Graphs

Part I: Basic algorithms

Laura Toma
Algorithms (csci2200), Bowdoin College

Undirected graphs

Concepts:

- connectivity, connected components
- paths
- (undirected) cycles

Basic problems, given undirected graph G:

- is G connected
- how many connected components (CC) are in G?
- label each vertex with its CC id
- find a path between u and v
- does G contain a cycle?
- compute a spanning forest for G

(Undirected) DFS

Idea: explore the graph "depth-first". Similar to how you'd try to find a path out of a maze.

Easiest to write it recursively. Usually keep track of the vertex that first discovered a vertex u; we call that the parent of u.

DFS(vertex v)

- mark v
- for each adjacent edge (v, u):
  - if u is not marked:
    - mark u
    - parent$(u) = v$
    - DFS(u)

(Undirected) DFS

DFS-tree: Each vertex, except the source vertex v, has a parent $\Rightarrow$ these edges define a tree, called the DFS-tree.

During $DFS(v)$ each edge in G is classified as:

- tree edge: an edge leading to an unmarked vertex
- non-tree edge: an edge leading to a marked vertex.

(Undirected) DFS

Lemma

$DFS(u)$ visits all vertices in the connected component (CC) of u, and the DFS-tree is a spanning tree of $CC(u)$.

Proof sketch: Assume by contradiction that there is a vertex v in CC(u) that is not reached by DFS(u). Since u, v are in same CC, there must exist a path $v_0 = u, v_1, v_2, ..., v_k, v$ connecting u to v. Let v_i be the last vertex on this path that is reached by DFS(u) (v_i could be u). When exploring v_i, DFS must have explored edge (v_i, v_{i+1}),..., leading eventually to v. Contradiction.

(Undirected) DFS

Tree:

- ancestors of x: parent, grandparent,..., i.e., all vertices on the path from x to root
- descendants of x: chidlren, grandschildren,.., i.e. all vertices in the subtree rooted at x

Lemma

For a given call $DFS(u)$, all vertices that are marked between the start and end of this call are all descendants of u in the DFS-tree.

To understand why this is true, visualize how the recursion works in $DFS(v)$. The outgoing edges $(v, *)$ are visited one at a time, calling DFS recursively on nodes that will become v's children; when all edges outgoing from v are visited, $DFS(v)$ terminates and returns. While the children of v are explored, the call $DFS(v)$ stays on stack. Repeat this argument, and you'll see that when $DFS(x)$ is called, the following functions are active on stack: DFS(parent(x)), DFS(grandparent(x)),....,DFS(v).

(Undirected) DFS

Lemma

Non-tree edges encountered during undirected $DFS(v)$ go from a vertex to an ancestor of that vertex.

Proof: Let's say DFS(v) reaches a vertex x and explores edge (x, y), at which point it sees that y is marked. We want to show that y is an ancestor of x. We know that all ancestors of x are marked and "unfinished", i.e. their DFS frames are active, on stack, and the system will backtrack to finish them. In contrast, a vertex that is marked but not an ancestor of x is "finished", i.e. all its outgoing edges were exploredf and DFS will not backtrack there.

Now assume by contradiction, that y is marked but is not an ancestor of x: then DFS has finished exploring y; when y was visited, edge must have been (y, x) explored and x could not have been marked at that time, so x would have been made a child of y —- contradiction.

(Undirected) DFS

As written above, DFS explores only the CC of v. DFS can be used to explore all the CCs in G:

- mark all vertices as "unmarked"
- for each vertex v
  - if v is marked, skip
  - if v is not marked: $DFS(v)$

Lemma

DFS runs in $O(|V| + |E|)$ time.

Proof: It explores every vertex once. Once a vertex is marked, it's not explored again. It traverses each edge twice. Overall, $O(|V| + |E|)$.

(Undirected) DFS

Undirected DFS can be used to solve in $O(|V| + |E|)$ time the following problems:

- is G connected?
- compute the number of CC of G
- compute a spanning forest of G
- compute a path between two vertices of G, or report that such a path does not exist
- compute a cycle, or report that no cycle exists

(Undirected) BFS

Idea: explore outwards, one layer at a time. Visualize a wave propagating outwards. BFS logically subdivides the vertices into layers.

BFS(vertex u)
mark u, d(u) =0, $Q = \{\}$
while Q not empty

- remove the next vertex v from Q
- for all edges (v, w) do
  - if w is not marked:
    - mark w
    - parent(w)=u //(v, w) is a tree edge
    - d(w) = d(v) + 1
    - add w to Q
  - //else: w is marked, (v, w) is non-tree edge

Undirected BFS

BFS-tree: Each vertex, except the source vertex v, has a parent $\Rightarrow$ these edges define a tree, called the DFS-tree.

During $BFS(v)$ each edge in G is classified as:

- tree edge: an edge leading to an unmarked vertex
- non-tree edge: an edge leading to a marked vertex.

Undirected BFS

Lemma

$BFS(u)$ visits all vertices in the connected component (CC) of u, and the BFS-tree is a spanning tree of $CC(u)$.

Proof sketch: Assume by contradiction that there is a vertex v in CC(u) that is not reached by BFS(u). Since u, v are in same CC, there must exist a path $v_0 = u, v_1, v_2, ..., v_k, v$ connecting u to v. Let v_i be the last vertex on this path that is reached by BFS(u) (v_i could be u). When exploring v_i, BFS must have explored edge (v_i, v_{i+1}),..., leading eventually to v. Contradiction.

Undirected BFS

As written above, BFS explores only the CC of v. BFS can be used to explore all the CCs in G:

- mark all vertices as "unmarked"
- for each vertex v
  - if v is marked, skip
  - if v is not marked: $BFS(v)$

Lemma

BFS runs in $O(|V| + |E|)$ time.

Proof: It explores every vertex once. Once a vertex is marked, it's not explored again. It traverses each edge twice. Overall, $O(|V| + |E|)$.

Undirected BFS

Lemma

Let x be a vertex reached in BFS(v). The path $v \rightarrow x$ contains d(i) edges and represents the shortest path from v to x in G.

Notation: length of shortest path from v to u is $\delta(v, u)$.

Proof idea: The complete proof is quite long.....The idea is contradiction: Assume there exists at least one vertex for which BFS(v) does not compute the right distance. Among these vertices, let u be the vertex with the smallest distance from v. Let $p = (v, ..., u)$ be the shortest path from v to u of length $\delta(v, u)$. The vertex x just before u on this path has shortest path to v of length $\delta(v, x) = \delta(v, u) - 1$ (subpaths of shortest paths are shortest paths bla bla..). This vertex is correctly labeled by BFS (because of our assumption), so $d(x) = \delta(v, x) = \delta(v, u) - 1$. But because of edge (x, u), BFS will find a path to u of length $d(v, x) + 1$, i.e. $d(u) = \delta(v, u)$.

Undirected BFS

Lemma

For any non-tree edge (x, y) in BFS(v), the level of x and y differ by at most one.

In other words, x, y are on the same level or on consecutive levels; there cannot be non-tree edges that jump over more than one level.

Proof idea: Intuitively, this is because all immediate neighbors that are not marked are put on the next level. Observe that, at any point in time, the vertices in the queue have distances that differ by at most 1. This can be shown easily with induction (bla bla). Let's say x comes out first from the queue; at this time y must be already marked (because otherwise (x, y) would be a tree edge). Furthermore y has to be in the queue, because, if it wasn't, it means it was already deleted from the queue and we assumed x was first. So y has to be in the queue, and we have $|d(y) - d(x)| \leq 1$ by above observation.

Undirected BFS

Undirected BFS can be used to solve in $O(|V| + |E|)$ time the following problems:

- is G connected?
- compute the number of CC of G
- compute a spanning forest of G
- compute shortest path between two vertices of G
- compute a cycle, or report that no cycle exists

Directed graphs (Digraphs)

Concepts:

- reachability
- directed paths and cycles
- strongly connected components (SCC)
- directed acyclyc graphs (DAGs)
- transitive closure (TC)

Problems:

- given u, v: does u reach v?
- given u: find all vertices reachabla from u
- is G strongly connected?
- is G acyclic?
- compute the SCCs
- compute the TC $G*$ of G

Directed DFS

Same as undirected, but visit the outgoing edges of a vertex

Properties:

- DFS(v) visits all vertices reachable from v.
- the DFS-tree contains directed paths from v to all vertices reachable from v
- runs in $O(|V| + |E|)$
- non-tree edges (x, y) are of 3 types:
  - back edge: y is an ancestor of x in DFS tree
  - forward edges: y is a descendant of x in the DFS tree
  - cross edges: y is neither ancestor nor descendant
- all 3 types of non-tree edges are possible in directed DFS
- a non-tree back edge defines a directed cycle

Directed DFS

Given a digraph G, directed DFS can be used to solve the following in $O(|V| + |E|)$:

- does the graph contain a directed cycle?
- find a directed cycle
- given v, find all vertices reachable from v
- given u, v: find a path from u to v or report that there is none

Directed BFS

Same as undirected, but visit the outgoing edges of a vertex

Properties:

- BFS(v) visits all vertices reachable from v.
- the BFS-tree contains directed paths from v to all vertices reachable from v
- BFS(v) computes the shortest paths from v to all vertices reachable from v
- runs in $O(|V| + |E|)$
- forward non-tree edges are not possible
  - When visiting x, all outgoing edges are explored at the same time. If y is not marked, y becomes the child of x. If y is marked, it's either a back edge or a cross edge. A forward edge (x, y) is not possible.

Directed BFS

Given a digraph G, directed BFS can be used to solve the following in $O(|V| + |E|)$:

- find directed cycles
- given v, find all vertices reachable from v
- given u, v: find the shortest path from u to v

Is G strongly connected?

Idea: for each v run BFS(v). G is SC if and only if every vertex reaches all other vertices. This runs in $O(V \cdot (V + E))$.

Can we do better?

YES. Run BFS twice. Total $O(V + E)$.

- run $BFS(v)$ from an arbitrary vertex v. If this does not reach all $u \in V (u \neq v)$, G is not SC.
- If it does, run a modified $BFS(v)$ that visits the *incoming* edges in a vertex. If all $u \in V (u \neq v)$ are reached by this BFS, then we conclude G is SC.
  - given $x, y \in V$: x reaches v, and v reaches y, so x reaches y. The other way around, too.

Note: Can use BFS or DFS. SCCs can be found in $O(V + E)$ but it's more complicated.

Transitive closure

Idea: for each v, run BFS(v), and for all vertices w reached, output an edge (v, w). Runs in $O(V \cdot (V + E))$.

Can we do better?

YES. Idea: If a reaches b and b reaches c, then a reaches c. Use this to avoid recomputation. Dynamic programming.

Transitive closure with dynamic programming

Ideas:

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, 3, ..., n$

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, 3, ..., n$

We'll compute G^* in rounds, starting from G.

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, 3, ..., n$

We'll compute G^* in rounds, starting from G.

In round 1: $G \rightarrow G_1$

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, 3, ..., n$

We'll compute G^* in rounds, starting from G.

In round 1: $G \rightarrow G_1$

- We start with all edges in G, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertex 1.

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, 3, ..., n$

We'll compute G^* in rounds, starting from G.

In round 1: $G \rightarrow G_1$

- We start with all edges in G, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertex 1.
- How does a path (i, j) that goes only through vertex 1 look like?

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, 3, ..., n$

We'll compute G^* in rounds, starting from G.

In round 1: $G \rightarrow G_1$

- We start with all edges in G, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertex 1.
- How does a path (i, j) that goes only through vertex 1 look like? $(i, j), (i, 1, j)$

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, 3, ..., n$

We'll compute G^* in rounds, starting from G.

In round 1: $G \rightarrow G_1$

- We start with all edges in G, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertex 1.
- How does a path (i, j) that goes only through vertex 1 look like? $(i, j), (i, 1, j)$
- When does it exist?

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, 3, ..., n$

We'll compute G^* in rounds, starting from G.

In round 1: $G \to G_1$

- We start with all edges in G, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertex 1.
- How does a path (i, j) that goes only through vertex 1 look like? $(i, j), (i, 1, j)$
- When does it exist? When (i, j) exists, or $(i, 1)$ and $(1, j)$ both exist

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, 3, ..., n$

We'll compute G^* in rounds, starting from G.

In round 1: $G \to G_1$

- We start with all edges in G, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertex 1.
- How does a path (i, j) that goes only through vertex 1 look like? $(i, j), (i, 1, j)$
- When does it exist? When (i, j) exists, or $(i, 1)$ and $(1, j)$ both exist
- This comes down to checking, if there exist edges $(i, 1)$ and $(1, j)$ in G.

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, 3, ..., n$

We'll compute G^* in rounds, starting from G.

In round 2: $G_1 \rightarrow G_2$

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, 3, ..., n$

We'll compute G^* in rounds, starting from G.

In round 2: $G_1 \rightarrow G_2$

- We start with all edges in G, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertices $1, 2$.

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, 3, ..., n$

We'll compute G^* in rounds, starting from G.

In round 2: $G_1 \rightarrow G_2$

- We start with all edges in G, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertices $1, 2$.
- How does a path (i, j) that goes only through $1, 2$ looks like?

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, 3, ..., n$

We'll compute G^* in rounds, starting from G.

In round 2: $G_1 \rightarrow G_2$

- We start with all edges in G, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertices $1, 2$.
- How does a path (i, j) that goes only through $1, 2$ looks like? $(i, j), (i, 1, j), (i, 2, j), (i, 1, 2, j), (i, 2, 1, j))$

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, 3, ..., n$

We'll compute G^* in rounds, starting from G.

In round 2: $G_1 \rightarrow G_2$

- We start with all edges in G, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertices $1, 2$.
- How does a path (i, j) that goes only through $1, 2$ looks like? $(i, j), (i, 1, j), (i, 2, j), (i, 1, 2, j), (i, 2, 1, j))$
- When does it exist?

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, 3, ..., n$

We'll compute G^* in rounds, starting from G.

In round 2: $G_1 \rightarrow G_2$

- We start with all edges in G, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertices $1, 2$.
- How does a path (i, j) that goes only through $1, 2$ looks like? $(i, j), (i, 1, j), (i, 2, j), (i, 1, 2, j), (i, 2, 1, j))$
- When does it exist? When (i, j) exists in G_1, or $(i, 2)$ and $(2, j)$ both exist in G_1

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, 3, ..., n$

We'll compute G^* in rounds, starting from G.

In round 2: $G_1 \rightarrow G_2$

- We start with all edges in G, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertices $1, 2$.
- How does a path (i, j) that goes only through $1, 2$ looks like? $(i, j), (i, 1, j), (i, 2, j), (i, 1, 2, j), (i, 2, 1, j))$
- When does it exist? When (i, j) exists in G_1, or $(i, 2)$ and $(2, j)$ both exist in G_1
- This comes down to checking, if there exist edges $(i, 2)$ and $(2, j)$ in G_1.

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, ..., n$

We'll compute G^* in rounds, starting from G.

In round k+1: $G_k \rightarrow G_{k+1}$

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, ..., n$

We'll compute G^* in rounds, starting from G.

In round k+1: $G_k \rightarrow G_{k+1}$

- We start with all edges in G_k, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertices $1, 2, ..., k, k+1$.

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, ..., n$

We'll compute G^* in rounds, starting from G.

In round k+1: $G_k \rightarrow G_{k+1}$

- We start with all edges in G_k, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertices $1, 2, ..., k, k+1$.
- How does a path (i, j) that goes only through $1, 2, ..., k, k+1$ look like?

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, ..., n$

We'll compute G^* in rounds, starting from G.

In round k+1: $G_k \rightarrow G_{k+1}$

- We start with all edges in G_k, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertices $1, 2, ..., k, k+1$.
- How does a path (i, j) that goes only through $1, 2, ..., k, k+1$ look like? $(i....j), (i..., k+1, ...j)$

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, ..., n$

We'll compute G^* in rounds, starting from G.

In round k+1: $G_k \rightarrow G_{k+1}$

- We start with all edges in G_k, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertices $1, 2, ..., k, k+1$.
- How does a path (i, j) that goes only through $1, 2, ..., k, k+1$ look like? $(i....j), (i..., k+1, ...j)$
- When does it exist?

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, ..., n$

We'll compute G^* in rounds, starting from G.

In round k+1: $G_k \rightarrow G_{k+1}$

- We start with all edges in G_k, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertices $1, 2, ..., k, k+1$.
- How does a path (i, j) that goes only through $1, 2, ..., k, k+1$ look like? $(i....j), (i..., k+1, ...j)$
- When does it exist? When (i, j) exists in G_k, or $(i, k+1)$ and $(k+1, j)$ both exist in G_k

Transitive closure with dynamic programming

Ideas:

Number the vertices $1, 2, ..., n$

We'll compute G^* in rounds, starting from G.

In round k+1: $G_k \rightarrow G_{k+1}$

- We start with all edges in G_k, and we add a new edge (i, j) for every pair of vertices such that i can reach j going only through vertices $1, 2, ..., k, k+1$.
- How does a path (i, j) that goes only through $1, 2, ..., k, k+1$ look like? $(i....j), (i..., k+1, ...j)$
- When does it exist? When (i, j) exists in G_k, or $(i, k+1)$ and $(k+1, j)$ both exist in G_k
- This comes down to checking, if there exist edges $(i, k+1)$ and $(k+1, j)$ in G_k.

Transitive closure with dynamic programming

Idea 1: Number the vertices $1, 2, ..., n$
Idea 2: Consider paths that only use vertices numbered $1, 2, ..., k$ as intermediate vertices.

Lemma

A path from v_i to v_j that uses vertices numbered $v_1, v_2, ..., v_k$, v_{k+1} either:

- *does not include vertex V_{k+1}: is a path from v_i to v_j that uses vertices numbered $v_1, v_2, ..., v_k$; or*
- *includes vertex v_{k+1}: is a path from v_i to v_{k+1} that uses vertices numbered $v_1, v_2, ..., v_k$; and a path from v_{k+1} to v_j that uses vertices numbered $v_1, v_2, ..., v_k$.*

Transitive closure with dynamic programming

Floyd-Warshall's algorithm:

- $G_0 = G$
- for $k = 1$ to n
  - $G_k = G_{k-1}$
  - for $i = 1..n$ $(i \neq k)$ and $j = 1..n$ $(j \neq i, k)$:
    - if edge (v_i, v_j) exists in G_{k-1}, then add edge (v_i, v_j) to G_k
    - else: if both edges (v_i, v_k) and (v_k, v_j) exist in G_{k-1}, then add edge (v_i, v_j) to G_k

Lemma

G_n represents the TC of G.

Transitive closure with dynamic programming

Floyd-Warshall's algorithm:

- $G_0 = G$
- for $k = 1$ to n
  - $G_k = G_{k-1}$
  - for $i = 1..n$ $(i \neq k)$ and $j = 1..n$ $(j \neq i, k)$:
    - if edge (v_i, v_j) exists in G_{k-1}, then add edge (v_i, v_j) to G_k
    - else: if both edges (v_i, v_k) and (v_k, v_j) exist in G_{k-1}, then add edge (v_i, v_j) to G_k

Lemma

G_n represents the TC of G.

Paths in G_n contain all possible vertices and more iterations will not change G_n.

Transitive closure with dynamic programming

Floyd-Warshall's algorithm:

- $G_0 = G$
- for $k = 1$ to n
  - $G_k = G_{k-1}$
  - for $i = 1..n$ $(i \neq k)$ and $j = 1..n$ $(j \neq i, k)$:
    - if edge (v_i, v_j) exists in G_{k-1}, then add edge (v_i, v_j) to G_k
    - else: if both edges (v_i, v_k) and (v_k, v_j) exist in G_{k-1}, then add edge (v_i, v_j) to G_k

Lemma

G_n represents the TC of G.

Paths in G_n contain all possible vertices and more iterations will not change G_n.

Analysis: if G is represented by adjacency matrix: $O(n^3)$

Transitive closure

Two algorithms:

- Run BFS from each vertex:
  $O(V(V+E)) = O(n(n+m))$
- Floyd-Warshall: $O(n^3)$

Which one is better, when?

Topological order

Definition

A topological ordering of G is an ordering $v_1, v_2, ..., v_n$ of its vertices such that for any edge (v_i, v_j) in G, $v_i < v_j$ in the order.

A topological order is an order such that any directed path traverses the vertices in increasing order.

There may be more than one topological orderings.

Topological order

Lemma

If a digraph G admits a topological ordering then it is acyclic.

Proof idea: Assume by contradiction that G contains a cycle. Let v_i, v_j two vertices on this cycle. Then v_i reaches v_j and v_j reaches v_i. This means that v_i must come before v_j, and v_j must come before v_i in the ordering. Contradiction.

Topological order

Lemma

If a digraph G is acyclic then it admits a topological ordering.

Proof:
(1) If a graph is acyclic, there must be a vertex of indegree 0: Assume by contradiction that this is not true. If all vertices have indegree ≥ 1, we could travel from a vertex v using one of its incoming edge, $x \leftarrow v$, and from x using one of its incoming edges, and so on, and we could do this indefinitely since all vertices have at least one incoming edge. There's only a finite number of vertices, so there must be a cycle. Contradiction.

(2) Let v be a vertex of indegree 0. We put v first in topological order, and delete all its outgoing edges from G: For each (v, u), decrement indegree(u). The remaining graph $G - \{v\}$ is still acyclic. Repeat.

Topological order

Theorem

A digraph G admits a topological ordering if and only if G is acyclic.

Algorithm:

- for each $v \in V$: compute indegree(v)
- $L = \{v \in V | indegree(v) = 0\}$
- while L not empty :
  - delete a vertex from L (doesn't matter which one). Call it v.
  - v is the next vertex in topological ordering
  - for every edge (v, u): decrement indegree(u); if indegree(u)==0 insert u in L.

Topological order

Theorem

A digraph G admits a topological ordering if and only if G is acyclic.

Algorithm:

- for each $v \in V$: compute indegree(v)
- $L = \{v \in V | indegree(v) = 0\}$
- while L not empty :
  - delete a vertex from L (doesn't matter which one). Call it v.
  - v is the next vertex in topological ordering
  - for every edge (v, u): decrement indegree(u); if indegree(u)==0 insert u in L.

Analysis: $O(V + E)$