CMSC 510: BFS & DFS — Pseudocode

Acacia Ackles

September 23, 2022

Lecture notes adapted from *Introduction to Algorithms 4e*, Cormen, Leiserson, Rivest, and Stein.

Review

Recall last class we came up with something we called a disjoint-set data structure to represent some buildings and their shared connections.

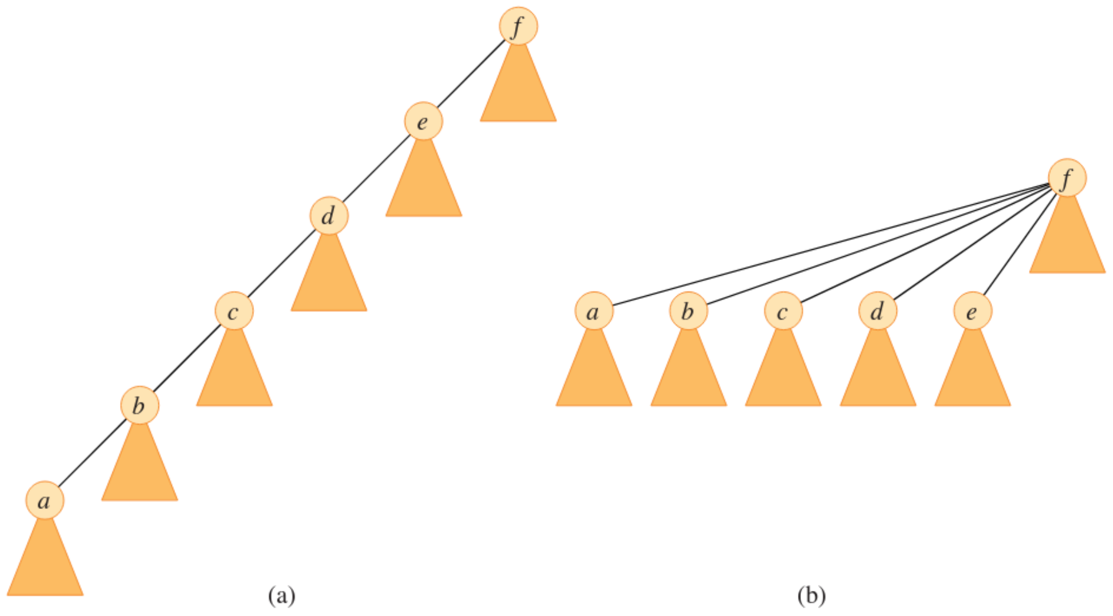


Figure 19.5 Path compression during the operation FIND-SET. Arrows and self-loops at roots are omitted. **(a)** A tree representing a set prior to executing FIND-SET(a). Triangles represent subtrees whose roots are the nodes shown. Each node has a pointer to its parent. **(b)** The same set after executing FIND-SET(a). Each node on the find path now points directly to the root.

We also ended up with an algorithm to do the exact operations we were interested in, namely:

1. `FIND-SET(x)`, to determine whether two nodes are in the same set, and
2. `MAKE-SET(x)` and `UNION(x,y)`, to add new nodes and new edges to our sets

This data structure and its associated algorithm are good if those are the two operations you want to do the most. However, sometimes we want to do some more general things with network-like data structure.

Commonly, we might want to know more than just whether two nodes are in the same set; we would like to understand how they are connected and what the structure of the connections is.

In this case, we turn to some general *graph algorithms.*

Graph Representations

What is a Graph?

Before we perform any sort of algorithms on our graphs, we need to be able to represent them computationally.

Visually, a graph is a series of nodes or vertices connected by edges. We can draw this quite simply.

Representing this in the computer is a bit more challenging. First, we have to turn our simple image into a more formal concept of what a graph is, mathematically.

A graph G is a collection of vertices, V, and a collection of edges, E, where each edge in E connects exactly two vertices in V. Essentially, a graph can be represented by listing its vertices and its edges as follows:

```
V = {1, 2, 3, 4, 5}
E = {(1, 2), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (4, 5)}
```

EXERCISE: Draw the graph described above.

How Do We Represent a Graph?

Last class we discussed at length some representations of the graph of buildings at Lawrence University and their connections. While some of those ideas were incomplete for the exact task we wanted, they were very good for general graph algorithms.

In general, there are two common representations for graphs: adjacency-list and adjacency-matrix.

Def. The **adjacency-list representation** of a graph consists of an array, A, of $|V|$ lists. For each $u \in V$ the array $A[u]$ contains all vertices v such that $(u, v) \in E$.

Def. The **adjacency-matrix representation** of a graph arbitrarily numbers the vertices in V and constructs a $|V| \times |V|$ matrix, called A, such that

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

Construct the two different representations of the graph described above.

QUESTION: When might we want to use one representation over another?

We might want to use an adjacency list when we are worried about the size of memory required, when our graph is very sparse, or when we are not so interested in doing a lookup to see if two nodes are connected.

Conversely, we might want to use an adjacency matrix when our graph is densely connected or when we would like to quickly look up a node connection.

DISCUSSION: How might you change these representations from undirected to directed graphs? From unweighted to weighted?

What Can We Do With Graphs?

As mentioned earlier a common task for graphs is to find out the structure of the graph so that we can use this information to our advantage. After all, if the structure was unimportant, we could just store this information in a simple list. So usually we want to understand how the edges and vertices connect to one another.

To do this, we will have to turn to some graph search algorithms.

Search Algorithms for Graphs

In-Class Exercise: Design an Algorithm

What would you do in order to search a graph?

Imagine I give you this graph. In words, describe how you might traverse the graph such that you visit every vertex at least once.

Your approach should also provide some information on the structure of the graph. This means you should either be able to:

1. Provide some information about the shortest possible distance between some vertices in the graph, or
2. Provide some information about the order of your search's traversal through the graph

After you attempt this, we can turn to looking at common algorithms for these approaches.

Breadth-First Search

The first algorithm we will look at is Breadth-First Search. This algorithm is commonly used when we are more interested in the first requirement: providing some information about distance between vertices. In particular, it's commonly used when we're interested in exploring the connections to a single *source vertex* in the graph.

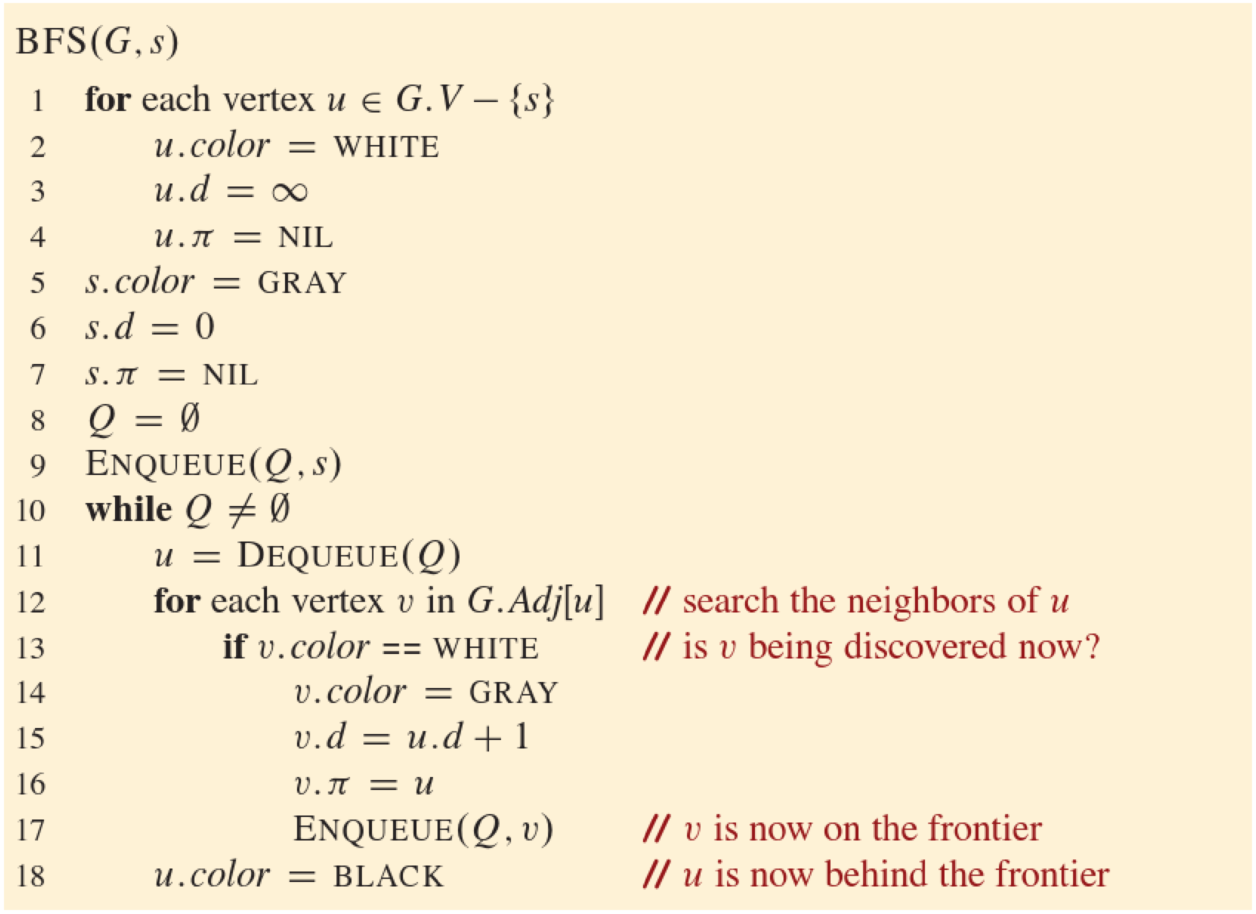
BFS explores the graph in "waves", starting with those vertices closest to the source, then two away, then three away, etc.

BFS also uses "colorings" to keep track of the vertices it is exploring.

- Vertices begin colored `white` and remain white until they are discovered in the search
- The first time a vertex is reached in the search, it is *discovered* and becomes `grey`.
- Once all edges of a grey vertex have been explored, it becomes `black`.

Finally, BFS keeps track of the distance from the source vertex to each other vertex.

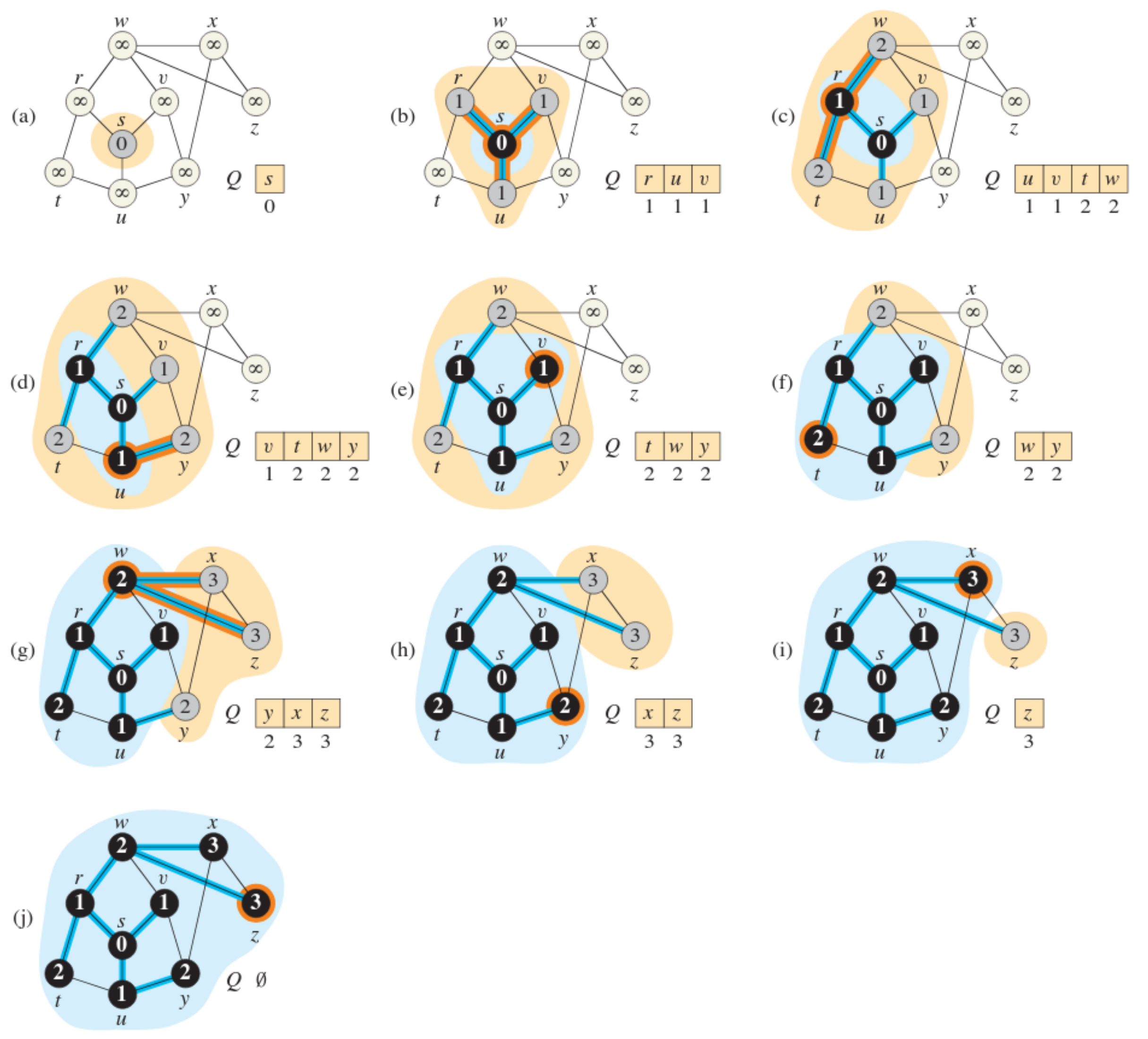
Let's look at the pseudocode and then discuss some of the implementation details involved.



```
BFS($G, s$)
for each vertex $u \in G.V - \{s\}$
    $u.color$ = WHITE
    $u.d = \infty$
    $u.\pi$ = NIL
$s.color$ = GRAY
$s.d = 0$
$s.\pi$ = NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u$ = DEQUEUE($Q$)
    for each vertex $v$ in $G.Adj[u]$      // search the neighbors of $u$
        if $v.color$ == WHITE              // is $v$ being discovered now?
            $v.color$ = GRAY
            $v.d = u.d + 1$
            $v.\pi = u$
            ENQUEUE($Q, v$)                // $v$ is now on the frontier
    $u.color$ = BLACK                      // $u$ is now behind the frontier
```

Notice that BFS maintains a *queue* of the next vertices to be processed, and adds to the queue while there are still vertices to search.

Here's a graphical representation of BFS.



QUESTION: Where is the part in this algorithm where we move on to the next vertex to process?

QUESTION: Will this algorithm visit every vertex? Will this algorithm visit every edge? Make an argument.

Depth-First Search

An alternative to breadth-first search is depth-first search. This search is commonly used when we are less interested in finding shortest paths and more interested in doing an exhaustive search as part of some larger algorithm. (There are other reasons to use one over the other, but this is a common one.)

Depth-First Search (DFS) explores edges out of the most recent vertex that still has unexplored edges.

The coloring for DFS works in the same fashion as BFS.

Another difference is what is tracked at each vertex. In DFS, we are not tracking distance from the source, but rather *timestamps* for each vertex. Vertices are timestamped first when they are visited, and second when they are finished.

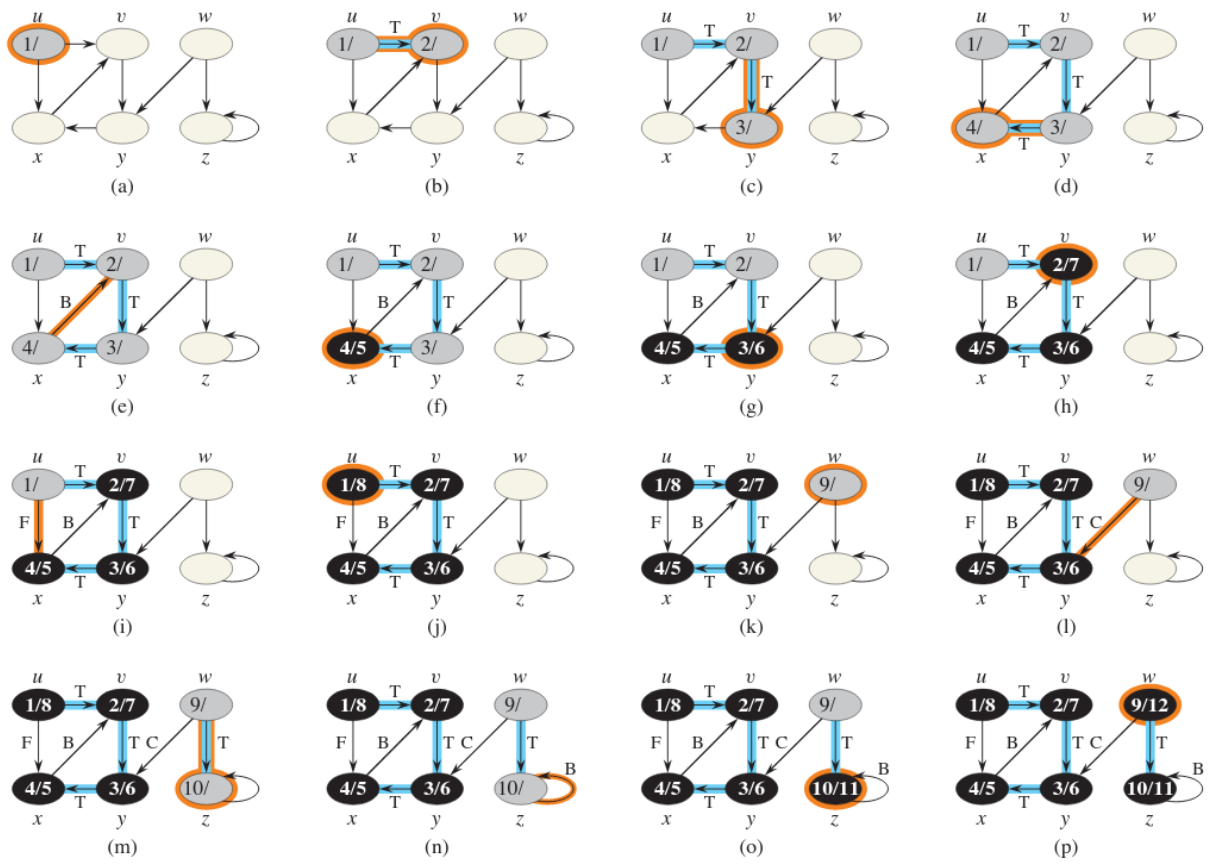
```
DFS($G$)
for each vertex $u \in G.V$
    $u.color$ = WHITE
    $u.\pi$ = NIL
$time = 0$
for each vertex $u \in G.V$
    if $u.color$ == WHITE
        DFS-VISIT($G, u$)

DFS-VISIT($G, u$)
$time = time + 1$                   // white vertex $u$ has just been discovered
$u.d = time$
$u.color$ = GRAY
for each vertex $v$ in $G.Adj[u]$   // explore each edge $(u, v)$
    if $v.color$ == WHITE
        $v.\pi = u$
        DFS-VISIT($G, v$)
$time = time + 1$
$u.f = time$
$u.color$ = BLACK                   // blacken $u$; it is finished
```

Notice here there is no queue to maintain. Why?

Here's a graphical representation of DFS.

QUESTION: Looking at the image of DFS, where does the recursion 'bottom out'? On which step?

QUESTION: Looking at the image of DFS, where does the recursion finish unfurling and move on to the next node? On which step?



Applications of DFS: Toplogical Sort

Here's an example of an algorithm that uses Depth-First Search as a subroutine for a larger problem.

Suppose we have some directed acyclic graph (dag). Topological sorting ensures that we produce a linear ordering of all the vertices in the graph such that if G contains an edge (u, v), then u appears before v in the ordering. In general, topological sorts are helpful if we have some sort of dependency structure.

The "pseudocode" for topological sort is very brief:

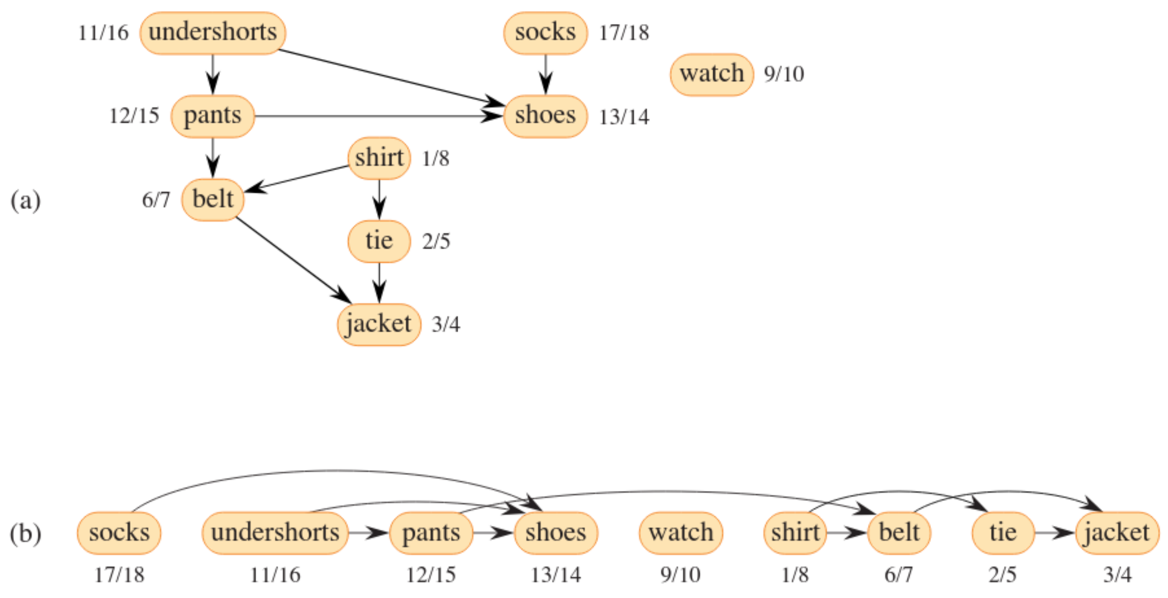
```
TOPOLOGICAL-SORT(G):
    call DFS(G) to compute finish times
    as a vertex is finished, insert it ot the front of a linked list
    return the linked list
```

This is an example of an algorithm that takes advantage of the depth property of DFS.

QUESTION: Why use DFS here and not BFS?

Since we can be sure that the first vertex finished is the deepest vertex from our start, we basicaly reverse-order the depth for topological sort.

Here is the example the book gives of topological sorting, using clothing:



Can you come up with another example for which topological sort might be the right algorithm to use?