Algorithms 2024: Advanced Graph Algorithms

(Based on [Manber 1989])

Yih-Kuen Tsay

November 11, 2024

1 Strongly Connected Components

Strongly Connected Components

- A directed graph is *strongly connected* if there is a directed path from every vertex to every other vertex.
- A *strongly connected component* (SCC) is a maximal subset of the vertices such that its induced subgraph is strongly connected (namely, there is no other subset that contains it and induces a strongly connected graph).

Strongly Connected Components (cont.)

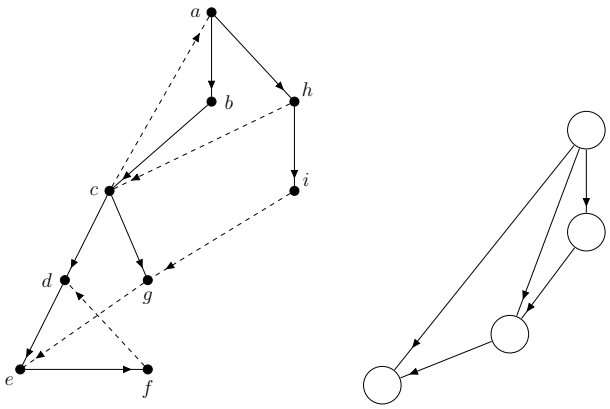


Figure: A directed graph and its strongly connected component graph.
Source: redrawn from [Manber 1989, Figure 7.30].

/* The strongly connected component graph (SCC graph) is a directed acyclic graph (DAG). There is an edge from one SCC to another if in the original graph there is an edge from some vertex in the first SCC to some vertex in the second SCC.

Look closely at the graph and its SCC graph above and ask yourself a question: if you start a DFS anywhere in the graph, which SCC will you first completely exhaust and leave forever (without returning)? */

Strongly Connected Components (cont.)

Lemma 1 (7.11)**.** *Two distinct vertices belong to the same SCC if and only if there is a circuit containing both of them.*

/* An important application of this lemma is that, using a DFS, one vertex will find out that it can reach either the other vertex eventually via a back edge, which indicates the existence of a directed cycle, or some other vertex on the cycle. This property will be untilized in the algorithm we will study later for determining the SCCs of a graph. */

Lemma 2 (7.12)**.** *Each vertex belongs to exactly one SCC.*

/* All the SCCs of a graph form a partition of the set of vertices of the graph. */

Strongly Connected Components (cont.)

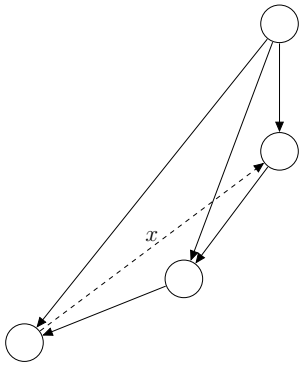


Figure: Adding an edge connecting two different strongly connected components.
Source: redrawn from [Manber 1989, Figure 7.31].

/* An induction on the edges goes by considering the edges one by one to see how they may change the structure of the SCC graph observed so far. DFS turns out to provide a convenient ordering of the edges (that is, the induction sequence). */

Strongly Connected Components (cont.)

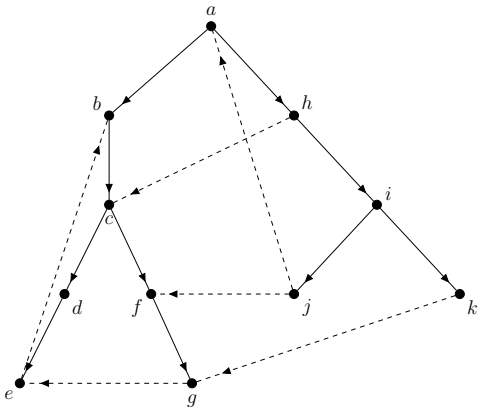


Figure: The effect of cross edges.
Source: redrawn from [Manber 1989, Figure 7.32].

/* A cross edge may point to a vertex in the same SCC under exploration or another SCC that has already been identified. */

Strongly Connected Components (cont.)

Algorithm Strongly_Connected_Components(G, n);
begin
  for every vertex v of G **do**
    $v.DFS\_Number := 0$;
    $v.Component := 0$;
  $Current\_Component := 0$; $DFS\_N := n$;
  while $v.DFS\_Number = 0$ for some v **do**
    $SCC(v)$
end

procedure SCC(v);
begin
  $v.DFS\_Number := DFS\_N$;
  $DFS\_N := DFS\_N - 1$;
  insert v into $Stack$;
  $v.High := v.DFS\_Number$;

Strongly Connected Components (cont.)

  for all edges (v, w) **do**
    if $w.DFS\_Number = 0$ **then**
      $SCC(w)$;
      $v.High := \max(v.High, w.High)$
    else if $w.DFS\_Number > v.DFS\_Number$
        and $w.Component = 0$ **then**
      $v.High := \max(v.High, w.DFS\_Number)$
      // $\max(v.High, w.High)$ also works
  if $v.High = v.DFS\_Number$ **then**
    $Current\_Component := Current\_Component + 1$;
    repeat
      remove x from the top of $Stack$;
      $x.component := Current\_Component$
    until $x = v$
end

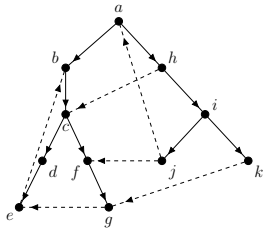
Time complexity: $O(|E| + |V|)$.

/* This is essentially a DFS with constant extra work per vertex. */

/* For an arbitrary SCC, the vertex v that is visited first (the "leader" of the SCC) during the DFS will acquire the largest/highest DFS number among all the vertices in the same SCC. Every vertex in the SCC reports the largest number it may reach to its parent (according to the DFS tree); in the actual code, it is the parent vertex who collects the information by reading the $High$ value of its child when the corresponding recursive call returns. When the recursive call with v as the input is about to return, v will see that $v.High = v.DFS\_Number$ and thus find itself the leader of an SCC.

Due to the nature of DFS, the deeper an SCC is in the SCC graph (which is a DAG), the later it will be visited/touched but the earlier it will be determined (i.e., all vertices in the SCC exhausted and assigned a component number). */

Strongly Connected Components (cont.)



	a	b	c	d	e	f	g	h	i	j	k
	11	10	9	8	7	6	5	4	3	2	1
a	11	-	-	-	-	-	-	-	-	-	-
b	11	10	-	-	-	-	-	-	-	-	-
c	11	10	9	-	-	-	-	-	-	-	-
d	11	10	9	8	-	-	-	-	-	-	-
e	11	10	9	8	10	-	-	-	-	-	-
d	11	10	9	10	10	-	-	-	-	-	-
c	11	10	10	10	10	-	-	-	-	-	-
f	11	10	10	10	10	6	-	-	-	-	-
g	11	10	10	10	10	6	7	-	-	-	-
f	11	10	10	10	10	7	7	-	-	-	-
c	11	10	10	10	10	7	7	-	-	-	-
(b)	11	10	10	10	10	7	7	-	-	-	-
a	11	10	10	10	10	7	7	-	-	-	-
h	11	10	10	10	10	7	7	4	-	-	-
i	11	10	10	10	10	7	7	4	3	-	-
j	11	10	10	10	10	7	7	4	3	11	-
i	11	10	10	10	10	7	7	4	11	11	-
(k)	11	10	10	10	10	7	7	4	11	11	1
i	11	10	10	10	10	7	7	4	11	11	1
h	11	10	10	10	10	7	7	11	11	11	1
(a)	11	10	10	10	10	7	7	11	11	11	1

Figure: An example of computing *High* values and strongly connected components.
Source: redrawn from [Manber 1989, Figure 7.34].

Odd-Length Cycles

Problem 3. *Given a directed graph $G = (V, E)$, determine whether it contains a (directed) cycle of odd length.*

- A cycle must reside completely within a strongly connected component (SCC), so we exam each SCC separately.
- Mark the nodes of an SCC with "even" or "odd" using DFS.
- If we have to mark a node that is already marked in the opposite, then we have found an odd-length cycle.

2 Biconnected Components

Biconnected Components

- An undirected graph is *biconnected* if there are at least two vertex-disjoint paths from every vertex to every other vertex.
- A graph is *not* biconnected if and only if there is a vertex whose removal disconnects the graph. Such a vertex is called an *articulation point*.
- A *biconnected component* (BCC) is a *maximal* subset of the edges such that its induced subgraph is biconnected (namely, there is no other subset that contains it and induces a biconnected graph).

Biconnected Components (cont.)

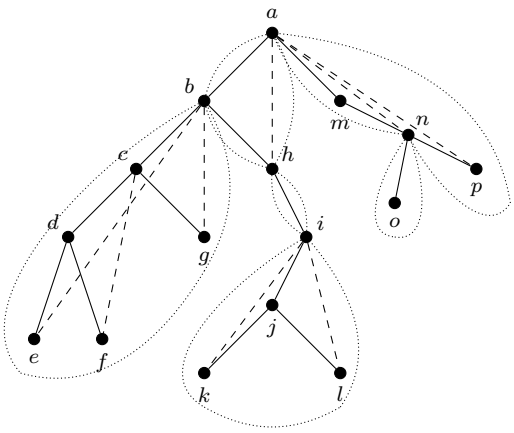


Figure: The structure of a nonbiconnected graph.
Source: redrawn from [Manber 1989, Figure 7.25].

Biconnected Components (cont.)

Lemma 4 (7.9)**.** *Two distinct edges e and f belong to the same BCC if and only if there is a cycle containing both of them.*

Lemma 5 (7.10)**.** *Each edge belongs to exactly one BCC.*

Biconnected Components (cont.)

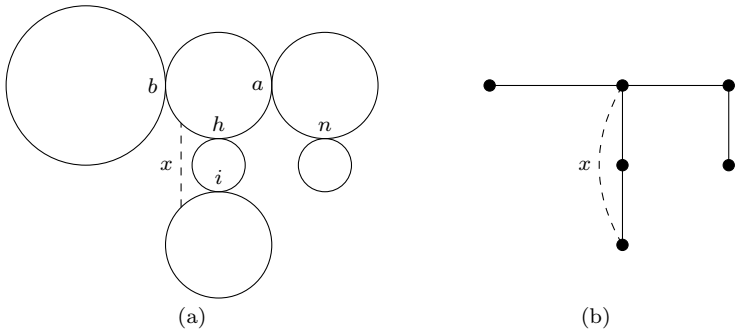


Figure: An edge that connects two different biconnected components. (a) The components corresponding to the graph of Figure 7.25 with the articulation points indicated. (b) The biconnected component tree.
Source: redrawn from [Manber 1989, Figure 7.26].

Biconnected Components (cont.)

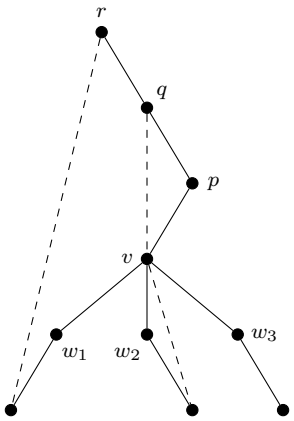


Figure: Computing the *High* values.
Source: redrawn from [Manber 1989, Figure 7.27].

Biconnected Components (cont.)

Algorithm Biconnected_Components(G, v, n);
begin
  for every vertex w **do** $w.DFS\_Number := 0$;
  $DFS\_N := n$;
  $BC(v)$
end

procedure BC(v);
begin
  $v.DFS\_Number := DFS\_N$;
  $DFS\_N := DFS\_N - 1$;
  insert v into *Stack*;
  $v.High := v.DFS\_Number$;

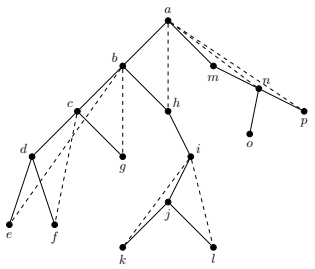
Biconnected Components (cont.)

  for all edges (v, w) **do**
    insert (v, w) into *Stack*;
    if w is not the parent of v **then**
      if $w.DFS\_Number = 0$ **then**
        $BC(w)$;
        if $w.High \leq v.DFS\_Number$ **then**
          remove all edges and vertices
            from *Stack* until v is reached;
          insert v back into *Stack*;
        $v.High := \max(v.High, w.High)$
      else
        $v.High := \max(v.High, w.DFS\_Number)$
        // $\max(v.High, w.High)$ would not work, unlike in SCC
end

Biconnected Components (cont.)

procedure BC(v);
begin
  $v.DFS\_Number := DFS\_N$;
  $DFS\_N := DFS\_N - 1$;
  $v.High := v.DFS\_Number$;
  for all edges (v, w) **do**
    if w is not the parent of v **then**
      insert (v, w) into $Stack$;
      if $w.DFS\_Number = 0$ **then**
        $BC(w)$;
        if $w.high \leq v.DFS\_Number$ **then**
          remove all edges from $Stack$
            until (v, w) is reached;
        $v.High := \max(v.High, w.High)$
      else
        $v.High := \max(v.High, w.DFS\_Number)$
end

Biconnected Components (cont.)



	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
a	16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
b	16	15	-	-	-	-	-	-	-	-	-	-	-	-	-	-
c	16	15	14	-	-	-	-	-	-	-	-	-	-	-	-	-
d	16	15	14	13	-	-	-	-	-	-	-	-	-	-	-	-
e	16	15	14	13	15	-	-	-	-	-	-	-	-	-	-	-
d	16	15	14	15	15	-	-	-	-	-	-	-	-	-	-	-
f	16	15	14	15	15	14	-	-	-	-	-	-	-	-	-	-
d	16	15	14	15	15	14	-	-	-	-	-	-	-	-	-	-
e	16	15	15	15	15	14	-	-	-	-	-	-	-	-	-	-
g	16	15	15	15	15	14	15	-	-	-	-	-	-	-	-	-
c	16	15	15	15	15	14	15	-	-	-	-	-	-	-	-	-
ⓑ	16	15	15	15	15	14	15	-	-	-	-	-	-	-	-	-
h	16	15	15	15	15	14	15	16	-	-	-	-	-	-	-	-
i	16	15	15	15	15	14	15	16	8	-	-	-	-	-	-	-
j	16	15	15	15	15	14	15	16	8	7	-	-	-	-	-	-
k	16	15	15	15	15	14	15	16	8	7	8	-	-	-	-	-
j	16	15	15	15	15	14	15	16	8	8	8	-	-	-	-	-
l	16	15	15	15	15	14	15	16	8	8	8	8	-	-	-	-
j	16	15	15	15	15	14	15	16	8	8	8	8	-	-	-	-
ⓘ	16	15	15	15	15	14	15	16	8	8	8	8	-	-	-	-
ⓑ	16	15	15	15	15	14	15	16	8	8	8	8	-	-	-	-
b	16	16	15	15	15	14	15	16	8	8	8	8	-	-	-	-
ⓐ	16	16	15	15	15	14	15	16	8	8	8	8	-	-	-	-
m	16	16	15	15	15	14	15	16	8	8	8	8	4	-	-	-
n	16	16	15	15	15	14	15	16	8	8	8	8	4	16	-	-
o	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	-
ⓝ	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	-
p	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	16
n	16	16	15	15	15	14	15	16	8	8	8	8	4	16	2	16
m	16	16	15	15	15	14	15	16	8	8	8	8	16	16	2	16
ⓐ	16	16	15	15	15	14	15	16	8	8	8	8	16	16	2	16

Figure: An example of computing the *High* values and biconnected components.
Source: redrawn from [Manber 1989, Figure 7.29].

Even-Length Cycles

Problem 6. *Given a connected undirected graph $G = (V, E)$, determine whether it contains a cycle of even length.*

Theorem 7. *Every biconnected graph that has more than one edge and is not merely an odd-length cycle contains an even-length cycle.*

Even-Length Cycles (cont.)

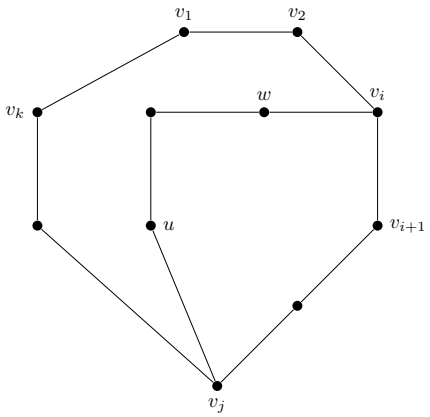


Figure: Finding an even-length cycle.
Source: redrawn from [Manber 1989, Figure 7.35].

3 Network Flows

Network Flows

- Consider a directed graph, or network, $G = (V, E)$ with two distinguished vertices: s (the source) with indegree 0 and t (the sink) with outdegree 0.
- Each edge e in E has an associated positive weight $c(e)$, called the *capacity* of e.

Network Flows (cont.)

- A **flow** is a function f on E that satisfies the following two conditions:
  1. $0 \leq f(e) \leq c(e)$.
  2. $\sum_u f(u, v) = \sum_w f(v, w)$, for all $v \in V - \{s, t\}$.
- The **network flow problem** is to maximize the flow f for a given network G.

Network Flows (cont.)

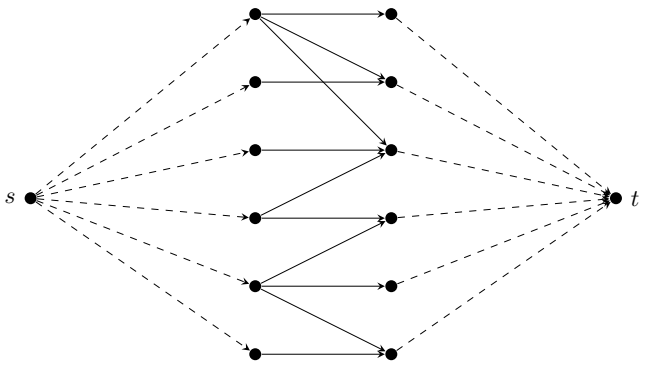


Figure: Reducing bipartite matching to network flow. Every edge has capacity 1 (not shown on the network).

Source: redrawn from [Manber 1989, Figure 7.39].

/* A graph is *partite* if its set of vertices can be divided into two disjoint subsets V_1 and V_2 such that every edge in the graph connects a vertex in V_1 with a vertex in V_2. The *bipartite matching* problem asks for a selection of as many edges as possible where no two edges share a common end vertex.

We will study this reduction in greater detail later in the course. */

Augmenting Paths

- An **augmenting path** w.r.t. a given flow f (of a network G) is a directed path from s to t consisting of edges from G, but not necessarily in the same diretion; each of these edges (v, u) satisfies exactly one of:
  1. (v, u) is in the same direction as it is in G, and $f(v, u) < c(v, u)$. (*forward edge*)
  2. (v, u) is in the opposite direction in G (namely, $(u, v) \in E$), and $f(u, v) > 0$. (*backward edge*)
- If there exists an augmenting path w.r.t. a flow f (f *admits* an augmenting path), then f is not maximum.

Augmenting Paths (cont.)

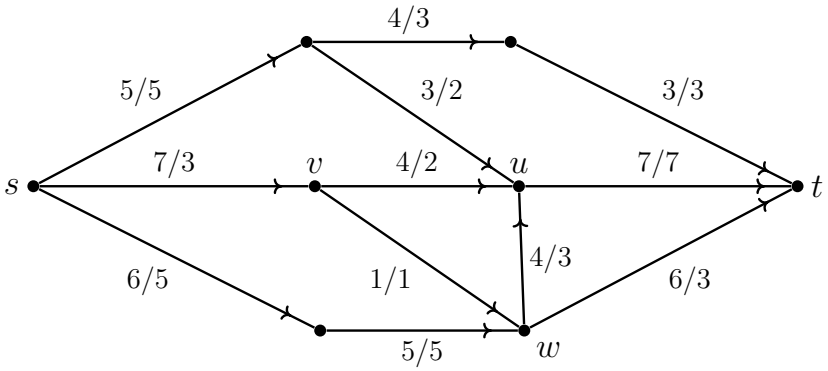


Figure: An example of a network with a (nonmaximum) flow.

Source: redrawn from [Manber 1989, Figure 7.40].

/* The flow admits an augmenting path: $s \to v \to u \to w \to t$. */

Augmenting Paths (cont.)

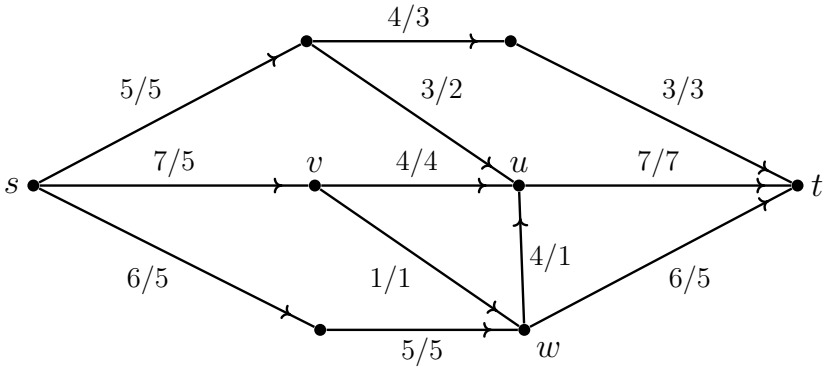


Figure: The result of augmenting the flow of the preceding figure.
Source: redrawn from [Manber 1989, Figure 7.41].

Properties of Network Flows

Theorem 8 (Augmenting-Path)**.** *A flow f is maximum if and only if it admits no augmenting path.*

A *cut* is a set of edges that separate s from t, or more precisely a set of the form $\{(v, w) \in E \mid v \in A \text{ and } w \in B\}$, where $B = V - A$ such that $s \in A$ and $t \in B$.

Theorem 9 (Max-Flow Min-Cut)**.** *The value of a maximum flow in a network is equal to the minimum capacity of a cut.*

/* The capacity of a cut is the sum of the capacities of all edges in the cut. */

Properties of Network Flows (cont.)

Theorem 10 (Integral-Flow)**.** *If the capacities of all edges in the network are integers, then there is a maximum flow whose value is an integer.*

Residual Graphs

- The **residual graph** with respect to a network $G = (V, E)$ and a flow f is the network $R = (V, F)$, where F consists of all forward and backward edges and their capacities are given as follows:
  1. $c_R(v, w) = c(v, w) - f(v, w)$ if (v, w) is a forward edge and
  2. $c_R(v, w) = f(w, v)$ if (v, w) is a backward edge.
- An augmenting path is thus a regular directed path from s to t in the residual graph.

Residual Graphs (cont.)

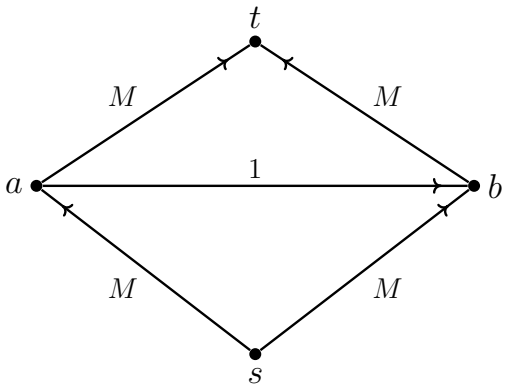


Figure: A bad example of network flow.
Source: redrawn from [Manber 1989, Figure 7.42].

/* Without any particular search tactics, one might first find the augmenting path $s \to a \to b \to t$ and assign a flow of 1 to each of the three edges. After the flow assignment, $s \to b \to a \to t$ becomes an augmenting path, which can accommodate a flow of at most 1 on each edge. Suppose that path is taken. Now after two tries of flow assignment, one in effect gets a flow of 1 on each of (s, a), (s, b), (a, t), and (b, t). If bad luck persists, one would need $2M$ tries to find the maximum flow, while the best possible is just 2 tries. */