxiliary code on the course website.

- Additional Algorithms:
  - Flood fill
  - Finding articulation points and bridges
  - Finding SCC has another algorithm by Tarjan