Design & Analysis of Algorithms Study Notes

Covers:

Textbook pages refer to:

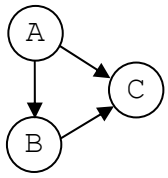
Cormen, Thomas H. (Editor), et al. *Introduction to Algorithms, Second Edition*. Cambridge, Massachusetts: The MIT Press, 2001.

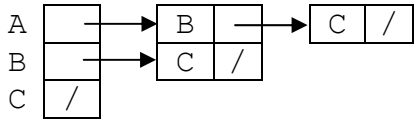
Adjacency-List and Adjacency-Matrix Representations

Textbook: pp. 528-529

The textbook only shows lists and matrices without edge costs.

Example: w/o edge costs, directed graph

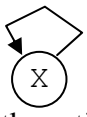




Adjacency-list

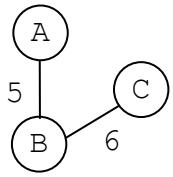
	A	B	C
A	0	1	1
B	0	0	1
C	0	0	0

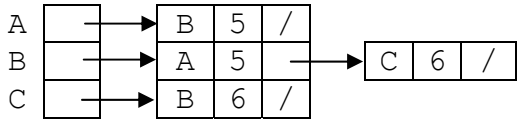
Adjacency-matrix

- There is no edge from X to itself unless shown by 

- In Adj-matrix, 0 means there is no edge linking both vertices
  1 means there is an edge linking both vertices
- / = nil

Example: w/ edge costs, undirected graph

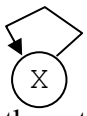




Adjacency-list

	A	B	C
A	∞	5	∞
B	5	∞	6
C	∞	6	∞

Adjacency-matrix

- There is no edge from X to itself unless shown by 

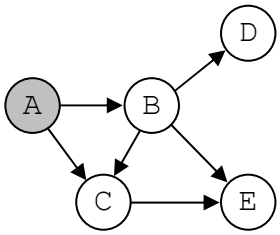
- In Adj-matrix, ∞ means there is no edge linking both vertices
- / = nil

BFS, Breadth-First Search

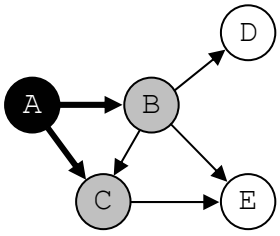
Textbook: pp. 531-539

```
BFS(G, s)
1   for each vertex u ∈ V[G] – {s}
2       do color[u] ← WHITE
3          d[u] ← ∞
4          Π[u] ← NIL
5   color[s] ← GRAY
6   d[s] ← 0
7   Π[s] ← NIL
8   Q ← Ø
9   ENQUEUE(Q, s)
10  while Q ≠ Ø
11      do u ← DEQUEUE(Q)
12         for each v ∈ Adj[u]
13             do if color[v] = WHITE
14                then color[v] ← GRAY
15                     d[v] ← d[u] + 1
16                     Π[u] ← u
17                     ENQUEUE(Q, s)
18         color[u] ← BLACK
```

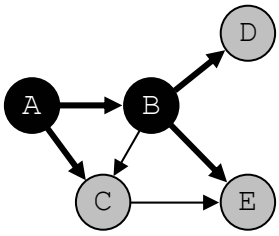
Picks a start vertex and keeps all adjacent vertices in a queue, adding new entries at the end of the queue (unless it would form a cycle).



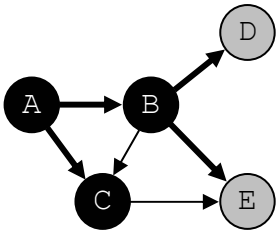
S = NIL
Q= A



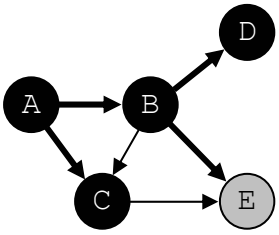
S = A
Q= B, C



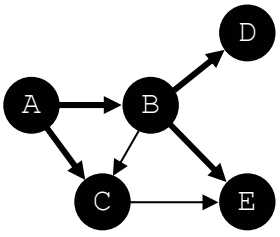
S = A, B
Q= C, D, E



S = A, B, C
Q= D, E



S = A, B, C, D
Q= E



S = A, B, C, D, E
Q= NIL

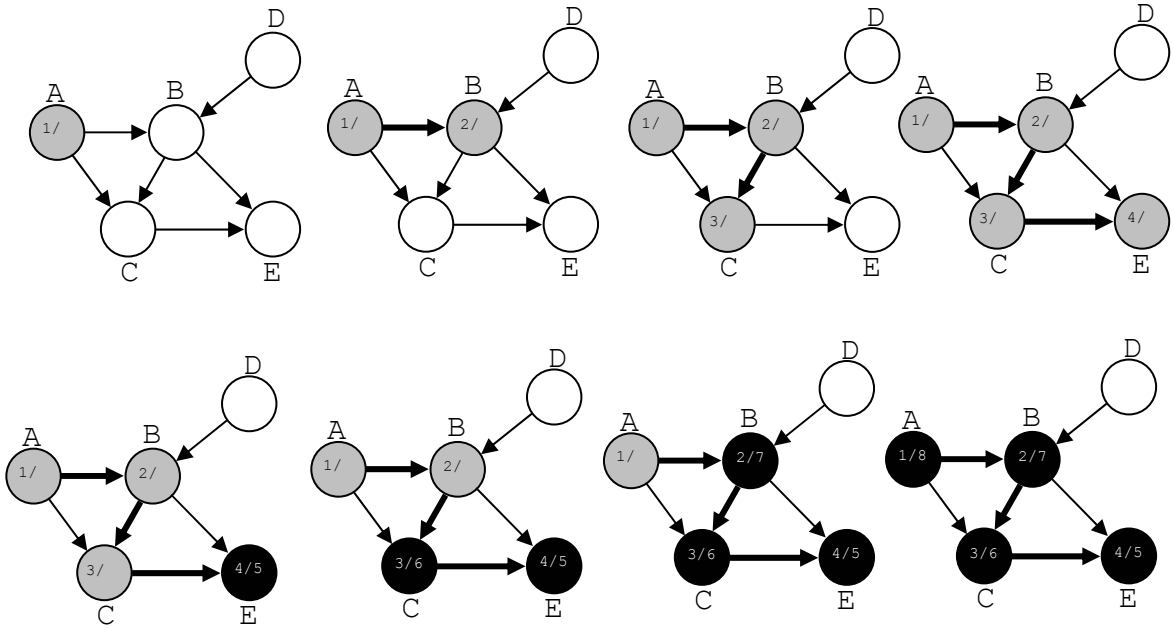
DFS, Depth-First Search

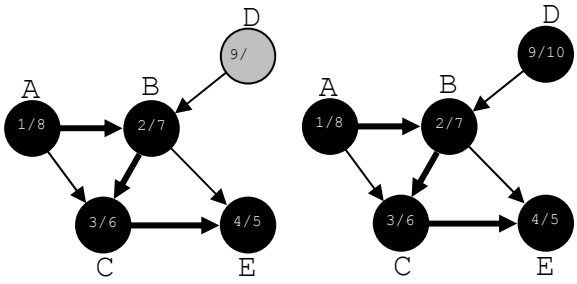
Textbook: pp. 540-545

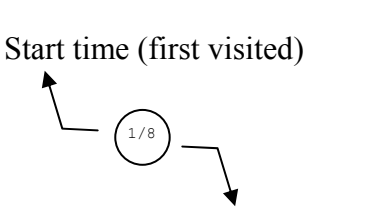
```
DFS(G)
1   for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3               Π[u] ← NIL
4   time ← 0
5   for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7               then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1   color[u] ← GRAY          ► White vertex u has just been discovered.
2   time ← time + 1
3   d[u] ← time
4   for each v ∈ Adj[u]      ► Explore edge (u, v).
5       do if color[v] = WHITE
6               then Π [v] ← u
7                       DFS-VISIT(v)
8   color[u] ← BLACK         ► Blacken u; it is finished.
9   f[u] ← time ← time + 1
```

Picks a start vertex and keeps picking the lowest cost edge from that vertex (unless it would form a cycle) until all vertices have been visited. (May be viewed as BFS but adding new vertices at the front of the queue rather than at the end.)







Complete DFS traversal:

```
 1. A – B (tree)
 2.     B – C (tree)
 3.         C – E (tree)
 4.             E – B (back)
 5.             E done
 6.         C done
 7.     B done
 8. A – C (forward)
 9. A done
10. Jump to D...
11. D – B (cross)
12. D done
```

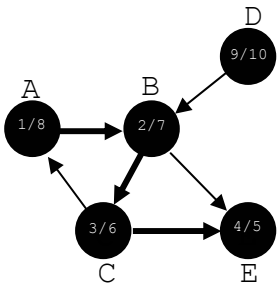
See Edge and Vertex ID for explanation of edge IDs.

Forefathers

v is a forefather of *u* (denoted by ff(*u*) = *v*) if:

a) There is a path from *u* to *v*, and

b) *f*[*v*] is maximum (the latest finishing time) among all *v* that can be reached from *u* by a path



ff(B) = A
ff(C) = A

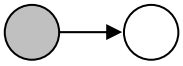
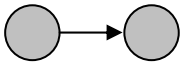
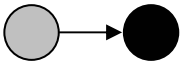
A vertex can be it's own forefather: ff(D) = D

There must be a cycle for there to be a forefather.

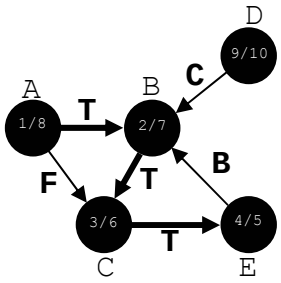
Edge and Vertex ID

Textbook: p. 546

(⟶ means "leads to", that is the traversal is at *u* and is heading towards *v*)

u	*v*	*Color*	*State*	*Edge classification*
		white	unvisited	$d[u] < d[v] < f[v] < f[u]$ if *u* is visited before *v*, but finishes after *v* > tree edge
		grey	being visited	$d[v] < d[u] < f[u] < f[v]$ if *u* is visited after *v*, but finishes before *v* > back edge
		black	visited	$d[v] < f[v] < f[u]$ if *u* finishes after *v*, and 1) *u* is visited after *v*, $d[v] < d[u]$ > cross edge 2) *u* is visited before *v*, $d[u] < d[v]$ > forward edge

Edge ID Example:



See DFS notes for Vertex ID example

Topological Sort

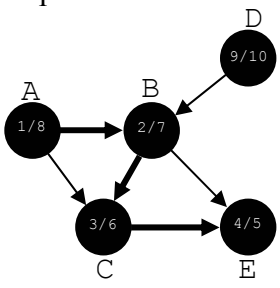
Textbook: p. 549-551

Topological sort is run on a directed acyclic graph (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

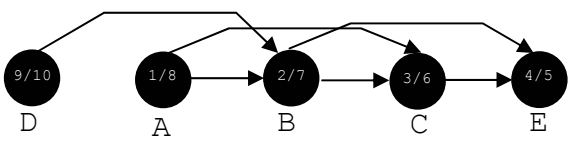
In other words, run DFS and sort the vertices in descending finishing time.

Example:



Topological sort:
{ D, A, B, C, E }

Or visually:

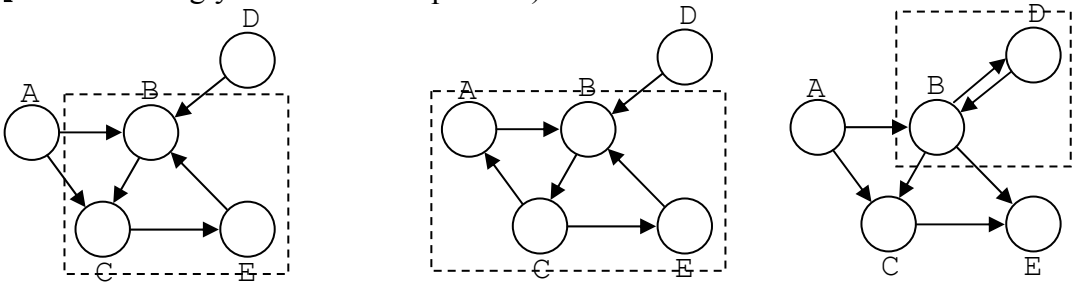


SCC, Strongly Connected Components

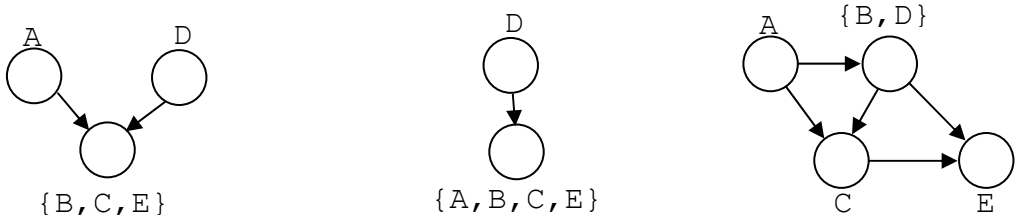
Textbook: p. 552-556

Two vertices *u* and *v* are in the same strong component of a directed graph if there is:
a path from *u* to *v* and
a path from *v* to *u*.

Example:
(denotes strongly connected components)



A graph with strong components can also be represented by a dag:



Note: A dag cannot have strongly connected components since a cycle is required to have an scc.

Also, a *strongly connected graph* is a directed graph whose vertices are all in the same strong component.

Example:

A strongly connected graph

Not a strongly connected graph

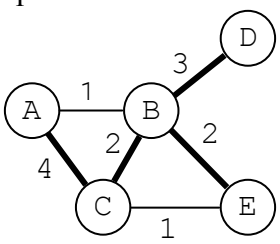
MST, Minimum Spanning Tree

Textbook: p. 561-579

For a mathematical nonsensical definition of an MST (as with everything else), refer to the textbook. For an oversimplified definition, read on:

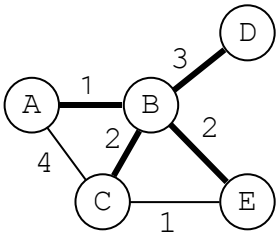
> An MST is an acyclic tree that contains every vertex in the graph by connecting them in the shortest distance possible (such that the total distance – edge weights – connecting all the vertices is at a minimum).

Example:



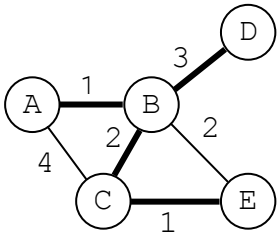
Not an MST
Total weight: 11

Must replace A-C with A-B and B-E with C-E



Not an MST
Total weight: 8

Must replace B-E with C-E



An MST
Total weight: 7

Note: There cannot be a gap in the MST, otherwise all the vertices would no longer be connected together; you would have two or more disjoint sets, which cannot have a tree.

If you add a new vertex *v* to *G* and it has more than one outgoing edge, connect *v* to the MST through the edge with the smallest weight.

Given a graph *G*, you can make an MST with algorithms such as Kurskal's and Prim's.