

BASIC GRAPH ALGORITHMS

1. A SLEW OF DEFINITIONS ABOUT GRAPHS

A **graph** G is a pair (V, E) of things: a set V of *vertices* and a set E of *edges*. Vertices are an abstraction: they can represent anything from cities, people, jobs, classes, etc. to other vertices. The edges give relational information: if two vertices u, v are related then there will be an edge saying as much, usually written (u, v) , in the set of edges. If the relational information is symmetric¹ - u related to v is the same as v related to u - then the edges are **undirected** and the graph is an undirected graph. If the relational information isn't symmetric² - u related to v isn't the same as v related to u - then the edges are ordered pairs, meaning $(u, v) \neq (v, u)$ (just like the xy plane ordered pairs $(1, 2), (2, 1)$ aren't equal), are called **directed** edges or arrows, and the graph is a directed graph. Figure 1.1 is a directed graph. The vertices are clothes you would put on when getting dressed and the edges (arrows) represent precedence (you have to put on your underwear before your pants and your pants before your belt).

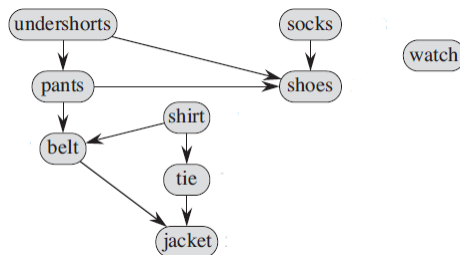


FIGURE 1.1. Order of dressing graph

In an undirected graph the jargon is that the edge (u, v) is *incident* on the vertices u and v . In a directed graph we need to convey information about the direction of the edge so we say that edge (u, v) is *incident from* u and *incident to* v . I've never heard anyone say this but it is true that it's important to convey direction for all edges when describing edges in an undirected graph. If two vertices have an edge connecting them then they're **adjacent**.

A **subgraph** $G' = (V', E')$ of a graph G is any $E' \subset E$ and $V' \subset V$. Given some $V' \subset V$ the subgraph *induced* on G by V' is the graph $G' = (V', E')$ where E' is the set of edges in E connecting vertices in V' , i.e. $E' = \{(u, v) \in E \mid u, v \in V'\}$. It's basically the graph you get by throwing away some vertices.

In a graph the **degree** of a vertex is the number of edges incident on it (i.e. just the number of edges connected to it). In a directed graph we need to also distinguish between edges into the vertex and edges leaving (but just the degree

¹Obvious example: relatives - if you're related to someone then they're related to you.

²Obvious example: parents - if you're someone's parent they're not your parent.

is still the total number): the **in-degree** of a vertex is number of edges coming into the vertex and the **out-degree** is the number of edges leaving the vertex. An **isolated** vertex is one whose degree is 0 (like ‘watch’ in figure 1.1).

A **path** of length k from vertex v_0 to v_k is a sequence of vertices $\left(\underbrace{v_0, v_1, \dots, v_k}_{k+1} \right)$ (i.e. a walk along the edges from v_0 to v_k). The “ k length” comes from the k edges connecting vertices not the $k+1$ vertices: $\left\{ \underbrace{(v_0, v_1)}_{1\text{st}}, \underbrace{(v_1, v_2)}_{2\text{nd}}, \dots, \underbrace{(v_{k-1}, v_k)}_{k\text{th}} \right\}$. Note also that there’s no requirement that the edges or vertices be distinct, meaning a path that retraces its steps is fine. A **simple** path is one where all the vertices in (v_0, v_1, \dots, v_k) are distinct, i.e. that doesn’t retrace its steps. A **cycle** is a path where $v_0 = v_k$, i.e. ends up where it starts. A **simple cycle** is a cycle that doesn’t retrace its steps. A graph with no cycles is called **acyclic**.

A graph is **connected** if there’s a path from any vertex to any other vertex. A graph that isn’t connected isn’t “disconnected”! It’s just not connected. The graph in figure 1.1 is not connected because there’s no path from any vertex to ‘watch’. Note in a directed graph you can’t “go backwards” along edges so also there’s no path from ‘pants’ to ‘undershorts’. A **connected component** $V' \subset V$ of an undirected graph is a set of vertices such that the graph G' induced on G by V' is connected, i.e. it’s a subset of the original vertices where any vertex in that subset is reachable from any other in that subset. In figure 1.1 if you flip the edge (undershorts, shoes) to (shoes, undershorts) then $V' = \{\text{undershorts, shorts, shoes}\}$ becomes a connected component because you can just “go around the path” from any of them to any other. There can be several connected components in a graph. In figure 1.1 if you also flip the edges (shirt, tie) and (tie, jacket) then $V' = \{\text{undershorts, pants, shoes}\}$ and $V'' = \{\text{shirt, tie, jacket}\}$ become connected components (but are not connected to each other).

Two graphs G, G' are **isomorphic** if there’s a bijection $f : V \rightarrow V'$ such that $(u, v) \in E \iff (f(u), f(v)) \in E'$, i.e. essentially the same except for a relabeling of the vertex names. Figure 1.2 shows two isomorphic graphs because the mapping

$$\begin{aligned} f(u_1) &= v_1 \\ f(u_2) &= v_4 \\ f(u_3) &= v_3 \\ f(u_4) &= v_2 \end{aligned}$$

works (all you do is “twist” vertices u_2, u_4).

Graphs with particular properties get fancy names. A **complete** graph is a graph where every vertex has an edge to every other vertex (how many edges is that?). Figure 1.3 shows complete graphs for³ $|V| = 2, \dots, 7$, i.e. 2 vertices, 3 vertices, ..., 7 vertices. Are these unique? I.e. is every other complete graph with 7 vertices isomorphic to K_7 ? A **bipartite** graph is a graph that can be divided into two sets V_1, V_2 such that the edges of the graph only connect vertices in V_1 to V_2 and vice versa (i.e. vertices in V_1 are not connected to each other). A **bipartite** graph is a graph that can be divided into two sets V_1, V_2 such that the edges of the

³ $|V|$ is the number of vertices and $|E|$ is the number of edges.

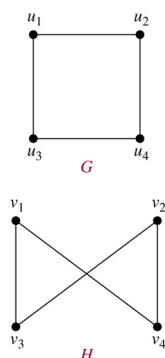
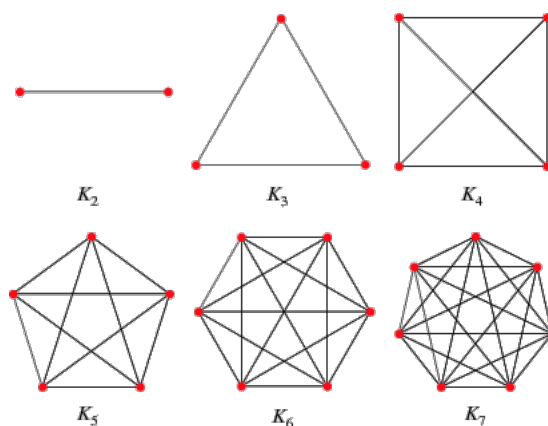


FIGURE 1.2. Isomorphic graphs

FIGURE 1.3. Complete graphs for number of vertices $2, \dots, 7$

graph only connected vertices in V_1 to V_2 and vice versa (i.e. vertices in V_1 are not connected to each other). Figure 1.4 is a bipartite graph. An acyclic undirected graph is a **forest** (need not be connected) and a connected, acyclic, undirected graph is a **tree**⁴. A **directed acyclic graph** is usually shortened to **dag**.

Finally the **contraction** of an undirected graph by an edge (u, v) is formed by removing the edge and making $u = v$, i.e. contract u and v to be one vertex, with the edges connected to each of u, v now connected to the contracted vertex.

A **weighted** graph is one where the edges have numbers associated with them called “weights”. For example if vertices were cities in Florida and edges represented “being connected to by freeway” (any number of freeways but only freeways) then the edge weights might represent the number of highway changes you have to make when going from one city to another.

⁴There are tons of refinements of tree, e.g. binary tree, complete binary tree, full binary tree, etc.

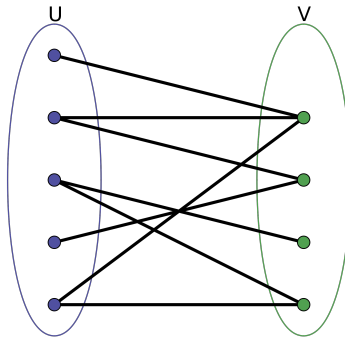


FIGURE 1.4. Bipartite graph

2. REPRESENTATIONS OF GRAPHS

A representation of a graph is a data structure of some sort that captures information about the vertices and the edges connecting the vertices. The two standard representations are an **adjacency list** (which uses a list of linked lists) and an **adjacency matrix** (which uses a matrix obviously).

How to construct the adjacency list representation: first enumerate all the vertices of the graph (i.e. give each vertex a unique numerical “name”), then create a list or array type data structure of length n and for each entry in this list store another list. The index of the first list corresponds to some vertex and the list fetched upon indexing corresponds to all of the vertices its connected to. For example consider the graph in figure 2.1. The structure on the right is the adjacency list: the first column is an array or vector or something with pointers to the heads of linked lists. For example the 3rd position in the array corresponds to vertex 3 and hence has a pointer to a linked list that contains nodes (linked list nodes not graph nodes) that store the number identifying vertices that vertex 3 is connected to (a node with the number 6 and a pointer to another node with the number 5 [and the null pointer because vertex 3 isn’t connected to any other vertices]). Note that the order of the vertices as they appear in the linked lists don’t mean anything (for example 2 precedes 4 in vertex 1’s linked list but vertex 6 precedes vertex 5 in vertex 3’s linked list). Also note that since this is a directed graph the adjacency lists respect this: vertex 2 doesn’t have vertex 1 in its adjacency list.

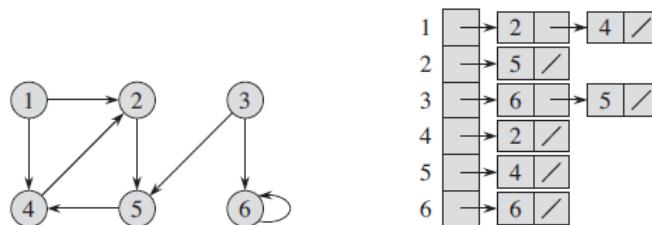


FIGURE 2.1. Adjacency list

How to construct adjacency matrix representation: create a 2d array (matrix) of size $|V|^2$ filled with zeros. Then fill in 1s on row i position j if vertex i is connected to j . Figure 2.2 is the adjacency matrix representation for the graph in figure 2.1. Note that since this graph is directed the adjacency matrix representation respects this: just like for the adjacency list representation row 2 (corresponding to vertex 2) doesn't have a 1 in the first position because there's no edge from vertex 2 to vertex 1, but there is a 1 in the first row's (the row corresponding to vertex 1) second position because there is an edge from vertex 1 to vertex 2. What would the graph look like if the graph was undirected? Hint: an undirected graph is like a directed graph but for any directed edge (i, j) there also exists directed edge (j, i) .

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

FIGURE 2.2. Adjacency matrix

When is an adjacency list representation better than an adjacency graph representation? When the graph is “sparse”, which means vertices are on average adjacent to a small percentage of the other vertices. Why? Because adjacency lists store only information about present edges while the adjacency matrix explicitly stores zeros to indicate when 2 vertices aren't adjacent. Notice that the matrix in figure 2.2 has many more numbers in it (if you count the zeros) than the adjacency list in figure 2.1. Conversely when a graph is “dense”, which means vertices are on average connected to a large percentage of the other vertices, the adjacency matrix representation is better because the added cost of traversing a linked list isn't worthwhile for example if you're trying to figure out if two vertices are adjacent (in the adjacency matrix representation you would just index directly to the other vertex and check for a 0 or 1 whereas in the adjacency list you have to traverse the list). For the adjacency list the space cost⁵ is $\Theta(|V| + |E|)$ which means you need space proportional to the number of vertices (the length of the “top” array) and the number of edges (one linked list node for each edge). For the adjacency matrix representation the space cost is $\Theta(|V|^2)$ regardless of the density or sparseness of the graph. Notice that if $|E| = |V|(|V| - 1)$, the maximum number of edges, then

$$\Theta(|V| + |E|) = \Theta(|V| + |V|(|V| - 1)) = \Theta(|V|^2)$$

So worst case the space cost of an adjacency list representation is the same as that of the adjacency matrix representation.

Both representation can be adapted to represent weighted graphs. The linked list nodes in the adjacency list can be pointers to some other data structure that has fields that store weight attributes and the adjacency 2d array can similarly store

⁵Big Θ means best and worst case. Big O means worst case. Big Ω means best case.

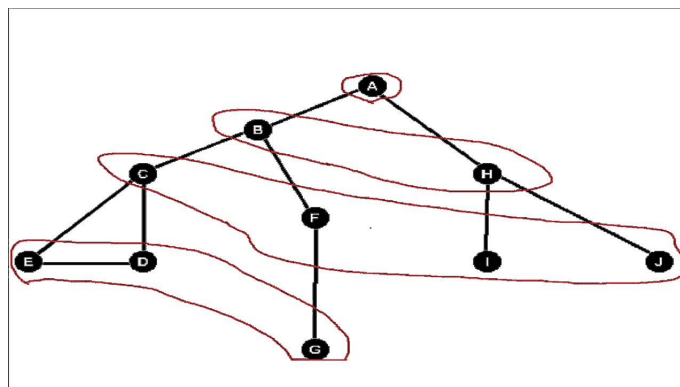


FIGURE 3.1. BFS

pointers or simply have binary entries (the entries can be real numbers but note that in that case 0 weight might be semantically distinct from “not adjacent to” in which the case appropriate thing might be something like some entry representing ∞).

3. BREADTH-FIRST SEARCH

Breadth-first search (BFS) is better called breadth-first discovery because you’re not necessarily searching for anything you’re just exploring the graph. It’s the simplest thing you could think of doing: if you’re at some vertex “discover” all of its adjacent vertices then for each one of those discover each of its adjacent vertices, and so on. Note you go in order of discovery. So first discover all neighboring vertices v_1, v_2, \dots, v_k of some **source** vertex s , then discover all the neighboring vertices $v_{1,1}, v_{1,2}, \dots, v_{1,k'}$ of v_1 and then move on to discovering all adjacent vertices of v_2 , not all adjacent vertices $v_{1,1}$. So the discovery process goes in frontiers of number of edges from the source: discover all vertices within one edge of s , then within two edges of s , etc. Consider figure 3.1: The source is A from which B,H are discovered. From B the vertices C,F are discovered and then from H the vertices I,J are discovered. The levels are circled in red and correspond to frontiers of discovery. The order of visiting the vertices in BFS suggests an implementation: you want to discover the first vertex that’s one edge away, then the second vertex that’s one edge away, etc. until there are no more vertices one edge away. Then you want to “visit” the first vertex discovered at the one edge frontier and discover its “children”, then move on to the second vertex discovered at the one edge frontier and discover its “children” and so on. Hence you want to do the discovery process in a first in first out order: the vertices discovered earlier should be visited earlier. Obviously this suggests using a queue data structure. It’s also important to distinguish between vertices that have been discovered and those which have been fully explored so that vertices that have been discovered don’t get “re-discovered” (since a vertex might have an incoming edge from two other vertices in a preceding frontier) and those which have been fully explored don’t get explored again. Algorithm 1 lists pseudocode for a function that takes a graph G and a source s (from which to start the BFS) and performs the breadth-first search That’s it - that’s basic BFS. What’s the runtime of BFS? Since you have to traverse each edge it’s $O(|E|)$.

Algorithm 1 Breadth-first search

```

// G is a graph and s is the vertex from which
// the search "emanates"
BFS(G, s)
// initialize all vertices as undiscovered and unexplored
// G.V is the set of vertices of the graph
for v in G.V
    v.status = undiscovered

Q = Queue
// the first vertex discovered is the source
s.status = discovered
Q.push(s)
while Q.empty != true
    v = Q.pop
    // G's adjacency information is abstracted away here
    // i.e. G.adj(v) queries either the adjacency list
    // or adjacency matrix for all vertices adjacent to v
    for u in G.adj(v)
        // u might be adjacent to some other vertex and hence
        // might have already been discovered
        if u.status == undiscovered
            u.status = discovered
            Q.push(u)
    // discovered all vertices adjacent to v hence
    // v is fully explored
    v.status = explored

```

Now there are variations on the theme: for example you could use BFS to compute the number of edges from the source to every other vertex it's connected to. How? Note that every vertex is one frontier deeper than the vertex "through which" it was discovered: so if the number of edges $s.d$ from the source to itself is 0 then the number of edges from the source to each of the vertices discovered after exploring the source is $s.d + 1$. Setting this "distance" upon discovering the vertices adjacent to some vertex gives the correct distance of the discovered vertex from the source. Why? Because no vertex is discovered twice and so when $u.d$ is set it's set as early as it can be. There is also the notion of a **breadth-first predecessor tree**. This is basically a representation of which vertices were discovered by exploring which vertices in a breadth-first search. You can imagine rearranging the graph such that s at the center and the frontiers of vertices form concentric circles around s . Figure 3.2 demonstrates this. Formally for a graph $G = (V, E)$ and s the breadth-first predecessor tree G' consists of all vertices $V' \subset V$ that were discovered in a BFS emanating from s and edges $(u, v) \in E' \subset E$ if v was discovered through exploring u . It's called a tree because there are no cycles (why?), it's connected (why?). The "root" of the tree (figure 3.2) will be the source vertex s . How would you construct the tree? Just keep track of which vertex a vertex was discovered through by storing a "parent" pointer that's set when the vertex is discovered. Using the BFS

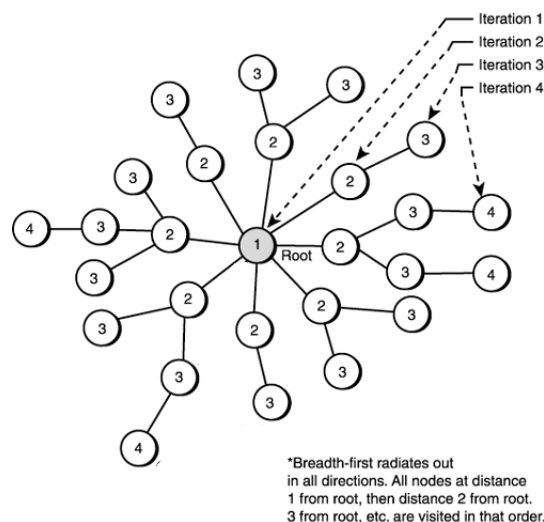


FIGURE 3.2. Breadth-first predecessor tree

predecessor tree you can also print out shortest paths to the source, in the sense of fewest edges, by following the parent pointers back to the source.

4. DIJKSTRA'S SHORTEST PATH (OPTIONAL)

You haven't read the section on depth-first search, nor topological sort, nor strongly connected components, but this section appears here because topological sort and strongly connected components are applications of depth-first search and Dijkstra's shortest path algorithm is an application of breadth-first search.

As alluded to at the end of the BFS section you can use BFS to compute shortest paths from a single source to all vertices using a BFS tree. One way to think about the problem that that solves is computing the shortest path from a single source to all vertices on a weighted graph where all the edges have weight 1. But what if the edge weights aren't one? It won't work. BFS only gives you shortest path in the sense of fewest number of edges, not in terms of sum of edge weights (obviously since the algorithm never checks the edge weights). How could you use BFS to solve the problem of single source all vertices⁶ in terms of sum of all edge weights? Well if the edge weights are integral (have whole number values) one hack would be to insert $k - 1$ vertices on any edge with weight k and then the number of edges distance matches the edge weight distance. But that would blow up the run time of BFS since it's $O(|E|)$ and $|E|$ will blow up.

The smarter way is to make BFS a little smarter: expand the frontier in the direction of the "nearest" vertices, where nearest now means those for which the path is shortest in the sense of sum of edge weights. In addition to expanding in the direction of the nearest vertices you should always update your current assessments of shortest paths to all vertices seen so far. Effecting the first part is easy: just replace the Queue in BFS with a Min-Heap where the priority ranking of each vertex is its current shortest path distance from the source. Then the nearest vertices

⁶Compute shortest paths from a single source to all other vertices.

will be popped. To effect the second part is easy too: after popping a vertex update all of the distances to all of its neighbors if the shortest path to them (from the source) is shorter through the vertex that just got popped. The code appears in Algorithm 2.

Algorithm 2 Dijkstra's shortest path

```

Dijkstras(G, s)
// each entry in this array will store
// the distance from the source to this vertex
// the vertices are numbered 1 to |G.V|
dist = array(|G.V|)
prev = array(|G.V|)
Q = minHeap

dist[s] = 0

// for each vertex except the source
for v in {G.V - s}
    // initialize every vertex to be infinitely far from the source
    dist[v] = ∞
    // in order to construct the paths we need the predecessor vertex for every vertex
    // on the shortest path to that vertex from the source
    // initialize to null in case some vertex is unreachable and to prevent
    // garbage in the array from giving incorrect answers
    prev[v] = null
    // the current priority rank of each vertex except the source
    // is ∞
    Q.add(v, dist[v])

while Q.empty != true
    // pop the closest (lowest rank) vertex
    v = Q.pop
    // update only those vertices we're not done with
    for u in G.adj(v) where u ∈ Q
        // if the distance (sum of edge weights) along path path from s to u through v)
        // is less than the current distance to u then shortest path should go through
        // v
        dvu = dist[v] + weight(v, u)
        if dvu < dist[u]
            dist[u] = dvu + weight(v, u)
            prev[u] = v
            Q.change_priority(u, dvu)

```

What's the runtime of Dijkstra's shortest path algorithm? The breadth-first search costs $O(|E|)$ but since you have to do $|V|$ insertions in the MinHeap, which each cost at most⁷ $\log(|V|)$ (for example when the heap is almost full) the total runtime is $O(|E| + |V| \log(|V|))$.

⁷Using a Fibonacci heap.

5. DEPTH-FIRST SEARCH

Depth-first search is almost exactly like breadth-first search except that it explores as deeply as possible (as opposed to as broadly as possible) on each iteration. Practically speaking the only difference is the replacement of the Queue with a Stack. Algorithm 3 shows the algorithm. Why should it be a Stack? The easiest way to see it intuitively is to go through an example on a small graph. Consider figure 5.1: first A is pushed (so it's at the top of the stack), then popped and explored, then C and B are pushed (so B is at the top of the stack), then B is popped and explored, then E and D are pushed (so D is at the top of the stack, followed by E). D has no neighbors so E is the next to be popped and explored, pushing I and then H. The current state of the stack from bottom to top is (C,I,H). You get the idea. Essentially the Stack orders which vertices are to be explored by which were explored most recently (they appear at the top of the Stack!).

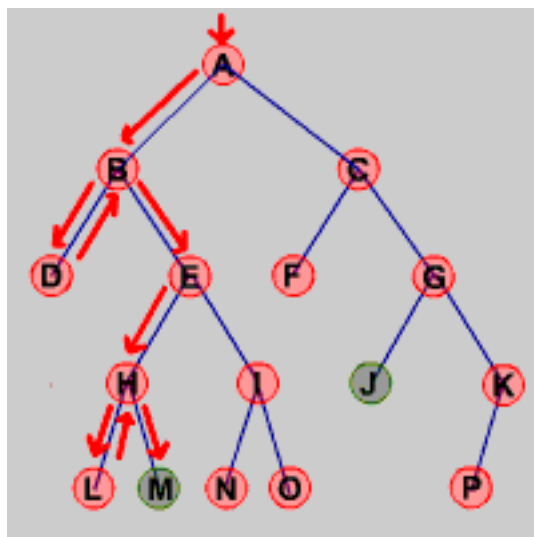


FIGURE 5.1. Depth-first search example

Algorithm 3 Depth-first search

```

DFS( $G, s$ )

for  $v$  in  $G.V$ 
     $v.status = undiscovered$ 

 $T = \text{Stack}$ 
 $s.status = discovered$ 
 $T.push(s)$ 
while  $T.empty \neq \text{true}$ 
     $v = T.pop$ 
    for  $u$  in  $G.adj(v)$ 
        if  $u.status == undiscovered$ 
             $u.status = discovered$ 
             $T.push(u)$ 
     $v.status = explored$ 

```

What's the runtime of the algorithm? Since you need to traverse each edge in the worst case it's $O(|E|)$. Finally just like for breadth-first search you can construct a depth-first predecessor tree by keeping pointers.

6. TANGENT: PARENTHESES

The standard DFS algorithm can be easily modified to make it infinitely more useful: let's keep track of when a vertex is first discovered, its "open" time, and when all of its children are done being explored, i.e. when it is "closed". To enable this bookkeeping some trivial modifications need to be made. Algorithm 4 is the modified algorithm.

How does the modified algorithm work? Keeping track of open times is easy: just keep a global timer (which you increment constantly) and assign the value of the timer to each vertex when it is discovered. The more difficult task is keeping track of the close time. Before even attempting to figure out what times to assign as close times note that in the unmodified algorithm the vertex is popped and disappears (in the sense that we don't keep track of it anymore) so we couldn't assign a time to it (whatever time) even if we wanted to. The solution is to "save" the vertex, i.e. don't pop it. Hence peeking at it instead of outright popping it.

But now we need to distinguish between vertices on the stack because they are yet to be explored and those we're saving just in order to assign close times to; hence the conditional. If the top of the stack is a vertex that hasn't been explored then we do the same thing as the unmodified algorithm and if it's one that has been explored then we know it's one of those that we're holding onto in order to assign it a close time. Also (coincidentally enough) if the top of stack is a vertex we've already explored that means that all of its children have been explored too (that's the only way you "come back" to a vertex) and it should be closed. Note the counter is always incremented so that every open and close time is unique (i.e. no set of vertices has the same open time for example; it's not possible for two vertices to have the same close time since the algorithm isn't parallel).

What's the use of all of this bookkeeping? We can use the open and close times to "parenthesize" the vertices:

$$\text{MDFS}(G, s)$$

Theorem. Parenthesis theorem. Given a DFS of a graph $G = (V, E)$ for any two $u, v \in V$ **either** the intervals $[u.open, u.close]$ and $[v.open, v.close]$ are entirely disjoint, i.e. $u.close < v.open$ **or** one of $[u.open, u.close], [v.open, v.close]$ is entirely contained in the other, i.e. $u.open < v.open < v.close < u.close$.

What this theorem says is that for any two vertices u, v either they're completely unrelated (e.g. u opens and closes before v opens and closes) or one of u, v is an ancestor of the other (e.g. v is "down the graph" from u and hence u opens, sometime later v opens, then eventually v closes, and sometime later u closes). Note this is not just about direct descendants, i.e. parents and children, but applies to any depth of descendance. Why is this called the parenthesis theorem? Look at this sequence of parentheses

 $((()())(((((())()))))))$

and notice how for any two sets of parentheses are disjoint (for example the bolded ones) or one is nested in the other.

In a sort-of but not completely related interesting tidbit if you wanted to test whether a set of parentheses was parenthesized correctly, i.e. nothing like $)()()$, you'd use a stack. Can you think up how?

7. TOPOLOGICAL SORT

A topological sort of a directed graph is a sorting of the vertices by precedence (i.e. which vertices “globally” precede which). For example figure 7.1 is a topological sort of the graph in figure 1.1 that represents an order in which you can successfully get dressed: first socks, then undershorts, then pants, then shoes, then watch, etc. Note the edges are just reproduced from the graph - the topological sort is the order of vertices from left to right. I say an order because for example undershorts could be put on before socks because there's no precedence relationship (no edge or sequence of edges) between socks and undershorts. Another name for a topological sort is a *linearization* of a graph (because it kind of arranges the vertices in a line).

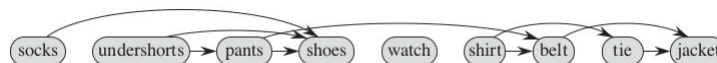


FIGURE 7.1. Topological sort of clothing precedence

A topological sort only makes sense for a directed graph - there is no notion of precedence in an undirected graph. Furthermore a topological sort is only possible in a directed acyclic graph. Why? If there's a cycle in the graph then which vertex in the cycle really precedes which other vertex? The precedence of the vertices in the cycle with respect to each other is ill-defined. How do you detect a cycle in a directed graph? Easy: if at some point during a depth-first search you were about to “discover” a vertex that had already been discovered (i.e. but not explored) then there's a cycle. Why? Since all discovered vertices get turned into explored vertices as the DFS “comes back up” the only way to see a discovered vertex that hasn't been fully explored is if the DFS has looped back. Therefore assume without loss of generality that the directed graphs in this are directed acyclic graphs (meaning before you try to do a topological sort of a graph check to see if it's acyclic).

How do you do a topological sort of an acyclic graph? Easily! Do a depth-first search and just list the vertices in reverse order of their closing times. I'll repeat that again: list the vertices in reverse order of their closing times **not their opening times!!!** Why? As a warm-up think about it like this: what is the closing time of ‘undershorts’ in figure 1.1 if I start the depth-first search at it? It'll be one of the last ones⁸ because everything below it will close before it (since everything below it is a descendent of it and by the parenthesis theorem). Now figure 1.1 is kind of strange because ‘socks’ is never discovered by a depth-first search starting at ‘undershorts’. The solution is essentially to rerun depth-first search start at vertices that remain undiscovered while keeping the timer global. The algorithm appears in Algorithm 5.

⁸And in fact if you look at figure 7.1 you'll see it's the second item.

Algorithm 5 Topological Sort

TopSort (G)

```

for v in G.V
    v.status = undiscovered

    // for use in keeping track of whether
    // some vertices haven't been discovered in
    // the DFS, it's kind of a made up data structure
    // that supports indexing and direct removal
    unsearched_vertices = Set(G.V)

    // this linked list will store
    // the vertices in reverse order
    // of their closing times
    L = LinkedList

    time = 1
    // the outer loop reinitializes everything
    // except time for each DFS
    while unsearched_vertices.empty != true
        s = unsearched_vertices[1]
        // this removes s from unsearched_vertices
        // since it's been discovered (so that another DFS
        // won't be run starting at s)
        unsearched_vertices.remove(s)
        T = Stack
        s.status = discovered
        s.open = time
        T.push(s)
        while T.empty != true
            v = T.peak
            if v.status != explored
                for u in G.adj(v)
                    time = time + 1
                    if u.status == undiscovered
                        u.status = discovered
                        unsearched_vertices.remove(u)
                        u.begin = time
                        T.push(u)
                v.status = explored
            else
                time = time + 1
                v.close = time
                // the first stored v will eventually
                // be pushed all the way to the end of the
                // list
                L.push(v)
            T.pop

```

What's the runtime? Since it's essentially a DFS the runtime is $O(|E|)$.

8. STRONGLY CONNECTED COMPONENTS

Recall that a connected component of a undirected graph G is a subgraph G' that's connected, i.e. each vertex can reach each other vertex. A *strongly* connected component is the same thing in a directed graph. A graph could have many distinct (unreachable from each other) connected components. How would you find them all? For an undirected graph it's easy: use either breadth-first search or depth-first search. Since the graph is undirected any path from a vertex u to a vertex v can be similarly used by v to reach u . For directed graphs it's a little more difficult...

How about an intuition pump⁹: consider the directed graph in figure 8.1

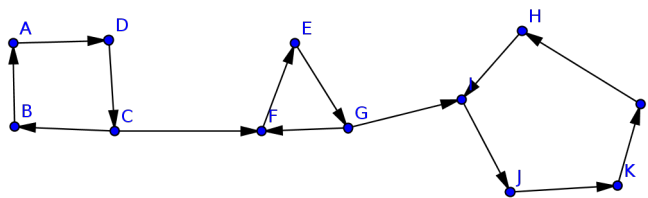


FIGURE 8.1. Strongly connected components

. First of all do you recognize the connected components? They are

$$C_1 = \{A, B, C, D\}$$

$$C_2 = \{E, F, G\}$$

$$C_3 = \{H, I, J, K, L\}$$

Note that the entire graph is not strongly connected (since, for example, A is unreachable from K).

So how would algorithmically decide all of the connected components in this graph? Maybe a first guess would be to perform a depth-first search from and see which vertices are discovered by that depth first search. That clearly doesn't work because all vertices are reachable from A . In contrast how about a performing a

⁹An intuition pump is a thought experiment structured to allow the thinker to use their intuition to develop an answer to a problem. The term was coined by Daniel Dennett."

depth-first search that starts at H ? Well then the only vertices discovered will be $\{H, I, J, K, L\}$. That's C_1 the connected component H is a member of! This is very close to what we need but this doesn't work if there's a cycle. All of the vertices you can reach from some fixed vertex (e.g. F) are not all in that vertex's strongly connected component. To drive the point home: if you had started the DFS at F you would have discovered

$$\{E, F, G, H, I, J, K, L\} = C_1 \cup C_2$$

Not all of the discovered vertices are in F 's connected component but all of vertices in F 's connected component are discovered!

What's the key feature here? Look again at C_3 and think about what happens when you perform the depth-first search starting at H . The DFS discovers all the vertices in the "last" part of the graph only because it's stuck; because the edge (G, I) only goes one direction the DFS can't get out. This is the key. If you knew what the "last" part of the graph was you could run DFS on it, collect all the vertices discovered and label them as one strongly connected component, remove all of them, then rerun DFS on the "second to last" part, etc.

This is almost the answer **but it's not, unfortunately**. Consider the graph in figure 8.2

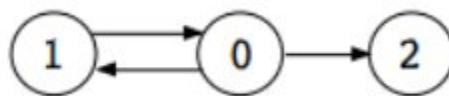


FIGURE 8.2. Counter-example

. A plausible DFS that starts at 0 gives finishing ordering of first 1, then 2, then 0. Then your second DFS, starting at 1, would produce $\{1, 0, 2\}$ as a strongly connected component since both 0 and 2 are reachable from 1. The reason for this, vaguely, is that there's no well defined "last" part of the graph. But there is a well defined "first" part to the graph (at least relative to where you start the DFS from)! A topological sort¹⁰ gives you the "first" part as the vertex with the highest finishing time. But how can we use this since the whole algorithm hinges on "getting stuck" and clearly for graphs like in figure 8.1 you wouldn't get stuck if you started at A .

The solution is to use, for the second depth-first search the transpose graph¹¹, i.e. flip all the edges in the original graph. Note that the strongly connected components of G^T and G are the same¹² so there's no loss of reachable paths. So now you can effect the same idea (using the "getting stuck" nature of the second DFS) on the "first" part of the graph: just start the second DFS on the vertex with the highest finishing time (i.e. the one that's topologically sorted to be the first in order). Once you identify the strongly connected component of the "first" part of

¹⁰We're not really going to do a topological sort since the graph obviously has cycles, but what is true is that the graph of strongly connected components is a dag.

¹¹ $G^T = (V, E^T)$ where $E^T = \{(v, u) \mid (u, v) \in E\}$.

¹²Since if you can reach v from u and u from v then flipping all the edges means the paths have just switched roles; the path from u to v because the path from v to u and vice-versa.

the graph, remove all those vertices and do the same thing with the next highest finishing time vertex that's still left. The source appears in Algorithm 6.

Algorithm 6 Strongly connected components

```

StronglyConnected(G)

// augment TopSort to return a stack with the vertices in
// topologically sorted order, i.e. first at the top
T = TopSort(G)
G' = GT

// the rest is basically DFS except in topological order
for v in G'.V
    v.status = undiscovered
    v.parent = nil

while T.empty != true
    v = T.pop
    for u in G'.adj(v)
        if u.status == undiscovered
            u.status = discovered
            // construct a DFS predecessor tree
            // so that you can identify the strongly connected
            // components
            u.parent = v

// each connected component is rooted at the vertex
// that was explored in the above dfs
dfsForest = Map

for v in G'.V
    pred = v.parent
    while pred != nil
        pred = pred.parent
    v.root = pred
    // v.root will be "name" of the connected
    // component that v is a member of
    dfsForest[v.root].add(v)
  
```

What's the runtime of the algorithm? Each DFS takes $O(|E|)$ but the computation of the transpose graph takes $O(|V| + |E|)$ so the total runtime is $O(|V| + |E|)$.