Data structures and algorithms for graphs

Graph traversal. Applications

December 2020

Terminology

Graph $G = (V, E)$ where

- V: finite set of nodes of vertices
- E: list of edges $(a, b) \in V \times V$

Types of graphs:

Undirected: edges have no direction: $(a, b) = (b, a)$.

Directed: edges have direction: if $a \neq b$ then $(a, b) \neq (b, a)$. Usually, we write $a \rightarrow b$ instead of (a, b) and call it arc from a to b.

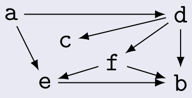
Weighted: a graph $G = (V, E)$ together with a weight function $w : E \rightarrow \mathbb{R}$, $w(e)$ is the weight of edge $e \in E$. Usually, we write $w(a, b)$ instead of $w((a, b))$.

Glossary

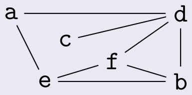
Assumption: $G = (V, E)$ is a given graph.

- Adjacency list of $x \in V$: $\texttt{adj}[x] = [y \in V \mid (x, y) \in E]$

Examples of representations with adjacency lists



adj[a] = [d, e]
adj[b] = [ ]
adj[c] = [ ]
adj[d] = [b, c, f]
adj[e] = [b]
adj[f] = [b, e]



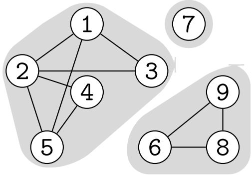
adj[a] = [d, e]
adj[b] = [d, e, f]
adj[c] = [d]
adj[d] = [a, b, c, f]
adj[e] = [a, b, f]
adj[f] = [b, d, e]

ASSUMPTION: $G = (V, E)$ is a given graph; $x, y \in V$

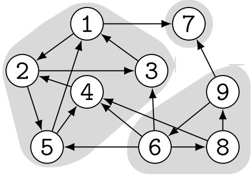
- Path from x to y = list of nodes $[x_1, x_2, \ldots, x_n]$ s.t.
  $x_1 = x$, $x_2 = y$, and $(x_i, x_{i+1} \in E)$ for all $1 \leq i < n$.
  The length of this path is $n - 1$.
- We write $x \leadsto y$ if there is a path from x to y, and $x \not\leadsto y$ otherwise.
- x, y are strongly connected, and we write $x \sim_{sc} y$, if $x \leadsto y$ and $y \leadsto x$.

Remarks:

1. $\leadsto$ is an equivalence relation on V in undirected graphs. The equivalence classes of $\leadsto$ for an undirected graph G are the connected components of G.
2. $\sim_{sc}$ is an equivalence relation on V in digraphs. The equivalence classes of $\leadsto$ for an digraph G are the strongly connected components of G.



Connected components:
$\{1,2,3,4,5\}$, $\{6,8,9\}$ and $\{7\}$



Strongly connected components:
$\{1,2,3,4,5\}$, $\{6,8,9\}$ and $\{7\}$

Graph traversals

Given $G = (V, E)$ and $s \in V$

Find the set of nodes $S = \{x \in V \mid s \rightsquigarrow x\}$. Also, for every $x \in S$, find a path from s to x.

Graph traversals

Given $G = (V, E)$ and $s \in V$

Find the set of nodes $S = \{x \in V \mid s \rightsquigarrow x\}$. Also, for every $x \in S$, find a path from s to x.

This problem can be solved with a tree traversal strategy.

Given $G = (V, E)$ and $s \in V$

Find the set of nodes $S = \{x \in V \mid s \rightsquigarrow x\}$. Also, for every $x \in S$, find a path from s to x.

This problem can be solved with a tree traversal strategy.

- The most important tree traversal strategies are depth first search (DFS) and breadth first search (BFS).

Graph traversals

Given $G = (V, E)$ and $s \in V$

Find the set of nodes $S = \{x \in V \mid s \rightsquigarrow x\}$. Also, for every $x \in S$, find a path from s to x.

This problem can be solved with a tree traversal strategy.

- The most important tree traversal strategies are depth first search (DFS) and breadth first search (BFS).
- Both strategies build a search tree T with root s, with the following properties:
  - The set of nodes in T is $S = \{x \in V \mid s \rightsquigarrow x\}$.
  - For every $x \in S$: the branch from s to x in T is a path from s to x in G.

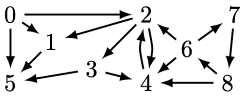
Depth first search from a source node s

- Start by visiting the source node s.
- Visiting a node x is a recursive process:
  1. Mark node x as visited.
  2. Visit recursively all unvisited neighbors of x. Usually, for every unvisited neighbor y that gets visited, we set $\mathrm{p}[y] = x$ to record the fact that graph traversal proceeds from x to y.

```
dfs(G, x)
    visited[x] = true;
    for y ∈ adj[x] do
        if not(visited[y])
            p[y] = x;
            dfs(G, y);
```

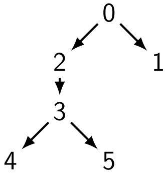
Depth first search from a source node s

Illustrated example



$\mathtt{adj}[0] = [2, 1, 5]$, $\mathtt{adj}[1] = [5]$, $\mathtt{adj}[2] = [3, 1, 4]$,
$\mathtt{adj}[3] = [4, 5]$, $\mathtt{adj}[4] = [2]$, $\mathtt{adj}[5] = []$,
$\mathtt{adj}[6] = [4, 2]$, $\mathtt{adj}[7] = [8]$, $\mathtt{adj}[8] = [6]$.

DFS from node 0 yields the depth first search tree

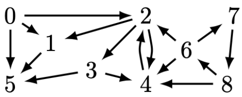


Paths from source node 0:

[0], [0, 2], [0, 2, 3], [0, 2, 3, 4], [0, 2, 3, 5], [0, 1]

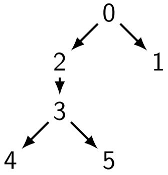
Depth first search from a source node s

Illustrated example



$\texttt{adj}[0] = [2, 1, 5]$, $\texttt{adj}[1] = [5]$, $\texttt{adj}[2] = [3, 1, 4]$,
$\texttt{adj}[3] = [4, 5]$, $\texttt{adj}[4] = [2]$, $\texttt{adj}[5] = []$,
$\texttt{adj}[6] = [4, 2]$, $\texttt{adj}[7] = [8]$, $\texttt{adj}[8] = [6]$.

DFS from node 0 yields the depth first search tree



Paths from source node 0:

$[0], [0, 2], [0, 2, 3], [0, 2, 3, 4], [0, 2, 3, 5], [0, 1]$

Remarks

1. The paths computed by DFS are **not** shortest paths from source node 0.
2. We can compute shortest paths from the source node with BFS (see next slide).

Breadth first search from a source node s

Breadth first traversal from a source node s proceeds in rounds

- In the first round we visit s and mark s as visited.
- In every next round we visit the unvisited nodes of the nodes visited in the previous round.

Breadth first search from a source node s

Breadth first traversal from a source node s proceeds in rounds

- In the first round we visit s and mark s as visited.
- In every next round we visit the unvisited nodes of the nodes visited in the previous round.

BFS can be implemented with a queue where we record the visited nodes in the order in which we will visit their unvisited neighbors.

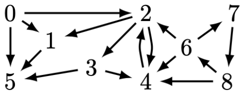
Breadth first search from a source node s

Breadth first traversal from a source node s proceeds in rounds

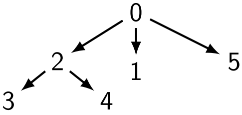
- In the first round we visit s and mark s as visited.
- In every next round we visit the unvisited nodes of the nodes visited in the previous round.

BFS can be implemented with a queue where we record the visited nodes in the order in which we will visit their unvisited neighbors.

```
bfs(G, s)
    visited[s] = true;
    Q :=empty queue;
    add s to Q;
    while nonempty(Q)
            v := pop(Q);
            for w ∈ adj[v]
                 if not(visited[w])
                    p[w] = v;
                    visited[w] = true;
                    add w to Q;
```



$\texttt{adj}[0] = [2,1,5],\ \texttt{adj}[1] = [5],\ \texttt{adj}[2] = [3,1,4],$
$\texttt{adj}[3] = [4,5],\ \texttt{adj}[4] = [2],\ \texttt{adj}[5] = [],$
$\texttt{adj}[6] = [4,2],\ \texttt{adj}[7] = [8],\ \texttt{adj}[8] = [6].$



Remarks

- The paths computed by BFS are shortest paths from the source node.

DFS traversal orders

We can use dfs() to visit all nodes of $G = (V, E)$ and produce a forest of depth first search trees:

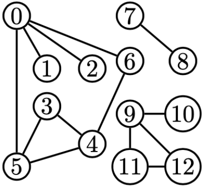
for $s \in V$
    if not($visited[s]$) dfs(G, s)

$\Rightarrow$ we define three DFS traversal orders:

1. Preorder: nodes are added in a queue before the recursive call of dfs(), and assume $x <_{\mathrm{pre}} y$ if x occurs before y in queue.
2. Postorder: nodes are added in a queue after the recursive call of dfs(), and assume $x <_{\mathrm{post}} y$ if x occurs before y in queue.
3. Reverse postorder: we have $x <_{\mathrm{revpost}} y$ if $y <_{\mathrm{post}} x$.

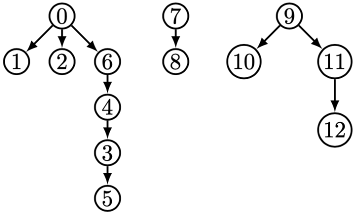
DFS traversal orders

Example



$\texttt{adj}[0] = [1, 2, 5, 6], \texttt{adj}[1] = [0], \texttt{adj}[2] = [0],$
$\texttt{adj}[3] = [4, 5], \texttt{adj}[4] = [3, 5, 6], \texttt{adj}[5] = [0, 3, 4],$
$\texttt{adj}[6] = [0, 4], \texttt{adj}[7] = [8], \texttt{adj}[8] = [7],$
$\texttt{adj}[9] = [10, 11, 12], \texttt{adj}[10] = [9], \texttt{adj}[11] = [9, 12],$
$\texttt{adj}[12] = [9, 11].$

DFS yields a forest of 3 depth first search trees



with the following orderings:

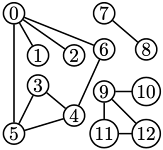
Preorder: [0,1,2,6,4,3,5,7,8,9,10,11,12]
Postorder: [1,2,5,3,4,6,0,8,7,10,12,11,9]
Reverse postorder:
[9,11,12,10,7,8,0,6,4,3,5,2,1]

Applications of DFS

Assumption: $G = (V, E)$ is an undirected graph.

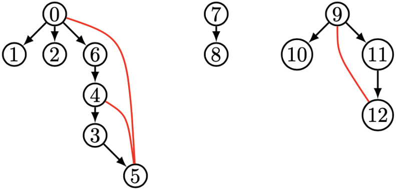
1. Detection of connected components in undirected graphs.
   Main idea: Build a forest of depth first search trees
   - The connected components are the sets of nodes in the individual depth first search trees.
2. Cycle detection in undirected graphs.
   1. Build a forest of depth first search trees.
   2. All edges of G which are not in the forest of depth first search trees, are between a node and a non-parent ancestor.
   3. G has a cycle **iff** there is a DFS tree with an edge between a node and a non-parent predecessor.

See illustrated example on next slide.



$\mathtt{adj}[0] = [1, 2, 5, 6], \mathtt{adj}[1] = [0], \mathtt{adj}[2] = [0],$
$\mathtt{adj}[3] = [4, 5], \mathtt{adj}[4] = [3, 5, 6], \mathtt{adj}[5] = [0, 3, 4],$
$\mathtt{adj}[6] = [0, 4], \mathtt{adj}[7] = [8], \mathtt{adj}[8] = [7],$
$\mathtt{adj}[9] = [10, 11, 12], \mathtt{adj}[10] = [9], \mathtt{adj}[11] = [9, 12],$
$\mathtt{adj}[12] = [9, 11].$

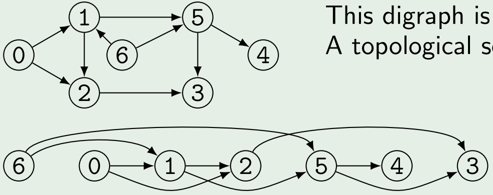
The forest of trees produced by DFS is



- The red-colored edges indicate cycles in G.

- A directed acyclic graph, or DAG, is a digraph $G = (V, E)$ without cycles.
- A topological sort of a DAG is an enumeration $[x_1, x_2, \ldots, x_n]$ of all nodes in G such that all arcs in E are of the form $x_i \to x_j$ with $1 \leq i < j \leq n$.

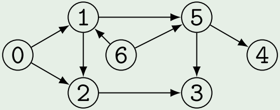
Example



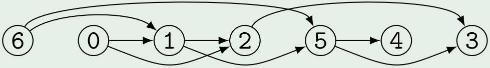
This digraph is a DAG.
A topological sort is $[6, 0, 1, 2, 5, 4, 3]$

- A directed acyclic graph, or DAG, is a digraph $G = (V, E)$ without cycles.
- A topological sort of a DAG is an enumeration $[x_1, x_2, \ldots, x_n]$ of all nodes in G such that all arcs in E are of the form $x_i \rightarrow x_j$ with $1 \leq i < j \leq n$.

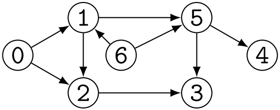
Example



This digraph is a DAG.
A topological sort is $[6, 0, 1, 2, 5, 4, 3]$

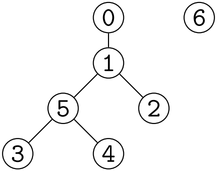


REMARK: For a DAG $G = (V, E)$, the nodes of V listed in reverse postorder are a topological sort of G.



$\texttt{adj}[0] = [1, 2]$, $\texttt{adj}[1] = [5, 2]$, $\texttt{adj}[2] = [3]$,
$\texttt{adj}[3] = \texttt{adj}[4] = []$, $\texttt{adj}[5] = [3, 4]$,
$\texttt{adj}[6] = [1, 5]$

The forest of depth first search trees of this digraph is



Postorder: $[3, 4, 5, 2, 1, 0, 6]$.
Reverse postorder: $[6, 0, 1, 2, 5, 4, 3]$.

Given a digraph $G = (V, E)$

Find the strongly connected components of G.

Given a digraph $G = (V, E)$

Find the strongly connected components of G.

This problem can be solved with Kosaraju's algorithm:

Given a digraph $G = (V, E)$

Find the strongly connected components of G.

This problem can be solved with Kosaraju's algorithm:

1. Compute the reverse digraph $G^r = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}$.

Given a digraph $G = (V, E)$

Find the strongly connected components of G.

This problem can be solved with Kosaraju's algorithm:

1. Compute the reverse digraph $G^r = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}$.
2. Let $[x_1, x_2, \ldots, x_n]$ be the enumeration of the nodes of G^r in the reverse postorder of DFS of G^r

Given a digraph $G = (V, E)$

Find the strongly connected components of G.

This problem can be solved with Kosaraju's algorithm:

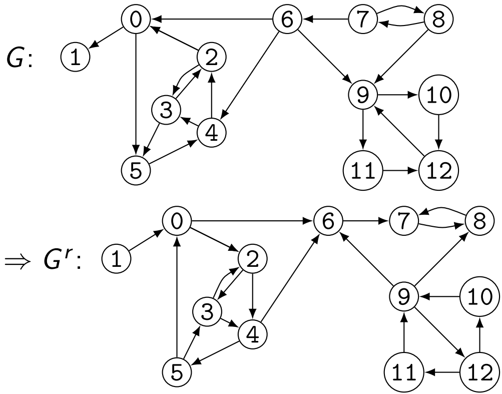
1. Compute the reverse digraph $G^r = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}$.
2. Let $[x_1, x_2, \ldots, x_n]$ be the enumeration of the nodes of G^r in the reverse postorder of DFS of G^r
3. Let $T_1, \ldots, T_r$ be the forest of depth-first traversal trees of G produced by visiting the unvisited nodes on G in the order $[x_1, x_2, \ldots, x_n]$.

Given a digraph $G = (V, E)$

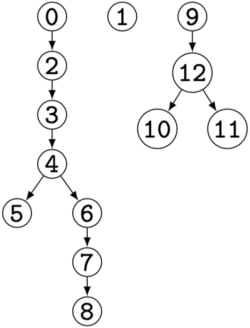
Find the strongly connected components of G.

This problem can be solved with Kosaraju's algorithm:

1. Compute the reverse digraph $G^r = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}$.
2. Let $[x_1, x_2, \ldots, x_n]$ be the enumeration of the nodes of G^r in the reverse postorder of DFS of G^r
3. Let $T_1, \ldots, T_r$ be the forest of depth-first traversal trees of G produced by visiting the unvisited nodes on G in the order $[x_1, x_2, \ldots, x_n]$.
4. The strongly connected components of G are the sets of nodes of the trees T_1, T_2, ..., T_r

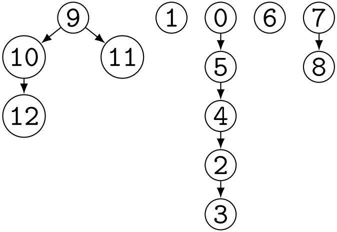

G:
0
1
2
3
4
5
6
7
8
9
10
11
12
⇒ G^r:
0
1
2
3
4
5
6
7
8
9
10
11
12

DFS of the nodes of G^r with nodes ordered by $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$ yields the forest of trees



$\Rightarrow$ reverse postorder $[9, 12, 11, 10, 1,\ 0, 2, 3, 4, 6, 7, 8, 5]$.

3. DFS of the nodes of G with nodes ordered by $[9, 12, 11, 10, 1, 0, 2, 3, 4, 6, 7, 8, 5]$ yields the forest of trees



4. We conclude that the strongly connected components of G are $\{9, 10, 11, 12\}$, $\{1\}$, $\{0, 2, 3, 4, 5\}$, $\{6\}$ and $\{7, 8\}$.
   - The strongly connected components of G are illustrated on the next slide.

Detection of strongly connected components

Application 4: Kosaraju's algorithm – illustration of the final result