Lecture 12: Graph Algorithms III

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

October 24, 2023

Overview

- Directed Graphs
  - Reachability
  - BFS/DFS trees
  - Directed Acyclic Graphs (DAGs) & Topological Sort
  - Strongly Connected Components
- Acknowledgements

Directed Graphs

- Now each edge has a direction, and we say that (u, v) goes from u (tail) to v (head)

Directed Graphs

- Now each edge has a direction, and we say that (u, v) goes from u (tail) to v (head)
- Useful to model situations with asymetry:
  - web page links
  - one-way streets
  - dependencies in parallel computation

Directed Graphs

- Now each edge has a direction, and we say that (u, v) goes from u (tail) to v (head)
- Useful to model situations with asymetry:
  - web page links
  - one-way streets
  - dependencies in parallel computation
- Notation:
  - $\deg_{in}(u) = \#$ vertices $s \in V$ such that $(s, u) \in E$ (in-degree/fanin)
  - $\deg_{out}(u) = \#$ vertices $t \in V$ such that $(u, t) \in E$ (out-degree/fanout)

Reachability in Directed Graphs

Let $G(V, E)$ be a directed graph and $s, t \in V$.

- t is *reachable* from s, if there is a directed $s - t$ path in G

Reachability in Directed Graphs

Let $G(V, E)$ be a directed graph and $s, t \in V$.

- t is *reachable* from s, if there is a directed $s - t$ path in G
- G is *strongly connected* if $\forall s, t \in V$, we have that t is reachable from s and s is reachable from t

Reachability in Directed Graphs

Let $G(V, E)$ be a directed graph and $s, t \in V$.

- t is *reachable* from s, if there is a directed $s - t$ path in G
- G is *strongly connected* if $\forall s, t \in V$, we have that t is reachable from s and s is reachable from t
- $S \subset V$ is strongly connected if $\forall s, t \in S$, we have s reachable from t and t reachable from s

Reachability in Directed Graphs

Let $G(V, E)$ be a directed graph and $s, t \in V$.

- t is *reachable* from s, if there is a directed $s - t$ path in G
- G is *strongly connected* if $\forall s, t \in V$, we have that t is reachable from s and s is reachable from t
- $S \subset V$ is strongly connected if $\forall s, t \in S$, we have s reachable from t and t reachable from s
- $S \subset V$ is a *strongly connected component* (SCC) if S is a maximal strongly connected set

Reachability in Directed Graphs

Let $G(V, E)$ be a directed graph and $s, t \in V$.

- t is *reachable* from s, if there is a directed $s - t$ path in G
- G is *strongly connected* if $\forall s, t \in V$, we have that t is reachable from s and s is reachable from t
- $S \subset V$ is strongly connected if $\forall s, t \in S$, we have s reachable from t and t reachable from s
- $S \subset V$ is a *strongly connected component* (SCC) if S is a maximal strongly connected set
- We are interested in following reachability/structural questions:
  1. Given $s \in V$ what are the vertices reachable from s?
  2. Is a given graph strongly connected?
  3. What are all strongly connected components in a given directed graph?

Reachability in Directed Graphs

Let $G(V, E)$ be a directed graph and $s, t \in V$.

- t is *reachable* from s, if there is a directed $s - t$ path in G
- G is *strongly connected* if $\forall s, t \in V$, we have that t is reachable from s and s is reachable from t
- $S \subset V$ is strongly connected if $\forall s, t \in S$, we have s reachable from t and t reachable from s
- $S \subset V$ is a *strongly connected component* (SCC) if S is a maximal strongly connected set
- We are interested in following reachability/structural questions:
    1. Given $s \in V$ what are the vertices reachable from s?
    2. Is a given graph strongly connected?
    3. What are all strongly connected components in a given directed graph?
- Just as with undirected graphs, we will find $O(n + m)$ time algorithms for these and other problems.

Checking Reachability

- **Input:** directed graph $G(V, E)$, $s \in V$
- **Output:** all vertices reachable from s

Checking Reachability

- **Input:** directed graph $G(V, E)$, $s \in V$
- **Output:** all vertices reachable from s
- Could use either **BFS** or **DFS** for this question. We will use DFS.

Checking Reachability

- **Input:** directed graph $G(V, E)$, $s \in V$
- **Output:** all vertices reachable from s
- EXPLORE(u, visited, p, S, F, τ):
  1. $S[u] = \tau$, and $\tau \leftarrow \tau + 1$
  2. for each $v \in N_{out}(u)$: (only outgoing neighbors)
     - If visited$[v] = 0$, then
       visited$[v] = 1$, $p[v] = u$
       and EXPLORE(v, visited, p, S, F, τ).
  3. $F[u] = \tau$, $\tau \leftarrow \tau + 1$ and

Checking Reachability

- **Input:** directed graph $G(V, E)$, $s \in V$
- **Output:** all vertices reachable from s
- EXPLORE(u, visited, p, S, F, τ):
  1. $S[u] = \tau$, and $\tau \leftarrow \tau + 1$
  2. for each $v \in N_{out}(u)$: (only outgoing neighbors)
     - If visited$[v] = 0$, then
       visited$[v] = 1$, $p[v] = u$
       and EXPLORE(v, visited, p, S, F, τ).
  3. $F[u] = \tau$, $\tau \leftarrow \tau + 1$ and
- Main algorithm:
  1. initialize

     $$\text{visited}[v] = 0,\ S[v] = F[v] = \infty \text{ and } p[v] = \text{NULL}$$

     for all $v \in V$
  2. set visited$[s] = 1$ and $\tau = 1$
  3. EXPLORE(s, visited, p, S, F, τ)

Checking Reachability

- **Input:** directed graph $G(V, E)$, $s \in V$
- **Output:** all vertices reachable from s
- EXPLORE(u, visited, p, S, F, τ):
  1. $S[u] = \tau$, and $\tau \leftarrow \tau + 1$
  2. for each $v \in N_{out}(u)$: (only outgoing neighbors)
     - If visited$[v] = 0$, then
       visited$[v] = 1$, $p[v] = u$
       and EXPLORE(v, visited, p, S, F, τ).
  3. $F[u] = \tau$, $\tau \leftarrow \tau + 1$ and
- Main algorithm:
  1. initialize

     $$\text{visited}[v] = 0,\ S[v] = F[v] = \infty \text{ and } p[v] = \text{NULL}$$

     for all $v \in V$
  2. set visited$[s] = 1$ and $\tau = 1$
  3. EXPLORE(s, visited, p, S, F, τ)
- Time complexity $O(n + m)$, and similarly to undirected case, t reachable iff visited$[t] = 1$.

Directed Cuts

- Set of all visited vertices forms a “directed cut”
  - no outgoing edges
  - possibly incoming edges

- Directed Graphs
  - Reachability
  - BFS/DFS trees
  - Directed Acyclic Graphs (DAGs) & Topological Sort
  - Strongly Connected Components

- Acknowledgements

DFS Trees

- Just as in undirected graph case, we have (directed and undirected) DFS trees, given by edges $\{u, p[u]\}$ (or $(p[u], u)$ if we keep directions).

DFS Trees

- Just as in undirected graph case, we have (directed and undirected) DFS trees, given by edges $\{u, p[u]\}$ (or $(p[u], u)$ if we keep directions).
- In *undirected* graph case, we have proved all non-tree edges are *back edges* (last lecture)

DFS Trees

- Just as in undirected graph case, we have (directed and undirected) DFS trees, given by edges $\{u, p[u]\}$ (or $(p[u], u)$ if we keep directions).
- In *undirected* graph case, we have proved all non-tree edges are *back edges* (last lecture)
- However, in *directed* graph case, we can have "**cross edges**" and "**forward edges**" (we cannot choose orientation now)

DFS Trees

- Just as in undirected graph case, we have (directed and undirected) DFS trees, given by edges $\{u, p[u]\}$ (or $(p[u], u)$ if we keep directions).
- In *undirected* graph case, we have proved all non-tree edges are *back edges* (last lecture)
- However, in *directed* graph case, we can have "**cross edges**" and "**forward edges**" (we cannot choose orientation now)
- Still plenty of structure left:

  Parenthesis lemma still holds!

BFS Trees

- Just as in undirected graph case, we have (directed and undirected) BFS trees, given by edges $\{u, p[u]\}$ (or $(p[u], u)$ if we keep directions).

BFS Trees

- Just as in undirected graph case, we have (directed and undirected) BFS trees, given by edges $\{u, p[u]\}$ (or $(p[u], u)$ if we keep directions).
- In *undirected* graph case, we have proved all non-tree edges are between *consecutive layers* (last lecture)

BFS Trees

- Just as in undirected graph case, we have (directed and undirected) BFS trees, given by edges $\{u, p[u]\}$ (or $(p[u], u)$ if we keep directions).
- In *undirected* graph case, we have proved all non-tree edges are between *consecutive layers* (last lecture)
- However, in *directed* graph case, we can have non-tree edges between *arbitrary layers* ("backward edges")

BFS Trees

- Just as in undirected graph case, we have (directed and undirected) BFS trees, given by edges $\{u, p[u]\}$ (or $(p[u], u)$ if we keep directions).
- In *undirected* graph case, we have proved all non-tree edges are between *consecutive layers* (last lecture)
- However, in *directed* graph case, we can have non-tree edges between *arbitrary layers* ("backward edges")
- Still, plenty of structure left

Shortest paths from source.

- Directed Graphs
  - Reachability
  - BFS/DFS trees
  - Directed Acyclic Graphs (DAGs) & Topological Sort
  - Strongly Connected Components

- Acknowledgements

Directed Acyclic Graphs (DAGs)

- DAGs are directed graphs without directed cycles

Directed Acyclic Graphs (DAGs)

- DAGs are directed graphs without directed cycles
- Very useful in modelling dependency relations
  - course pre-requisites
  - software installation
  - sequence of algebraic operations

Directed Acyclic Graphs (DAGs)

- DAGs are directed graphs without directed cycles
- Very useful in modelling dependency relations
  - course pre-requisites
  - software installation
  - sequence of algebraic operations
- Very useful to find ordering of vertices so that all edges "go forward"

  Topological Ordering

Topological Ordering

Proposition

A directed graph is acyclic $\Leftrightarrow$ *there is a topological ordering.*

Topological Ordering

Proposition

A directed graph is acyclic $\Leftrightarrow$ *there is a topological ordering.*

- ($\Leftarrow$) given topological ordering, no edge goes backwards, therefore no cycles

Topological Ordering

Proposition

A directed graph is acyclic $\Leftrightarrow$ there is a topological ordering.

- ($\Rightarrow$) if we prove that any DAG has a vertex u with $\deg_{in}(u) = 0$, then can construct topological order by putting u in first position, then iterating over graph $G \setminus \{u\}$

Topological Ordering

Proposition

A directed graph is acyclic $\Leftrightarrow$ there is a topological ordering.

- $(\Rightarrow)$ if we prove that any DAG has a vertex u with $\deg_{in}(u) = 0$, then can construct topological order by putting u in first position, then iterating over graph $G \setminus \{u\}$
- **Proof of indegree zero vertex:**
  - Suppose (for sake of contradiction) that every vertex u has $\deg_{in}(u) \geq 1$.
  - Starting from vertex $t =: u_0$, go to an in-neighbour u_1, and then to an in-neighbour u_2 and so on. (possible since $\deg_{in}(u_i) > 0$)
  - Since graph is finite, at some point must repeat a vertex $\Rightarrow$ found a cycle. (contradiction)

Topological Ordering

Proposition

A directed graph is acyclic $\Leftrightarrow$ there is a topological ordering.

- ($\Rightarrow$) if we prove that any DAG has a vertex u with $\deg_{in}(u) = 0$, then can construct topological order by putting u in first position, then iterating over graph $G \setminus \{u\}$
- **Proof of indegree zero vertex:**
  - Suppose (for sake of contradiction) that every vertex u has $\deg_{in}(u) \geq 1$.
  - Starting from vertex $t =: u_0$, go to an in-neighbour u_1, and then to an in-neighbour u_2 and so on. (possible since $\deg_{in}(u_i) > 0$)
  - Since graph is finite, at some point must repeat a vertex $\Rightarrow$ found a cycle. (contradiction)
- Can use above procedure to topologically sort a DAG (exercise)

Constructing a Topological Ordering

- Algorithm:
  1. Run DFS on the whole graph
  2. Output the ordering with decreasing finishing time
  3. Check if this is a topological ordering. If not, return **not acyclic**.

Constructing a Topological Ordering

- Algorithm:
  1. Run DFS on the whole graph
  2. Output the ordering with decreasing finishing time
  3. Check if this is a topological ordering. If not, return **not acyclic**.
- Why does this work? (parenthesis lemma)

Constructing a Topological Ordering

- Algorithm:
  1. Run DFS on the whole graph
  2. Output the ordering with decreasing finishing time
  3. Check if this is a topological ordering. If not, return **not acyclic**.
- Why does this work? (parenthesis lemma)

Lemma

If G is a DAG, then for any $(u, v) \in E$, $F[v] < F[u]$ for any DFS.

Constructing a Topological Ordering

- Algorithm:
  1. Run DFS on the whole graph
  2. Output the ordering with decreasing finishing time
  3. Check if this is a topological ordering. If not, return **not acyclic**.
- Why does this work? (parenthesis lemma)

Lemma

If G is a DAG, then for any $(u, v) \in E$, $F[v] < F[u]$ for any DFS.

- We have 2 cases:
  - Case 1: $S[v] < S[u]$.
  - Since graph is a DAG (no cycles) u not reachable from v
  - Hence u not descendant of v. By parenthesis property, must have

$$[S[v], F[v]] \cap [S[u], F[u]] = \emptyset \Rightarrow F[v] < F[u]$$

Constructing a Topological Ordering

- Algorithm:
  1. Run DFS on the whole graph
  2. Output the ordering with decreasing finishing time
  3. Check if this is a topological ordering. If not, return **not acyclic**.
- Why does this work? (parenthesis lemma)

Lemma

If G is a DAG, then for any $(u, v) \in E$, $F[v] < F[u]$ for any DFS.

- We have 2 cases:
  - Case 2: $S[v] > S[u]$.
  - Since visited$[v] = 0$ when we start u and $(u, v) \in E$, v will be a descendant of u in DFS tree.
  - Parenthesis lemma implies $[S[v], F[v]] \subset [S[u], F[u]]$

Constructing a Topological Ordering

- Algorithm:
  1. Run DFS on the whole graph
  2. Output the ordering with decreasing finishing time
  3. Check if this is a topological ordering. If not, return **not acyclic**.
- Why does this work? (parenthesis lemma)

Lemma

If G is a DAG, then for any $(u, v) \in E$, $F[v] < F[u]$ for any DFS.

- Correctness:
  1. By lemma G is a DAG $\Rightarrow$ all edges go forward in this ordering
  2. G has a cycle, then there is no topological order by proposition

Constructing a Topological Ordering

- Algorithm:
  1. Run DFS on the whole graph
  2. Output the ordering with decreasing finishing time
  3. Check if this is a topological ordering. If not, return **not acyclic**.
- Why does this work? (parenthesis lemma)

Lemma

If G is a DAG, then for any $(u, v) \in E$, $F[v] < F[u]$ for any DFS.

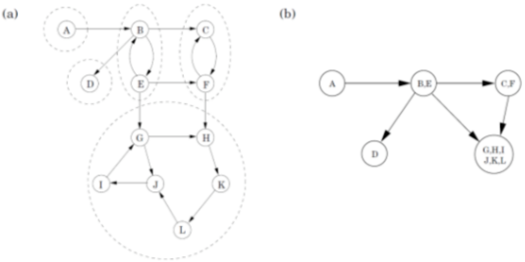
- Correctness:
  1. By lemma G is a DAG $\Rightarrow$ all edges go forward in this ordering
  2. G has a cycle, then there is no topological order by proposition
- Running time: $O(n + m)$ (can obtain sorted list within algorithm)

Strongly Connected Components (SCCs)

- **Input:** directed graph $G(V, E)$
- **Output:** Strongly connected components of G

Strongly Connected Components (SCCs)

- **Input:** directed graph $G(V, E)$
- **Output:** Strongly connected components of G
- **Observation 1:** SCCs are vertex disjoint

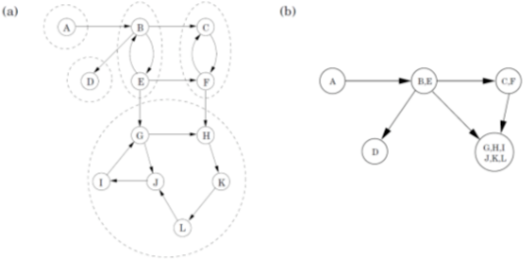


picture from
[DPV 3.4]

Exercise: Prove this

Strongly Connected Components (SCCs)

- **Input:** directed graph $G(V, E)$
- **Output:** Strongly connected components of G
- **Observation 1:** SCCs are vertex disjoint



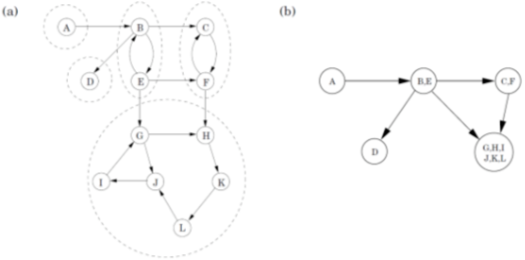
picture from [DPV 3.4]

Exercise: Prove this

- **Observation 2:** general directed graph is a DAG on its SCCs!

Strongly Connected Components (SCCs)

- **Input:** directed graph $G(V, E)$
- **Output:** Strongly connected components of G
- **Observation 1:** SCCs are vertex disjoint



picture from

[DPV 3.4]

Exercise: Prove this

- **Observation 2:** general directed graph is a DAG on its SCCs!
- Can we find a "topological sorting" of the SCCs? Need to find one component...

Strongly Connected Components

- **Idea 1:** If we started a DFS/BFS in a "sink component" Γ (with no outgoing edges), then we will certainly find only Γ and then we can recurse on $G \setminus \Gamma$.

Can we find such a component?

Strongly Connected Components

- **Idea 1:** If we started a DFS/BFS in a "sink component" Γ (with no outgoing edges), then we will certainly find only Γ and then we can recurse on $G \setminus \Gamma$.

Can we find such a component?

- **Attempt 1:** "topological sort" the components.
  1. If components are a DAG, then must have a sink.

Strongly Connected Components

- **Idea 1:** If we started a DFS/BFS in a "sink component" Γ (with no outgoing edges), then we will certainly find only Γ and then we can recurse on $G \setminus \Gamma$.

Can we find such a component?

- **Attempt 1:** "topological sort" the components.
  1. If components are a DAG, then must have a sink.
  2. From DAG discussion, node with earlierst finish time will be a sink.

Strongly Connected Components

- **Idea 1:** If we started a DFS/BFS in a "sink component" Γ (with no outgoing edges), then we will certainly find only Γ and then we can recurse on $G \setminus \Gamma$.

Can we find such a component?

- **Attempt 1:** "topological sort" the components.
  1. If components are a DAG, then must have a sink.
  2. From DAG discussion, node with earlierst finish time will be a sink.
  3. run DFS and obtain ordering in increasing finishing time, let s be element with earliest finish time

Strongly Connected Components

- **Idea 1:** If we started a DFS/BFS in a "sink component" Γ (with no outgoing edges), then we will certainly find only Γ and then we can recurse on $G \setminus \Gamma$.

Can we find such a component?

- **Attempt 1:** "topological sort" the components.
  1. If components are a DAG, then must have a sink.
  2. From DAG discussion, node with earlierst finish time will be a sink.
  3. run DFS and obtain ordering in increasing finishing time, let s be element with earliest finish time
  4. apply idea 1 to s, and obtain its SCC

Strongly Connected Components

- **Idea 1:** If we started a DFS/BFS in a "sink component" Γ (with no outgoing edges), then we will certainly find only Γ and then we can recurse on $G \setminus \Gamma$.

Can we find such a component?

- **Attempt 1:** "topological sort" the components.
  1. If components are a DAG, then must have a sink.
  2. From DAG discussion, node with earlierst finish time will be a sink.
  3. run DFS and obtain ordering in increasing finishing time, let s be element with earliest finish time
  4. apply idea 1 to s, and obtain its SCC

  Doesn't work: node of earliest finishing time need not be in sink component.

Strongly Connected Components

- **Idea 1:** If we started a DFS/BFS in a "sink component" Γ (with no outgoing edges), then we will certainly find only Γ and then we can recurse on $G \setminus \Gamma$.

  Can we find such a component?

- **Attempt 1:** "topological sort" the components.
  1. If components are a DAG, then must have a sink.
  2. From DAG discussion, node with earlierst finish time will be a sink.
  3. run DFS and obtain ordering in increasing finishing time, let s be element with earliest finish time
  4. apply idea 1 to s, and obtain its SCC
- **Observation 3:** note that node with largest finishing time will be in a *source component*!

Strongly Connected Components

Lemma

If Γ and Γ' are two SCCs and we have edges from Γ to Γ', then largest finish time of Γ is larger than largest finish time of Γ'.

Strongly Connected Components

Lemma

If Γ and Γ' are two SCCs and we have edges from Γ to Γ', then largest finish time of Γ is larger than largest finish time of Γ'.

- **Proof:** two cases

Strongly Connected Components

Lemma

If Γ and Γ' are two SCCs and we have edges from Γ to Γ', then largest finish time of Γ is larger than largest finish time of Γ'.

- **Proof:** two cases
  - Case 1: first visited vertex $u \in \Gamma \sqcup \Gamma'$ is in Γ
    - Since vertices in $\Gamma \sqcup \Gamma'$ are reachable from u, all vertices in $\Gamma \sqcup \Gamma'$ will be finished before u, so largest finishing time will be of u

Strongly Connected Components

Lemma

If Γ and Γ' are two SCCs and we have edges from Γ to Γ', then largest finish time of Γ is larger than largest finish time of Γ'.

- **Proof:** two cases
  - Case 1: first visited vertex $u \in \Gamma \sqcup \Gamma'$ is in Γ
    - Since vertices in $\Gamma \sqcup \Gamma'$ are reachable from u, all vertices in $\Gamma \sqcup \Gamma'$ will be finished before u, so largest finishing time will be of u
  - Case 2: first visited vertex $u \in \Gamma \sqcup \Gamma'$ is in Γ'
    - Since vertices from Γ unreachable from Γ', DFS needs to finish exploring Γ' before starting any vertex in Γ.

Strongly Connected Components

- Were looking for sinks, but found sources... how to deal with it?

Strongly Connected Components

- Were looking for sinks, but found sources... how to deal with it?
- Reverse the edges of the graph! Then sources become sinks (and vice-versa)!

Strongly Connected Components

- Were looking for sinks, but found sources... how to deal with it?
- Reverse the edges of the graph! Then sources become sinks (and vice-versa)!
- Let G^R be the graph obtained from G by reverting all edges

G and G^R have *same SCCs*!!

Strongly Connected Components

- Were looking for sinks, but found sources... how to deal with it?
- Reverse the edges of the graph! Then sources become sinks (and vice-versa)!
- Let G^R be the graph obtained from G by reverting all edges

G and G^R have *same SCCs*!!

- Can follow the ordering of the finishing times of DFS applied to G^R to get our sink components in G! (or vice-versa!)

Strongly Connected Components - Algorithm

- **Input:** directed graph $G(V, E)$
- **Output:** Strongly connected components of G
- **Algorithm:**
  1. Run DFS on G using arbitrary ordering of vertices
  2. Order vertices by decreasing order of finishing times, label vertices by $u_1, \ldots, u_n$ with $F[u_i] > F[u_{i+1}]$
  3. Reverse G to obtain G^R
  4. Follow ordering in Step 2 to explore G^R and cut out one SCC at a time
     - Let $\gamma = 1$ (counts # SCCs)
     - For $1 \leq i \leq n$ do:
       If visited$[u_i] = 0$, then: $DFS(G^R, u_i)$ and mark all vertices reachable from u_i in G^R to be in component Γ_γ. Then set $\gamma \leftarrow \gamma + 1$

Acknowledgement

- Based on Prof. Lau's lecture 07

  `https://cs.uwaterloo.ca/~lapchi/cs341/notes/L07.pdf`

References I

Cormen, Thomas and Leiserson, Charles and Rivest, Ronald and Stein, Clifford. (2009)

Introduction to Algorithms, third edition.

MIT Press

Kleinberg, Jon and Tardos, Eva (2006)

Algorithm Design.

Addison Wesley