

Abstract Data Types and Depth-First Search

Let's review the main abstract data types that we might use when implementing various algorithms.

Linked Lists

Consider a list $L = [x_1, x_2, \dots, x_n]$ where each x_i is an element in the list. In a *singly-linked list*, we keep a pointer to the first element of the list—that is, $\text{head}(L) = x_1$, and each element x_i has a pointer to the element after it, so $\text{next}(x_i) = x_{i+1}$ and $\text{next}(x_n) = \text{null}$.

There is no reason for singly-linked lists to be used in practice. You will never see them, with the exception of perhaps a coding interview question or a puzzle.

A *doubly-linked list* also has a pointer to the last element of the list ($\text{tail}(L) = x_n$) as well as pointers from each element to the previous element ($\text{prev}(x_i) = x_{i-1}$ and $\text{prev}(x_1) = \text{null}$).

It's constant to do the actual insertion or deletion of an element, and at most linear ($O(n)$) to find an element by starting at the head or tail and moving along the list until it is found.

Queues

Queues are what's known as First-In, First-Out (FIFO) linked lists. They support the following additional operations:

- $\text{enqueue}(q, x)$: insert element x to the back of the queue q . Formally, $q = q \circ x$.
- $\text{dequeue}(q)$: delete the element at the front of the queue q and return it. Formally, $q = [x_2, \dots, x_n]$, return x_1 .

Stacks

Stacks are what's known as Last-In, First-Out (LIFO) linked lists. They support the following additional operations:

- $\text{push}(s, x)$: insert element x to the top (back) of the stack s . Formally, $s = s \circ x$.
- $\text{pop}(s)$: delete the element at the top (back) of the stack s and return it. Formally, $s = [x_1, \dots, x_{n-1}]$, return x_n .

Graphs

Definition 1. A (directed) *graph* $G = (V, E)$ is defined by a set of vertices V and a set of (ordered) edges $E \subseteq V \times V$.

Definition 2. A *directed edge* is an ordered pair of vertices (u, v) and is usually indicated by drawing a line between u and v , with an arrow pointing towards v .

Definition 3. An *undirected edge* is an unordered pair of vertices $\{u, v\}$ and is usually indicated by drawing a line between u and v . It indicates the existence of ordered edges (u, v) and (v, u) .

Typically undirected edges will also be notated (u, v) out of sloppiness.

Some conventions:

- We will refer to the number of vertices (or the *size* of the vertex set $|V|$) as n .
- We will refer to the number of edges (or the *size* of the edge set $|E|$) as m .
- Often we will simply name the vertices $V = \{1, \dots, n\}$ so an edge (i, j) is an edge from the i^{th} vertex to the j^{th} vertex.
- You may also hear vertices referred to as “nodes” or edges referred to as “arcs.”

Definition 4. We call vertices i and j *adjacent* or *neighbors* if there is an edge $(i, j) \in E$. In directed graphs, we may explicitly refer to *out-neighbors* ($\{j : (i, j) \in E\}$) or *in-neighbors* ($\{j : (j, i) \in E\}$).

Definition 5. The *degree* of a vertex v is the number of neighbors it has. That is, $d_v = |\{u : (v, u) \in E\}|$. For directed graphs, we may refer to a vertex’s *in-degree* or *out-degree*, and its *degree* is the sum of these.

Definition 6. A *path* from u to w is a sequence of edges e_1, e_2, \dots, e_k such that $e_1 = (u, v_1)$, $e_i = (v_{i-1}, v_{i+1})$, and $e_k = (v_{k-1}, w)$. That is, the first edge starts at u , the last edge ends at w , and each proceeding edge ends where the previous edge starts.

Definition 7. We say that a pair of vertices are *connected* if there exists a path between them.

We see graphs all over; networks are an entire field of study! What can you represent with graphs?

- Transportation networks (roads, airlines)
- Communication networks (Bitcoin peer-to-peer network)
- Information network (internet with links)
- Social networks
- Dependency network (course prerequisites, food chain)

What graph problems do you know?

- Shortest path
- Traveling salesman
- Scheduling

Abstract Data Types for Graphs

There are two primary ways that we represent graphs in the computer.

Definition 8. An *adjacency matrix* for $G = (V, E)$ is an $n \times n$ binary matrix A where $A_{ij} = 1$ if and only if $(i, j) \in E$.

Pros of using an adjacency matrix:

- Look-up of a specific (i, j) edge is $O(1)$.

Cons of using an adjacency matrix:

- Space is $\Omega(n^2)$, independent of m . This can be very wasteful for sparse graphs where m is small.
- Listing all of i 's edges is $\Omega(n)$ time, which can again be wasteful if i has small degree.

Definition 9. An *adjacency list* for $G = (V, E)$ is an array A of length n where the i^{th} entry contains a linked list of i 's neighbors. That is, j is in the list $A[i]$ if and only if $(i, j) \in E$.

Pros of using an adjacency list:

- Listing all of i 's edges is $O(d_i)$ time, hence $O(1)$ per neighbor.
- Space is $O(n + m)$.

Cons of using an adjacency list:

- Look-up of a specific (i, j) edge is $O(d_i) = O(n)$.

Depth-First Search

If we want to explore a graph, determine whether two nodes are connected, or determine some properties regarding the ordered structure of a directed graph, we use *graph search* algorithms. The two most basic graph search algorithms are *depth-first* and *breadth-first*. Depth-First Search is the algorithm that shoots as far away from a node possible to see if it results in a successful path, and only turns around if it dead ends.

For now, we ignore the calls $\text{previsit}(v)$ and $\text{postvisit}(v)$.

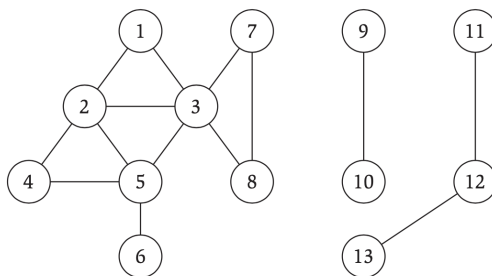


Figure 1: Example graph G . From Kleinberg Tardos.

Consider the below pseudocode when called on the above example, that is, what happens when we run $\text{DFS}(1, G)$ where G is the graph above?

Algorithm 1 $\text{search}(v, G)$

Input: Graph $G = (V, E)$ and vertex v .
mark v as explored
previsit(v)
for $(v, w) \in E$ **do**
 if w is unexplored **then**
 search(w)
 end if
end for
postvisit(v)

Algorithm 2 $\text{DFS}(s, G)$

Input: Graph $G = (V, E)$ and vertex v .
for each $v \in V$ **do**
 v is unexplored
end for
search(s, G)

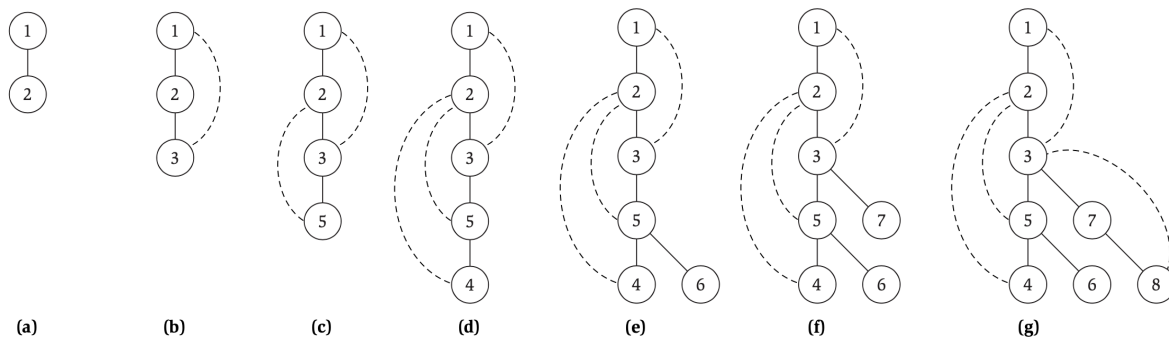


Figure 2: The DFS tree is shown in solid edges as constructed in stages by the above algorithm. The dashed edges are the edges that do not belong to the DFS tree but do belong to G . From Kleinberg Tardos.

Exercise. This DFS algorithm implicitly uses a stack. Can you draw the stages of the stack as it runs on the example graph?