Graph Traversal

We wish to visit and list every node in a connected graph by following some path. We don't mind if we reuse arcs. If we revisit nodes, this is fine, but we don’t want to write them down again.

This process is called graph traversal.

Depth-first Search

In depth-first search (DFS):

- Start from a particular node *start*
- Continue outwards along a path from start until we reach a node which has no adjacent unvisited nodes.
- Backtrack to the previous node and try a different path
- The procedure continues until we have backtracked to *start* and all nodes adjacent to *start* are visited
- At this point every node will have been visited

An Example of DFS

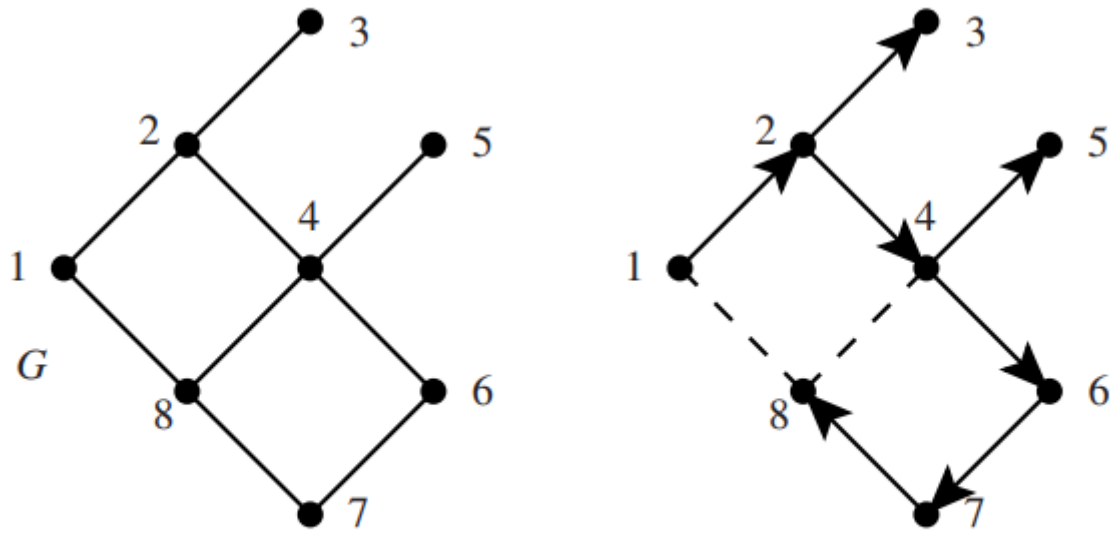


Figure 1.16: Depth-first Search

- *start* = 1
- From 1 we go to 2, next we go to 3
- The path can go no further, and so we backtrack to 2
- From here we go to 4 and 5
- We backtrack to 4 and then go to 6, 7 and 8
- Here, we see that all adjacent nodes are visited
- We backtrack to 7, 6, 4, 2 and 1
- The DFS is complete.

The right hand diagram in Figure 1.16 shows the forward direction of the search. It forms a spanning tree. The root of the tree is the start node. Notice that node 8 is at depth 5 in the search tree, while its shortest distance from the start node is only 1.

Breadth-First Search

- Start from a node start
- Fan out from there to all adjacent nodes
- From which we then run breadth-first searches

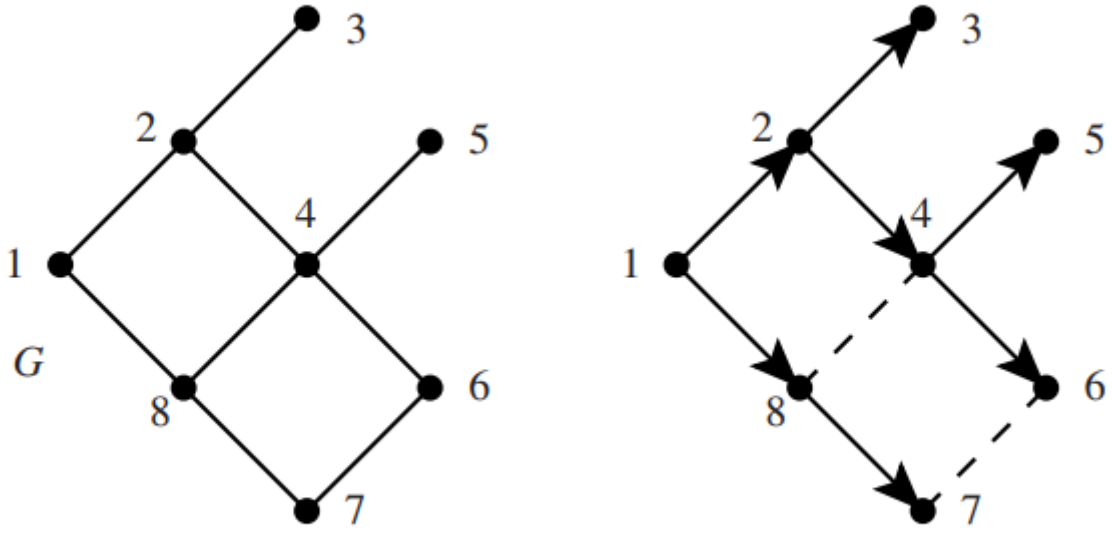


Figure 1.17: Breadth-first Search

An Example of BFS

- *start* = node 1
- Go to 2 and 8 (in either order)
- Fan out from 2 to visit 3 and 4 from 2, and 7 from 8
- We don't visit 4 from 8, since we already visited 4 from 2
- Next we visit 5 and 6 from 4
- Summing up, the order of visiting the nodes is 1,2,8,3,4,7,5,6

We get a different spanning tree from BFS than from DFS.

In the BFS tree, we visit all the nodes at depth k before visiting any node at depth $k + 1$. The depth of a node in the BFS tree is its shortest distance from the start node.

Both search procedures traverse all nodes in a connected graph, but the order of visiting is different.

Algorithms for DFS and BFS

Assume that the graph to be traversed is given as an adjacency list.

Create a Boolean array to keep track of visited nodes, and a parent function to keep track of the search tree. We shall output nodes in the order visited. Initially no nodes are visited.

The DFS algorithm is naturally expressed using recursion:

```
procedure dfs(x):
  visited[x] = true
  print x
  for y in adj[x]:
    if not visited[y]:
      parent[y] = x
      dfs(y)
      # backtrack to x
```

Once DFS is completed, every node will have a parent except for the start node. For instance, in our example the parent of 8 is 7.

If we don't need the parent information we can omit the line `parent[y] = x` from the code.

$\mathrm{dfs}(x)$ is applied to each node at most once (it will be exactly once if the graph is connected). Each application of $\mathrm{dfs}(x)$ runs through the arcs incident on x (i.e. $\mathrm{adj}[x]$) exactly once.

Therefore the running time of DFS is $O(n + m)$, where n is the number of nodes and m is the number of arcs.

BFS is naturally expressed using a FIFO queue of nodes. The queue is initialised to contain just the start node x. We then process nodes from the front of the queue. For each node we visit its immediate neighbours, adding them to the back of the queue.

```
visited[x] = true
print x
enqueue(x,Q)
while not isempty(Q):
  y = front(Q)
  for z in adj[y]:
    if not visited[z]:
      visited[z] = true
      print z
      parent[z] = y
      enqueue(z,Q)
  dequeue(Q)
```

The nodes are added to the queue in the order 1,2,8,3,4,7,5,6 and removed in the same order.

The queue grows and shrinks during the computation. The size of the queue represents the “breadth” of the front on which the traversal is being carried out.

Each node is processed once and each adjacency list is processed once. Therefore the running time of BFS is again $O(n + m)$.

Applications

Determining Whether a Graph Is Connected

So far we have assumed that the graph to be traversed is connected. We can traverse non-connected graphs as well, but we will only visit nodes which are in

the same connected component as the start node.

We can adapt either DFS or BFS to return a list of visited nodes.

The graph is connected if and only if this list is the same (perhaps in a different order) as the complete list of nodes.

This gives an $O(n + m)$ algorithm to determine whether a graph is connected.

Determining Whether a Graph Has a Cycle

Suppose a connected graph has n nodes. If it has $\geq n$ arcs then it contains a cycle (this holds even if the graph is not connected). Use this to check whether a graph has a cycle.

To find the cycle itself, we can use DFS.

Let G be a connected graph. Let T be a spanning tree of G obtained by DFS starting at node *start*.

If a is any arc of G, with endpoints x and y, then either x is an ancestor of y in T or y is an ancestor of x in T.

'*x is an ancestor of y in T*' means that x lies on the (unique) path from *start* to y in T.

- Suppose that we visit x before visiting y
- Then we must visit y from x directly (x is the parent of y) unless we have already visited y via calling DFS on some other node z adjacent to x
- In either case x is an ancestor of y
- Similarly, if we visit y before x then y is an ancestor of x
- Suppose that we are using DFS to traverse a connected graph
- If when at node x we encounter a node y which we have already visited (except by backtracking), this tells us that the node can be approached by two different routes from the start node
- Hence there is a cycle in the graph
- Conversely, if we never encounter an already visited node, then the graph is a tree, with no cycles

Therefore, we can use DFS to test whether a graph has a cycle.

```
procedure cycleDfs(x):
  visited[x] = true
  # print x
  for y in adj[x]:
    if visited[y] and y != parent[x]:
      # cycle found involving x and y
      return (x,y)
    if not visited[y]:
      parent[y] = x
      pair = cycleDfs(y)
      # backtrack to x
      if pair:
        return pair
```

The algorithm can be used to find a cycle if one exists.

Suppose that nodes x and y are returned. Then y must be an ancestor of x, as in Figure 1.18.

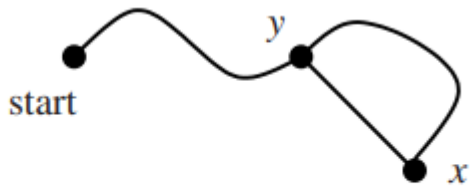


Figure 1.18: Using DFS to test for cycles

The cycle will be $x, parent[x], \ldots, y, x$.

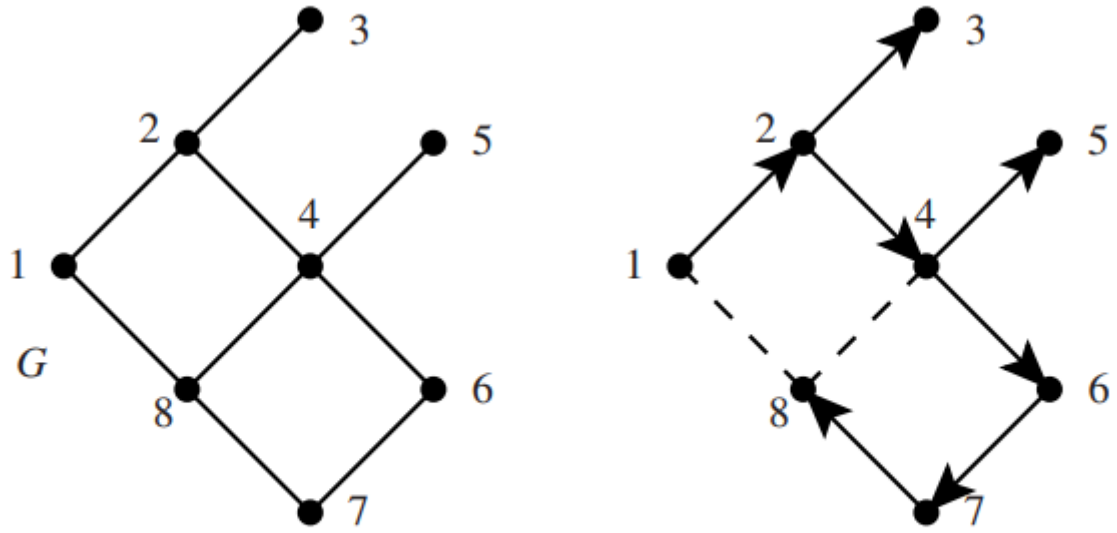


Figure 1.16: Depth-first Search

Here, the first cycle we find is when processing node 8. We find that node 1 is already visited, and return (8,1). This corresponds to the cycle $8, 7, 6, 4, 2, 1, 8$.

We could have used BFS instead of DFS.

Calculating Distances From the Start Node

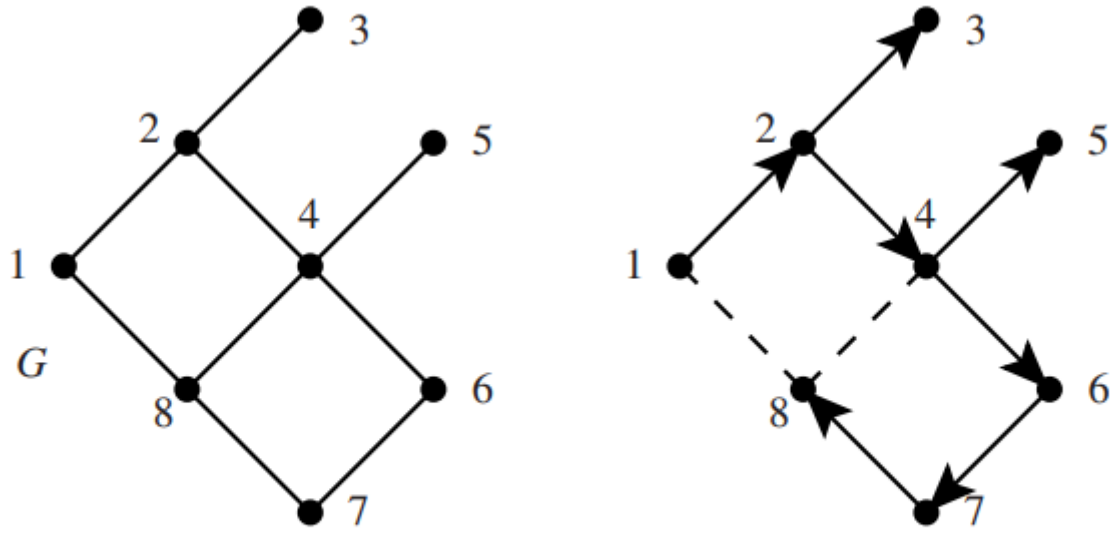


Figure 1.16: Depth-first Search

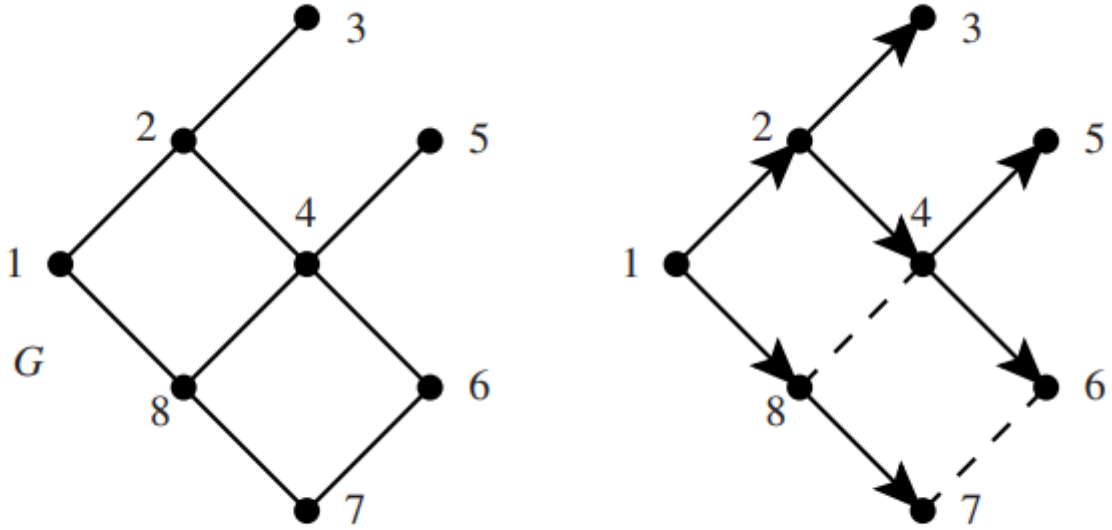


Figure 1.17: Breadth-first Search

Here, we see that BFS finds the shortest path from the start node to any reachable node, while DFS may give a longer distance than necessary.

We can adapt BFS to calculate the distance from the start node.

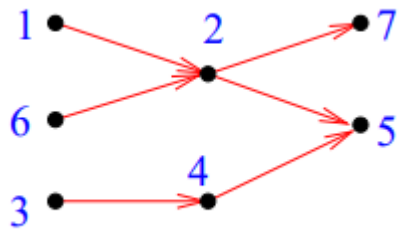
We add a new function `distance` which records the depth of each visited node in the BFS tree.

```
visited[x] = true
distance[x] = 0
enqueue(x,Q)
while not isempty(Q):
  y = front(Q)
  for z in adj[y]:
    if not visited[z]:
      visited[z] = true
      parent[z] = y
      distance[z] = distance[y] + 1
      enqueue(z,Q)
  dequeue(Q)
```

The shortest path from a node y to the start node can be read off from the parent array as $y, parent[y], parent[parent[y]], \ldots, start$

Topological Sorting

Suppose that we have a list of tasks to be completed and some tasks have to be completed before others. We can view them as a directed graph, with arcs representing the dependencies, so that an arc (x, y) denotes that task x must be completed before task y.

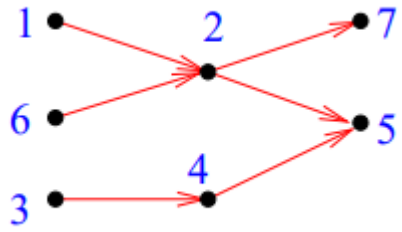


This graph must be acyclic, otherwise we cannot find an order in which to complete the tasks.

The *topological sorting problem* is:

- Given a directed acyclic graph (DAG) G with n nodes, find a total ordering of the nodes $x_1, \ldots, x_n$ such that for any $i, j \leq n$:
  - if $j > i$ then there is no path from x_j to x_i in G

Such a total ordering is called a topological sort of G. It could be presented as a list or array of nodes.



Possible topological sorts include:

- $1, 6, 3, 2, 4, 7, 5$
- $6, 1, 2, 7, 3, 4, 5$
- etc

DAGs are closely related to partial orderings.

A weak partial ordering on a set X is a binary relation $\leq$ satisfying:

- Reflexive: $\forall x \in X(x \leq x)$
- Transitive: $\forall x, y, z \in X(x \leq y \cap y \leq z \implies x \leq z)$
- Antisymmetric: $\forall x, y \in X(x \leq y \cap y \leq x \implies x = y)$

Given a DAG G, let $x \leq y$ if and only if there is a path from x to y. Then $\leq$ is a (weak) partial ordering on $\mathrm{nodes}(G)$.

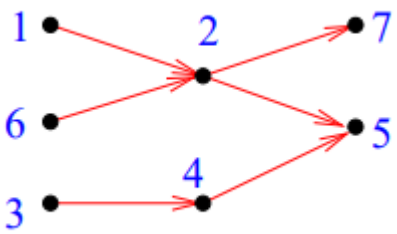
Conversely, if $(X, \leq)$ is a partial ordering, let G be the directed graph with nodes X and arcs $\{ (x, y) : x \leq y \}$. Then G is acyclic.

So a topological sorting of a DAG amounts to a linearisation of a partial ordering, i.e. a linear order which extends the partial ordering.

We can perform topological sorting using DFS.

The idea:

- When we have finished processing a node x we must have finished with all nodes which are reachable from x (the nodes which must come after x in the sorting)
- Therefore, we can add a node to the sorted list (starting from the top end) once we have finished processing it (once we exit / backtrack out of the node).



If we apply DFS we visit the nodes in the order $1, 2, 5, 7, 3, 4, 6$ assuming that the adjacency lists are given in numerical order.

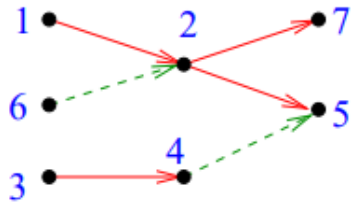


Figure 1.20: The forest got from applying DFS to the graph in Figure 1.19

Here is the resulting DFS forest. We exit the nodes in the order $5, 7, 2, 1, 4, 3, 6$.

By reversing the order in which the nodes are exited we get the sort $6, 3, 4, 1, 2, 7, 5$.

The pseudocode for DFS adapted to perform topological sorting:

```
# Given: a directed graph G with n nodes
# Return: topological sort of G as array ts of nodes if G acyclic
(else abort)
procedure dfsts(x)
  entered[x] = true
  for y in adj[x]:
    if entered[y]:
      if not exited[y]: abort # cycle
    else:
      parent[y] = x
      dfsts(y)
  exited[x] = true
  ts[index] = x
  index = index - 1

index = n - 1
for x in nodes(G):
  if not entered[x]:
    dfsts(x)
```

Note that we add nodes to the array working downwards as they are exited.
We now show:

- if the algorithm terminates then it produces a topological sort
- if the algorithm aborts then there is a cycle

When performing this algorithm on a directed graph, when we exit a node x we have already exited all nodes reachable from x.

Proof. Suppose that this is not the case. Let x be the first node that we exit but where there is some node z reachable from x that has not been exited. Since z 6= x, clearly z must be reachable from some y in adj[x]. But by the code we have already exited from y. By assumption this means that we have exited from all nodes reachable from y, including z. Contradiction.

If there is a cycle then the computation can never exit from any node of the cycle, and so must abort.

From the code we can see that if the computation does not abort:

- Each node is entered exactly once
- Each node is exited after entry
- Upon exit each node is added to the array `ts` in descending order.

Suppose for a contradiction that x is added to the array after y but that there is a path from y to x. Then we exited x after we exited y. But by Proposition 1.6.3 we can only exit y after exiting x since x is reachable from y. Contradiction. We conclude that if the computation does not abort then it produces a topological sort.

When performing DFS on a directed graph, if we enter a node x after entering a node y but before exiting y then x is reachable from y.

Proof. Suppose that this is not the case. Let x be the first node that we enter after entering some node y but before exiting y where x is not reachable from y. Let z be the last node which is entered but not exited before entering x (so that z may or may not be y). Since y has not been exited when entering x, the invocation of DFS on x is not at the top (non-recursive) level and so we must have entered x as part of processing z, and so x is in adj[z]. But by our assumption z is reachable from y, and so x is reachable from y. Contradiction.

Finally we argue that if the computation aborts then there is a cycle. Suppose we abort while running dfsts(x) and discovering that y in adj[x] has been entered but

not exited. By Proposition 1.6.4 we see that x is reachable from y. Hence we have a cycle involving x and y.