CHAPTER 22

Elementary Graph Algorithms
22.1 Representations of graphs

There are two standard ways to represent a graph $G = (V, E)$: as a collection of adjacency lists or as an adjacency matrix. Either way is applicable to both directed and undirected graphs. The adjacency-list representation is usually preferred, because it provides a compact way to represent sparse graphs—those for which $|E|$ is much less than $|V|^2$. Most of the graph algorithms presented in this book assume that an input graph is represented in adjacency-list form.

An adjacency-matrix representation may be preferred, however, when the graph is dense—$|E|$ is close to $|V|^2$—or when we need to be able to tell quickly if there is an edge connecting two given vertices. For example, two of the all-pairs shortest-paths algorithms presented in Chapter 25 assume that their input graphs are represented by adjacency matrices.

The adjacency-list representation of a graph $G = (V, E)$ consists of an array $\text{Adj}$ of $|V|$ lists, one for each vertex in $V$. For each $u \in V$, the adjacency list $\text{Adj}[u]$ contains all the vertices $v$ such that there is an edge $(u, v) \in E$. 

ستعيش سعيداً وناجحاً إذا كان ما تريده هو ما تحتاجه بالفعل.
That is, Adj[u] consists of all the vertices adjacent to u in G. (Alternatively, it may contain pointers to these vertices.) The vertices in each adjacency list are typically stored in an arbitrary order.

The following figure (b) is an adjacency-list representation of the undirected graph in figure (a).

The following figure (b) is an adjacency-list representation of the directed graph in figure (a).

Note that (u,v) means that vertex u is adjacent to v. In undirected graphs, it means there’s an edge connecting u and v. In directed graphs, it means there’s an edge from u to v.
If $G$ is a directed graph, the sum of the lengths of all the adjacency lists is $|E|$, since an edge of the form $(u, v)$ is represented by having $v$ appear in $\text{Adj}[u]$.

If $G$ is an undirected graph, the sum of the lengths of all the adjacency lists is $2|E|$, since if $(u, v)$ is an undirected edge, then $u$ appears in $v$’s adjacency list and vice versa.

For both directed and undirected graphs, the adjacency-list representation has the desirable property that the amount of memory it requires is $\Theta(V + E)$.

Adjacency lists can readily be adapted to represent weighted graphs, that is, graphs for which each edge has an associated weight, typically given by a weight function $w$. For example, let $G = (V, E)$ be a weighted graph with weight function $w$. The weight $w(u, v)$ of the edge $(u, v) \in E$ is simply stored with vertex $v$ in $u$’s adjacency list.
A potential disadvantage of the adjacency-list representation is that there is no quicker way to determine if a given edge \((u, v)\) is present in the graph than to search for \(v\) in the adjacency list \(\text{Adj}[u]\). This disadvantage can be remedied by an adjacency-matrix representation of the graph, at the cost using asymptotically more memory.

(See Exercise 22.1-8 for suggestions of variations on adjacency lists that permit faster edge lookup.)

For the **adjacency-matrix representation** of a graph \(G = (V, E)\), we assume that the vertices are numbered \(1, 2, ..., |V|\) in some arbitrary manner.

Then the adjacency-matrix representation of a graph \(G\) consists of a \(|V| \times |V|\) matrix \(A = (a_{ij})\) such that

\[
a_{ij} = \begin{cases} 
1 & \text{if } (i, j) \in E, \\
0 & \text{otherwise.}
\end{cases}
\]

The previous two figures (c) are the adjacency matrices of the undirected and directed graphs in both figures respectively. The adjacency matrix of a graph requires \(\Theta(|V|^2)\) memory, independent of the number of edges in the graph.
Since in an undirected graph, \((u, v)\) and \((v, u)\) represent the same edge, the adjacency matrix \(A\) of an undirected graph is its own transpose: \(A = A^T\).

In some applications, it pays to store only the entries on and above the diagonal of the adjacency matrix, thereby cutting the memory needed to store the graph almost in half.

Like the adjacency-list representation of a graph, the adjacency-matrix representation can be used for weighted graphs. For example, if \(G = (V, E)\) is a weighted graph with edge-weight function \(w\), the weight \(w(u, v)\) of the edge \((u, v) \in E\) is simply stored as the entry in row \(u\) and column \(v\) of the adjacency matrix. If an edge does not exist, a NIL value can be stored as its corresponding matrix entry, though for many problems it is convenient to use a value such as 0 or \(\infty\).
Although the adjacency-list representation is asymptotically at least as efficient as the adjacency-matrix representation, the simplicity of an adjacency matrix may make it preferable when graphs are reasonably small.

Moreover, if the graph is unweighted, there is an additional advantage in storage for the adjacency-matrix representation, which is rather than using one word of computer memory for each matrix entry, the adjacency matrix uses only one bit per entry.

An algorithm that increases the weight of each vertex by one:

Using adjacency list representation

\[
\text{INCREASE (G)}
\]

for each \( u \) in \( G \)

\[
\text{do for each } v \text{ in } \text{adj}[u]
\]

\[
\text{do } w = w + 1
\]
Complexity: $O(V+E)$

Using adjacency matrix representation

**INCREASE(G)**

for each $u$ in $G$

    do for each $v$ in $G$

        do if ($A[u,v] \neq \infty$)


Complexity: $O(V^2)$
Exercise 22.1-1

Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?

Solution

Given the adjacency-list representation of a graph, the out and in-degree of every node can easily be computed in $O(V+E)$ time.
Exercise 22.1-3

The transpose of a directed graph $G = (V,E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(v,u) \in V \times V : (u,v) \in E\}$. Thus, $G^T$ is $G$ with all its edges reversed. Describe efficient algorithms for computing $G^T$ from $G$, for both the adjacency-list and adjacency-matrix representations of $G$. Analyze the running times of your algorithms.

Solution

The transpose of a directed graph $G^T$ can be computed as follows: If the graph is represented by an adjacency-matrix $A$, simply compute the transpose of the matrix $A^T$ in time $O(V^2)$. If the graph is represented by an adjacency-list, a single scan through this list is sufficient to construct the transpose. The time used is $O(E+V)$. 

Exercise 22.1-5

The square of a directed graph $G = (V,E)$ is the graph $G^2 = (V,E^2)$ such that $(u,w) \in E^2$ if and only if for some $v \in V$, both $(u,v) \in E$ and $(v,w) \in E$. That is, $G^2$ contains an edge between $u$ and $w$ whenever $G$ contains a path with exactly two edges between $u$ and $w$. Describe efficient algorithms for computing $G^2$ from $G$ for both the adjacency-list and adjacency-matrix representations of $G$. Analyze the running times of your algorithms.

Solution

If the graph is represented as an adjacency-matrix $A$, simply compute the product $A^2$ where multiplication and addition is exchanged by and’ing and or’ing. Using the trivial algorithm yields a running time of $O(n^3)$.

If its representing as an adjacency-list representation, we can simply append lists eliminating duplicates. Assuming the lists are sorted, we can proceed as follows: For each node $v$ in some list, replace the $v$ with $\text{Adj}[v]$ and merge this into the list. Each list can be at most $V$ long and therefore each merge operation takes at most $O(V)$ time. Thus, the total time is $O(E+V^2)$. 

أَسْأَلْ أَحَدًا بِأَهْمِيَّتِهِ، وَسَتَمْلِكُ قَلْبِهِ
Exercise 22.1-6

When an adjacency-matrix representation is used, most graph algorithms require time $\Omega(V^2)$, but there are some exceptions. Show that determining whether a directed graph $G$ contains a universal sink – a vertex with in-degree $|V|-1$ and out-degree 0 – can be determined in time $O(V)$, given an adjacency-matrix for $G$.

Solution

Notice that if edge is present between vertices $v$ and $u$, then $v$ cannot be a sink, and if the edge is not present then $u$ cannot be a sink. Searching the adjacency list in a linear fashion enables us to exclude one vertex at a time.
Exercise 22.1-7

The incidence matrix of a directed graph $G = (V,E)$ is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that

$$
B_{ij} = \begin{cases} 
-1 & \text{if edge } j \text{ leaves vertex } i, \\
1 & \text{if edge } j \text{ enters vertex } i, \\
0 & \text{otherwise.}
\end{cases}
$$

Describe what the entries of the matrix product $B B^T$ represent, where $B^T$ is the transpose of $B$.

**Solution**

$$
B B^T (i,j) = \sum b_{ie} b_{ej}^T = \sum b_{ie} b_{je}
$$

- If $i = j$, then $b_{ie} b_{je} = 1$ (it is 1 . 1 or (-1) . (-1)) whenever $e$ enters or leaves vertex $i$, and 0 otherwise.

- If $i \neq j$, then $b_{ie} b_{je} = -1$ when $e = (i,j)$ or $e = (j,i)$, and 0 otherwise.

Thus,

$$
B B^T (i,j) = \begin{cases} 
\text{degree of } i = \text{in-degree} + \text{out-degree} & \text{if } i=j \\
-(\# \text{ of edges connecting } i \text{ and } j) & \text{if } i \neq j.
\end{cases}
$$
22.2 Breadth-first search

**DEFINITION**

**Breadth-first search** is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms.

Given a graph $G = (V, E)$ and a distinguished source vertex $s$, breadth-first search systematically explores the edges of $G$ to "discover" every vertex that is reachable from $s$. It computes the distance (smallest number of edges) from $s$ to each reachable vertex. It also produces a "breadth-first tree" with root $s$ that contains all reachable vertices.

For any vertex $v$ reachable from $s$, the path in the breadth-first tree from $s$ to $v$ corresponds to a "shortest path" from $s$ to $v$ in $G$, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs.

Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
That is, the algorithm discovers all vertices at distance $k$ from $s$ before discovering any vertices at distance $k+1$.

To keep track of progress, breadth-first search colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is **discovered** the first time it is encountered during the search, at which time it becomes nonwhite. Gray and black vertices, therefore, have been discovered, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner. If $(u, v) \in E$ and vertex $u$ is black, then vertex $v$ is either gray or black; that is, all vertices adjacent to black vertices have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex $s$. Whenever a white vertex $v$ is discovered in the course of scanning the adjacency list of an already discovered vertex $u$, the vertex $v$ and the edge $(u, v)$ are added to the tree.
We say that \( u \) is the \textbf{predecessor} or \textbf{parent} of \( v \) in the breadth-first tree. Since a vertex is discovered at most once, it has at most one parent.

\textbf{Ancestor} or \textbf{descendant} relationships in the breadth-first tree are defined relative to the root \( s \) as usual: if \( u \) is on a path in the tree from the root \( s \) to vertex \( v \), then \( u \) is an ancestor of \( v \) and \( v \) is a descendant of \( u \).

The breadth-first-search procedure BFS below assumes that the input graph \( G = (V, E) \) is represented using adjacency lists. It maintains several additional data structures with each vertex in the graph. The \textbf{color} of each vertex \( u \in V \) is stored in the variable \( \text{color}[u] \), and the \textbf{predecessor} of \( u \) is stored in the variable \( \pi[u] \).

If \( u \) has no predecessor (for example, if \( u = s \) or \( u \) has not been discovered), then \( \pi[u] = \text{NIL} \).

The \textbf{distance} from the source \( s \) to vertex \( u \) computed by the algorithm is stored in \( d[u] \). The algorithm also uses a first-in, first-out queue \( Q \) to manage the set of gray vertices.
BFS (G, s)

1. for each vertex $u \in V \setminus \{s\}$
2. do color [u] ← WHITE
3. $d[u] \leftarrow \infty$
4. $\pi [u] \leftarrow \text{NIL}$
5. color[s] ← GRAY
6. $d[s] \leftarrow 0$
7. $\pi [s] \leftarrow \text{NIL}$
8. $Q \leftarrow \emptyset$
9. ENQUEUE ($Q$, s)
10. While $Q \neq \emptyset$
11. do $u \leftarrow \text{DEQUEUE} (Q)$
12. for each $v \in \text{Adj}[u]$
13. do if color [v] = WHITE
14. then color[v] ← GRAY
15. $d[v] \leftarrow d[u] + 1$
16. $\pi [v] \leftarrow u$
17. ENQUEUE ($Q$, v)
18. color[u] ← BLACK

من الوقاحة أن تسأل صديقك الذي وقع في ورطة إن كان يحتاج لمساعدتك.
من نكد الدنيا على المرء أن ترى عدوا لك ما من صداقته بد.
The figure illustrates the progress of BFS on a sample graph.

The procedure BFS works as follows.

With the exception of the source vertex s, lines 1-4 paint every vertex white, set \( d[u] \) to be infinity for each vertex \( u \), and set the parent of every vertex to be NIL.

Line 5 paints \( s \) gray, since it is considered to be discovered when the procedure begins. Line 6 initializes \( d[s] \) to 0, and line 7 sets the predecessor of the source to be NIL.

Line 8-9 initialize \( Q \) to the queue containing just the vertex \( s \).
The **while** loop of lines 10-18 iterates as long as there remain gray vertices, which are discovered vertices that have not yet had their adjacency lists fully examined. This **while** loop maintains the following invariant:

At the test in line 10, the queue $Q$ consists of the set of gray vertices.

Although we won’t use this loop invariant to prove correctness, it is easy to see that it holds prior to the first iteration and that each iteration of the loop maintains the invariant. Prior to the first iteration, the only gray vertex, and the only vertex in $Q$, is the source vertex $s$.

Line 11 determines the gray vertex $u$ at the head of the queue $Q$ and removes it from $Q$. The **for** loop of lines 12-7 considers each vertex $v$ in the adjacency list of $u$.

If $v$ is white, then it has not yet been discovered, and the algorithm discovers it by executing lines 14-17.

It is first grayed, and its distance $d[v]$ is set to $d[u] + 1$. Then, $u$ is recorded as its parent. Finally, it is placed at the tail of the queue $Q$.  

لا ستلام نسست نلكتلونسة ن منن سنلاتناكمنقعنمجاساًن علىنةمةلكنق بزةاوة نبزةاوة نبزةاوة نبزةاوة نبزةاوة نبزةاوة نبزةاوة نبزةاوة نبزةاوة نبزةاوة نبزةاوة نبزةاوة نبزةاوة

The truth has a beautiful taste, but most of us have grown used to the taste of lies.
When all the vertices on u’s adjacency list have been examined, u is blackened in line 18.

The loop invariant is maintained because whenever a vertex is painted gray (in line 14) it is also enqueued (in line 17), and whenever a vertex is dequeued (in line 11) it is also painted black (in line 18).

**The results of breadth-first search may depend upon the order in which the neighbors of a given vertex are visited** in line 12: the breadth-first tree may vary, but the distances d computed by the algorithm will not.

**Analysis**

Before proving the various properties of breadth-first search, we take on the easier job of analyzing its running time on an input graph G = (V, E). We use aggregate analysis.

After initialization, no vertex is ever whitened, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once.
The operations of enqueuing and dequeuing take $O(1)$ time, so the total time devoted to queue operations is $O(V)$. Because the adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned at most once.

Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(E)$. The overhead for initialization is $O(V)$, and thus the total running time of BFS is $O(V + E)$. Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of $G$. 

لا تصبح الكذبة حقيقة إذا تم تكرارها بما يكفي.

لاستلم نسخ إلكترونية من نواتن الموقع مجاناً على إيميلك فم بزيارة

eng-hs.net

النواتن متوفرة مجانا في كل فصول 프رسون ولمانو 24814916 أو تصوير الجامعية بحسب الرسوم بـ 24926388

eng-hs.com, eng-hs.net
Exercise 22.2-1

Show the d and π values that result from running breadth-first search on the directed graph shown, using vertex 3 as the source.

Solution
لا حينما لا يكون لديك ما تقول له تخزن، الأمر سهل، عليك أن تتصمت.

لا ستلام نسست نلكنتلنسنمننمنقعنمجاساًن علىنةلكنقبزةاوة

51941842

اننلصنةوناكجمعة ناكوئةستة

61862999

ناحمادةنشعبان

2604449

info@eng-hs.com

النواتن متوفرة مجناً بالموقعين

eng-hs.net, eng-hs.net

المؤتمرات الهندسية shine on 248149164 249263888

M. حمادة شعبان 260 4444

eng-hs.com
Exercise 22.2-3

What is the running time of BFS if its input graph is represented by an adjacency matrix and the algorithm is modified to handle this form of input?

Solution

If breadth-first search is run on a graph represented by an adjacency matrix, the time used scanning for neighbor can increase to $O(V^2)$ yielding a total running time of $O(V^2+E)$. 
**Exercise 22.2-4**

Argue that in a breadth-first search, the value $d[u]$ assigned to a vertex $u$ is independent of the order in which the vertices in each adjacency list are given. Using the previous figure as an example, show that the breadth-first tree computed by BFS can depend on the ordering within adjacency lists.

**Solution**

The correctness of proof for the BFS algorithm shows that $d[u] = \delta(s,u)$, and the algorithm doesn’t assume that the adjacency lists are in any particular order. In the figure, if $t$ precedes $x$ in $\text{Adj}[w]$, we can get the breadth-first tree shown in the figure. But if $x$ precedes $t$ in $\text{Adj}[w]$ and $u$ precedes $y$ in $\text{Adj}[x]$, we can get edge $(x,u)$ in the breadth-first tree.
Exercise 22.2-5

Give an example of a directed graph \( G = (V,E) \), a source vertex \( s \in V \), and a set of tree edges \( E_\pi \subseteq E \) such that for each vertex \( v \in V \), the unique path in the graph \( (V,E_\pi) \) from \( s \) to \( v \) is a shortest path in \( G \), yet the set of edges \( E_\pi \) cannot be produced by running on \( G \), no matter how the vertices are ordered in each adjacency list.

Solution

The edges in \( E_\pi \) are shaded in the following graph

![Graph Diagram]

To see that \( E_\pi \) cannot be a breadth-first tree, let’s suppose that \( \text{Adj}[s] \) contains \( u \) before \( v \). BFS adds edges \((s,u)\) and \((s,v)\) to the breadth-first tree. Since \( u \) is enqueued before \( v \), BFS then adds edges \((u,w)\) and \((u,x)\). (The order of \( w \) and \( x \) in \( \text{Adj}[u] \) doesn’t matter). Symmetrically, if \( \text{Adj}[s] \) contains \( v \) before \( u \), then BFS adds edges \((s,v)\) and \((s,u)\) to the breadth-first tree, \( v \) is enqueued before \( u \), and BFS adds edges \((v,w)\) and \((v,x)\). BFS will never put both edges \((u,w)\) and \((v,x)\) into the breadth-first tree. In fact, it will also never put both edges \((u,x)\) and \((v,w)\) into the breadth-first tree.
Exercise 22.2-6

There are two types of professional wrestlers: “good guys” and “bad guys” between any pair of professional wrestlers, there may or may not be rivalry. Suppose we have \( n \) professional wrestlers and we have a list of \( r \) pairs of wrestlers for which there are rivalries. Give an \( O(n+r) \)-time algorithm that determines whether it is possible to designate some of the wrestlers ad good guys and the remainder as bad guys such that each rivalry is between a good guy and a bad guy. If it is possible to perform such a designation, your algorithm should produce it.

Solution

Perform as many BFS’s as needed to visit all vertices. Assign all wrestlers whose distance is even to be good guys and all wrestlers whose distance is odd to be bad guys. Then check each edge to verify that it goes between a good guy and a bad guy. This solution would take \( O(n+r) \) time for the BFS, \( O(n) \) time to designate each wrestler as a good guy or bad guy, and \( O(n) \) time to check edges, which is \( O(n+r) \) time overall.
Exercise 22.2-7

The diameter of a tree $T = (V, E)$ is given by

$$\max \delta(u, v);$$

that is, the diameter is the largest of all shortest-path distances in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyze the running time of your algorithm.

Solution

The diameter of a tree can be computed in a bottom-up fashion using a recursive solution. If $x$ is a node with a depth $d(x)$ in the tree, then the diameter $D(x)$ must be:

$$D(x) = \begin{cases} \max \{ \max_i \{ D(x.\text{child}_i) \}, \max_{ij} \{ d(x.\text{child}_i) + d(x.\text{child}_j) \} + 2 \}, & \text{if } x \text{ is an internal node.} \\ 0, & \text{if } x \text{ is a leaf.} \end{cases}$$

Since the diameter must be in one of the subtrees or pass through the root and the longest path from the root must be the depth. The depth can easily be computed at the same time. Using dynamic programming, we obtain a linear solution.

Actually, the problem can also be solved by computing the longest shortest path from an arbitrary node. The node farthest away will be the endpoint of a diameter and we can thus compute the longest shortest path from this node to obtain the diameter.
22.3 Depth-first search

The strategy followed by depth-first search is, as its name implies, to search “deeper” in the graph whenever possible. In depth-first search, edges are explored out of the most recently discovered vertex $v$ that still has unexplored edges leaving it. When all of $v$’s edges have been explored, the search “backtracks” to explore edges leaving the vertex from which $v$ was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source. This entire process is repeated until all vertices are discovered.

As in breadth-first search, whenever a vertex $v$ is discovered during a scan of the adjacency list of an already discovered vertex $u$, depth-first search records this event by setting $v$’s predecessor field $\pi[v]$ to $u$. 
Unlike breadth-first search, whose predecessor subgraph forms a tree, the predecessor subgraph produced by a depth-first search may be composed of several trees, because the search may be repeated from multiple sources.

The **predecessor subgraph** of a depth-first search is therefore defined slightly differently from that of a breadth-first search: we let $G_\pi = (V, E_\pi)$, where

$$E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq \text{NIL}\}.$$ 

The predecessor subgraph of a depth-first search forms a **depth-first forest** composed of several **depth-first trees**. The edges in $E_\pi$ are called **tree edges**.

As in breadth-first search, vertices are colored during the search to indicate their state. Each vertex is initially white, is grayed when it is **discovered** in the search, and is blackened when it is **finished**, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

من الأفضل أن تغلق فمك وتترك الناس يعتقدون أنك أحمق، من أن تفتحه وتمحو كل شكل.
Beside creating a depth-first forest, depth-first search also **timestamps** each vertex. Each vertex \( \mathcal{V} \) has two timestamps: the first timestamp \( d[\mathcal{V}] \) records when \( \mathcal{V} \) is first discovered (and grayed), and the second timestamp \( f[\mathcal{V}] \) records when the search finishes examining \( \mathcal{V} \)’s adjacency list (and blackens \( \mathcal{V} \)). These timestamps are used in many graph algorithms and are generally helpful in reasoning about the behavior of depth-first search.

The procedure DFS below records when it discovers vertex \( u \) in the variable \( d[u] \) and when it finishes vertex \( u \) in the variable \( f[u] \). These timestamps are integers between 1 and \( 2|V| \), since there is one discovery event and one finishing event for each of the \( |V| \) vertices. For every vertex \( u \), \( d[u] < f[u] \) vertex \( u \) is WHITE before time \( d[u] \), GRAY between time \( d[u] \) and time \( f[u] \), and BLACK thereafter.

The following pseudo code is the basic depth-first-search algorithm. The input graph \( G \) may be undirected or directed. The variable time is a global variable that we use for timestamping.
DFS(G)

1. for each vertex \( u \in V[G] \)
2. do color \([u]\) \(\leftarrow\) WHITE
3. \(\pi[u] \leftarrow\) NIL
4. time \(\leftarrow\) 0
5. for each vertex \( u \in V[G] \)
6. do if color\([u]\) = WHITE
7. then DFS-VISIT \((u)\)

DFS-VISIT\((u)\)

1. color \([u]\) \(\leftarrow\) Gray
2. time \(\leftarrow\) time +1
3. \(d[u] \leftarrow\) time
4. For each \( v \in Adj[u] \)
5. do if color \([v]\) = WHITE
6. then \(\pi[v] \leftarrow u\)
7. DFS-VISIT \((v)\)
8. color \([u]\) \(\leftarrow\) BLACK
9. \(f[u] \leftarrow\) time \(\leftarrow\) time +1
The following figure illustrates the progress of DFS on the graph shown in the previous figure.

procedure DFS works as follows. Lines 1-3 paint all vertices white and initialize their $\pi$ fields no NIL. Line 4 resets the global time counter. Lines 5-7 check each vertex in $V$ in turn and, when a white vertex is found, visit it using DFS-VISIT. Every time DFS-VISIT ($u$) is called in line 7, vertex $u$ becomes the root of a new tree in the depth-first forest. When DFS returns, every vertex $u$ has been assigned a discovery time $d[u]$ and a finishing time $f[u]$. 

لاستلام نسخ إلكترونية من نوتنات الموقع مجاناً على إيميلك قم بزيارة eng-hs.net

لاستلام نسخ إلكترونية من نوتنات الموقع مجاناً على إيميلك قم بزيارة eng-hs.com

النواتت متوفرة مجاناً بالموقعين info@eng-hs.com

م. حمادة شبان 260 4444

4926388248149616

سيكون بإمكاننا يوما ما حساب حركة الأجرام السماوية، ولكن ليس جنون البشر.
In each call DFS-VISIT (u), vertex u is initially white. Line 1 paints u gray, line 2 increments the global variable time, and line 3 records the new value of time as the discovery time \( d[u] \). Lines 4-7 examine each vertex \( \nu \) adjacent to u and recursively visit \( \nu \) if it is white. As each vertex \( \nu \in Adj[u] \) is considered in line 4, we say that edge \( (u, \nu) \) is \textbf{explored} by the depth-first search. Finally, after every edge leaving u has been explored, lines 8-9 paint u black and record the finishing time in \( f[u] \).

Note that the results of depth-first search may depend upon the order in which the vertices are examined in line 5 of DFS, and upon the order in which the neighbors of a vertex are visited in line 4 of DFS-VISIT. These different visitation orders tend not to cause problems in practice, as any depth-first search result can usually be used effectively, with essentially equivalent results.
What is the running time of DFS? The loops on lines 1-3 and lines 5-7 of DFS take time \( \Theta(V) \), exclusive of the time to execute the calls to DFS-VISIT. As we did for breadth-first search, we use aggregate analysis. The procedure DFS-VISIT is called exactly once for each vertex \( v \in V \), since DFS-VISIT is invoked only on white vertices and the first thing it does is paint the vertex gray. During an execution of DFS-VISIT \((v)\), the loop on lines 4-7 is executed \(|\text{Adj}[v]| \) times. Since

\[
\sum_{v \in V} |\text{Adj}[v]| = \Theta(E),
\]

The total cost of executing lines 4-7 of DFS-VISIT is \( \Theta(E) \). The running time of DFS is therefore \( \Theta(V+E) \).

**Properties of depth-first search**

Depth-first search yield valuable information about the structure of a graph. Perhaps the most basic property of depth-first search is that predecessor subgraph \( G_\pi \) does indeed from a forest of trees, since the structure of the depth-first trees exactly mirrors the structure of recursive calls of DFS-VISIT.
That is, \( u = \pi[v] \) if and only if DFS-VISIT \( (v) \) was called during a search of \( u \)’s adjacency list. Additionally, vertex \( v \) is a descendant of vertex \( u \) in the depth-first forest if and only if \( v \) is discovered during the time in which \( u \) is gray.

**Classification of edges**

Another interesting property of depth-first search is that the search can be used to classify the edges of the input graph \( G = (V, E) \). This edge classification can be used to glean important information about a graph. For example, in the next section, we shall see that a directed graph is acyclic if and only if a depth-first search yields no “back” edges.

We can define four edge types in terms of the depth-first forest \( \pi \) produced by a depth-first search on \( G \).

1. **Tree edges** are edges in the depth-first forest \( \pi \).

Edge \( (u, v) \) is a tree edge if \( v \) was first discovered by exploring edge \( (u, v) \).
2. **Back edges** are those edges \((u, v)\) connecting a vertex \(u\) to an ancestor \(v\) in a depth-first tree. Self-loops, which may occur in directed graphs, are considered to be back edges.

3. **Forward edges** are those nontree edges \((u, v)\) connecting a vertex \(u\) to a descendant \(v\) in a depth-first tree.

4. **Cross edges** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.
لاستلام نسخ إلكترونية من نوّات الموقع مجانًا على إيفيك ثم زيارة
الجميع يتمنى أن يتغير الناس، القليل يتمنى تغيير نفسه.
The DFS algorithm can be modified to classify edges as it encounters them. The key idea is that each edge \((u, \nu)\) can be classified by the color of the vertex \(\nu\) that is reached when the edge is first explored (except that forward and cross edges are not distinguished):

1. WHITE indicates a tree edge,
2. GRAY indicates a back edge, and
3. BLACK indicates a forward or cross edge.

The first case is immediate from the specification of the algorithm. For the second case, observe that the gray vertices always form a linear chain of descendants corresponding to the stack of active DFS-VISIT invocations; the number of gray vertices is one more than the depth in the depth-first forest of the vertex most recently discovered. Exploration always proceeds from the deepest gray vertex, so an edge that reaches another gray vertex reaches an ancestor. The third case handles the remaining possibility; it can be shown that such an edge \((u, \nu)\) is a forward edge if \(d[u] < d[\nu]\) and a cross edge if \(d[u] > d[\nu]\). (See Exercise 22.3-4.)
In an undirected graph, there may be some ambiguity in the type classification, since \((u,v)\) and \((v,u)\) are really the same edge. In such a case, the edge is classified as the first type in the classification list that applies. Equivalently, the edge is classified according to whichever of \((u,v)\) or \((v,u)\) is encountered first during the execution of the algorithm.

Note that forward and cross edges never occur in a depth-first search of an undirected graph.

**Definitions**

**Ancestor**: if \(u\) is on a path in the tree from the root \(s\) to vertex \(x\), then \(u\) is an ancestor of \(v\) and \(v\) is a descendant of \(u\).

**Tree edges**: An edge \((u,v)\) is a tree edge if \(v\) was first discovered by exploring edge \((u,v)\).

**Black edges**: Those edges \((u,v)\) connecting a vertex \(u\) to an ancestor \(v\) in a depth-first tree.

**Forward edges**: Those nontree edges \((u,v)\) connecting a vertex \(u\) to a descendant \(v\) in depth-first tree.

**Cross edges**: All other edges.
### Exercise 22.3-1

Make a 3-by-3 chart with row and column labels WHITE, GRAY, and BLACK. In each cell (i,j), indicate whether, at any point during a depth-first search of a directed graph, there can be an edge from a vertex of color i to a vertex of color j. For each possible edge, indicate what edge types it can be. Make a second such chart for depth-first search of an undirected graph.

### Solution

For the directed case:

<table>
<thead>
<tr>
<th>(i,j)</th>
<th>WHITE</th>
<th>GRAY</th>
<th>BLACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHITE</td>
<td>TBFC</td>
<td>BC</td>
<td>C</td>
</tr>
<tr>
<td>GRAY</td>
<td>TF</td>
<td>TFB</td>
<td>TFC</td>
</tr>
<tr>
<td>BLACK</td>
<td>BC</td>
<td>TFBC</td>
<td></td>
</tr>
</tbody>
</table>

For the undirected case:

<table>
<thead>
<tr>
<th>(i,j)</th>
<th>WHITE</th>
<th>GRAY</th>
<th>BLACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHITE</td>
<td>TB</td>
<td>TB</td>
<td></td>
</tr>
<tr>
<td>GRAY</td>
<td>TB</td>
<td>TB</td>
<td>TB</td>
</tr>
<tr>
<td>BLACK</td>
<td>TB</td>
<td>TB</td>
<td>TB</td>
</tr>
</tbody>
</table>
Exercise 22.3-4

Show that edge \((u,v)\) is

a. A tree edge or forward edge if and only if \(d[u] < d[v] < f[v] < f[u]\).

b. A back edge if and only if \(d[v] < d[u] < f[u] < f[v]\).

c. A cross edge if and only if \(d[v] < f[v] < d[u] < f[u]\).

Solution

a. Edge \((u,v)\) is a tree edge or forward edge if and only if \(v\) is a descendant of \(u\) in the depth-first forest. (If \((u,v)\) is a back edge, then \(u\) is a descendant of the other). Therefore, \((u,v)\) is a tree edge or forward edge if and only if \(d[u] < d[v] < f[v] < f[u]\).

b. First, supposed that \((u,v)\) is a back edge.
   A self-loop is by definition a back edge. If \((u,v)\) is a self-loop, then clearly \(d[v] = d[u] < f[u] = f[v]\).
   If \((u,v)\) is not a self-loop, then \(u\) is a descendant of \(v\) in the depth-first forest, and \(d[v] < d[u] < f[u] < f[v]\).
Now, suppose that \( d[v] \leq d[u] < f[u] \leq f[v] \). If \( u \) and \( v \) are the same vertex, then \( d[v] - d[u] < f[u] = f[v] \), and \( (u,v) \) is a self-loop and hence a back edge. If \( u \) and \( v \) are distinct, then \( d[v] < d[u] < f[u] < f[v] \). Interval \([d[u], f[u]]\) is contained entirely within the interval \([d[v], f[v]]\), and \( u \) is a descendant of \( v \) in a depth-first tree. Thus, \( (u,v) \) is a back edge.

d. First, supposed that \( (u,v) \) is a cross edge. Since neither \( u \) not \( v \) is an ancestor of the other, the intervals \([d[u], f[u]]\) and \([d[v], f[v]]\) are entirely disjoint. Thus, we must have either \( d[v] < f[u] < d[v] < f[v] \) or \( d[v] < f[v] < d[u] < f[u] \). We claim that we cannot have \( d[u] < f[v] \) if \( (u,v) \) is a cross edge. Why? If \( d[u] < d[v] \), then \( v \) is white at time \( d[u] \). \( v \) is a descendant of \( u \), which contradicts \( (u,v) \) being a cross edge. Thus, we must have \( d[v] < f[v] < d[u] < f[u] \).

Now, supposed that \( d[v] < f[v] < d[u] < f[u] \). Neither \( u \) not \( v \) is a descendant of the other, which means that \( (u,v) \) must be a cross edge.
Exercise 22.3-7

Give a counterexample to the conjecture that if there is a path from \( u \) to \( v \) in a directed graph \( G \), and if \( d[u] \) in a depth-first search of \( G \), then \( v \) is a descendant of \( u \) in the depth-first forest produced.

**Solution**

Let’s consider the example graph and depth-first search below.

<table>
<thead>
<tr>
<th></th>
<th>( d )</th>
<th>( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w )</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>( u )</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>( v )</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Clearly, there is a path from \( u \) to \( v \) in \( G \). the bold edges are in the depth-first forest produced. We can see that 

\[ d[u] < d[v] \]

in the depth-first search but \( v \) is not a descendant of \( u \) in the forest.
Exercise 22.3-8

Give a counterexample to the conjecture that if there is a path from u to v in a directed graph G, then any depth-first search must result in d[v] ≤ f[u].

Solution

Let’s consider the example graph and depth-first search below.

<table>
<thead>
<tr>
<th></th>
<th>d</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>u</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>v</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Clearly, there is a path from u to v in G. the bold edges are in the depth-first forest produced by the search. However, d[v] > f[u] and the conjecture is false.
Exercise 22.3-10

Explain how a vertex \( u \) of a directed graph can end up in a depth-first tree containing only \( u \), even though \( u \) has both incoming and outgoing edges in \( G \).

Solution

Let’s consider the example graph and depth-first search below.

<table>
<thead>
<tr>
<th></th>
<th>d</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>u</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>v</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Clearly, \( u \) has both incoming and outgoing edges in \( G \) but a depth-first search of \( G \) produced a depth-first forest where \( u \) is in a tree by itself.
22.4 Topological sort

This section shows how depth-first search can be used to perform a topological sort of a directed acyclic (no cycles) graph, or a “dag” as it is sometimes called.

Definition

A topological sort of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if $G$ contains an edge $(u, v)$, then $u$ appears before $v$ in the ordering. (If the graph is not acyclic, then no linear ordering is possible.)

A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right. Topological sorting is thus different from the usual kind of “sorting” studied in part II.

Directed acyclic graphs are used in many applications to indicate precedence among events.
The figure gives an example that arises when professor Bumstead gets dressed in the morning. The professor must don certain garments before others (e.g., socks before shoes). Other items may be put on in any order (e.g., socks and pants). A directed edge $(u, v)$ in the dag of Figure (a) indicates that garment $u$ must be donned before garment $v$. A topological sort of this dag therefore gives an order for getting dressed. Figure (b) shows the topologically sorted dag as an ordering of vertices along a horizontal line such that all directed edges go from left to right.

The following simple algorithm topologically sorts a dag.
TOPOLOGICAL-SORT(G)

1. Call DFS(G) to compute finishing times f[v] for each vertex v
2. As each vertex is finished, insert it onto the front of a linked list
3. **Return** the linked list of vertices

The previous figure (b) shows how the topologically sorted vertices appear in reverse order of their finishing times.

We can perform a topological sort in time $\Theta(V + E)$, since depth-first search takes $\Theta(V + E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.

A dag got topological sorting.
Exercise 22.4-3

Give an algorithm that determines whether or not a given undirected graph $G = (V,E)$ contains a cycle. Your algorithm should run in $O(v)$ time, independent of $|E|$.

Solution

An undirected graph is acyclic if and only if a DFS yields no back edges.

- If there’s a back edge, there’s a cycle.
- If there’s no back edge, then there are only tree edges. Hence, the graph is acyclic.

Thus, we can run DFS: if we find a back edge, there’s a cycle.

Time: $O(V)$. (Not $O(V+E)$)

If we ever see $|V|$ distinct edges, we must have seen a back edge because in an acyclic (undirected) forest, $|E| \leq |V| - 1$. 

أحيانا نحتاج إلى لطمه قوية على جانب الرأس كي نزيل الافتراضات التي نتفقنا نفكر بنفس الطريقة.
22.5 Strongly connected components

We now consider a classic application of depth-first search: decomposing a directed graph into its strongly connected components. This section shows how to do this decomposition using two depth-first searches.

Many algorithms that work with directed graphs begin with such a decomposition. After decomposition, the algorithm is run separately on each strongly connected component. The solutions are then combined according to the structure of connections between components.

Definitions

A strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices $u$ and $v$ in $C$, we have both $u \rightarrow v$ and $v \rightarrow u$; that is, vertices $u$ and $v$ are reachable from each other.

The transpose of a graph is another graph that has the same vertices and edges but with every edge having opposite direction.
The figure (a) shows a directed graph $G$. The strongly connected components of $G$ are shown as shaded regions.

Our algorithm for finding strongly connected components of a graph $G = (V, E)$ uses the transpose of $G$, which is defined to be the graph $G^T = (V, E^T)$, where $E^T = \{(u, v) : (v, u) \in E\}$. That is, $E^T$ consists of the edges of $G$ with their directions reversed. Given an adjacency-list representation of $G$, the time to create $G^T$ is $O(V + E)$.
It is interesting to observe that $G$ and $G^T$ have exactly the same strongly connected components: $u$ and $v$ are reachable from each other in $G$ if and only if they are reachable from each other in $G^T$. Previous figure (b) shows the transpose of the graph in figure (a), with the strongly connected components shaded.

The following linear-time (i.e., $\Theta(V + E)$-time) algorithm computes the strongly connected components of a directed graph $G = (V, E)$ using two depth-first searches, one on $G$ and one on $G^T$.

**STRONGLY-CONNECTED-COMPONENTS(G)**

1. Call DFS($G$) to compute finishing times $f[u]$ for each vertex $u$
2. Compute $G^T$
3. Call DFS($G^T$), but in the main loop of DFS, consider the vertices in order of decreasing $f[u]$ (as computed in line 1)
4. Output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component.
Here is another way to look at how the second depth-first search operates.

Consider the component graph \((G^T)_{SCC}\) of \(G^T\). If we map each strongly connected component visited in the second depth-first search to a vertex of \((G^T)_{SCC}\), the vertices are visited in the reverse of a topologically sorted order.

If we reverse the edges of \((G^T)_{SCC}\), we get the graph \(((G^T)_{SCC})^T\). Because \(((G^T)_{SCC})^T = G_{SCC}\), the second depth-first search visits the vertices of \(G_{SCC}\) in topologically sorted order.
Exercise 22.5-1

How can the number of strongly connected components of a graph change if a new edge is added?

Solution

There are two cases to consider:

Case 1: If this added edge will connect two vertices of a strongly connected component.

This will not change that strongly connected component and there will be no effect for other components either.

Case 2: If this added edge will connect two vertices from different strongly connected components.

Let’s say A and A’ are two strongly connected components.

If there are no paths between any vertices of A and A’, then again this edge will not have any effects on the number of strongly connected components.

But if there is a path, say from u to A to u’ A’ and if this added edge will create a reverse path from A’ to A, then A will become a one strongly connected component.

If this added edge creates a path from A to A’ instead, then the number of strongly connected components will remain unchanged.
Exercise 22.5-3

Professor Deaver claims that the algorithm for strongly connected components can be simplified by using the original (instead of the transpose) graph in the second depth-first search and scanning the vertices in order of increasing finishing times. Is the professor correct?

Solution

For this, the first DFS will give a list 1 2 0 for the second DFS. All vertices will be incorrectly reported to be in the same strong connected component. For the DFS, the first DFS give a list 0 2 1 for the second DFS. After reversing edges, the correct strongly connected components \{0,1\} and \{2\} will be reported.
**Exercise 22.5-5**

Give an $O(V+E)$-time algorithm to compute the component graph of directed graph $G = (V,E)$. Make sure that there is at most one edge between two vertices in the component graph your algorithm produces.

**Solution**

We have an $O(V+E)$-time algorithm that computes strongly connected components. Let us assume that the output of this algorithm is a mapping $scc[u]$, giving the number of the strongly connected component containing vertex $u$, for each vertex $u$. Without loss of generality, assume that $scc[u]$ is an integer in the set $\{1,2,..|V|\}$.

Construct the multiset (a set that can contain the same object more than once)

$T = \{scc[u]: u \in V\}$, and sort it by using counting sort. Since the values we are sorting are integers in the range 1 to $|V|$, the time to sort is $O(V)$. Go through the sorted multiset $T$ and every time we find an element $x$ that is distinct from the one before it, add $x$ to $V^{SCC}$ (Consider the first element of the sorted set as “distinct from the one before it”). It takes $O(V)$ time to construct $V^{SCC}$.
Continue

Construct the set of ordered pairs

\[ S = \{(x, y) : \text{there is an edge } (u, v) \in E, x = \text{scc}[u], \text{ and } y = \text{scc}[v]\}. \]

We can easily construct this set in \( \Theta(E) \) time by going through all edges in \( E \) and looking up \( \text{scc}[u] \) and \( \text{scc}[v] \) for each edge \((u, v) \in E\).

Having constructed \( S \), remove all elements of the form \((x, x)\).

Alternatively, when we construct \( S \), do not put an element in \( S \) when we and an edge \((u, v)\) for which \( \text{scc}[u] = \text{scc}[v] \). \( S \) now has at most \(|E|\) elements.

Now sort the elements of \( S \) using radix sort. Sort on one component at a time. The order does not matter. In other words, we are performing two passes of counting sort.

The time to do so is \( O(V+E) \), since the values we are sorting on are integers in the range 1 to \(|V|\). Finally, go through the sorted set \( S \), and every time we find an element \((x, y)\) that is distinct from the element before it (again consider the first element of the set as distinct from the one before it) add \((x, y)\) to \( E^{\text{SCC}} \).

Sorting and them adding \((x,y)\) only if it is distinct from the element before it ensures that we add \((x,y)\) at most once. It takes \( O(E) \) time to go through \( S \) in this way, once \( S \) has been sorted.

The total time is \( O(V+E) \).
Exercise 22.5-6

Given a directed graph $G = (V,E)$, explain how to create another graph $G' = (V,E')$ such that (a) $G'$ has the same strongly connected components as $G$, (b) $G'$ has the same component graph as $G$, and (c) $E'$ is as small as possible. Describe a fast algorithm to compute $G'$.

Solution

The basic idea is to replace the edges with each SCC by one simple, directed cycle and then remove redundant edges between SCC’s. Since there must be at least $k$ edges within an SCC that has $k$ vertices, a single directed cycle of $k$ edges gives the $k$-vertex SCC with the fewest possible edges.

The algorithm works as follows:

1- Identify all SCC’s of $G$. Time $\Theta(V+E)$, using the SCC algorithm.
2- Form the component $G^{SCC}$. Time $O(V+E)$.
3- Start with $E' = \emptyset$. Time $O(1)$.
4- For each SCC of $G$, let the vertices in the SCC be $v_1,v_2,\ldots,v_k$, and add to $E'$ the directed edges $(v_1,v_2),(v_2,v_3),\ldots,(v_{k-1},v_k),(v_k,v_1)$. These edges form a simple, directed cycle that includes all vertices of the SCC. Time for all SCC’s: $O(V)$.
5- For each edge $(u,v)$ in the component graph $G^{SCC}$, select any vertex $x$ in $u$’s SCC and any vertex $y$ in $v$’s SCC, and add the directed edge $(x,y)$ to $E'$. Time: $O(E)$.

Thus, the total time is $\Theta(V+E)$. 

رغم أن أولادك سيفعلون غالبا عكس ما تقول لهم بالضبط، لكن عليك أن تستمر في حبهم بالقدر ذاته.
Problem 22-1

A depth-first forest classifies the edges of a graph into tree, back, forward, and cross edges. A breadth-first tree can also be used to classify the edges reachable from the source of the search into the same four categories.

Prove that in a breadth-first search of an undirected graph, the following properties hold:

1. There are no back edges and no forward edges.
2. For each tree edge \((u,v)\), we have \(d[v] = d[u] + 1\).
3. For each cross edge \((u,v)\), we have \(d[v] = d[u]\) or \(d[v] = d[u] + 1\).

Solution

1. Suppose \((u,v)\) is a back edge or a forward edge in a BFS of an undirected graph. Then one of \(u\) and \(v\), say \(u\), is a proper ancestor of the other (\(v\)) in the breadth-first tree. Since we explore all edges of \(u\) before exploring any edges of \(u\’s\) descendants, we must explore the edge \((u,v)\) at the time we explore \(u\). But then \((u,b)\) must be a tree edge.

2. In BFS, an edge \((u,v)\) is a tree edge when we set \(\pi[v] \leftarrow u\). But we only do so when we set \(d[v] \leftarrow d[u]+1\). Since neither \(d[u]\) not \(d[v]\) ever changes thereafter, we have \(d[v] = d[u]+1\) when BFS completes.
Continue

3. Consider a cross edge \((u,v)\) where, without loss of
generality, \(u\) is visited before \(v\). At the time we visit \(u\),
vertex \(v\) must already be on the queue, for otherwise \((u,v)\)
would be a tree edge. Because \(v\) is on the queue, we have
\(d[v] \leq d[u] + 1\). And we have \(d[v] \geq d[u]\).
Thus, either \(d[v] = d[u]\) or \(d[v] = d[u] + 1\).
Problem 22-3

An Euler tour of a connected, directed graph $F = (V,E)$ is a cycle that traverses each edge of $G$ exactly once, although it may visit a vertex more than once.

a. Show that $G$ has an Euler tour if and only if in-degree $(v) = \text{out-degree}(v)$ for each vertex $v$ in $V$.

b. Describe an $O(E)$-time algorithm to find an Euler tour of $G$ if one exists. (Hint: Merge edge-disjoint cycles.)

Solution

a. If $g$ has an Euler tour any path going “into” a vertex must leave it. Conversely, if the in and out-degrees of any vertex is the same we can construct a path that visits all edges.

b. Since the edges of any Euler graph can be split into disjoint cycles, we can simply find these cycles and “merge” them into an Euler tour.