Graph Algorithms

Bernhard von Stengel, LSE

Definitions:

Directed graph = (V, E)

V = set of **nodes**, e.g. V = {1,2,3..., n}.
E set of node pairs (u,v), called **edges** (for directed graph also: **arcs**).

If E is symmetric, i.e. (u,v) in E if and only if (v,u) in E: **undirected** graph, edges are unordered pairs {u, v}.

Path from u to v of length k = sequence of edges $u_0,u_1,u_2,...,u_k$, where $u_o=u$, (u_i,u_{i+1}) is an edge for i= 0,...,k-1, and $u_k=v$.

An undirected graph is called **connected** if there is a path between any two nodes.

A **cycle** in a graph is a path from a node to itself.

A **tree** is a directed or undirected graph with a distinguished node, the **root**, from which there is a unique path to every other node. Unique tree-predecessor also called “parent”.

Storing a graph:

1. Adjacency matrix

A[i][j] for i and j in V which is 1 if and only if (i,j) is an edge, and 0 otherwise.
Requires n^2 bits of storage if |V|=n.
Extends to graphs with multiple (parallel) edges by storing the number of edges between i and j in A[i][j].

2. Adjacency list

Adj[u] = { v in V | (u, v) in E }
Adj[u] is the set of all *neighbours* of u. Can be stored as a linked list.

(picture)

Storage requirement O(|V|+|E|), which is less than $|V|^2$ if E has relatively few edges (sparse graph).

Depth-first search (DFS) on undirected graphs

Initialisation:
first colour each node u "white" (unvisited),
and having no parent: p[u] = NULL

1. go through all nodes, and whenever one is white, start a DFS-visit

2. A DFS-visit of a node u
- marks the node "grey" (in process)
- assigns it a “timestamp” d[u] (a new number which afterwards is incremented; all timestamps are *distinct* numbers), the DFS-discovery time.
- looks at all neighbours v of u, and if v is white, marks u has the DFS-parent of v, assignment p[v]=u, and then recursively starts a
- DFS-visit(v)
- after all neighbours of u have been looked at, mark u "black" (finished), give u a new time-stamp f[u], the DFS-finishing time.

4. The result will be a collection of *trees*, the DFS-trees, whose roots are the nodes u with p[u]=NULL, and otherwise p[u] as parent of u.

Running time of the above procedure: O(|V|) for initialization 1., and then the sum over all nodes u of |Adj[u]|, which is |E|.
Altogether O(|V|+|E|).

Parenthesis-property of DFS-discovery and -finishing times (Thm 23.6 of CLR):
For any two nodes u,v of a directed graph, if d[u] < d[v] (i.e. u discovered before v), either f[v] < f[u] (i.e. visit-time-interval for v subset of interval for u) and then v is descendant of u in DFS-tree, or f[u] < d[v] (i.e. visit of u finished before the search of v begins).

Proof: if not f[u] < d[v], then u still grey while v is being visited. DFS-visit(v) will then terminate before DFS-visit(u).

Note: Each edge (u,v) of the graph will be visited once during DFS.

Classification of edges: according to colour of endpoint during DFS-visit:
white edge (u,v): white endpoint v (also called *tree* edge, with u=p[v])
grey edge (u,v): grey endpoint v (also called *back* edge)

black edge (u,v): black (already finished) endpoint v (sometimes called *forward/cross* edge).

Thm 23.9 of CLR: An *undirected* graph G has no black edges (i.e. only white or grey edges, i.e. tree or back edges)

Proof: (u,v) edge of G, w.l.o.g. d[u] < d[v]. Then f[u] < d[v] not possible since v is a neighbour of u, so DFS-visit of u cannot have finished when v is discovered.
Hence f[v] < f[u] by Parenthesis theorem. If the edge is explored in DFS as (u,v), it becomes a white (= tree) edge, if as (v,u), a grey (= back) edge.

Lemma 23.10 of CLR: A directed graph G has *no cycle* if and only if a DFS of G gives no grey (= back) edges.

Proof: A grey edge clearly gives a cycle. Conversely, consider a cycle and let v be the node with smallest d[v] in that cycle (the node that is discovered first).
Let u be its predecessor in the cycle, so that (u,v) is an edge.
Claim: this edge is grey, i.e. while exploring u the node v is still grey. If not, then by the parenthesis theorem f[v] < d[u] (v already finished before u was discovered).
We show that this is not possible (CLR call this the "white path theorem"): Then there must be an edge (x,y) on the cycle (path) from v to u: v ... x y ... u (where v=x or y=u or both are possible) so that f[x] < f[v] (i.e. x finished before v) but f[v] < d[y] (v finished before y discovered). But then f[x] < d[y], i.e. x finished before y discovered, contradicting that y is a neighbor of x. So (u,v) is a grey edge.

Application: Topological sorting

Given: a directed graph without cycles (**acyclic** graph).

Topological sorting means to sort the nodes of the graph in some order $u_1,u_2,...,u_n$ such that
whenever there is an edge from u_i to u_j, then i<j.

Picture: Dress-in-the morning topological sorting problem.

Topological-Sort(G):

1. Perform DFS(G), where
2. Whenever a node is finished (marked black), insert it at the *beginning* of a list.
3. Output the list, which is then topologically sorted (in effect in reverse order of their finishing times).

Running time O(|V|+|E|).

Why does topological sort work? Consider an edge (u,v) of G when encountered during DFS. Then v cannot be grey by Lemma 23.10 since G is acyclic, so v is either black, i.e. already finished and f[v] < f[u], or white, which means that (u,v) is a tree edge and v hence a DFS-descendant of u, and thus again f[v] < f[u] by the parenthesis theorem.

Connected components

Any undirected graph is a disjoint union of connected graphs, which are called its connected components.

Recognition of these components:

Array element c[v] for each node v, which is simply the root of its DFS-tree, updated whenever a new node is encountered. Then c[u]=x if and only if c[u] is in same component as x.
Compute components in time O(|V|+|E|).

Can be extended to **biconnected** components, which are sets of nodes such that between any two nodes there are two paths. This means that taking any one node out still keeps the graph connected (application: reliable networks).

Breadth first search (BFS)

Idea: unlike DFS, where the search continues at the most recently discovered node (so the nodes are stored on a stack), the search continues at the oldest not yet fully explored node. Any newly discovered node is therefore not put on a stack, but at the end of a **queue**.

Works for directed as well as undirected graphs.

BFS(G, s) s = starting node
d[v] = depth is BFS-tree

Initialize: For each node u:
- colour u "white"
- d[u] = infinity
- p[u] = NULL

mark s "grey";
d[s]=0;
Q = (s);
(Q is the queue of nodes to be processed)

while (Q is not empty)
u = first node in Q; remove u from Q;
for each neighbour u of v, if v is white:
- mark v "grey"
- d[v] = d[u] + 1;
- p[v] = u (parent of v is u in BFS tree)
- append v at the end of Q.

when all neighbours of u have been visited, mark u "black" (finished).

Running time:
- each white node is exactly once put into Q.
- Queue operation takes time O(1)
- All of them time O(|V|)
- each node u moved once out of Q, work on Adj[u], altogether O(|E|) time.

Running time thus O(|V|+|E|).

Theorem: After BFS,
d[v] is **shortest** length of a path from s to v.

Single-source shortest paths

Given: weight function from edges to reals, usually called the **length** l(e) of an edge, nonnegative.

Weight (or length) of a path: sum of length of edges in the path.

Shortest path from u to v = the path of shortest length.

Single source shortest path: source s (some node) is fixed, find

shortest paths from s to all nodes v.

Happens to be just as quickly computable as **single-pair** shortest path between any two nodes s and t.

Remark: Negative weights might create cycles with negative length, running through them again and again makes the path arbitarily "short".
(Picture of a negative-length cycle.)
Recognition of negative-length cycle is easy, finding shortest path avoiding negative-length cycles makes the problem difficult.

Single-source shortest path computation

d(u, v) = shortest length of a path from u to v
s start node

Lemmas (sub-paths of shortest paths are shortest paths):

1. If $v_0 v_1 \ldots v_k$ is a shortest path from v_0 to v_k, then $v_i \ldots v_j$ is also a shortest path from v_i to v_j. (A shorter path from v_i to v_j would also shorten the overall path.)
2. Let s...uv be a shortest path from s to v. Then
$$d(s,v) = d(s, u) + l(u, v)$$
(Recall that l(u, v) is the length of the edge from u to v.)
3. For all (u, v) in E:
$$d(s,v) \leq d(s, u) + l(u, v)$$

Dijkstra's Algorithm:
Given is a graph with node set V and nonnegative lengths l(e) for its edges e. Data used in the algorithm:

D[u] preliminary distance from s to u, more precisely: the shortest length of all paths from s to u, **which are, with the exception of u, completely in S.** (The set S grows over time, initially empty.)

pred[v] predecessor on shortest path from s to v

S set of nodes such that for all v in S D[v] = d(s,v)

Initialisation:

for all v in V: { D[v] = infinity; pred[v] = NULL; }
D[s] = 0 ;
S = empty set; Q = V;
Throughout: Q = V\S (the nodes not in S) form a **priority queue** sorted in ascending order of D[v] for the nodes v it contains.

Main algorithm:

```
while (Q is not empty)
{
        remove u with smallest D[u] from Q;
        S = S augmented by u;
        for (all nodes v adjacent to u and not in S)
                if (D[v] > D[u] + l(u, v) )
                {
                        D[v] = D[u] + l(u, v);
                        pred[v] = u ;
                }
}
```

The algorithm terminates when S==V.
The predecessor on the shortest path from s to u is given by pred[u], and the shortest distance by D[u].

Running time:
- visit all nodes during initialisation
- visit all edges (u,v) once for updating D[v]
- per node:
  - find the smallest D[u] for u in Q
  - has running time O(|V|) if linear search,
    overall running time $O(|V|^2)$
  - if the priority queue is implemented as **heap**: only logarithmic search time O(log |V|), then overall running time O(|E| log|V|).

Correctness: see reasoning at the board (use own notes).