Graphs and Graphs Traversal

Arash Rafiey

Graph (Basic Definition)

Graph : Represents a way of encoding pairwise relationships among a set of objects.

Graph G consists of a collection V of *nodes* and a collection E of *edges*, each of which joins two of the nodes.

$E(G) \subseteq \{\{u, v\} | u, v \in V(G)\}$

If the relation is not symmetric (directed graph) we have $E(G) \subseteq \{(u, v) | u, v \in V(G)\}$.

1. Transportation networks (airline carrier, airports as node and direct flights as edges (direct edge).
2. Communication networks (a collection of computers as nodes and the physical link between them as edges).
3. Information networks (World Wide Web can be viewed as directed graph, the Web pages are nodes and the hyperlink between the pages are directed edges).
4. Social Network (People are nodes and friendship is an edge).

Let G be a graph. For simplicity instead of edge $\{u, v\}$ we write edge uv.

Two vertices u and v are called *adjacent* if uv is an edge of G.

We say v is a *neighbor* of u if uv is an edge of G.

Let $G = (V, E)$ be an undirected graph.

A path is a sequence P of nodes $v_0, v_1, \ldots, v_{k-2}, v_{k-1}$ with the property that $v_i v_{i+1}$ is an edge of G and $v_i \neq v_j$, $0 \leq i \leq k-2$, $i \neq j$.

The length of P is $k - 1$.

Let $G = (V, E)$ be an undirected graph.

A path is a sequence P of nodes $v_0, v_1, \ldots, v_{k-2}, v_{k-1}$ with the property that $v_i v_{i+1}$ is an edge of G and $v_i \neq v_j$, $0 \leq i \leq k-2$, $i \neq j$.

The length of P is $k - 1$.

A cycle is a sequence C of nodes $v_1, v_2, \ldots, v_{k-1}, v_k, v_1$ with the property that $v_i v_{i+1}$ (sum module k) is an edge of G and $v_i \neq v_j$, $0 \leq i \leq k-2$, $i \neq j$.

The length of C is k.

In the directed path and directed cycle, each pair of consecutive nodes (v_i, v_{i+1}) is a directed edge, i.e. $v_i v_{i+1}$ is an *arc* .

A walk is an alternating sequence of nodes and edges, beginning and ending with a node.

A walk is *closed* if its first and last nodes are the same.

A trail is a walk in which all the edges are distinct.

A path is a *simple* walk (no two nodes repeated).

We say a path P in graph G is Hamiltonian if it goes through all the nodes.

We say a cycle C in graph G is Hamiltonian if it goes through all the nodes (starts from one nodes and goes through all the other nodes and come back to the same node).

Connectivity

We say (undirected) graph G is connected if, for every pair of nodes u and v, there is a path from u to v.

We say digraph D is strongly connected if for every pair of nodes u, v there is a directed path from u to v and there is a directed path from v to u.

A directed cycle is a strong digraph.

Distance between two nodes u, v, $dis(u, v)$ is the length of the shortest path between u and v.

For a vertex u, let $dis_{max}(u)$ denotes the *maximum* distance from vertex u.

The diameter of G is the :

$$\min_{v \in G} dis_{max}(v)$$

Let $G = (V, E)$ be a graph. The degree of node v, $d(v)$ is the number of neighbors of v in G.

We have :

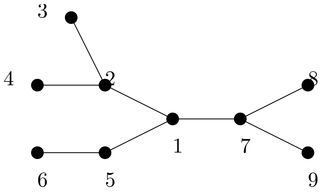
$$\sum_{v \in V} d(v) = 2|E|$$

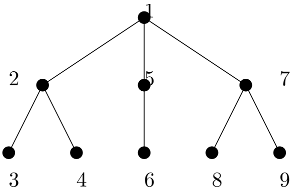
Let $D = (V, A)$ be a digraph. The outdegree of vertex v, $d^+(v)$ is the number of arcs going out of v and similarly the indegree of v, $d^-(v)$ is the number of arcs going out of v.

We have :

$$\sum_{v \in V} d^+(v) = \sum_{u \in V} d^-(u) = |E|$$

A tree is a connected graph that has no cycle. A tree with n nodes has exactly $n-1$ edges.
We usually consider a node as a root and the rest of the nodes hag downward from the root. The nodes that are at the end (have only one neighbor) are called leaves.





Two drawings of the same tree. On the right, the tree is rooted at node 1

A leaf in a tree is a vertex with degree 1.

Each tree T has at least two leaves.

A leaf in a tree is a vertex with degree 1.

Each tree T has at least two leaves.

A forest in a graph consist of a collection of trees.

A leaf in a tree is a vertex with degree 1.

Each tree T has at least two leaves.

A forest in a graph consist of a collection of trees.

If a forest F has k trees and n nodes then it has $n - k$ edges.

s-t connectivity and Graph Traversal

Given a graph G and two nodes s, t. The $s - t$ connectivity problem asks whether there is a path from s to t.

Breadth-First Search (BFS). Start with node s and set $L_0 = s$.

At step i let L_i be the set of nodes that are not in any of

$L_0, L_1, \ldots, L_{i-1}$ and have a neighbor in L_{i-1}.

Lemma

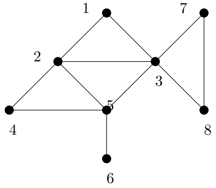
L_j is the set of nodes that are at distance exactly j from s.

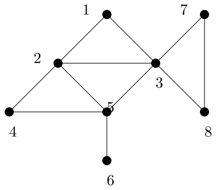
The BFS algorithm creates a tree with root s.

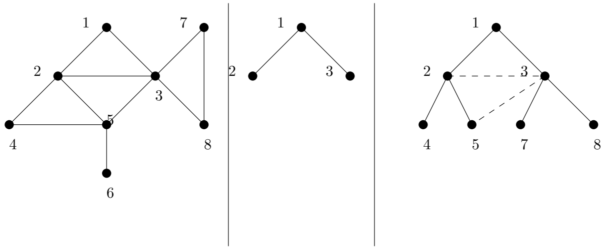
Once a node v is discovered by BFS algorithm we put an edge from v to all the nodes u that have not been considered. This way v is set to be the father of node u.

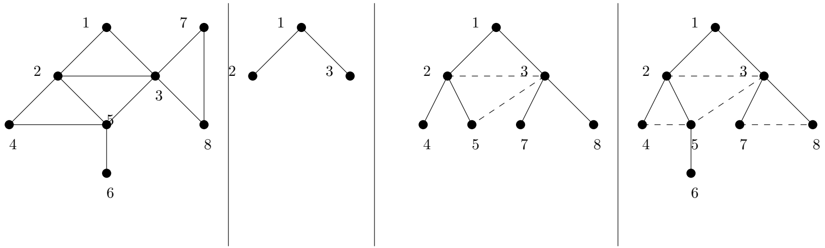
BFS (s)

1. Set Discover[s]=true and Discover[v]=false for all other v
2. Set $L[0] = \{s\}$
3. Set layer counter i=0
4. Set $T = \emptyset$
4. While $L[i]$ is not empty
5. Initialize empty set $L[i+1]$
6. For each node $u \in L[i]$
7. Consider each edge uv
8. If $Discover[v]$ = false then
9. Set $Discover[v]$ = true
10. Add edge uv to T
11. Add v to the list $L[i+1]$
12. Increase i by one


1
7
2
3
5
4
8
6


1
7
2
3
5
4
8
6


1
7
2
3
4
5
8
6
1
2
3
1
2
3
4
5
7
8


1
7
2
3
4
5
8
6
1
2
3
1
2
3
4
5
7
8
1
2
3
4
5
7
8
6

BFS running time

1) If we represent the graph G by adjacency matrix then the running time of BFS algorithm is $O(n^2)$, where n is the number of nodes.

2) If we represent the graph G by link lists then the running time of BFS algorithm is $O(m+n)$, where m is the number of edges and n is the number of nodes.

Bipartite Testing

Problem : Given a graph G decide whether G is bipartite or not.

A graph G is bipartite iff G does not contain an odd cycle.

Problem : Given a graph G decide whether G is bipartite or not.

A graph G is bipartite iff G does not contain an odd cycle.

Solution (Using BFS)
Start with node s and color it with red. Next color the neighbors of s by blue. Next color the neighbors of neighbors of s by red and so on.
If at the end there is an edge whose end points receive the same color G is not bipartite.

Bipartite Testing

Problem : Given a graph G decide whether G is bipartite or not.

A graph G is bipartite iff G does not contain an odd cycle.

Solution (Using BFS)

Start with node s and color it with red. Next color the neighbors of s by blue. Next color the neighbors of neighbors of s by red and so on.

If at the end there is an edge whose end points receive the same color G is not bipartite.

This is essentially is the BFS algorithm. We color the nodes in L_0 by red and the nodes in L_1 by blue and the nodes in L_3 by red and so on.

Next we read each edge uv of G. If both u, v have the same color then G is not bipartite. Otherwise G is bipartite.

Lemma

Let G be a connected graph, and let $L_0, L_1, L_2, \ldots, L_k$ be the layers produced by BFS algorithm starting at node s.

(i) *There is no edge of G joining two nodes of the same layer. In this case G is bipartite and $L_0, L_2, \ldots, L_{2i}$ can be colored red and the nodes in odd layers can be colored blue.*

(ii) *There is an edge of G joining two nodes of the same layer. In this case G contains an odd cycle and G is not bipartite.*

Proof :

Suppose (i) happens. In this case the red nodes and blue nodes give a bipartition, and all the edges of G are between the red and blue nodes.

Suppose (ii) happens. Suppose $x, y \in L_j$ and $xy \in E(G)$.

1) By definition there is a path P from s to x of length j and there is a path Q from s to y of length j.

Suppose (ii) happens. Suppose $x, y \in L_j$ and $xy \in E(G)$.

1) By definition there is a path P from s to x of length j and there is a path Q from s to y of length j.

2) Let i be the maximum index such that there is $z \in L(i)$ and z is in the intersection of P and Q, i.e. $z \in P \cap Q$ and $z \in L(i)$.

Suppose (ii) happens. Suppose $x, y \in L_j$ and $xy \in E(G)$.

1) By definition there is a path P from s to x of length j and there is a path Q from s to y of length j.

2) Let i be the maximum index such that there is $z \in L(i)$ and z is in the intersection of P and Q, i.e. $z \in P \cap Q$ and $z \in L(i)$.

3) Portion of P, say P' from z to x has length $j - i$ and portion of Q, say Q' from z to y has length $j - i$.

4) By adding xy edges into P', Q' we get a cycle of length $(j - i) + 1 + (j - i)$ which is of odd length.

Depth-First Search (backtracking approach)

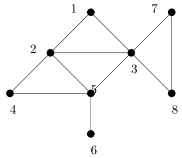
We don't visit the nodes level by level! As long as there is an unvisited node adjacent to the current visited node we continue! Once we are stuck, trace back and go to a different branch!

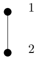
Depth-First Search (backtracking approach)

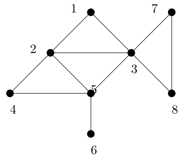
We don't visit the nodes level by level! As long as there is an unvisited node adjacent to the current visited node we continue! Once we are stuck, trace back and go to a different branch!

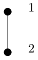
DFS (u)

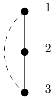
1. Mark u as Explored and add u to R
2. For every edge uv
3. If v is not Explored then call DFS (v)

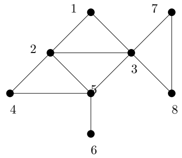

1
7
2
3
5
4
8
6

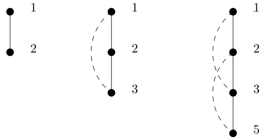

1
2

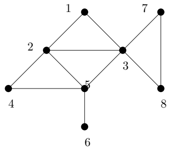

1
7
2
3
5
4
8
6

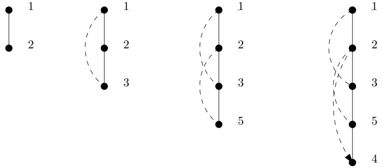

1
2

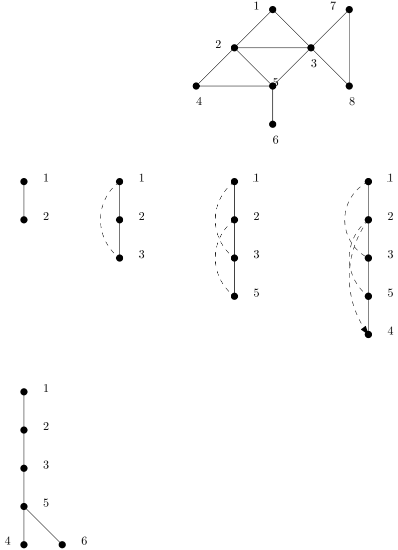

1
2
3

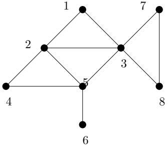

1
7
2
3
5
4
8
6

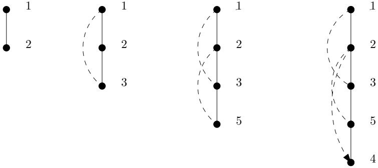

1
2
1
2
3
1
2
3
5

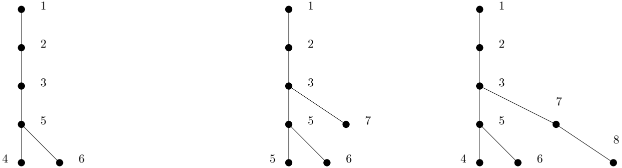

1
7
2
3
5
4
8
6


1
2
1
2
3
1
2
3
5
1
2
3
5
4


1 7
2 3
4 5 8
6
1 2
1 2 3
1 2 3 5
1 2 3 5 4
1 2 3 5 4 6


1
7
2
3
5
4
8
6





DFS(s)

1.Initialize S to be a stack with element s only.

2. While S is not empty

3. Take a node u from top of S.

4. If Explored[u] $=$false then

5. Set Explored[u]$=$true

6. For every uv edge add v to S.

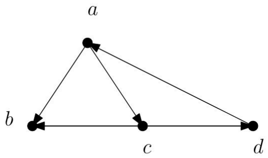
DFS running time

1) If we represent the graph G by adjacency matrix then the running time of DSF algorithm is $O(n^2)$, where n is the number of nodes.

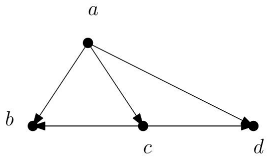
2) If we represent the graph G by link lists then the running time of DFS algorithm is $O(m+n)$, where m is the number of edges and n is the number of nodes.

Acyclic Digraphs and Topological Ordering

A digraph D is acyclic if it does not contain any directed cycle. D is called DAG (directed acyclic graph).



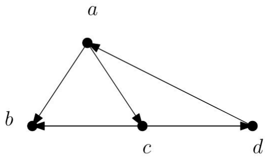
Not a DAG because there is a,c,d,a cycle



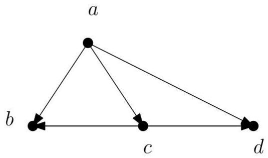
DAG, no directed cycle

Acyclic Digraphs and Topological Ordering

A digraph D is acyclic if it does not contain any directed cycle. D is called DAG (directed acyclic graph).



Not a DAG because there is a,c,d,a cycle



DAG, no directed cycle

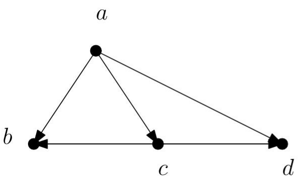
DAG can be used to model the job scheduling with precedence constraint.

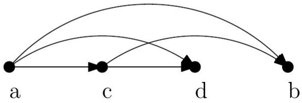
Suppose we want to schedule a set of jobs $\{J_1, J_2, \ldots, J_n\}$ with some dependencies between them (precedence constraint). For certain pair i, j, job J_i must be executed before job J_j.

We want to find an ordering of the jobs respecting the precedence constraints.

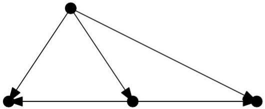
Topological ordering

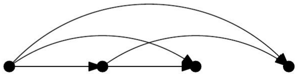
Let D be a digraph. We say an ordering $v_1, v_2, \ldots, v_n$ of the nodes in D is a topological ordering if whenever $v_i v_j$ is an arc of D, $i < j$. In other words, all the arcs are forward and there is no backward arc.





Let D be a digraph. We say an ordering $v_1, v_2, \ldots, v_n$ of the nodes in D is a topological ordering if whenever $v_i v_j$ is an arc of D, $i < j$. In other words, all the arcs are forward and there is no backward arc.





Lemma

Let D be an acyclic digraph. Then D has a node without in-degree.

Topological ordering

Theorem

Let D be a digraph. D has a topological ordering if and only if D is acyclic.

Proof.

If D contains a directed cycle C then in every ordering of the vertices of C at least one arc is backward. Conversely if D is acyclic we show that there is a topological ordering. We follow AC-Order algorithm.

AC-Order(D)

1.**If** D is empty then return.

2.**Else** Let v be a node without in neighbor in D.

3. Printout v.

4. Call AC-Order($D - v$)

AC-Order(D)

1. Initial queue Q to be empty.

2. For every node v set the $Indegree[v]$ to be the number of nodes having arc to v.

3. For every vertex v, If $(Indegree[v] = 0\ )\ \{\ Q.add(v);\ \}$

4. While Q is not empty

5. $\quad u = Q.delete();$

6. $\quad$ Printout(u);

7. $\quad$ For every arc $uw \in A(D)$

8. $\qquad Indegree[w] = Indegree[w] - 1;$

9. $\qquad$ If $(Indeegree[w] = 0)$

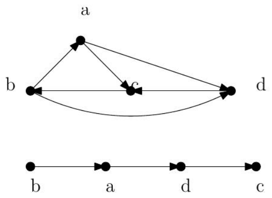
10 $\qquad\quad Q.add(w);$

Show that the AC-Order(D) algorithm runs in time $O(|D| + |E|)$.

A digraph T is called tournament if for every two nodes u, v of exactly one of the uv, vu is an arc in T.

We say u wins v if uv is an arc.

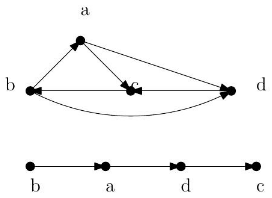
Problem 1: Show that in every tournament we can find an ordering $v_1, v_2, \ldots, v_n$ of the nodes such that v_i wins v_{i+1}, (for every $1 \leq i \leq n-1$).



A digraph T is called tournament if for every two nodes u, v of exactly one of the uv, vu is an arc in T.

We say u wins v if uv is an arc.

Problem 1: Show that in every tournament we can find an ordering $v_1, v_2, \ldots, v_n$ of the nodes such that v_i wins v_{i+1}, (for every $1 \leq i \leq n-1$).



Problem 2: Show that if a tournament is NOT acyclic (it has a directed cycle) then it has a directed cycle of length three.

Finding a longest path in a DAG

Let D be an acyclic digraph where each arc has a non-negative weight. For every node u of D compute a longest path (longest weighted path) from s to u.

Longest-Path-Algorithm(D, s)

1. Initial queue Q to be empty. For every nodes u, v if uv is not an arc then set $A[u][v] = 0$ otherwise $A[u][v] = 1$.

2. For every node v set the $Indegree[v]$ to be the number of nodes having arc to v.

3. For every node u set $\ell d(u) = 0$. // initially longest path ending at u has zero length.

4. For every node v, If $(Indegree[v] = 0)$ $\{$ $Q.add(v);$ $\}$

5. While Q is not empty

6. $\quad u = Q.delete();$

7. $\quad$ For every arc $uw \in A(D)$

8. $\quad\quad Indegree[w] = Indegree[w] - 1;$

9. $\quad\quad$ If $(\ell d(u) + A[u][w] > \ell d(w))$

10. $\quad\quad\quad \ell d(w) = \ell d(u) + A[u][w]$

11. $\quad\quad$ If $(Indegree[w] == 0)$

12. $\quad\quad\quad Q.add(w);$

Strong Digraph

A digraph D is called strong, if for every two vertices u, v of D there is a directed path from u to v and there is a directed path from v to u.

Strong Digraph

A digraph D is called strong, if for every two vertices u, v of D there is a directed path from u to v and there is a directed path from v to u.

A directed cycle is a strong digraph.

Strong Digraph

A digraph D is called strong, if for every two vertices u, v of D there is a directed path from u to v and there is a directed path from v to u.

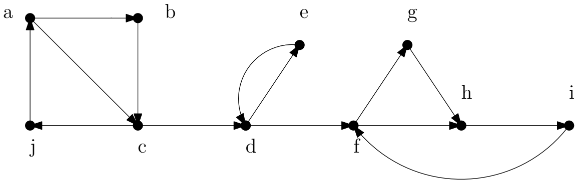
A directed cycle is a strong digraph.

A strong component of D is a maximal subset U of D which is strong.

A digraph D is called strong, if for every two vertices u, v of D there is a directed path from u to v and there is a directed path from v to u.

A directed cycle is a strong digraph.

A strong component of D is a maximal subset U of D which is strong.



Strong components are : $S_1 = \{a, b, c, j\}$, $S_2 = \{d, e\}$ and $S_3 = \{f, g, h, i\}$

Finding strong components in a digraph

Input: digraph $G = (V, E)$

Output: set of strongly connected components (sets of vertices)

Finding strong components in a digraph

Input: digraph $G = (V, E)$

Output: set of strongly connected components (sets of vertices)

Explaining Tarjan's algorithm :

1) The nodes are placed on a stack in the order in which they are visited.

Finding strong components in a digraph

Input: digraph $G = (V, E)$

Output: set of strongly connected components (sets of vertices)

Explaining Tarjan's algorithm :

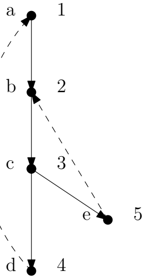
1) The nodes are placed on a stack in the order in which they are visited.

2) When the depth-first search recursively explores a node v and its descendants, we may not popped them from the stack. Because there maybe a node v descendant of u which may have a path to a node earlier on the stack.

3) Each node v is assigned a unique integer $index(v)$, The time when v is visited for the first time.

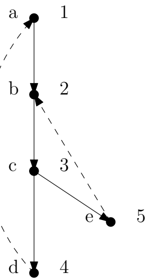
3) Each node v is assigned a unique integer $\mathit{index}(v)$, The time when v is visited for the first time.

4) We maintain a value $\mathit{lowlink}(v)$ that represents (roughly speaking) the smallest index of any node known to be reachable from v, including v itself.

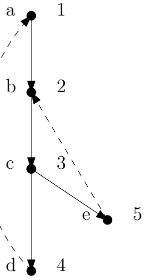


3) Each node v is assigned a unique integer $index(v)$, The time when v is visited for the first time.

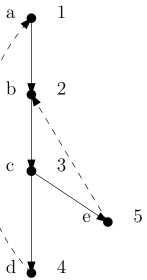
4) We maintain a value $lowlink(v)$ that represents (roughly speaking) the smallest index of any node known to be reachable from v, including v itself.



5) Therefore v must be left on the stack if $lowlink(v) < index(v)$



6) v must be removed as the root of a strongly connected component if $lowlink(v) = index(v)$.



6) v must be removed as the root of a strongly connected component if $lowlink(v) = index(v)$.

7) The value $lowlink(v)$ is computed during the depth-first search from v.

Strong Components-Tarjan's Algorithm(D)

function strongconnect(v)

```
1.  index(v) := index;    lowlink(v) := index;
2.  index:= index + 1;    Stack.push(v)
3.  for each arc vw ∈ E do
6.      if (index(w) is undefined)
7.        strongconnect(w)
8.        lowlink(v) :=min(lowlink(v), lowlink(w))
9.      else if (w is in Stack)
10.       lowlink(v) := min(lowlink(v), index(w))
// If v is a root node, pop the stack for new strong component
11. if (lowlink(v) = index(v))
12.     repeat
13.       w := Stack.pop()    add w to current strong component
14.     until (w = v)
15.     output the current strongly component
```

Strong Components-Tarjan's Algorithm(D)

```
1. index := 0
2. Stack := empty
3. for each v in V do
4.     if (index(v) is undefined)
5.        strongconnect(v)
```