Graphs and algorithms

Recall that graphs are pair (V,E), with:

- V is a set of *nodes*;
- E is a set of *edges*, meant to relate the nodes of the graph itself;

If the edges are *(un)directed*, so it is the graph.

A path is a sequence of edges, which are pairwise connected. A cycle is a path moving from a node and ending up in the very same node. A graph is acyclic if it doesn't contain cycles (**DAG** stands from directed acyclic graph).

A graph is *connected* if there is a path between every pair of nodes, a *connected component* of an undirected graph is a maximum connected sub-graph of G.

We can represent graph in two ways:

- **adjacency lists**, every element in the list corresponds to a specific node, the lists associated with an element represents edges moving from the current element (convenient for directed, sparse graphs, namely with not that much edges). There is an adjacency list for each vertex;
- **adjecency matrix**, each row represents the nodes reachable from that node using a single edge: $M_{ij} = 1$ if there is an edge moving from node i to node j, 0 otherwise (convenient for dense graphs).

The most trivial operation in a graph is **visit** it. A way to do it is **Breadth-First-Search (BFS)** (wave from the source), another is **Deapth-First-Search (DFS)** (searches deeper in the graph whenever possible).

BFS

Idea is to relate the visiting node to the distance from the source, the less the distance of a node from the source, the sooner it will be visited. Indeed it is also used to compute source-node distances. Visiting order is related to the distance from a *source*.

We can use colouring: nodes can be white, black or grey. White nodes have not been *discovered* yet (at the beginning all nodes are white), grey nodes have been discovered, but some of their neighbours are still undiscovered. Black nodes have been discovered and all their neighbours have been discovered. This procedure returns a tree, the *breadth-first tree* (tree of the shortest paths from the source).

Given a graph G=(V,E) and a distinguished source vertex s, BFS systematically explores the edges of G to discover every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a BF tree with root s that contains all reachable vertices. in such a tree, for every vertex v, the simple path from s to v corresponds to a shortest path from s to v in G (i.e. path containing the smallest number of edges).

BFS expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier (namely it discovers all vertices at distance k from s before discovering any vertex at distance $k+1$).

BFS constructs a tree, initially containing only its root, which is the source vertex s. Whenever the search discovers a white vertex v in the course of scanning the adjacency list of an already discovered vertex u, the vertex v, the vertex u and the edge (u, v) are added to the tree. u is called *predecessor/parent* of v. Since a vertex is discovered at most once, it has at most one parent.

The complexity of this operation is $T_{BFS}((V, E)) = \Theta(|V|) + O(|E|) + O(|V|) \in O(|E| + |V|)$, thus it runs in time linear in the size of the adjacency list representation of G.

Lemma: be $Q = [v_1, ..., v_n]$ be the queue of BFS at some point of the computation. Then $v_i \cdot d \leq v_{i+1} \cdot d$. For all $i \in [1, .., n-1]$ and $v_1 \cdot d + 1 \leq v_n \cdot d$.

proof (by induction on the number of iteration of the while loop, namely queue operations):

base case iteration 0: Q exclusively contains the source s, so the thesis holds.

inductive step: assume that the property holds before the i-th iteration. $Q = [v_1, .., , v_n]$ and $v_i \cdot d \leq v_{i+1} \cdot d \; \forall i \in [1, .., n-1]$ and $v_n \cdot d \leq v_1 + 1$.

If the head v_1 of the queue is id dequeued, v_2 becomes the new head, by the inductive hypothesis $v_1 \cdot d \leq v_2 \cdot d$, then $v_n \cdot d \leq v_1 \cdot d + 1 \leq v_2 \cdot d + 1$ and the remaining inequalities are unaffected. Thus the lemma holds with v_2 as the head.

Now we need to understand what happens when enqueuing a vertex. Let $v_{11}, .., v_{1m}$ be the white neighbours of v_1. So after the i-th iteration Q will be $[v_2, .., v_n, v_{11}, .., v_{1m}]$. So that $v_{1j} \cdot d = v_1 \cdot d + 1 \geq v_n \cdot d \geq v_1 \cdot d \; \forall j \in [1, .., m]$, $\forall i \in [2, .., n]$. Rename nodes as: $\forall v'_1, ., v'_{n-1}, v'_n, .., v'_{n+m+1}$, we proved that the $\forall i \in [1, .., m+n]$ $v'_i \cdot d \leq v'_{i+1} \cdot d$. The remaining inequalities are unaffected.

This lemma is useful to prove the following:

Theorem (Correctness of BFS): let $\delta(s, v)$ be the distance from s to v (i.e. the length of the shortest path). After BFS:

1. $v \cdot d \neq \infty$ iff it is reachable from s (BFS discovers every vertex reachable from the source);
2. if $v \cdot d \neq \infty$, then $v \cdot d = \delta(s, v)$;
3. the shortest path from s to v ends with $(v.pred, v)$.

proof:

1. by contradiction (all nodes in the graph G should be either black (meaning $\neq \infty$) or white (meaning $= \infty$) since BFS is finished by assumption). This holds of course for all the nodes on paths from s to v. Let w be the last

black node on one of these paths (all the neighbours of w should be black, in particular the ones that belong to the path from s to v), since w is black all the nodes reachable from it crossing an edge must be non-white including that on the path from s to v. This is a contradiction.

2. It comes from previous lemma + induction on the length of the shortest paths.
3. We observe that, if $v.pred = u$, than $v \cdot d = u \cdot d + 1$, thus we can obtain a shortest path from s to v by taking a shortest path from s to $v.pred$ and then traversing the edge $(v.pred, v)$.

With BFS we can compute the distance of a node from a source, and also the BFS tree.

DFS

All the nodes are visited:

- a non-visited node s is selected as source;
- a path is extended as much as possible by adding non-visited nodes;
- the process is repeated on all the branches left behind by previous step;
- if some node remain non-visited, a node among them is selected as new source.

DFS produces a *depth-first forest* (not a tree) of the predecessors. DFS labels all the nodes with **discovery time** and **finishing time**. Nodes are white, grey or black coloured: white nodes have not been discovered yet, grey nodes have been discovered, but not finished yet (they have discovery time set, but finishing time still undefined), black nodes have been finished (they have final time, reached when they don't have more whit neighbours).

The strategy followed by DFS is to search deeper in the graph whenever possible, it explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of the v's edges have been explored, the search backtracks to explore edges leaving the vertex from which v was discovered. This process continues until all we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertex remains, then DFS selects one of them as a new source, and repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex.

Since the search may be repeated from multiple sources, the predecessor subgraph produced by a depth-first search may be composed of several trees, forming a depth-first forest.

Beside creating a depth-first forest, DFS also *timestamps* each vertex: the first records when the vertex has been discovered (hence grayed), the second

records when the search finishes examining the adjacency list of the vertex (hence blackens the vertex).

The asymptotic complexity of this algorithm is $\Theta(|V| + |E|)$.

Theorem (parenthesis theorem): for any pair of nodes $(v, u) \in V$ either:

1. $[u.d, u.f] \cap [v.d, v.f] = \varnothing$ (time between discovery and finish) and neither u is a descendant (in the forest) of v, nor v of u (we don't discover v moving along u, and viceversa);
2. $[u.d, u.f] \subset [v.d, v.f]$ and u is descendant of v;
3. $[v.d, v.f] \subset [u.d, u.f]$ and v is descendant of u.

(need to look at discovering and finishing time when we have to look if u is descendant of v and vv).

If we represent the discovery time of a vertex with a left parenthesis [and the finishing time with a right parenthesis], then the history of discoveries and finishings makes a well-formed expression in the sense that the parenthesis are properly nested.

Theorem (white-path theorem): for any pair of nodes $(v, u) \in V$, u is a descendant of v iff at time $v.d - 1$ there is a white path (path containing exclusively white nodes) from v to u.

This theorem gives a characterization of when one vertex is a descendent of another in the depth-first forest.

Edge classification: *tree edges* that belongs to the depth first forest and *back edges* which connect a node to an ancestor or self loop, *forward edges* connect a node to a non-direct descendant, the remaining edges are called *cross edges*.

Topological sort

Dependency relation can be modelled using directed graphs. On DAGs there exists a topological order $\preceq_T$ on nodes such that: if $if(u, v) \rightarrow u \preceq_T v$.

A **topological sort** of a DAG G=(V,E) is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering (of course this is not possible in cyclic graphs). We can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.

In order to perform topological sort, we can call DFS to compute the finishing time of all the vertices, and once a vertex is finished we can insert it into a stack, we return this stack.

Sorting the elements according to this order takes $\Theta(|V| + |E|)$.

Theorem (Correctness of topological sort):

- let $[v_1, .., v_n]$ be the output of topological sort. Then $v_i.f > v_{i+1}.f \ \forall i \in [1, n-1]$;
- a graph G contains cycles iff DFS(G) yields back edges;
- if G is a DAG, then topological sort produces G's topological sort.

Strongly connected components

Consider a cycle: all nodes involved are reachable by the same set of nodes and can reach the same set of nodes (from the reachability point of view, all the nodes in a loop behave the same way). In some sense they are equivalent, need to discover such equivalent nodes.

Strongly connected components (SCCs) (we don't construct loops because they can be exponential in the number of nodes involved) are maximum subgraphs (of a directed graph) s.t. for every pair of their nodes, there is a path from one to the other and vice versa.

The relation of being reachable (in both direction) is an equivalence relation. So it induces a partition to the nodes of any graph. For every pair of nodes in the same SCC there is a path connecting the two in both directions.

Note that SCC can be defined exclusively on directed graphs.

Our goal is now to decompose a directed graph into its strongly connected components.

The idea is to identify a SCC using DFS, recall that the graph G contains a cycle iff DFS(G) yields black edges (in that case the black edge is the last edge in the loop). This is a way to discover loops. But this is just a part of strongly connected component, we need to label all the nodes in the loop. We may use the **Minimum Discovery Time of the subtree (lowlink)**: the minimum discovery time among all the nodes reachable from the current node, along one single back edge.

Since we are trying to search SCC we are searching for set of nodes such that for any pair of node, the first can reach the second and vice versa. Because of the white path theorem, for sure, thanks to this reachability property, any SCC must be discovered by one DFS_VISIT call. Not possible that we have two SCC such that DFS_VISIT interleave nodes of both SCC.

So we need to:

- perform a DFS_VISIT call and update lowlinks when possible;
- identify a SCC as soon as all its nodes have been visited;
- label the SCC nodes as "not available for lowlink updates".

Once the first node of the SCC has been discovered, its lowlink is the discovery time, hence once this first node has been finished, all of the node in the SCC

have been discovered and finished. Idea is to use a stack to store all the discovery times, as soon as we discover a node whose discovery time is equivalent to its lowlink, we can remove all nodes from the stack except this last. This is called **Tarjan SCCs Algorithm**, and its cost us $\Theta(|V| + |E|)$ (linear wrt number of nodes and number of edges of the original graph), at the same time it identifies the SCCs and topological order collapsing all components.

Transitive closure

Our goal is to know, given any pair of nodes (v, w), if there is a path between the two.

A naive solution can be to use BFS from all the nodes, in this way we end up with a complexity of $|V| \cdot O(|V| + |E|) = O(|V|^2 + |V| \cdot |E|)$.

Consider the matrix representation of the graph, we can easily detect if a node has distance 1 from another node. Moreover two nodes v,w are at distance 2 if $\exists z$ s.t. $A[v][z] \neq 0$ and $A[z][w] \neq 0 \leftrightarrow (AXX)[v][n] > 0$. Generalising: w has distance $\leq n$ from v iff $A^n[v][w] > 0$.

Since every acyclic path has at most length |V| (all the nodes on the graph once), we can solve the problem using the Strassen's algorithm, ending up with complexity $(|V| - 1) \cdot \Theta(|V|^{log_2^7})$, worst than the naive solution using BFS.

We think of a solution involving SCC, so we can collapse all nodes in SCCs and build the *SCCs graph* $\bar{G}$ (all nodes inside a SCC are identical from the reachability point of view). Now we end up for sure in a DAG, and we can apply topological sort (the original graph was not DAG), recall that if we apply Tarjan's algorithm, then the topological sort is for free. If we look at the adjacency matrix of the collapsed graph, and we order it according to the topological sort, we end up in a upper triangular matrix.

Summarising:

- Compute SCCs of the original graph with Tarjan's algortihm: $\Theta(|V| + |E|)$;
- build SCCs graph $\bar{G}$, using naive algorithm: $\Theta(|E|)$;
- topological sort: $\Theta(|V| + |E|)$;
- compute the infinite product $\bar{G}^\star$: $T(n) = 2 \cdot T(\frac{n}{2}) + 2 \cdot \Theta(n^{log_2 7})$;
- extend the transitive closure of $\bar{G}$: every node in a SCC will be able to reach every node reachable in the collapsed graph corresponding to nodes in that SCC, $O(|V|^2)$ (filling a matrix substantially).

Summing up, the overall asymptotic complexity is $\Theta(|E| + |V|^{log_2 7})$, depending on the number of edges, namely if the graph is not sparse, this algorithm can be quite efficient.