1 Graphs

A **graph** is a data structure that expresses relationships between objects. The objects are called "nodes" and the relationships are called "edges".

For example, social networks can be represented as graphs. In the Twitter follower graph, the nodes would be Twitter users, and there would be an edge pointing from me to Justin Bieber if I was following Justin Bieber. (We could write this edge as a 2-tuple: `(Jessica, Justin)`.)

Task dependencies can also be represented as graphs. Here the nodes might be tasks like "check your email," "put on your shoes," or "go to work," and you would draw an edge from "put on your shoes" to "go to work" because you need to put on your shoes before going to work.

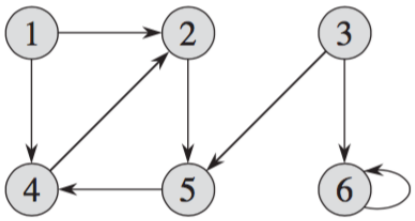

Figure 1: A directed graph.

1.0.1 Directed and undirected graphs

In a **directed graph**, the connection between two nodes is one-directional. The Twitter graph is a directed graph, because I might follow Justin Bieber, but Justin Bieber doesn't follow me.

In an **undirected graph**, all connections are bi-directional. Facebook friendships form an undirected graph, because if I was friends with Justin Bieber, he would also be friends with me. Undirected graphs can be represented as directed graphs, because if (u, v) is an edge in an undirected graph, it would be the same as having a directed graph with the edges (u, v) and (v, u). For this reason, we will mostly focus on directed graphs.

The degree of a node $deg(v)$ is the number of nodes connected to that node. In a directed graph, we have both outdegree (the number of edges pointing away from the node) and indegree (the number of edges pointing towards the node).

A cycle is a path in a graph that goes from a node to itself.

1.0.2 Weighted and unweighted graphs

The nodes and edges in a graph might also carry extra data with them. Often we will want to assign a node a certain color, like "white," "grey," or "black." Edges and nodes may carry

numeric values, known as weights.

For example, in the Google Maps graph (where nodes are locations, and edges are the roads connecting the locations), the weight on each edge might be the length of the road.

1.1 Representations of graphs

From here on out, we will often denote graphs as G, and they will have m edges and n nodes. The set of vertices (or nodes) will be denoted V, and the set of edges will be E. This is standard notation but we may use different notation depending on the circumstances.

There are two common ways to represent a graph on a computer: as an adjacency matrix, and as an adjacency list.

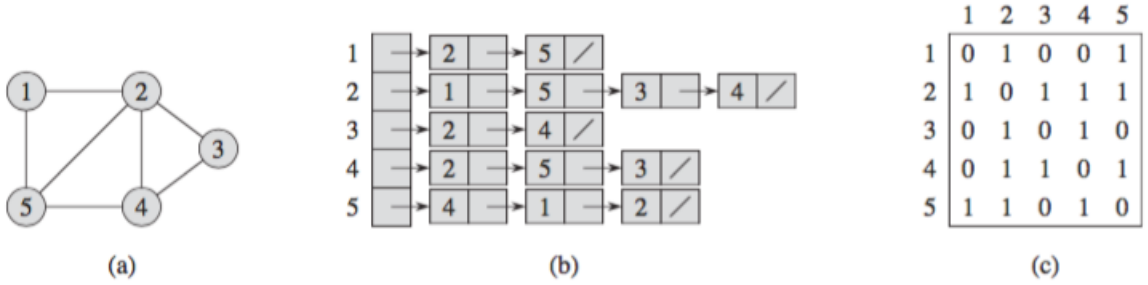

Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

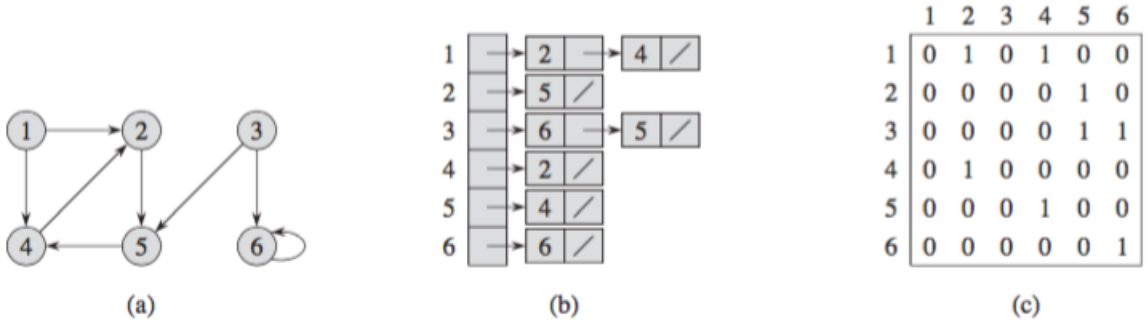

Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

1.1.1 Adjacency matrices

The adjacency matrix A is an n-by-n matrix, where the row and column numbers correspond to the nodes. Entry $A[i, j]$ is 1 if there is an edge from node i to node j, and 0 otherwise.

For weighted graphs, we may replace 1 with the weight on the edge, and replace 0 with a sensible value (like NIL, 0, or ∞, depending on the application).

Note that the adjacency matrix of an undirected graph is symmetric.

Adjacency matrices are easy to implement, but they suffer from a severe drawback. When $m << n^2$, most of the entries are 0, and this wastes a lot of space. For example, in the Twitter follower graph, there are hundreds of millions of users, so if we created a 100M-by-100M adjacency matrix, we would have 10^{16} entries. This costs ten thousand terabytes of space, and since most people follow less than a hundred people, most of this space is wasted.

You'd also use a lot of computation time. I might want to get the followers of a certain user, and to do that, I would have to iterate over all 100 million rows just to find 50 followers.

1.1.2 Adjacency lists

The adjacency list Adj is an array of linked lists. Each list contains the neighbors of a vertex. That is, v is in $Adj[u]$ if there is an edge from u to v. (For weighted graphs, we may store the weight $w(u, v)$ along with v in the linked list.)

Observe that the lengths of the linked lists sum to m, because each element in each list corresponds to an edge in the graph. Adj as a whole takes $\Theta(m + n)$ space (because there are n slots in the array, and the lengths of the linked lists sum to m). In many cases, this is much better than the $\Theta(n^2)$ space taken by the adjacency matrix.

On the other hand, testing whether two nodes are neighbors takes potentially $O(m)$ time, because you have to iterate through all the neighbors of a node. In the adjacency matrix, this would take constant time. So which representation to use really depends on your application. But most of the algorithms in the textbook assume that we have an adjacency list, because it is generally better.

2 Depth first search

Depth first search is a method for recursively traversing a graph that takes $\Theta(m + n)$ time. Whenever we visit a vertex, we visit its first neighbor. But before we visit its other neighbors, we visit the neighbor's neighbor, and so on. Effectively, we go "deeper" into the graph before finishing up business at the beginning.

Imagine you are reading the Wikipedia article on depth-first search.

Depth-first search

From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please help improve this article by ad
Unsourced material may be challenged and removed. *(July 2010) (Learn how and when to remove thi*

Depth-first search (**DFS**) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

A version of depth-first search was investigated in the 19th century by French mathematician Charles Pierre Trémaux[1] as a strategy for solving mazes.[2][3]

If you wanted to traverse the link structure in a depth-first manner, you would get to the first link, which says "algorithm," and then click on it.

Algorithm

From Wikipedia, the free encyclopedia

For other uses, see Algorithm (disambiguation).

In mathematics and computer science, an **algorithm** (/ˈælgərɪðəm/ *AL-gə-ri-dhəm*) is a self-contained step-by-step set of operations to be performed. Algorithms perform calculation, data processing, and/or automated reasoning tasks.

Now the first link on this page is "mathematics," so you would immediately click on that to get to the "mathematics" page. You would keep doing this until there were no more unexplored links. (If you got to a link that you'd seen before, you'd just ignore it.)

Once there are no more unexplored links, you would go back to the "algorithm" page and then click on "computer science," and explore that part of the graph. Then you would click on "calculation," etc.

When you were done with the links on the "algorithm" page, you would go back to the "depth-first-search" page and click on "tree," and perform depth-first search on the "tree" page.

2.1 Algorithm

When we write this algorithm, we want to do it recursively, so that when we call depth-first-search on a node, we call depth-first-search on each of its unvisited neighbors. Here we give each node a color, depending on its current state: white (the node is unexplored), grey (which means the node was seen, but we are not finished processing its descendants yet), or black (which means we have processed the node and all of its descendants).

If we wanted to write pseudocode for this algorithm, it might look like

```
Let G be a graph with vertices that are all white
DepthFirstSearch(graph G, node u):
```

```
    u.color = grey
    for each neighbor in Adj[u]:
        if neighbor.color == white:
            DepthFirstSearch(G, neighbor)
    u.color = black
```

You'll notice this DepthFirstSearch routine doesn't really do anything except color the vertices. If you wanted to use depth-first-search in real life, you would probably want to do something to a vertex every time you processed it. For instance, if you were using depth-first-search to traverse the Wikipedia graph, you might want to save each article to your hard drive every time you colored that article black.

It might seem kind of useless to have separate colors for grey and black, because they are treated identically in the algorithm, but the vertices can fundamentally be in three different states, and having different names for different states helps us prove theorems about the algorithm.

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
 1  time = time + 1            // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]      // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK            // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```

The full version of depth-first search keeps track of a few additional things. First, it keeps track of the parent node π of each vertex. The parent node of v is the node that triggers a visit to v (e.g. in the Wikipedia example, "depth-first-search" is the parent node of "algorithm"). Knowing all the parent nodes allows us to reconstruct a "depth-first-search tree," where there is an edge from u to v if u is the parent of v.

It also keeps track of the "discovery" and "finishing" times of each vertex. The discovery

time is the moment we color the vertex grey and start iterating through all of its neighbors. The finishing time is when we finish iterating through all the neighbors and color the vertex black.

Finally, it accounts for the possibility that not all vertices can be reached from the start node. If we finish exploring from the start node and there are still unexplored vertices left, we just repeat the process at one of the unexplored vertices, and we keep doing this until the graph is exhausted. Now the depth-first search tree becomes a depth-first search forest.

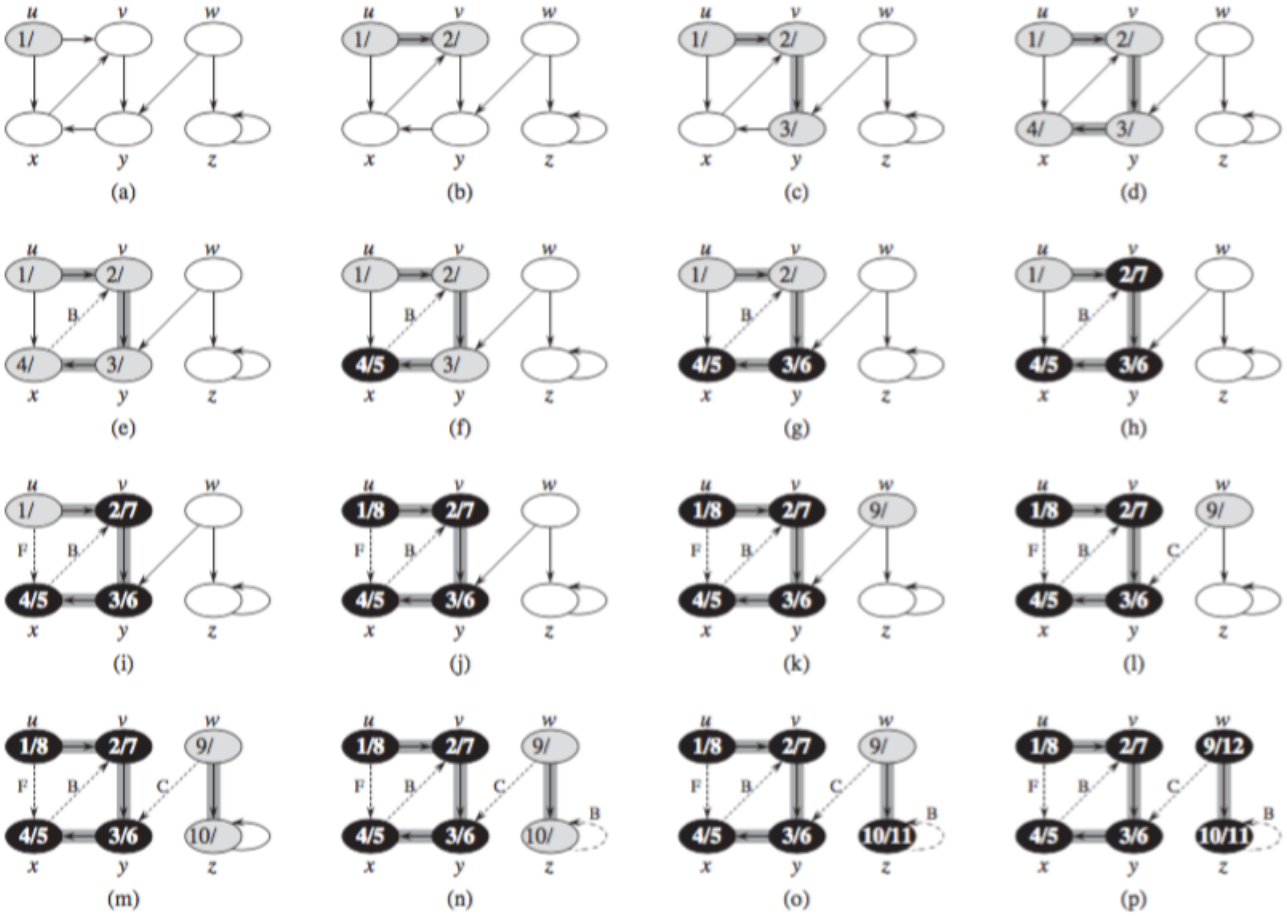

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Timestamps within vertices indicate discovery time/finishing times.

2.2 Runtime

The runtime of depth-first-search is $\Theta(m+n)$. The loops in DFS take $\Theta(n)$, plus the time taken to execute the calls to DFSVisit. DFSVisit is called exactly once for each vertex (taking $\Theta(n)$), since the vertex on which DFSVisit is invoked must be white, and the first

thing DFSVisit does is paint that vertex grey. The loop inside DFSVisit gets executed $Adj[u]$ times for each vertex u, and the total number of entries in Adj (across all the vertices) is just the number of edges in the graph. So this adds $\Theta(m)$ to the runtime.

2.3 Properties of depth-first search

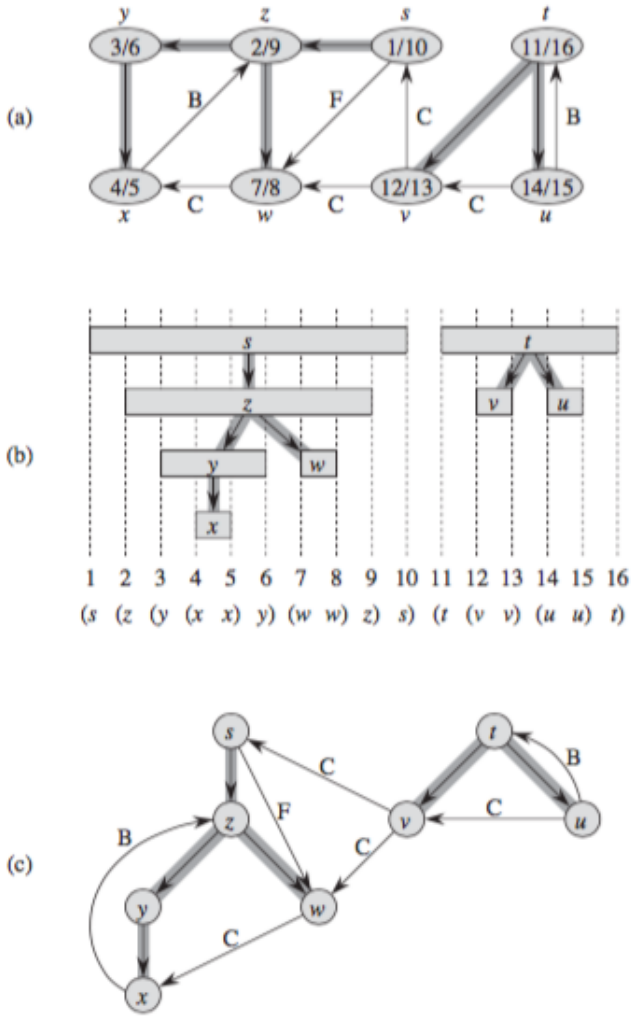

Figure 22.5 Properties of depth-first search. **(a)** The result of a depth-first search of a directed graph. Vertices are timestamped and edge types are indicated as in Figure 22.4. **(b)** Intervals for the discovery time and finishing time of each vertex correspond to the parenthesization shown. Each rectangle spans the interval given by the discovery and finishing times of the corresponding vertex. Only tree edges are shown. If two intervals overlap, then one is nested within the other, and the vertex corresponding to the smaller interval is a descendant of the vertex corresponding to the larger. **(c)** The graph of part (a) redrawn with all tree and forward edges going down within a depth-first tree and all back edges going up from a descendant to an ancestor.

1. Vertex v is a descendant of vertex u in the depth-first forest if and only if v is discovered during the time that u is grey.

2. **(Parenthesis structure)** If v is a descendant of u in the depth-first forest, then $[v.d, v.f]$ is contained entirely within $[u.d, u.f]$. If neither vertex is a descendant of the other, then the intervals are disjoint.
3. **(White path theorem)** In a depth-first forest, v is a descendant of u if and only if at the time $u.d$ that the search discovers u, there is a path from u to v consisting entirely of white vertices.

2.4 Correctness proofs for depth-first-search properties

2.4.1 v is a descendant of u if and only if v is discovered during the time that u is grey

This is because the structure of the depth-first forest precisely mirrors the structure of recursive calls of DFSVisit. Specifically, u is grey only during the time that we are processing the neighbors of u, and the neighbors of u's neighbors, etc.

2.4.2 Parenthesis structure

This mostly follows from the structure of the recursive calls.

Assume without loss of generality that $u.d < v.d$ (i.e. u is discovered before v). We consider two cases:

1. $v.d < u.f$ (i.e. v is discovered before u is finished). In this case v is discovered during the time u is grey, so v is a descendant of u. This also means that DFS-VISIT is called on v while we are still somewhere inside the DFS-VISIT call on u, which means we need to finish up v before finishing up u. Therefore $v.f < u.f$, and $[v.d, v.f]$ is entirely contained within $[u.d, u.f]$.
2. $u.f < v.d$ (i.e. u is finished before v is discovered). Because discovery always happens before finishing, $u.d < u.f < v.d < v.f$, and the intervals are disjoint. Since the intervals are disjoint, neither vertex was discovered while the other was grey, and so neither vertex is a descendant of the other.

2.4.3 White path theorem

If $v = u$, the statement is true, because u is white when we set the value of $u.d$.

($\Rightarrow$) If v is a proper descendant of u, then $u.d < v.d$, so v is white at time $u.d$. This applies to all the vertices on the path from u to v in the depth first forest, because all those vertices are descendants of u. Therefore, the path from u to v consists entirely of white vertices.

($\Leftarrow$) Now suppose there is a path from u to v consisting entirely of white vertices, but v is not a descendant of u in the depth first tree. Consider v', which is the first vertex along the

path from u to v that is not a descendant of u. Let $\pi(v')$ be the predecessor of v'. $\pi(v')$ must be a descendant of u, so by the parenthesis structure, $\pi(v').f \leq u.f$. Furthermore, because there is a path of white vertices from u to v', we know that v' is discovered after u is discovered, but before $\pi(v')$ is finished. So $u.d < v'.d < \pi(v').f \leq u.f$. By the parenthesis structure, $[v'.d, v'.f]$ must be contained within $[u.d, u.f]$, so v' is a descendant of u, producing a contradiction.

3 Topological sort (application of depth-first-search)

Consider a **directed acyclic graph**, which is a directed graph that has no cycles. If we have such a graph, it is possible to arrange the nodes in order, so that all of the edges point from left to right. This is called "topologically sorting" the graph.

Exercise: Why is it impossible to do this if the graph has a cycle?

As an example, directed acyclic graphs can be used to visualize dependencies between tasks. Maybe all your tasks involve putting on clothes, and there is an edge from your socks to your shoes because you have to put on your socks before putting on your shoes.

We can use a topological sort to figure out what order to do the tasks in. This guarantees that we don't try to put on shoes before socks.

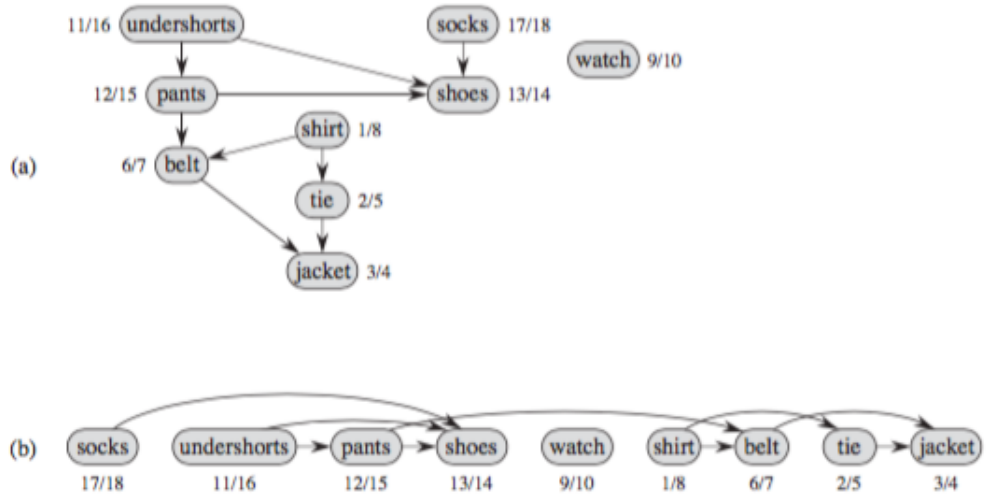

Figure 22.7 **(a)** Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge (u, v) means that garment u must be put on before garment v. The discovery and finishing times from a depth-first search are shown next to each vertex. **(b)** The same graph shown topologically sorted, with its vertices arranged from left to right in order of decreasing finishing time. All directed edges go from left to right.

Formally, a topological sort of a directed acyclic graph 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.

3.1 Algorithm

As it turns out, we can topologically sort a graph as follows:

TOPOLOGICAL-SORT(G)
1 call DFS(G) to compute finishing times $v.f$ 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

This takes $\Theta(m+n)$ time, because depth-first-search takes $\Theta(m+n)$, and inserting into a linked list takes $O(1)$ time for each vertex.

3.2 Correctness

Suppose we run DFS on a graph. It suffices to show that for any pair of distinct vertices u, v, if the graph contains an edge from u to v, then $v.f < u.f$. This way, sorting the vertices in reverse order of finishing time will cause all of the edges to point in the correct direction.

There are three cases:

1. v is a descendant of u in the DFS tree. In this case, $v.f < u.f$ by the parenthesis property.
2. u is a descendant of v in the DFS tree. This is impossible because we assumed the graph had no cycles.
3. Neither vertex is a descendant of the other. In this case, the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint, by the parenthesis property. However, we also know that $v.d < u.d$, because if u had been visited before v, then v would have been white when u was scanned, and by the white path theorem, it would become a descendant of u. Therefore, $v.f < u.f$.