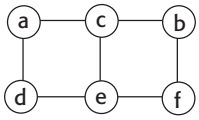
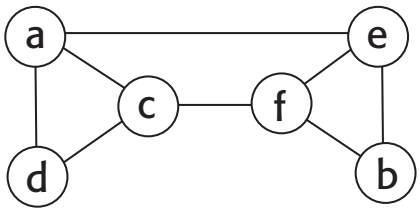
Depth First Search
Traversal Control

- DFS stack
- Push/pop order
- Spanning tree

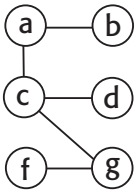


(a)

Eventually, we know that all verts were discovered when we backtrack to the starting point where we no longer see new ones.



(b)



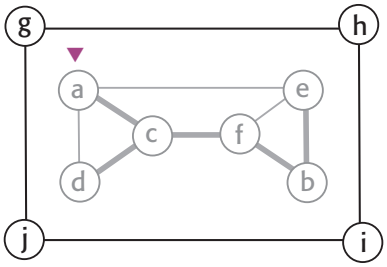
(c)

Exercise
Apply the pen-paper procedure using a stack to control steps in this acyclic graph, compare the resulting tree to the original graph. What can be observed?

Answer - tree (c)
Traversal leads to a spanning tree (span/include all verts). All edges in the acyclic graph appear in the DFS tree (are used), i.e., its own spanning tree. An acyclic graph is just a rootless tree. A DFS converts it to an explicit tree rooted at the start vertex (node).

Depth First Search
Traversal Restart

➪ Traversal forest



A natural depth-first traversal is only able to discover reachable (connected) verts.

Algorithm *DFS*
Input $G = \langle V, E \rangle$
Output A depth-first listing of vertices of G

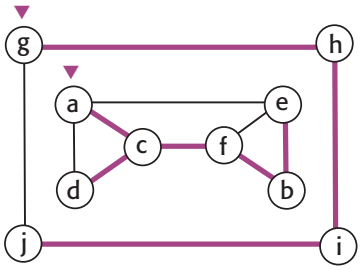
1: $\forall v \in V$, mark unvisited ▹ Loop implied
2: **for** $v \in V$ **do**
3: **if** v is unvisited **then** $dfs(v)$

Depth First Search Algorithm

Quiz
How can you use DFS to determine if graph is acyclic?

Algorithm *DFS*
Input $G = \langle V, E \rangle$
Output A depth-first listing of vertices of G

1: $\forall v \in V$, mark unvisited ▷ Loop implied
2: **for** $v \in V$ **do**
3: **if** v is unvisited **then** $dfs(v)$



A recursive statement seems natural (why?), is it necessary?

Algorithm *dfs*
Input $v \in V$
Output Recursively list unvisited vertices connected to v in DFS order

1: mark v visited, output v ▷ Any processing of v
2: **for** $w \in V \mid (v,w) \in E$ **do** ▷ w adjacent to v
3: **if** w unvisited **then** $dfs(w)$

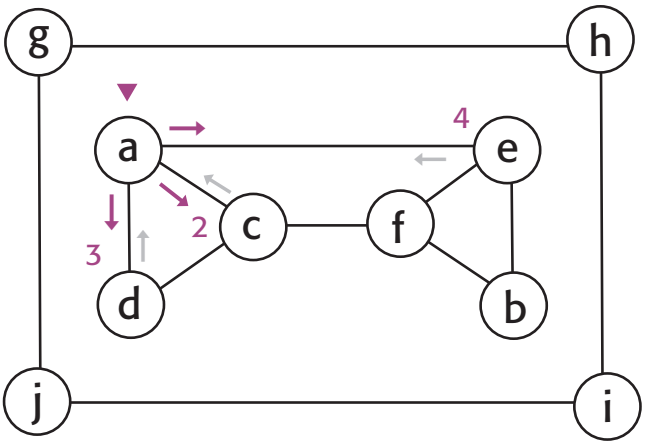
Quiz
What? Where is the traversal control structure (the stack)?

Some Answers - At least
for undirected, acyclic
if no back edges. An
iterative design with an
explicit stack maybe
less clear but equally
correct, perhaps more
efficient.

Graph Traversals
Breadth First Search

➪ Cross edge

Another natural approach to visiting all verts once is to first explore <u>all</u> just discovered paths leading to unseen verts.



Breadth First Search Algorithm

Compare to DFS (G) earlier.

Algorithm *BFS*
Input $G = \langle V, E \rangle$
Output A breadth-first listing of vertices of G

```
1: ∀v ∈ V, mark unvisited      ▷ iteration implied
2: for v ∈ V do
3:     if v is unvisited then bfs(v)
```

Quiz
How can you use BFS to determine if graph is connected?

Algorithm *bfs*
Input $v \in V$
Output List unvisited vertices connected to v in BFS order

```
1: mark v visited, intialize queue with v
2: while queue not empty do
3:     dequeue a vertex, u, output u
4:     foreach w adjacent to u do
5:         if w unvisited then mark w visited, queue w
```

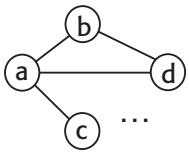
Graph Traversals
Conclusions

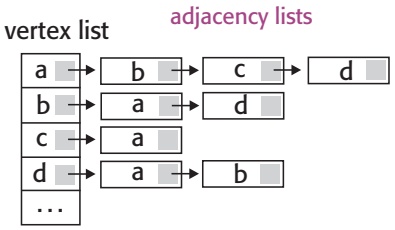
1.4

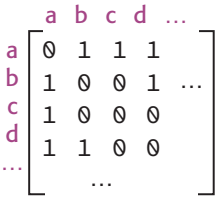
- Adjacency lists
- Adjacency matrix

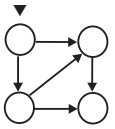
Quiz
Why is DFS/BFS also in Θ class of |V|+|E|, not just *O*?

➪ Performance









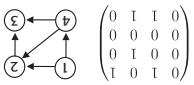
Exercise
Write the adjacency matrix. (Label clock-wise, answer below.)

Quiz
Why did your textbook consider DFS/BFS brute-force in its 3rd edition?

➪ Design pattern?

Quiz
List at least 3 common and 2 unique for DFS and BFS (google if needed).

➪ Applications



Brute Force Summary

Perform all operations a solution definition or specification calls for or clearly needs to get a result, or try all possible solution alternatives to identify a desired one (an optimal subset of items, or a graph condition).

Small optimizations (like skipping repeated comps or potential solutions) improve runtime but not efficiency.

- **Does the job (solves problem)**
- **Fairly simple, general**
- **Inefficient**

  In terms of arbitrarily large inputs, often worst in class

- **Solution approach, not design**