Lecture 23

Graph traversal algorithms

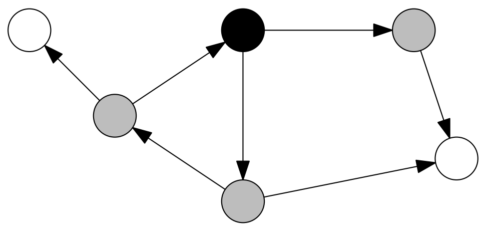
Traversals involve visiting each node of a digraph in a systematic way following only the arcs in the digraph. This is a very common task when dealing with digraphs and we will cover several applications in later lectures.

23.1 The general graph traversal algorithm

All graph traversal algorithms have the same basic structure that relies on keeping track of which nodes have been visited and whether they may be adjacent to nodes that have not yet been visited. We use a system of three colours:

- ***White nodes*** have not yet been visited;
- ***Grey nodes*** (or **frontier nodes**) have been visited but may have out-neighbours that are white;
- ***Black nodes*** have been visited and all their out-neighbours have been visited too (so are not white).

Example 23.1. Node states during a digraph traversal.



The traversal algorithm which visits all nodes in a digraph is loosely described as follows:

- All nodes are white to begin with.
- A starting white node is chosen and turned grey.
- A grey node is chosen and its out-neighbours explored.
- If any out-neighbour is white, it is visited and turned grey. If no out-neighbours are white, the grey node is turned black.
- The process of choosing grey nodes and exploring neighbours is continued until all nodes reachable from the initial node are black.
- If any white nodes remain in the digraph, a new starting node is chosen and the process continues until all nodes are black.

We keep track of the order in which nodes are visited by recording which arc was followed when a node first turned from white to grey. As we will see, this creates a sub-digraph of the original digraph which has a tree structure.

The traversal algorithm is formalised in the pseudocode of Algorithms 1 and 2. Algorithm 1 (`traverse`) initialises the process and is tasked with finding white nodes to start from. Most of the work is done in Algorithm 2 (`visit`) which takes a starting node and explores the portion of the subgraph reachable from that node.

Algorithm 1 Basic graph traversal main routine: `traverse`.

```
1: function TRAVERSE(digraph $G$)
2:     array colour[$0..n-1$]                    ▹ records node colour
3:     array pred[$0..n-1$]          ▹ records path of traversal as a tree
4:     for $u \in V(G)$ do                       ▹ make all nodes white
5:         colour[$u$] ← WHITE
6:     for $s \in V(G)$ do                          ▹ find a white node
7:         if colour[$s$] == WHITE then
8:             visit($s$)                      ▹ start traversal from $s$
       return pred
```

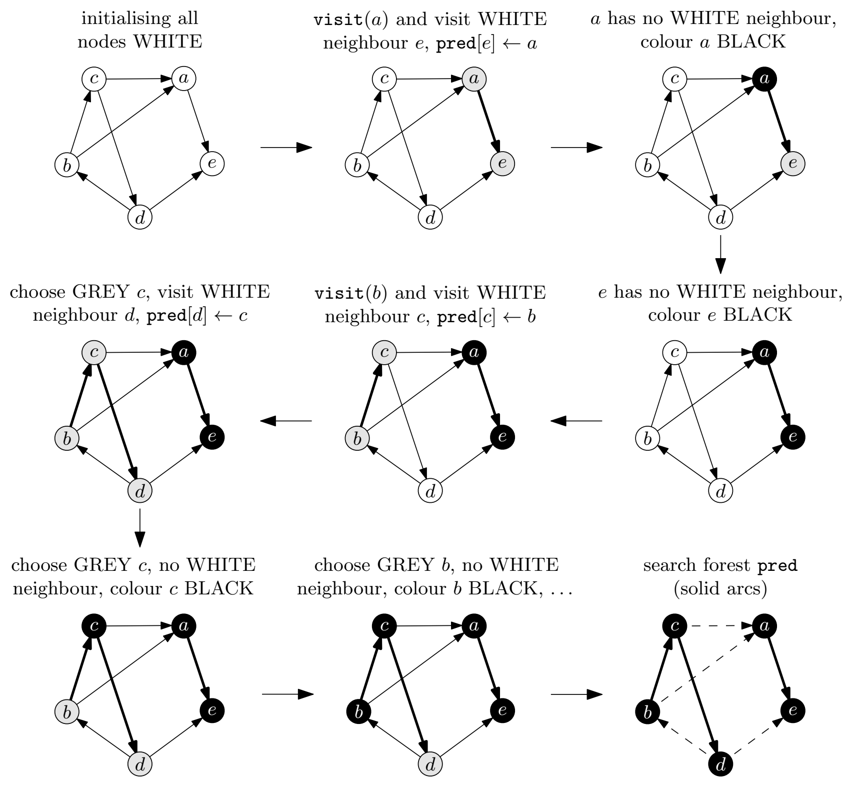
Algorithm 2 Basic graph traversal subroutine: `visit`.

```
1: function VISIT(node $s$ of digraph $G$)
2:     colour[$s$] ← GREY
3:     pred[$s$] ← null                ▷ make start node the root of the tree
4:     while there is a GREY node do   ▷ continues until all nodes are black
5:         choose a GREY node $u$
6:         if $u$ has a WHITE neighbour then
7:             choose a WHITE neighbour, $v$
8:             colour[$v$] ← GREY      ▷ $v$ is visited for first time, so turns grey
9:             pred[$v$] ← $u$ ▷ make $u$ the parent of $v$ in the traversal tree
10:        else
11:            colour[$u$] ← BLACK     ▷ $u$ has no white neighbours so done with $u$
```

A call to `visit` creates a subdigraph of G that is a tree: the nodes are precisely the black nodes (all nodes reachable from the initial node), and the arcs are those arcs followed when we found a white neighbour of a grey node.

Each white node chosen in `traverse` is the root of a tree in the call to `visit`. Eventually, we obtain a set of disjoint trees spanning the digraph (that is, it includes all the nodes of the digraph), which we call the ***search forest***. The search forest is returned by `traverse` in the array `pred`.

Example 23.2. Traversal of a digraph. Visiting a white vertex turns it grey.


initialising all nodes WHITE
visit(a) and visit WHITE neighbour e, pred[e] ← a
a has no WHITE neighbour, colour a BLACK
choose GREY c, visit WHITE neighbour d, pred[d] ← c
visit(b) and visit WHITE neighbour c, pred[c] ← b
e has no WHITE neighbour, colour e BLACK
choose GREY c, no WHITE neighbour, colour c BLACK
choose GREY b, no WHITE neighbour, colour b BLACK, . . .
search forest pred (solid arcs)
a
b
c
d
e

	a	b	c	d	e
Pred:	-1	-1	b	c	a

23.2 Classifying arcs after a traversal

It helps in analysing the traversal algorithm to classify the arcs of G based on their relationships in the search forest.

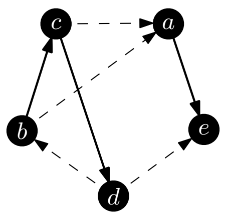
Definition 23.3. Suppose we have performed a traversal of a digraph G, resulting in a search forest F. Let $(u, v) \in E(G)$ be an arc. Then (u, v) is a

- ***tree arc*** if it belongs to one of the trees of F;
- or, if it is not a tree arc, it is a

- ***forward arc*** if u is an ancestor of v in F;
- ***back arc*** if u is a descendant of v in F;
- ***cross arc*** if neither u nor v is an ancestor of the other in F.

A cross arc may join two nodes in the same tree or point from one tree to another in the search forest. Tree, forward and back arcs require that u and v be in the same tree.

Example 23.4. Classify the arcs according to the definitions above.



(a, e) is a tree arc

(d, b) is a back arc

(c, a) is a cross arc

roots

$$\begin{tikzcd} \boxed{b} \arrow[d] & \boxed{a} \arrow[d] \\ c \arrow[d] & e \\ d & \end{tikzcd}$$

The following theorem collects all the basic facts we need for proofs in later sections. The proofs are simple and can be done as exercises.

Theorem 23.5. Suppose that we have carried out `traverse` on G, resulting in a search forest F. Let $v, w \in V(G)$.

- Let T_1 and T_2 be different trees in F and suppose that T_1 was explored before T_2. Then there are no arcs from T_1 to T_2.
- Suppose that G is a graph. Then there can be no edges joining different trees of F.
- Suppose that v is visited before w and w is reachable from v in G. Then v and w belong to the same tree of F.
- Suppose that v and w belong to the same tree T in F. Then any path from v to w in G must have all nodes in T.

Example 23.6. Assuming the first statement of Theorem 23.5, prove the third statement.

- suppose v was visited before w, and there's a walk $vu_1 \cdots u_k w$
- suppose v & w in different trees T_1 & T_2

and so along the walk $vu_1 \cdots u_k w$, there will be an arc connecting T_1 & T_2, ∴ in same tree

23.3 Running time for the general traversal algorithm

The generality of our traversal algorithm makes its exact running time impossible to determine. It depends on how one chooses the next grey node u and its white neighbour v. Some schemes for choosing u and v can be complex and depend on n. The schemes we consider here are simple and constant time.

- The initialization of the initialize 'pred' as well array `colour` takes time $\Theta(n)$ so `traverse` is in $\Theta(n + t)$, where t is the total time taken by all the calls to `visit`.
- We execute the while-loop of `visit` in total $\Theta(n)$ times since every node must eventually move from white through grey to black.
- The time taken in choosing grey nodes is $\Theta(n)$.
- The time taken to find a white neighbour involves examining each neighbour of u and checking whether it is white, then applying a selection rule.
- The total time in choosing white neighbours is in $\Theta(m)$ if adjacency lists are used and is in $\Omega(n^2)$ if an adjacency matrix is used.

In sparse digraph,
when is $n+m < n^2$?
When (m) is small enough

- So the running time of `traverse` is $\Theta(n + (n + m)) = \Theta(n + m)$ if adjacency lists are used, and $\Theta(n + n^2) = \Theta(n^2)$ if the adjacency matrix format is used.

So for simple selection rules and assuming a sparse input digraph, the adjacency list format seems preferable. But if more complex selection rules are used, for example, choosing a single grey node is of order n while choosing a single white node is still constant time, then the running time is asymptotically $\Theta(n^2)$ regardless of the data structure, so using the adjacency matrix is not clearly ruled out.