Representing Graphs

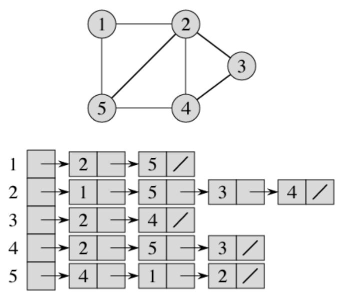
Adjacency Lists And Adjacency Matrices For Different Efficiencies

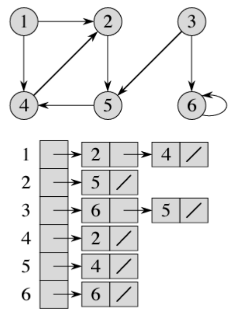
Review of Graph Theory Terminology

- Covered in discrete math course
- Reviewed extensively in CLRS Appendix B.4
  - Make sure to go over Appendix B.4
- Graph $G = (V, E)$
  - V: The vertex set of G, E: The edge set of G.
  - $E = \{(u, v) | u, v \in V\}$ in a digraph (directed graph): Self-loops are possible.
  - $E = \{\{u, v\} | u, v \in V, u \neq v\}$ in an undirected graph (edges are unordered pairs).
- Various terms and definitions to go over in Appendix B.4:
  - Degree, path, path length, reachability, simple path, cycle, simple cycle, connected graph, connected components, strongly connected components, …

Representing Graph: Adjacency List

- For $V = \{1, 2, \dots, n\}$, maintain an array $Adj[1..n]$ with each $Adj[i]$ pointing to a linked list of all vertices adjacent to vertex i.

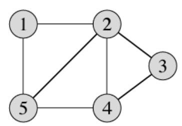




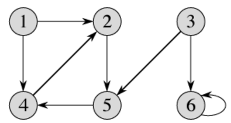
CLRS Fig. 22.1, 22.2

Representing Graph: Adjacency Matrix

- For $V = \{1, 2, \dots, n\}$, maintain an adjacency matrix (2D array) $A[1..n, 1..n]$ where $A[i, j] = 1$ if $(i, j) \in E$, and $A[i, j] = 0$ otherwise.



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



	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

CLRS Fig. 22.1, 22.2

Pros & Cons of Each Representation

- Adjacency list
  - $\Theta(|V| + |E|)$ space
    - Compact (in terms of space) representation of sparse ($|E| \ll |V|^2$) graphs
  - $O(|V|)$ time to check if vertex j is adjacent to vertex i: Need to traverse the list $Adj[i]$, which may contain as many vertices as $|V|$.
- Adjacency matrix
  - $O(1)$ time (always) to check if vertex j is adjacent to vertex i (if there's an edge from vertex i to vertex j)
  - $\Theta(|V|^2)$ space (always, even for very sparse graphs)

Summary

- An adjacency list is usually the method of choice, as most graphs we deal with are sparse.
- Adjacency matrix is preferred when the graph is dense or we need to tell quickly if there's an edge connecting two given vertices.
- Weighted graphs can also be easily represented with either choice:
  - Adjacency matrix entry $A[i,j]$ will be the weight $w(i,j)$ if $(i,j) \in E$.
  - Adjacency list $Adj[i]$ will point to the head of a linked list of nodes each of which has two properties: vertex j and weight $w(i,j)$ if $(i,j) \in E$.

Breadth-First Search

Know All Your Immediate Neighbors First Before Knowing Any Other Neighbors

Know Closer Neighbors First Before Knowing Farther Neighbors

Breadth-First Search (BFS) Overview

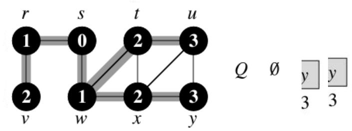
- One of the two widely used graph traversal methods
  - Also called breadth-first traversal
- Each vertex is (i) ***discovered***, and then later (ii) ***explored*** in a graph traversal.
  - Exploring a vertex is basically discovering its adjacent vertices!
    - "Exploring incident edges," as phrased in textbook.
  - There's a source vertex that's initially discovered at the very beginning.
  - You never discover/explore an already discovered/explored vertex!
    - Each vertex is discovered exactly once and explored exactly once.
- In BFS, ***First-Discovered, First-Explored***.
  - First-In, First-Out (FIFO): Queue data structure suits this well!

BFS Pseudocode Details

- Needs to distinguish each vertex's status:
  - Not discovered yet (white), discovered only (gray), explored (black)
  - At the beginning, the source vertex is marked gray (discovered) and all other vertices are marked white (not discovered yet).
- Breadth-first tree (predecessor graph) with root s (the source vertex) will be found after BFS, containing all reachable vertices from s.
- The simple path in the breadth-first tree from s to any vertex v (reachable from s) corresponds to a "shortest path" from s to v.
- "Breadth-first" because the frontier line of the search expands across the breadth of the current frontier.

BFS Illustration

- Using a queue to implement "First-Discovered, First-Explored" nature.
  - The queue holds only the vertices that are discovered only (gray), in the order of discovery.
  - The front of the queue (earliest discovered-only vertex) is dequeued and explored.
    - Which is to discover its adjacent non-discovered/explored vertices
  - The newly discovered vertices should be enqueued to the queue to be explored later in the order of discovery.
- CLRS Fig. 22.3 (pp.596)
  - See how shortest distance from s to each vertex is updated.



Actual Code

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

Lines 1–4: Marking all non-source vertices as not-yet-discovered, with unknown distance, and no parent (predecessor) yet in the breadth-first tree.

Lines 5–9: Marking source vertex as discovered-only, setting its distance from itself (0), and setting its predecessor (none). Also enqueue it so that it will be explored in the loop.

Line 10: Repeat as long as there's some vertex to be explored

Lines 11–12: Take the vertex to be explored (discovered earliest) and inspect all its adjacent vertices

Lines 13–17: Only for each not-yet-discovered vertex, mark it as discovered-only (gray), update its distance (+1), parent (predecessor), and enqueue it so that it can be explored later in the correct order.

Line 18: Done exploring all adjacent vertices. Mark as such (black). Not enqueued back!

Analysis

- Each vertex goes through white→gray transition exactly once.
- Only gray vertices are in the queue.
- Once dequeued, it'll never be enqueued again.
- Therefore, each vertex gets enqueued and dequeued exactly once in line 9-18.
- Therefore, the number of iterations of line 10 while loop is $O(V)$.
- Total number of iterations of line 12 for loop is $\Theta(E)$, because each vertex (u)'s adjacency list will be scanned exactly once (Sum of lengths of all adjacency lists is $\Theta(E)$).
- Therefore, it's $O(V+E)$.


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

Correctness Proofs

- Lemmas/Corollaries/Theorems 22.1-22.6 in pp.598-601 of CLRS
  - Study individually with the core properties below, and ask questions if any.
  - There may be exam problems related to these properties and insights.
- $v.d$ is the shortest distance from s to v in G.
- Predecessor subgraph G_π of G produced by the BFS procedure is a breadth-first tree of G.
  - $V_\pi = \{v \in V : v.\pi \neq NIL\} \cup \{s\}$
  - $E_\pi = \{(v.\pi, v) : v \in V - \{s\}\}$
- The G_π may vary depending on the order of vertices in the adjacency lists, but $v.d$ (distance) is unique.

Depth-First Search

Dig Deeper On One Neighbor Before Knowing Any Other Neighbors

Depth-First Search (DFS) Overview

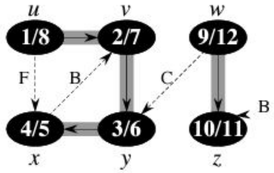
- The other widely used graph traversal method
  - Also called depth-first traversal
- Same process as in BFS: Discover and explore
- However, in DFS, ***Last-Discovered, First-Explored***!
  - Last-In, First-Out (LIFO): Stack data structure fits this.
  - In fact, we can perform DFS just by swapping the queue in BFS with a stack.
  - But remember “stack” of recursive calls can be used as well
    - So recursively implemented in the textbook.
    - Still iterative code with explicit stack data structure is preferred.

Details of Textbook’s DFS Implementation

- Each vertex’s predecessor is still maintained.
  - Gives depth-first forest (trees)
- Each vertex’s distance from source is no longer maintained (not meaningful).
- Two timestamps are maintained for each vertex:
  - Discovered time & finished time (done exploring incident edges)
- Try all remaining white vertices as sources
  - That’s why we get depth-first forest, not just a tree, if the given graph is not connected.
  - This is arbitrary (BFS could have been this way, or DFS could have been like BFS).
  - But it’s done this way to reflect how the results of these searches are typically used.

Actual Code and Example (CLRS pp.604-605)

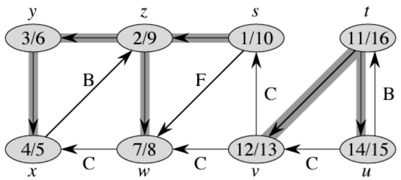
```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)

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

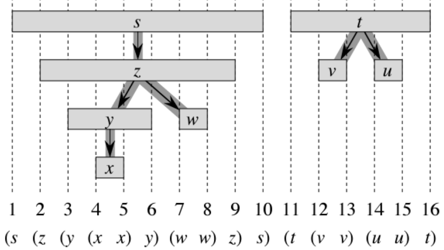


Time Complexity Analysis of DFS Code

- DFS-VISIT() is called exactly once for each vertex $v \in V$: $\Theta(V)$ calls
  - Same coloring argument as in BFS
- The for loop in DFS-VISIT(G,v) iterates $|Adj[v]|$ times for each $v \in V$
  - Across all vertices (all recursive calls), the total will be $\sum_{v \in V} |Adj[v]| = \Theta(E)$
    - $|E|$ for a digraph, $2|E|$ for an undirected graph
- Loops in DFS() will just iterate up to $\Theta(V)$ times
- Therefore, it's $\Theta(V + E)$, the same as in BFS.

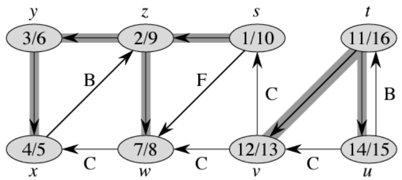
DFS Properties:
Parenthesis Structure (CLRS Fig. 22.5)

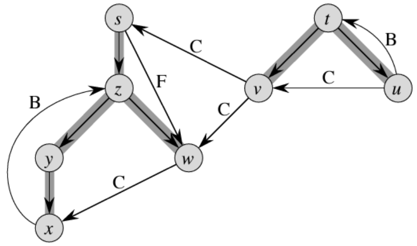




Theorem 22.7: Parenthesis theorem, about intervals, their inclusions, and their relationships.

Edge Classification from DFS





- Tree edge: When a white vertex is discovered by exploring that edge.
- Back edge: When a gray vertex is seen by exploring that edge.
- Forward edge: When a black vertex is seen by exploring that edge,
  but start time of the from-vertex of that edge is earlier than the finish time of the to-vertex of that edge.
- Cross edge: When a black vertex is seen by exploring that edge,
  but start time of the from-vertex of that edge is later than the finish time of the to-vertex of that edge.

Note about Formal Proofs

- Individually study the proofs of lemmas, corollaries, and theorems in the textbook, and ask questions if any.
- There might be exam problems related to those proofs.

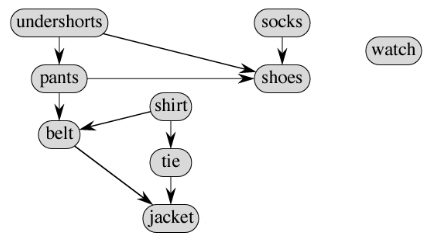
Applications of Graph Traversals

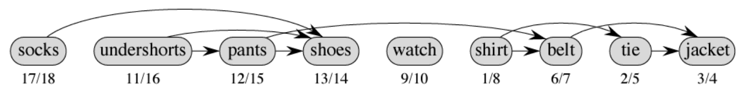
Topological Sort And Strongly Connected Components, Both With DFS

Topological Sort

- Directed-acyclic graph (DAG): A directed graph with no cycles
- Topological sort of a DAG $G = (V, E)$:
  - A linear ordering of all its vertices s.t. if G contains an edge (u, v), then u appears before v in the ordering.
  - If the graph contains a cycle, this is impossible.
- Many examples with precedence/prerequisite requirements on events can be represented using DAGs.
- Scheduling the events with precedence/prerequisite requirements satisfied can be done by topological sort.
- With our DFS() and DFS-VISIT(), it's simply ***a call to DFS() and listing vertices in descending order of their finish times!***

Dressing Sequence Example (CLRS Fig. 22.7)





Curriculum Prerequisite Structure Example

CSE101: INTR100
CSE111: INTR100
CSE221: CSE254 INTR100
CSE243: CSE254
CSE254: CSE111 MATH210 INTR100
CSE258: CSE254 CSE243 INTR100
ECE111: INTR100
ECE201: CSE111 INTR100
INTR100:
MATH210: INTR100

Implementation and Proofs

- Straightforward application of DFS: TOPOLOGICAL-SORT(G)
  - Call DFS(G) to compute finishing time $v.f$ for each vertex v
  - As each vertex is finished, insert it onto the front of a linked list
  - Return the linked list of vertices.
- Straightforward time complexity: $\Theta(V+E)$. Same as DFS.
  - O(1) to insert each of the $|V|$ vertices, so no effect to asymptotic complexity.
- Straightforward correctness proofs: Lemma 22.11 and Theorem 22.12
  - Show for any edge (u,v) in the dag, we have $v.f < u.f$.

Strongly Connected Components

- Definition (Recall from Appendix B)
  - A strongly connected component (SCC) of a *digraph* $G=(V,E)$ is a maximal set of vertices $C \subseteq V$ s.t. for every pair of vertices $u, v \in C$, they are reachable from each other.

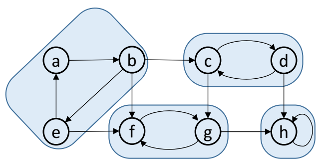


Fig. 22.9 (a) An example digraph G with its SCCs shaded

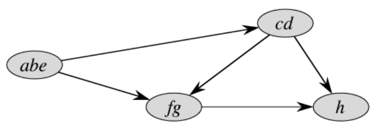
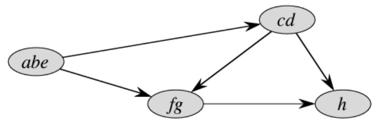


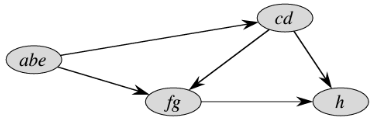
Fig. 22.9 (c) Acyclic component graph G^{SCC} obtained by contracting all edges within each SCC.

How To Find SCCs



- Note that the contracted acyclic component graph G^{SCC} is a dag!
- If we do a DFS on SCC w/ h (the last in the topological sort), we'll get a DF tree only for that SCC w/ h.
- Then if we do a DFS on SCC w/ fg (the second last in the topological sort), we'll get a DF tree only for that SCC w/ fg (h is already non-white, so won't be visited again).
- …
- That is, if we do DFS on this reverse topological sort order, we are guaranteed to get a depth-first forest whose trees correspond to SCCs!

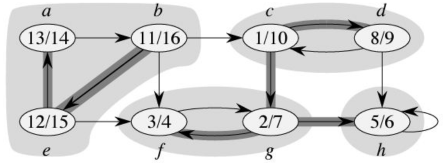
Implementation Details



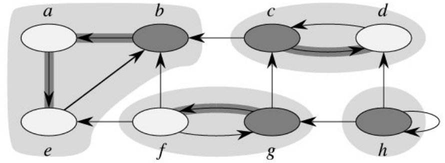
- It’s not easy to find the minimum of the finish times of SCCs (the maximum finish time of all vertices in SCC).
- It’s easy, though, to find the maximum of the finish times of SCCs.
  - It’s just the maximum finish time of the original graph’s DFS.
- So the question is how to use the original topological sort order
  - That is, start finding DF trees from abe and move forward, not start from h and move backward.
  - But if we start finding DF trees from abe, then cd, fg, and h will be all reachable (because forward directions)!
  - How can we avoid this? Simple. Reverse edges in original G! (Transpose G^T)

Double DFSs (CLRS Fig. 22.9)

- First DFS to sort vertices in descending order of finish times



- Get G^T and perform second DFS to find DF trees corresponding to connected components



Algorithm and Analysis

STRONGLY-CONNECTED-COMPONENTS(G)

1 call DFS(G) to compute finishing times $u.f$ for each vertex u
2 compute G^{T}
3 call DFS(G^{T}), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

- 2 DFSs, still $\Theta(V+E)$.
- Time to create G^T is still $O(V+E)$.
- Thus, this algorithm is still $\Theta(V+E)$.

Formal Correctness Proofs

- Lemmas/Corollaries/Theorems in CLRS Section 22.5
- Formalization of the intuitions presented in earlier slides
- Study the proof individually. Everyone is expected to understand the proofs.