More Examples and Applications on DFS

CSCI2100 Tutorial 12

Shangqi Lu

Department of Computer Science and Engineering
The Chinese University of Hong Kong

Adapted from the slides of the previous offerings of the course

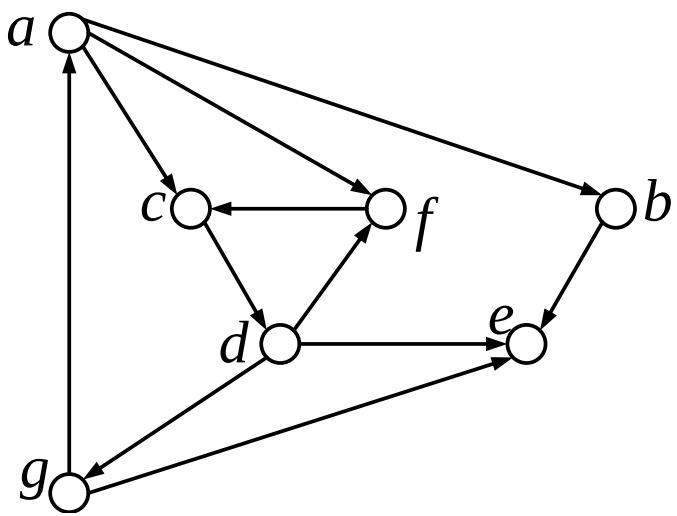
Intruction

In the previous lectures, we have already learned a surprisingly powerful algorithm – the depth first search (DFS), which is able to solve several classic problems elegantly, such as detecting cycles in a graph, topology sort and so on. Recall that our discussion focused on directed graph.

In this tutorial, we will first have a quick review about DFS algorithm through an example, then talk more about its applications and extend it to undirected graphs.

Let's first go over the DFS algorithm through a running example on directed graph.

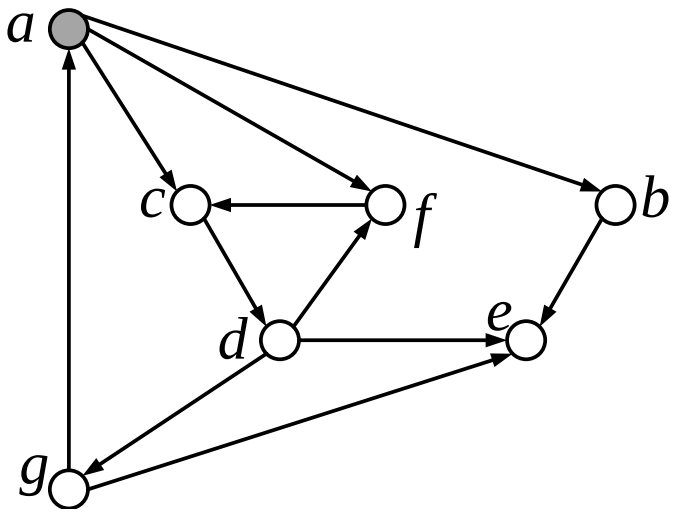
Input



Suppose we start from the vertex *a*, namely *a* is the root of DFS tree.

DFS

Firstly, create a stack S, push the starting vertex a into S and color it gray. Create a DFS Tree with a as the root. We also maintain the time interval $I(u)$ of each vertex u.

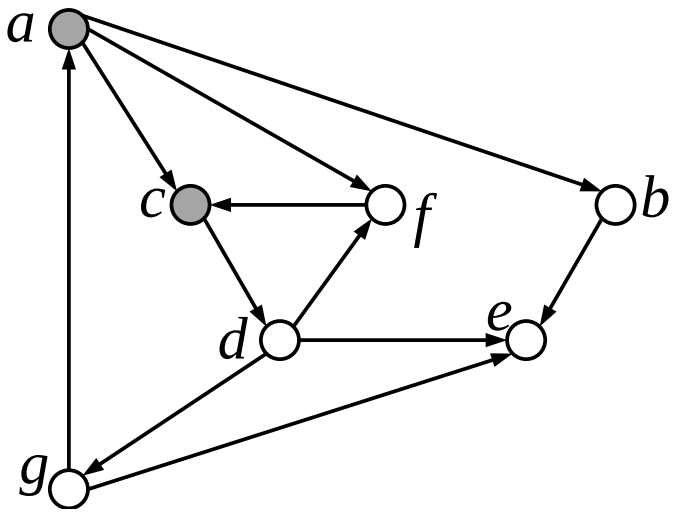


DFS Tree	Time Interval
a	$I(a) = [1, \ ]$

$S = (a)$.

DFS

Top of stack: a, which has white out-neighbors b, c, f. Suppose we access c first. Push c into S.



DFS Tree

a

|

c

Time Interval

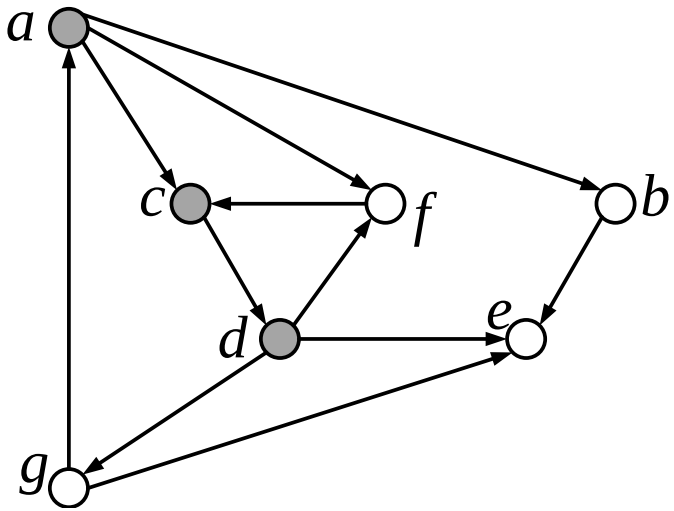
$I(a) = [1, \ ]$

$I(c) = [2, \ ]$

$S = (a, c)$.

DFS

After pushing d into S:

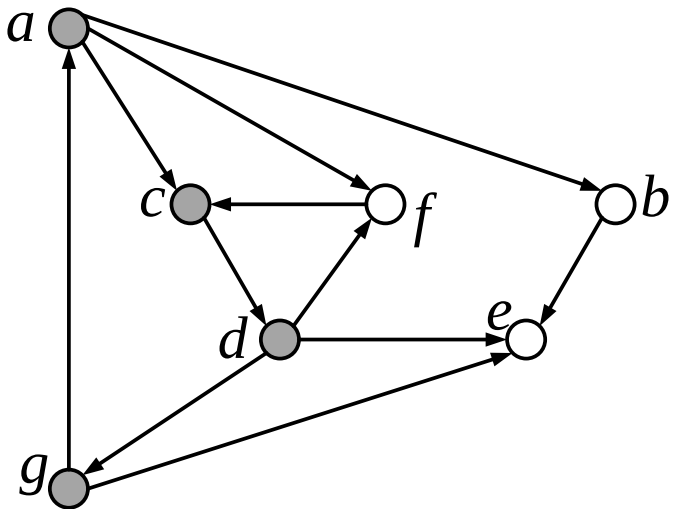


DFS Tree	Time Interval
a	$I(a) = [1, \ ]$
|	
c	$I(c) = [2, \ ]$
|	
d	$I(d) = [3, \ ]$

$S = (a, c, d)$.

DFS

Now d tops the stack. It has white out-neighbors e, f and g. Suppose we visit g first. Push g into S.

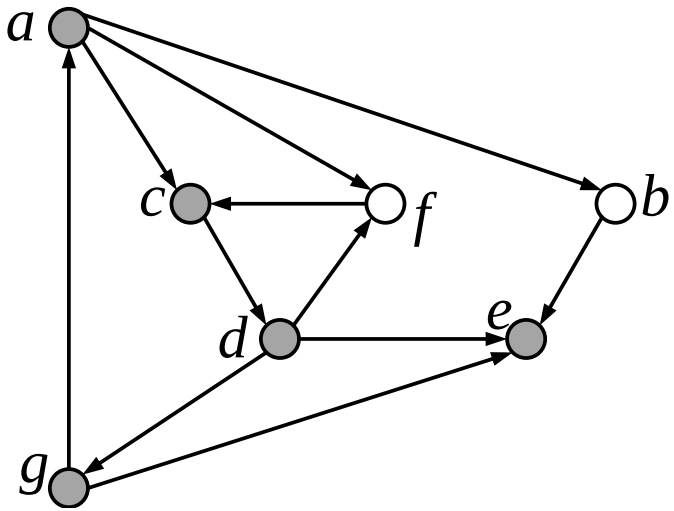


DFS Tree	Time Interval
a	$I(a) = [1, \ ]$
$\vert$	
c	$I(c) = [2, \ ]$
$\vert$	
d	$I(d) = [3, \ ]$
$\vert$	
g	$I(g) = [4, \ ]$

$S = (a, c, d, g)$.

DFS

After pushing e into S:

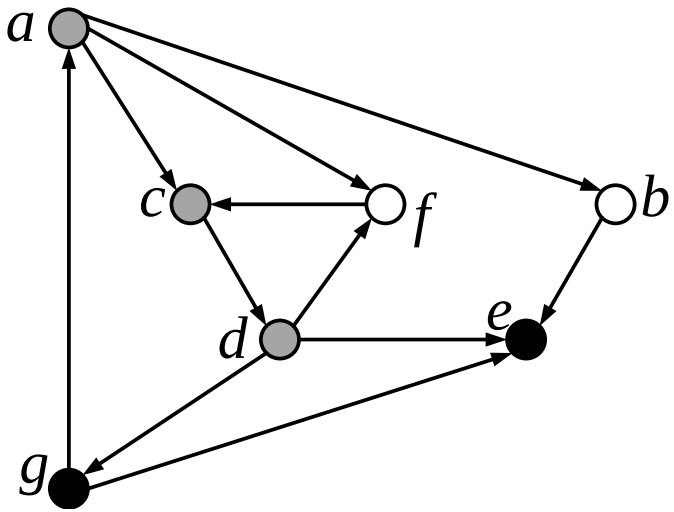


DFS Tree	Time Interval
a	$I(a) = [1, \ ]$
$\vert$	
c	$I(c) = [2, \ ]$
$\vert$	
d	$I(d) = [3, \ ]$
$\vert$	
g	$I(g) = [4, \ ]$
$\vert$	
e	$I(e) = [5, \ ]$

$S = (a, c, d, g, e)$.

DFS

e has no white out-neighbors. So pop it from S, and color it black.
Similarly, g has no white out-neighbors. Pop it from S, and color it black.

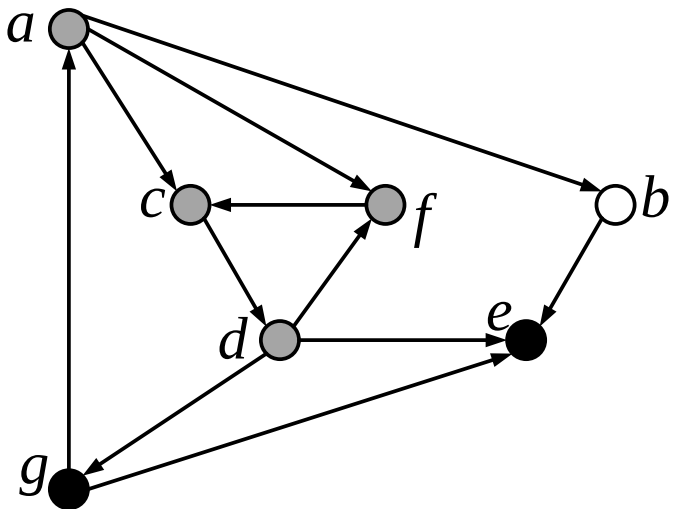


DFS Tree	Time Interval
a	$I(a) = [1, \ ]$
$\vert$	
c	$I(c) = [2, \ ]$
$\vert$	
d	$I(d) = [3, \ ]$
$\vert$	
g	$I(g) = [4, 7]$
$\vert$	
e	$I(e) = [5, 6]$

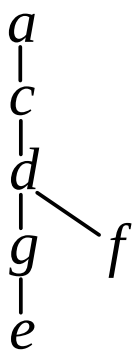
$S = (a, c, d)$.

DFS

Now d tops the stack again. It still has a white out-neighbor f. So, push f into S.



DFS Tree



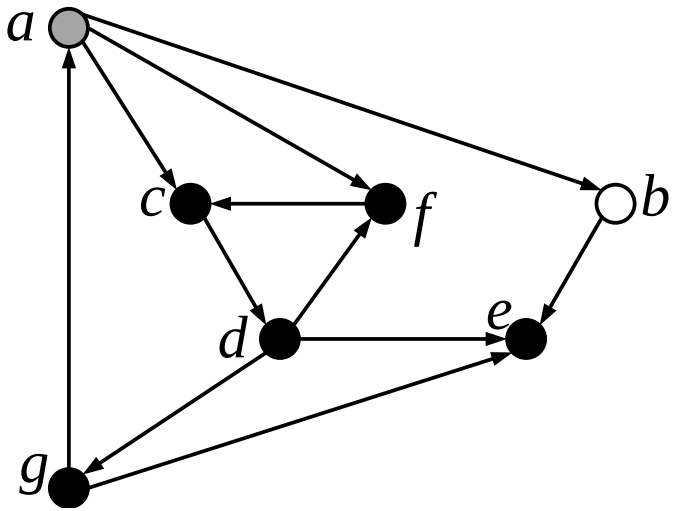
Time Interval

$I(a) = [1, \ ]$
$I(c) = [2, \ ]$
$I(d) = [3, \ ]$
$I(g) = [4, 7]$
$I(e) = [5, 6]$
$I(f) = [8, \ ]$

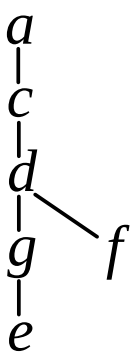
$S = (a, c, d, f)$.

DFS

After popping f, d, c:



DFS Tree



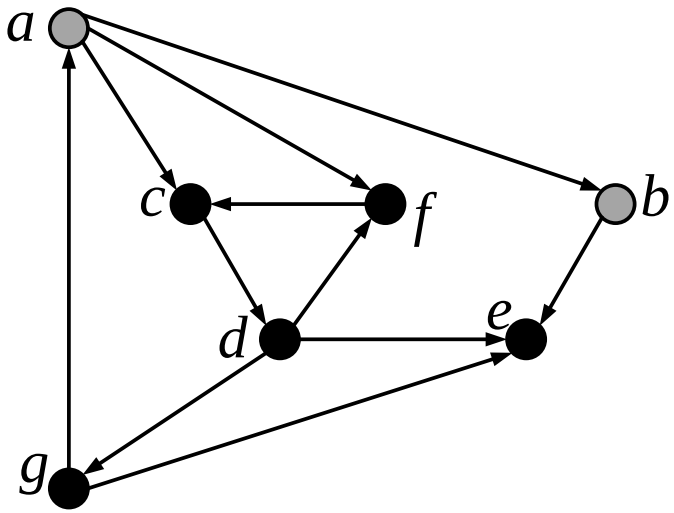
Time Interval

$I(a) = [1, \ ]$
$I(c) = [2, 11]$
$I(d) = [3, 10]$
$I(g) = [4, 7]$
$I(e) = [5, 6]$
$I(f) = [8, 9]$

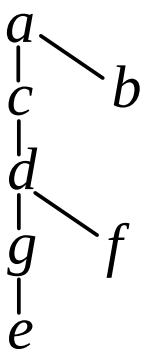
$S = (a)$.

DFS

Now a tops the stack again. It still has a white out-neighbor b. So, push b into S.



DFS Tree



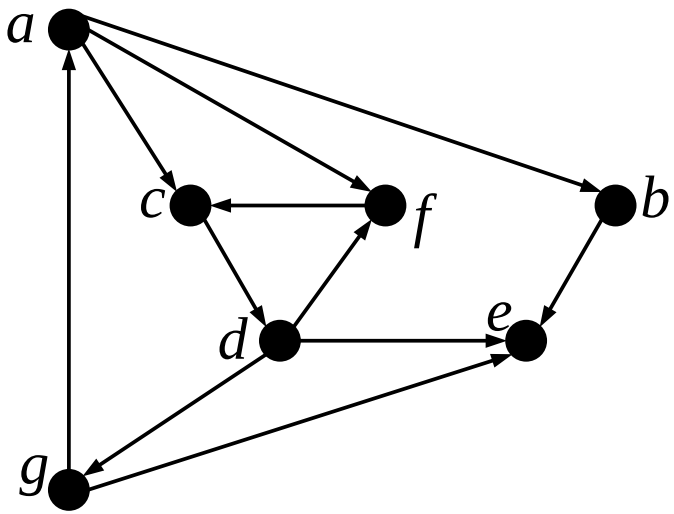
Time Interval

$I(a) = [1, \ ]$
$I(c) = [2, 11]$
$I(d) = [3, 10]$
$I(g) = [4, 7]$
$I(e) = [5, 6]$
$I(f) = [8, 9]$
$I(b) = [12, \ ]$

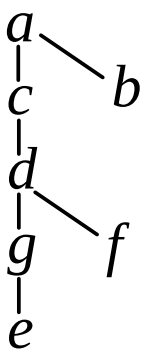
$S = (a, b)$.

DFS

After popping b and a:



DFS Tree



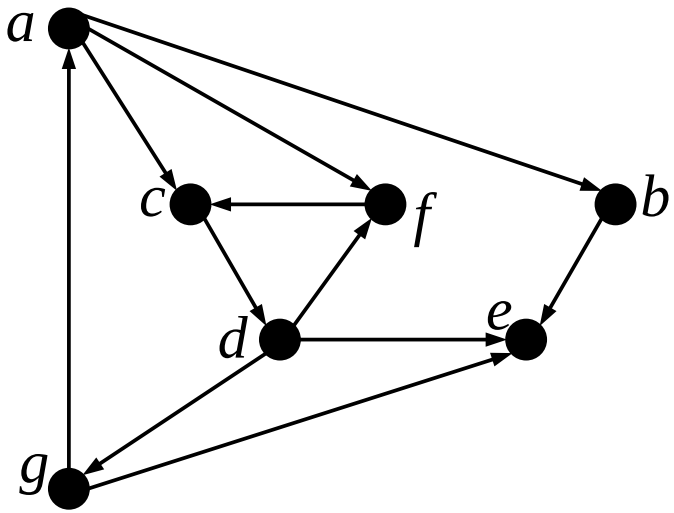
Time Interval

$I(a) = [1, 14]$
$I(c) = [2, 11]$
$I(d) = [3, 10]$
$I(g) = [4, 7]$
$I(e) = [5, 6]$
$I(f) = [8, 9]$
$I(b) = [12, 13]$

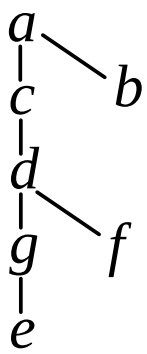
$S = ()$.

Now, there is no white vertex remaining, our algorithm terminates.

Observation



DFS Tree



Time Interval

$I(a) = [1, 14]$
$I(c) = [2, 11]$
$I(d) = [3, 10]$
$I(g) = [4, 7]$
$I(e) = [5, 6]$
$I(f) = [8, 9]$
$I(b) = [12, 13]$

From the obtained DFS tree and time interval of each vertex, we can easily determine by Parenthesis Theorem that the edge (f, c) is a backward edge, because $I(f)$ is contained in $I(c)$. Similarly, (g, a) is also a backward edge, which implies that this is a cyclic graph.

By the way, (b, e) is a cross edge because $I(b)$ and $I(e)$ are disjoint.

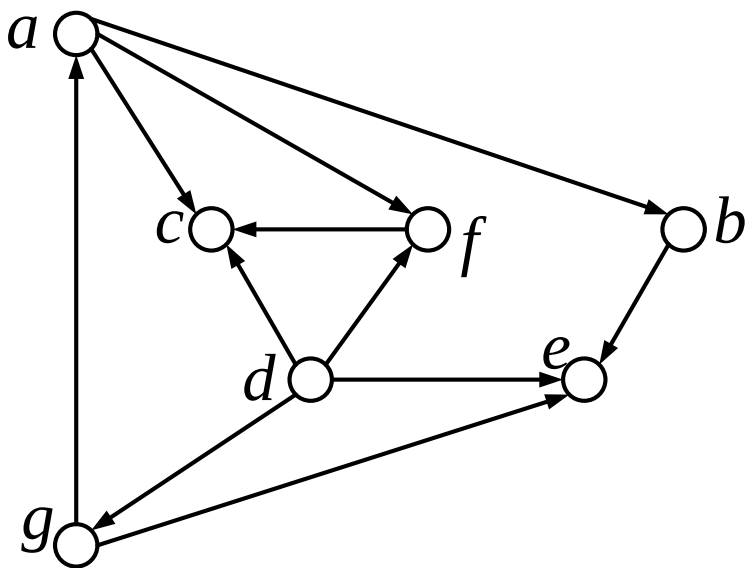
Topological Sort

Recall:

- $G = (V, E)$ is a directed acyclic graph (DAG), a topological order of G is an ordering of the vertices in V such that, for any edge (u, v), it must hold that u precedes v in the ordering.
- A directed cyclic graph has no topological orders.
- Every DAG has a topological order.

The goal of topological sort is to produce a topological order of G, this can be simply achieved by output the reverse order of L, where L is the order by which the vertices turn black when running DFS on G.

Example



Suppose that we run DFS on the above DAG starting from a, then restarting from g. The following is one possible order by which the vertices turn black:

- c, e, b, f, a, g, d.

Therefore, we output d, g, a, f, b, e, c as a topological order.

The White Path Theorem plays a vital role in proving the correctness of our DFS algorithms. Now, let's prove the theorem.

Proof of White Path Theorem

Recall:

White Path Theorem: Let u be a vertex in G. Consider the moment when u is pushed into the stack in the DFS algorithm. Then, a vertex v becomes a proper descendant of u in the DFS-forest if and only if the following is true:

- We can go from u to v by travelling only on white vertices.

Phrased differently, the theorem says that the search at u will get stuck only after discovering all the vertices that can still be discovered.

Proof of White Path Theorem

Proof:

1. only-if direction

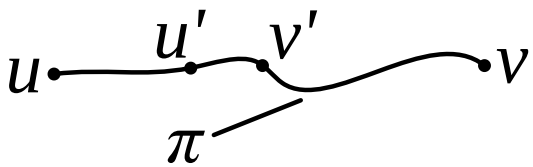
As v is a descendant of u, there is a moment in DFS when u and v were both in the stack with v being the top of the stack. It thus follows that there is a white path from u to v when u is discovered.

Proof of White Path Theorem

2. if direction

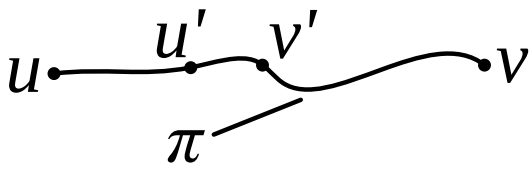
Let π be the path from u to v, we will prove that all the vertices on π must be descendants of u in the DFS forest. Let's prove this by contradiction.

Suppose this is not true. Let v' be the first vertex on π—in the order from u to v—that is not a descendant of u. Clearly $v' \neq u$. Let u' be the vertex that precedes v' on π.



Proof of White Path Theorem

2. if direction (cont.)



Consider the moment before u' turns black. As u' is a descendant of u in the DFS forest, we know u is in the stack currently.

1. The color of v' cannot be white.
   Otherwise, DFS must now push v' into stack, which contradicts the fact that u' is turning black.
2. The color of v' cannot be gray or black.
   Otherwise, it means that v' must have been pushed into the stack while u still remains in the stack, this contradicts the fact that v' is not a descendant of u.

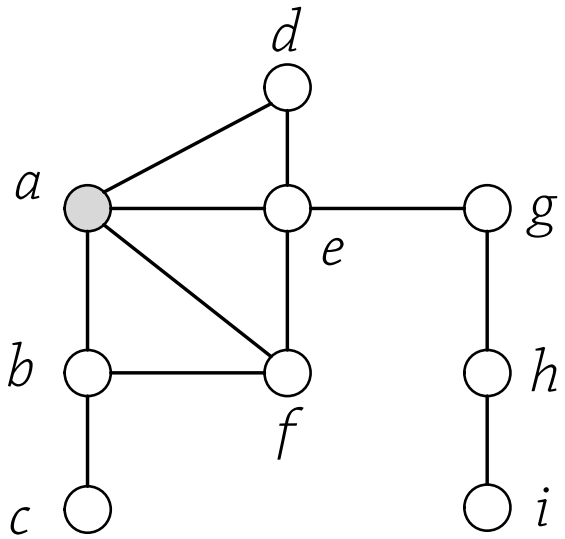
We now extend DFS to undirected graphs.

The algorithm runs in exactly the same way as DFS on a directed graph. The only difference is that, a vertex u is popped out of the stack, only if none of its neighbors (instead of out-neighbors) is still white.

We will illustrate the algorithm through an example.

DFS on Undirected Graphs

Firstly, create a stack S, push the starting vertex a into S and color it gray. Create a DFS Tree with a as the root.



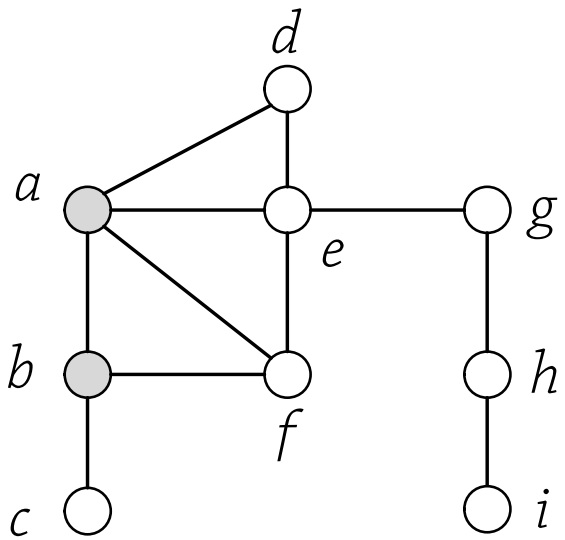
DFS Tree

a

$S = (a)$

DFS on Undirected Graphs

Top of stack: a, which has white neighbors b, f, e and d. Suppose that we access b first. Push b into S.

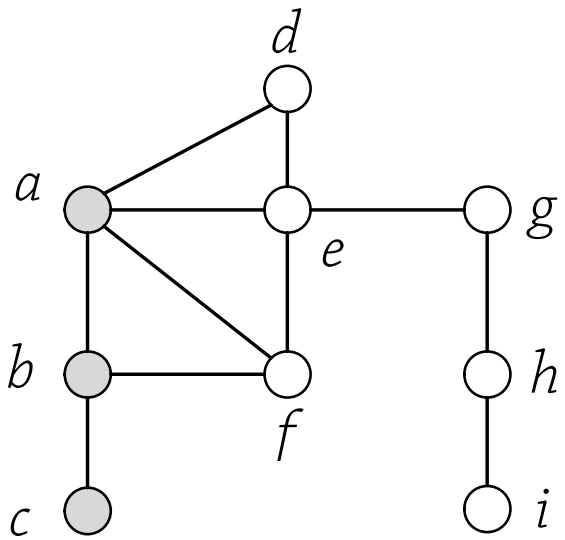


DFS Tree

$S = (a, b)$

DFS on Undirected Graphs

After pushing c.

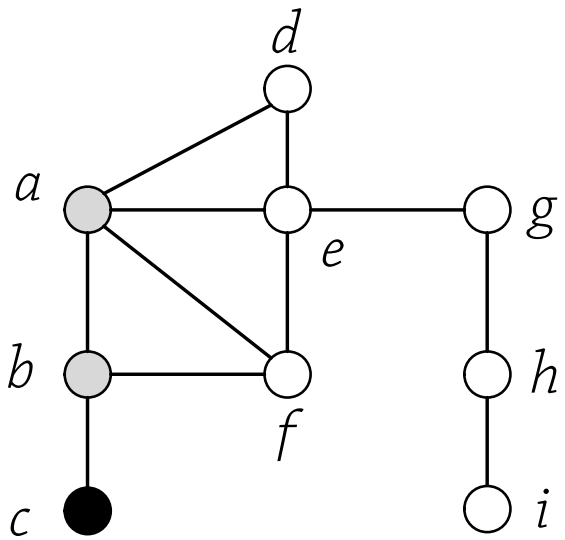


DFS Tree

$S = (a, b, c)$

DFS on Undirected Graphs

Since c has no white neighbors, pop it from S, and color it black.

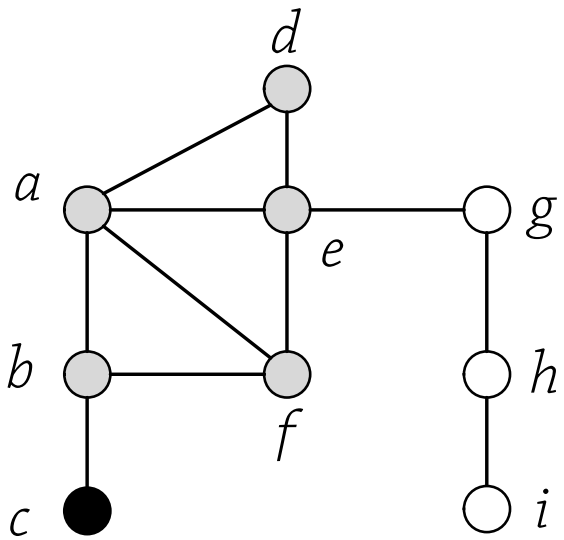


DFS Tree

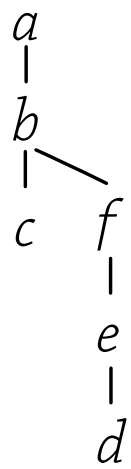
$S = (a, b)$

DFS on Undirected Graphs

After pushing f, e, d.



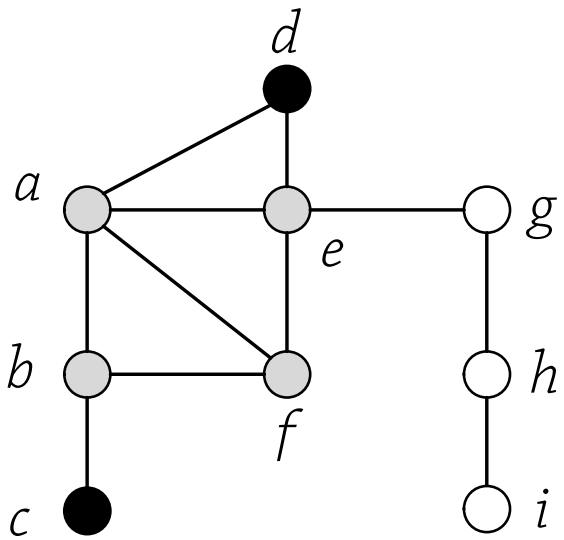
DFS Tree



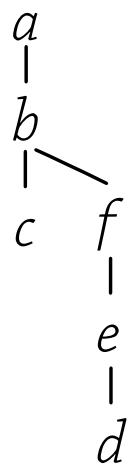
$S = (a, b, f, e, d)$

DFS on Undirected Graphs

Since d has no white neighbors, pop it from S, and color it black.



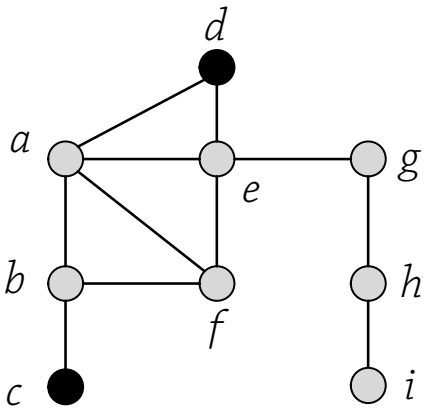
DFS Tree



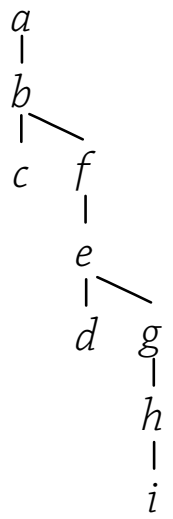
$S = (a, b, f, e)$

DFS on Undirected Graphs

Consecutively push g, h, i.



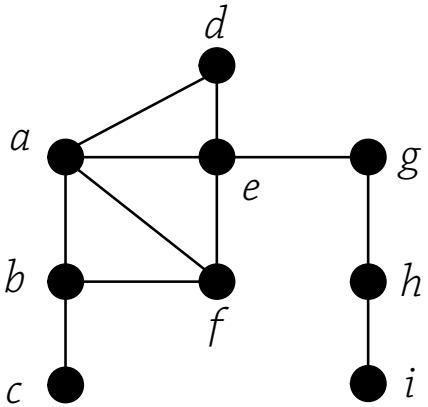
DFS Tree



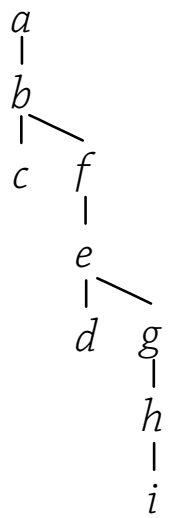
$S = (a, b, f, e, g, h, i)$

DFS on Undirected Graphs

After popping i, h, g, e, f, b, a.



DFS Tree



$S = ()$

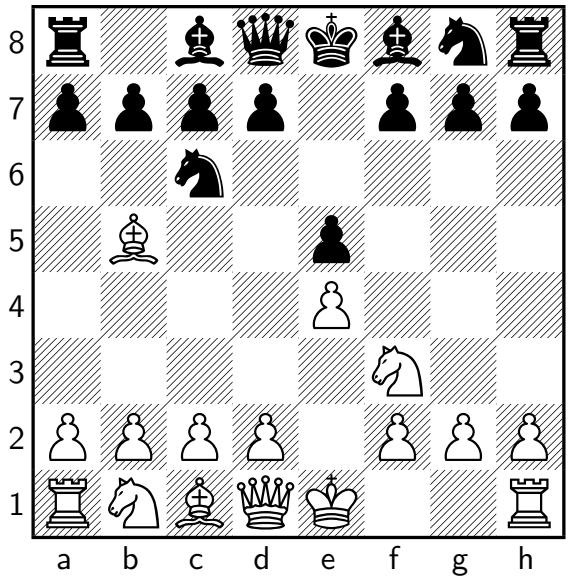
Done.

In the next few slides we will look at a "real-life" problem that can be modelled with a graph and solved using graph algorithm. The process essentially boils down to identifying:

1. where the graph is, answering questions like what are the vertices, what are the edges, are the edges directed, is the graph acyclic, etc; and
2. which graph algorithm to apply to solve the given problem.

Knight Placement

Suppose we were given a configuration of a chessboard, e.g.



Given a knight on the board, say the black knight on g8, the problem is to decide it is able to reach every unoccupied square on the board via only unoccupied squares (i.e. we can't take) if every other piece on the board are frozen in place.

Knight Placement

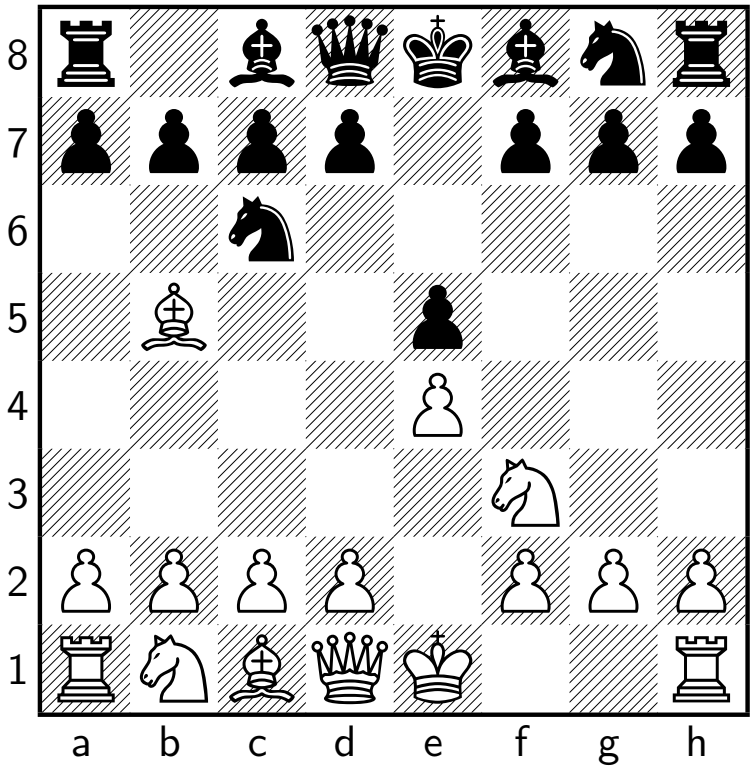
We will model the graph in the following way:

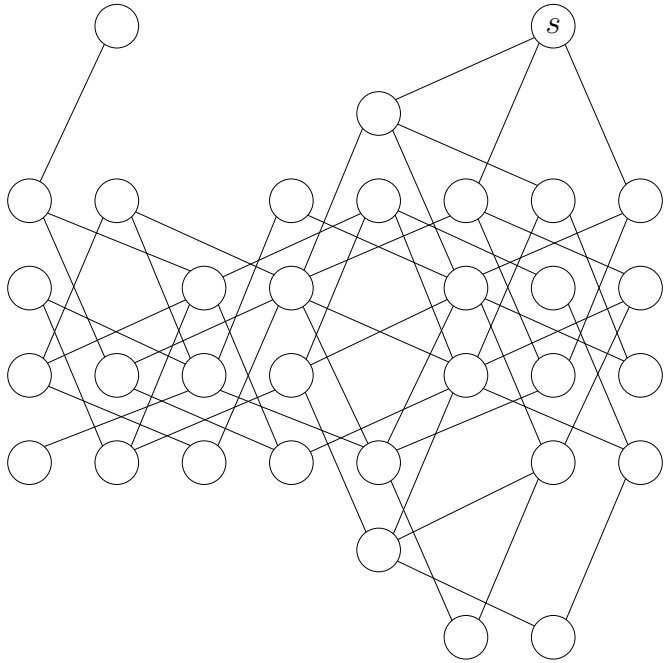
- Vertices: each unoccupied square on the board is a vertex, additionally the square the target knight is on is also a vertex.
- Edges: add an undirected edge for every pair of vertices u and v that are a knight's move apart.

The problem then is to check that the graph is connected, which we can do using DFS.

Knight Placement

Using the example shown earlier, the graph looks like:

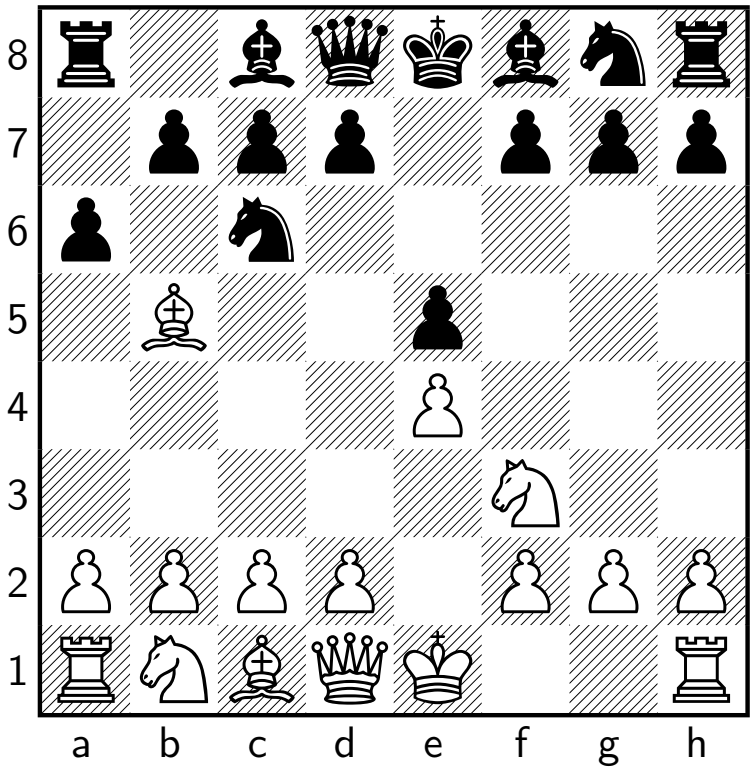


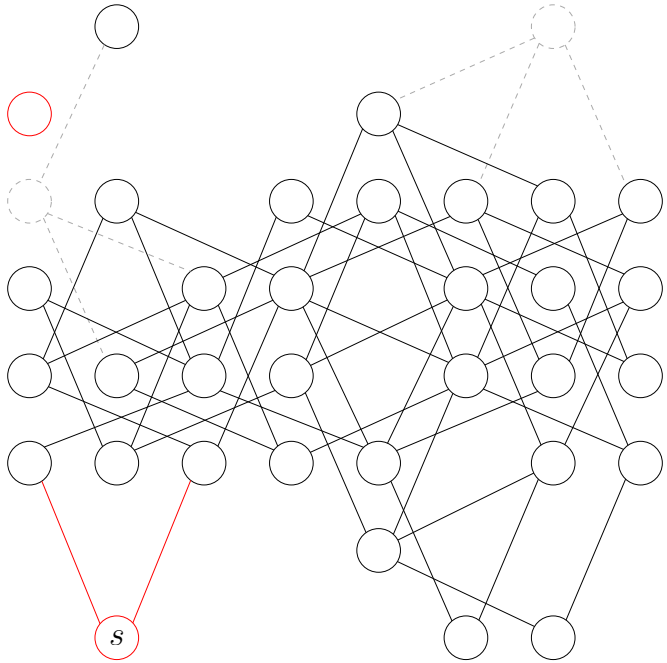


At the end of running DFS on this graph with s being our source node, every node in the graph would be coloured black and hence is reachable from s, thus the answer is yes.

Knight Placement

After moving a7 to a6, if we consider then the white knight on b1:





The graph is now disconnected with 3 connected components, and thus the answer is no.