Graph Traversals I

Instructor: Meng-Fen Chiang

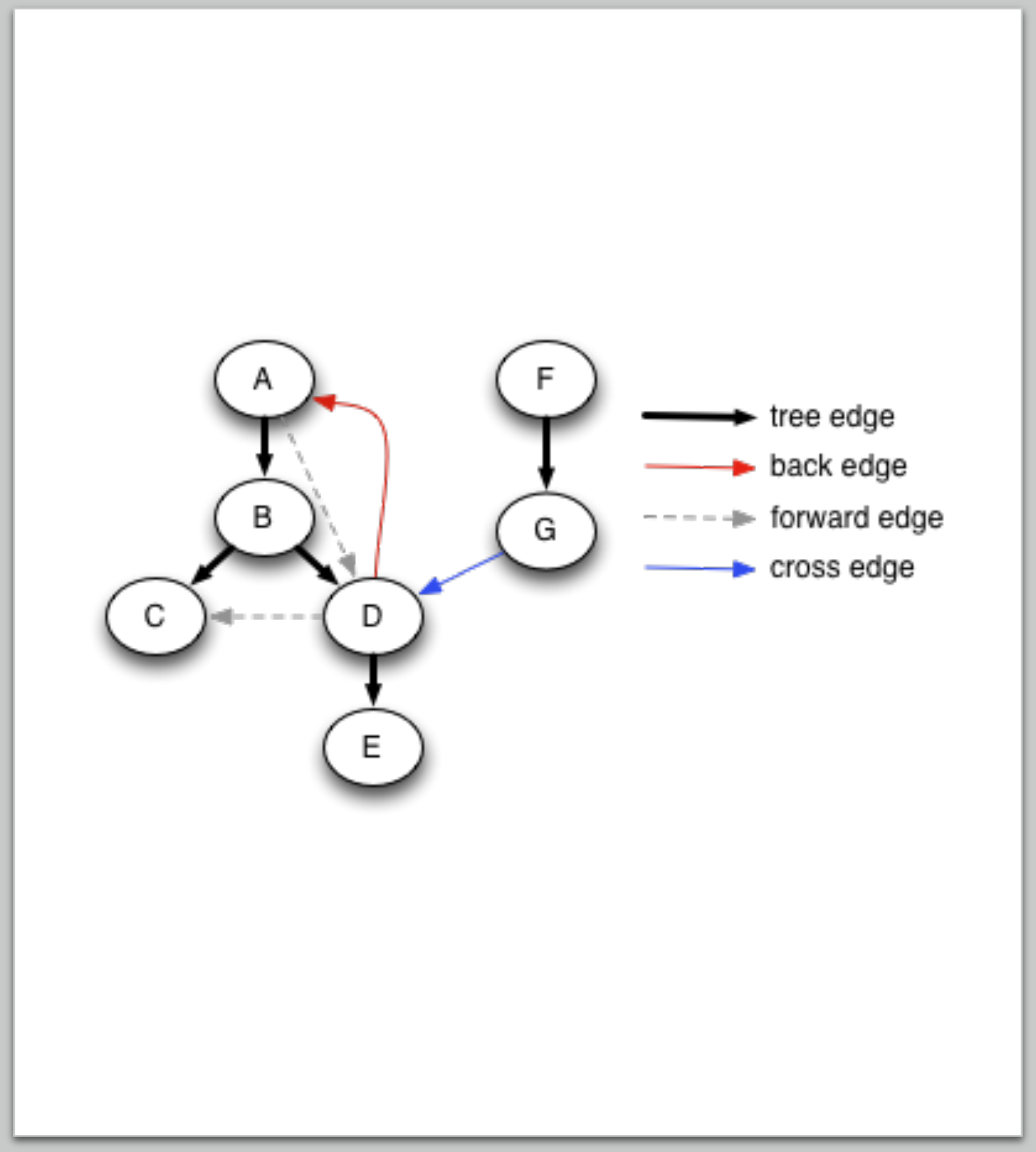
COMPCSI220: WEEK 10

THE UNIVERSITY OF AUCKLAND
Te Whare Wananga o Tamaki Makaurau
NEW ZEALAND

Slides adapted from Mark Wilson, Georgy Gimel'farb, Simone Linz and Tanya Gvozdeva

OUTLINE

- Graph Traversal Algorithm
- Facts about Traversal Trees
- Complexity Analysis
- Illustrative Example



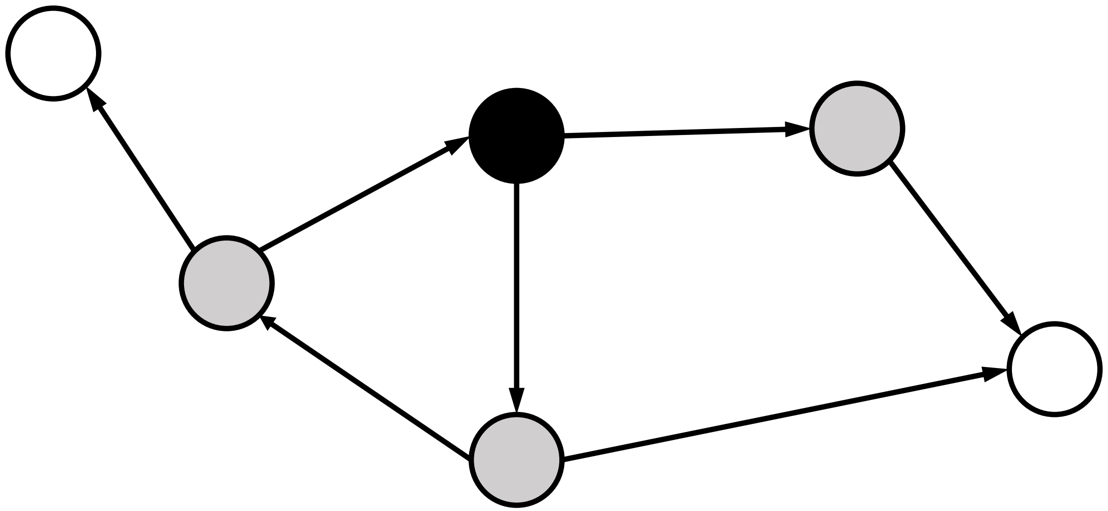
Motivation: Graph Traversals

- Want to visit each node of a digraph in a systematic and efficient way (e.g. to search a graph).

- We can walk only on arcs following their direction

General Graph Traversal: Colour Scheme

- All graph traversal algorithm follow the same structure which is called the The general graph traversal algorithm. This algorithm uses three types of nodes:
  - **White nodes**: have not yet been visited.
  - **Grey (frontier) nodes**: have been visited but may have adjacent nodes that are white.
  - **Black nodes**: have been visited and all their (out-)neighbors have been visited as well.



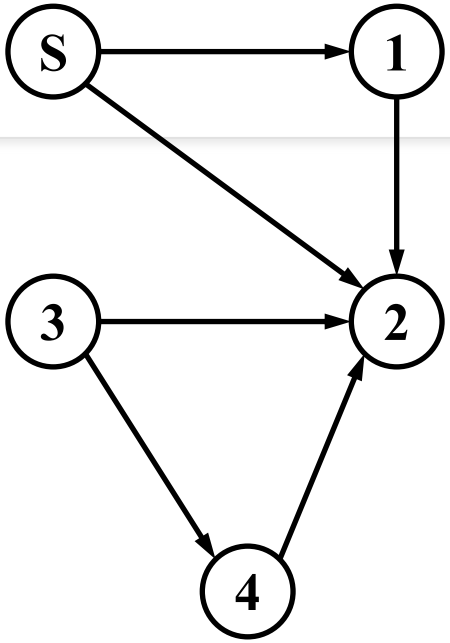
Graph Traversal Algorithm

- 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**.

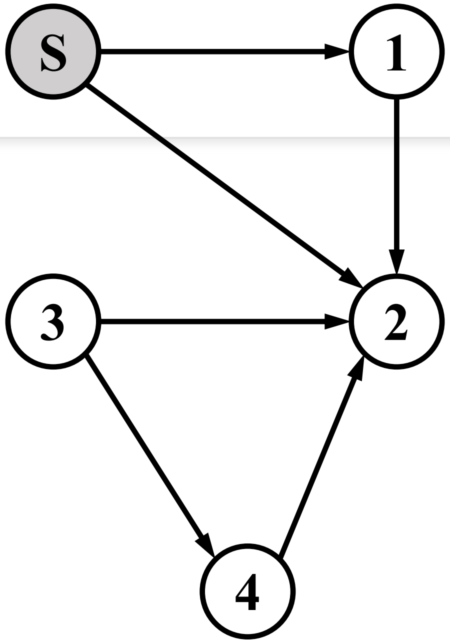
General Graph Traversal: Visit(s)



1. s is coloured grey and $pred[s]$=$null$.
2. choose a grey node u.
3. if u has a white (out)-neighbour v then colour v grey and $pred[v]$=u else colour u black.
4. if we have grey nodes go to 2).

* *pred - predecessor*

General Graph Traversal: Visit(s)



1. s is coloured grey and $pred[s]$=$null$.
2. choose a grey node u.
3. if u has a white (out)-neighbour v then colour v grey and $pred[v]$=u else colour u black.
4. if we have grey nodes go to 2).

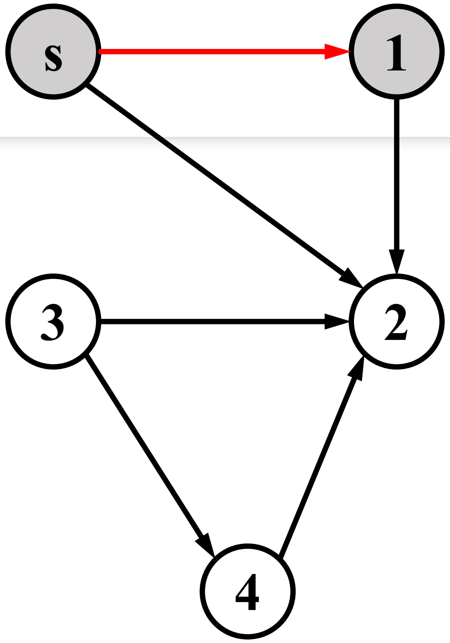
$\boldsymbol{Pred[s] = null}$

* *pred* - *predecessor*

General Graph Traversal: Visit(s)

1. s is coloured grey and $pred[s]$=$null$.
2. choose a grey node u.
3. if u has a white (out)-neighbour v then colour v grey and $pred[v]$=u else colour u black.
4. if we have grey nodes go to 2).

* $pred$ - $predecessor$

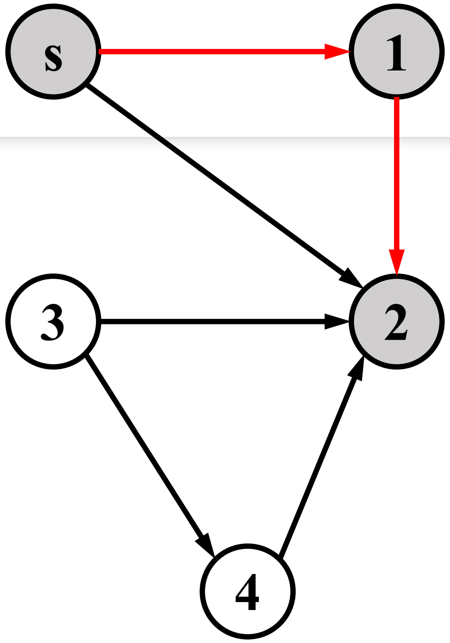


$\boldsymbol{Pred[1] = s}$

General Graph Traversal: Visit(s)

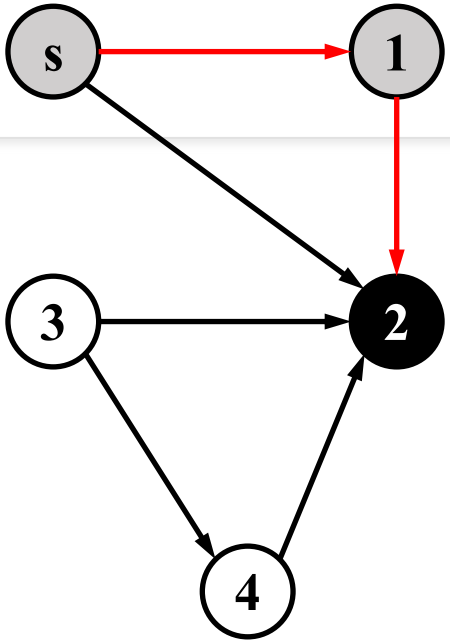
1. s is coloured grey and $pred[s]$=$null$.
2. choose a grey node u.
3. if u has a white (out)-neighbour v then colour v grey and $pred[v]$=u else colour u black.
4. if we have grey nodes go to 2).

* *pred - predecessor*



$\boldsymbol{Pred[2] = 1}$

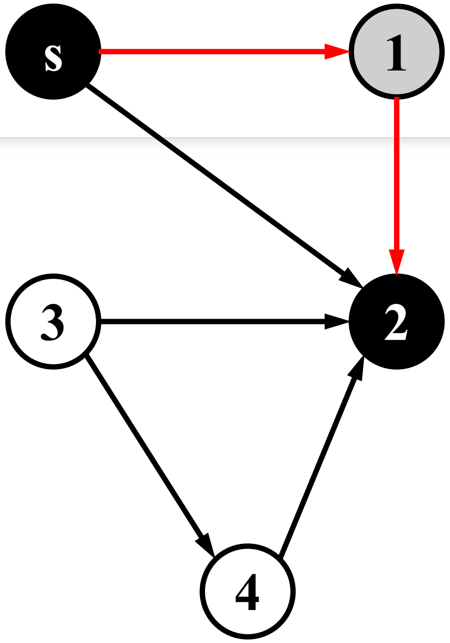
General Graph Traversal: Visit(s)



1. s is coloured grey and $pred[s]$=$null$.
2. choose a grey node u.
3. if u has a white (out)-neighbour v then colour v grey and $pred[v]$=u else colour u black.
4. if we have grey nodes go to 2).

* *pred - predecessor*

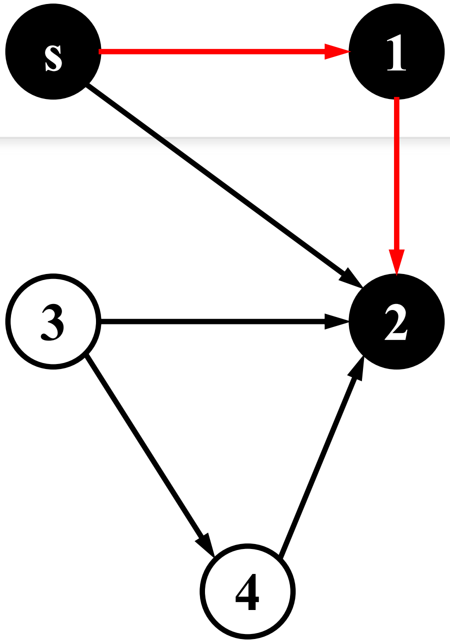
General Graph Traversal: Visit(s)



1. s is coloured grey and $pred[s]$=$null$.
2. choose a grey node u.
3. if u has a white (out)-neighbour v then colour v grey and $pred[v]$=u else colour u black.
4. if we have grey nodes go to 2).

* *pred* - *predecessor*

General Graph Traversal: Visit(s)



1. s is coloured grey and $pred[s]$=$null$.
2. choose a grey node u.
3. if u has a white (out)-neighbour v then colour v grey and $pred[v]$=u else colour u black.
4. if we have grey nodes go to 2).

* $pred – predecessor$

- $Visit(s)$ visits all nodes reachable from s.
- After the run of $visit(s)$ all reachable nodes are coloured black.

General Graph Traversal: Visit(s)

Algorithm 1 Visit.

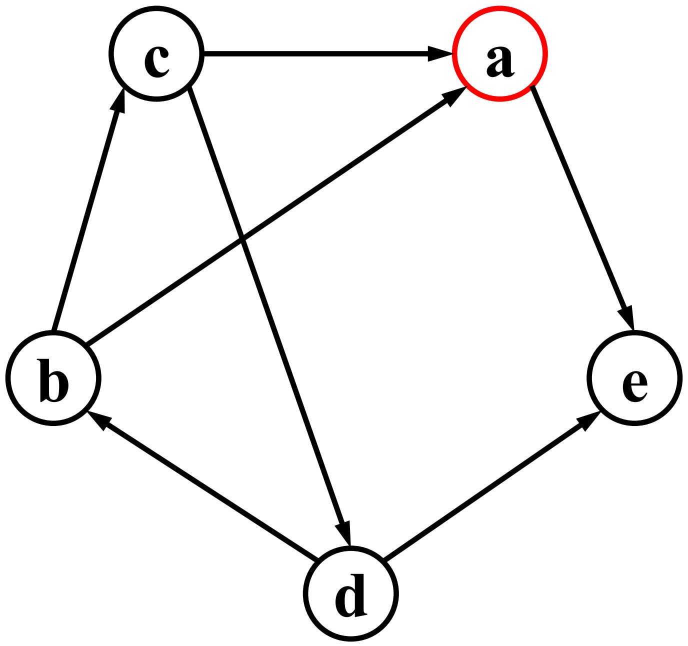
```
function VISIT(node s of digraph G)
    color[s] ← Grey
    pred[s] ← Null
    while there is a Grey node do
        choose a Grey node u
        if u has a WHITE (out-)neighbour then
            choose such a white (out-)neighbour v
            color[v] ← Grey
            pred[v] ← u
        else
            color[u] ← Black
```

General Graph Traversal Algorithm: Main

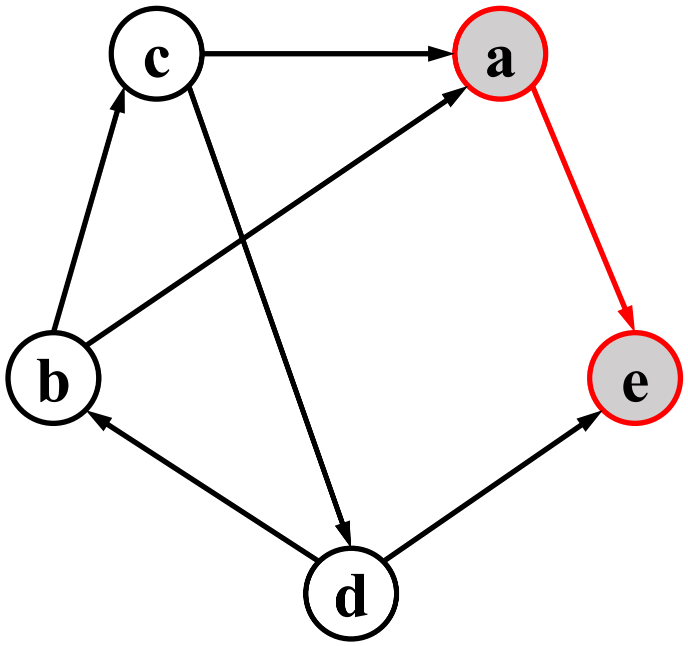
Algorithm 2 Traverse.

```
function TRAVERSE(digraph G)
    array color[0..n − 1]
    array pred[0..n − 1]
    for u ∈ V(G) do
        color[u] ← WHITE
    end for
    for s ∈ V(G) do
        if color[s] = WHITE then
            VISIT(s)
    return pred
```

Illustrating the general traversal algorithm



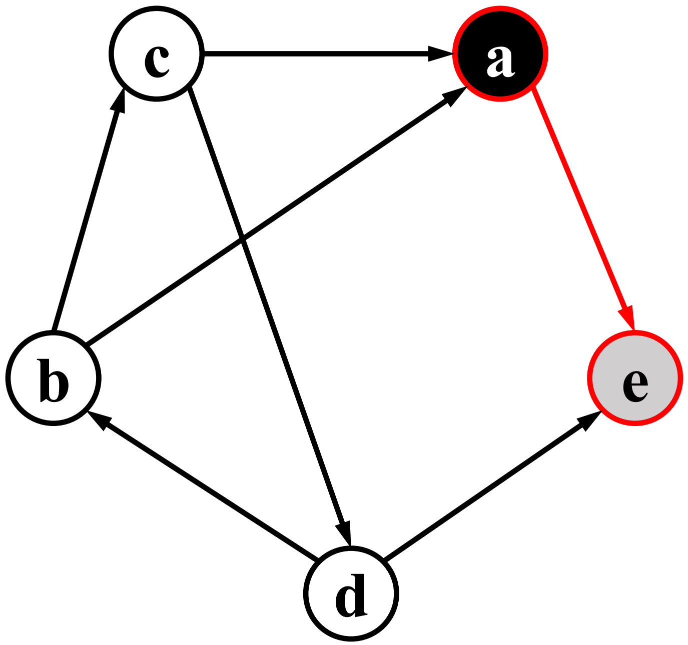
Illustrating the general traversal algorithm



- VISIT(a)

  e is the white neighbour of a

Illustrating the general traversal algorithm

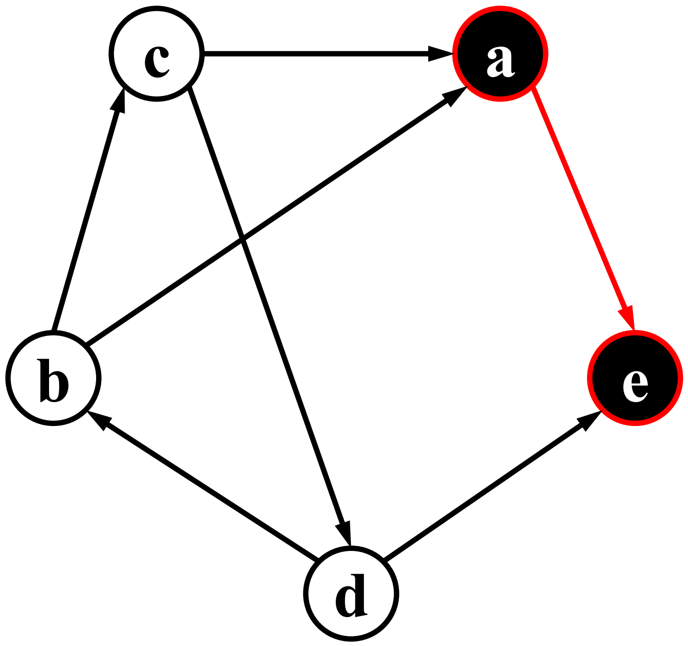


- VISIT(a)

  e is the white neighbour of a

  choose grey a; no white neighbour; colour black

Illustrating the general traversal algorithm



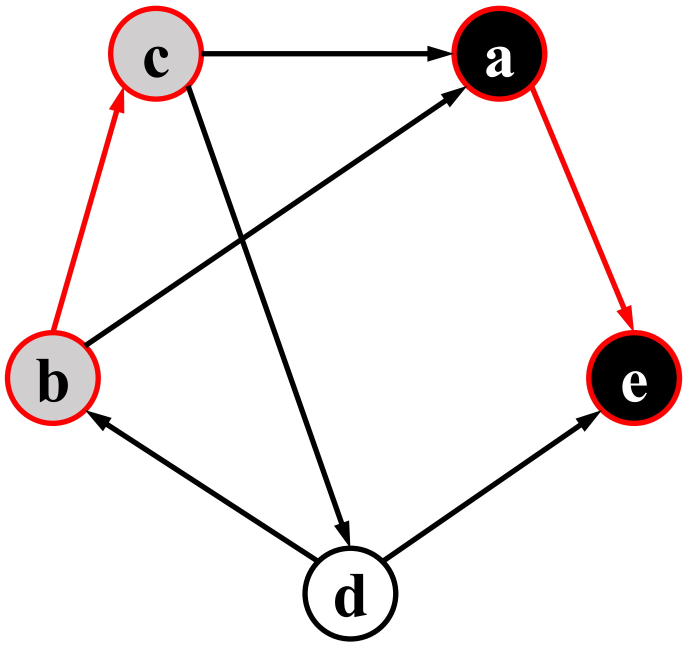
- VISIT(a)

  e is the white neighbour of a

  choose grey a; no white neighbour; colour black

  choose grey e; no white neighbour; colour black

Illustrating the general traversal algorithm



- VISIT(a)

  e is the white neighbour of a

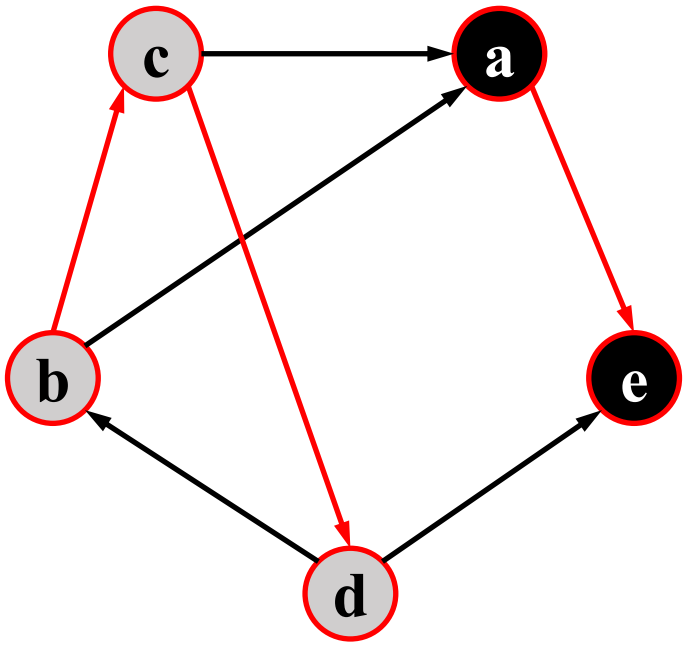
  choose grey a; no white neighbour; colour black

  choose grey e; no white neighbour; colour black

- VISIT(b)

  c is the white neighbour of b

Illustrating the general traversal algorithm



- VISIT(a)

  e is the white neighbour of a

  choose grey a; no white neighbour; colour black

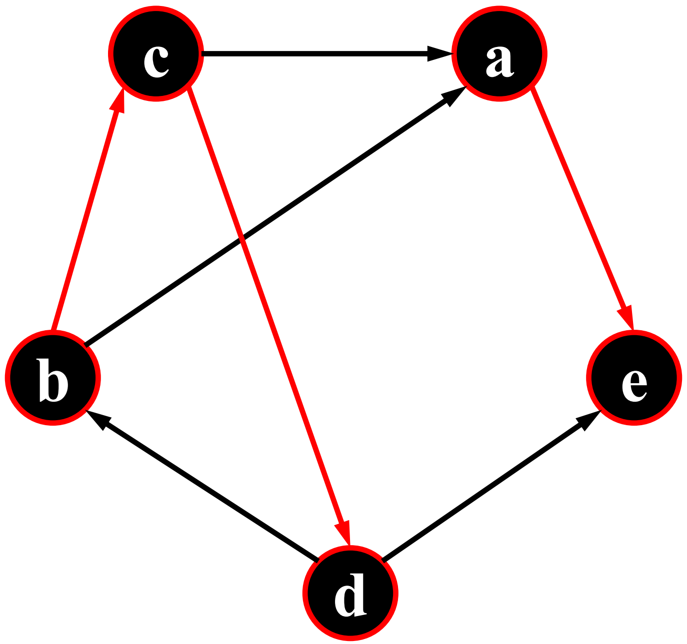
  choose grey e; no white neighbour; colour black

- VISIT(b)

  c is the white neighbour of b

  choose grey c; d is white neighbour

Illustrating the general traversal algorithm



- VISIT(a)

  e is the white neighbour of a

  choose grey a; no white neighbour; colour black

  choose grey e; no white neighbour; colour black

- VISIT(b)

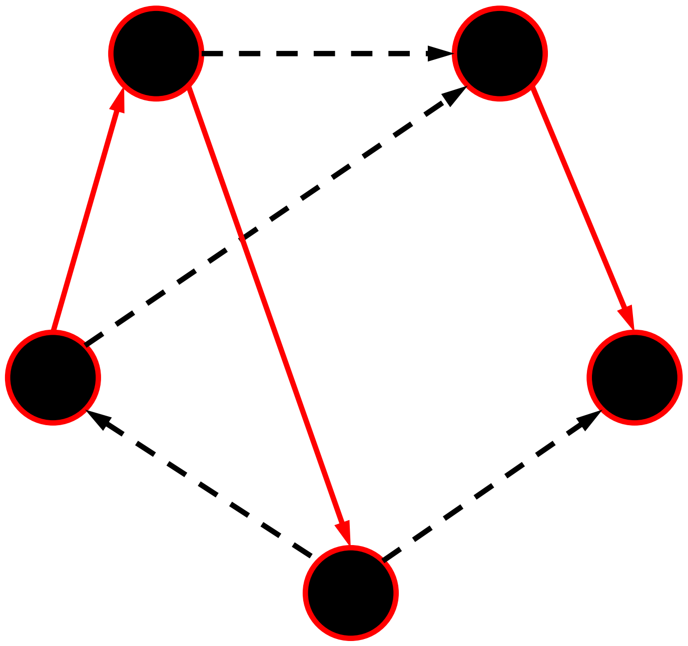
  c is the white neighbour of b

  choose grey c; d is white neighbour

  no more white nodes; all nodes turn black

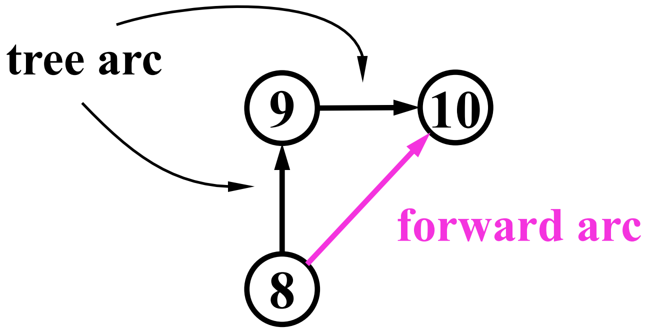
A search forest

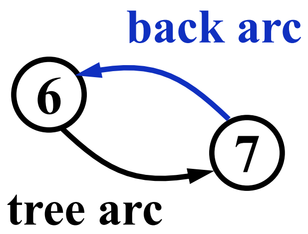
A search forest is a collection of node-disjoint trees that span the digraph and contain, for each node u with $pred[u] \neq NULL$, the arc $(pred[u], u)$.

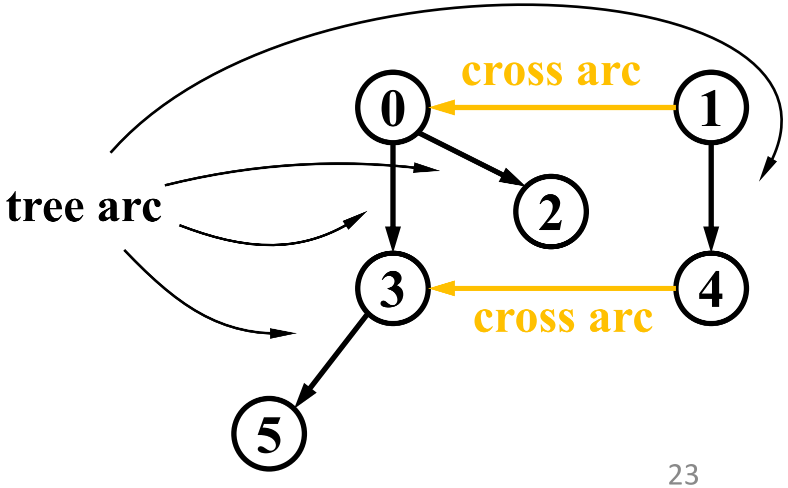


Traversal Arc Classifications

- 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.
- The arc is called a tree arc if it belongs to one of the trees of F. If the arc is not a tree arc, there are three possibilities:
  - a forward arc if u is an ancestor of v in F,
  - a back arc if u is a descendant of v in F, and
  - a cross arc if neither u nor v is an ancestor of the other in F.

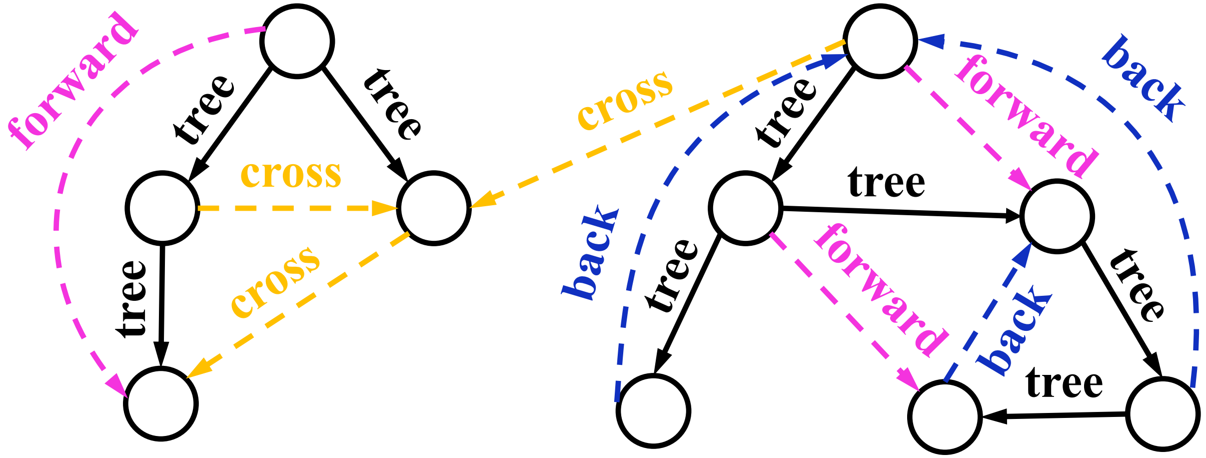






Traversal Arc Classifications

- The arc is called a tree arc if it belongs to one of the trees of F. If the arc is not a tree arc, there are three possibilities:
  - a forward arc if u is an ancestor of v in F,
  - a back arc if u is a descendant of v in F, and
  - a cross arc if neither u nor v is an ancestor of the other in F.

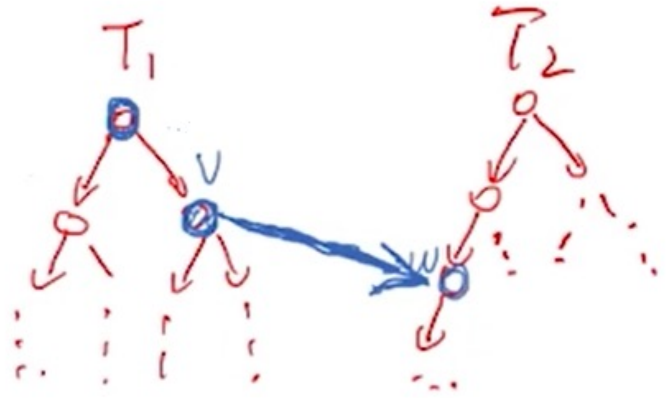


Facts about Traversal Trees

- **Theorem**: Suppose we run algorithm 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.

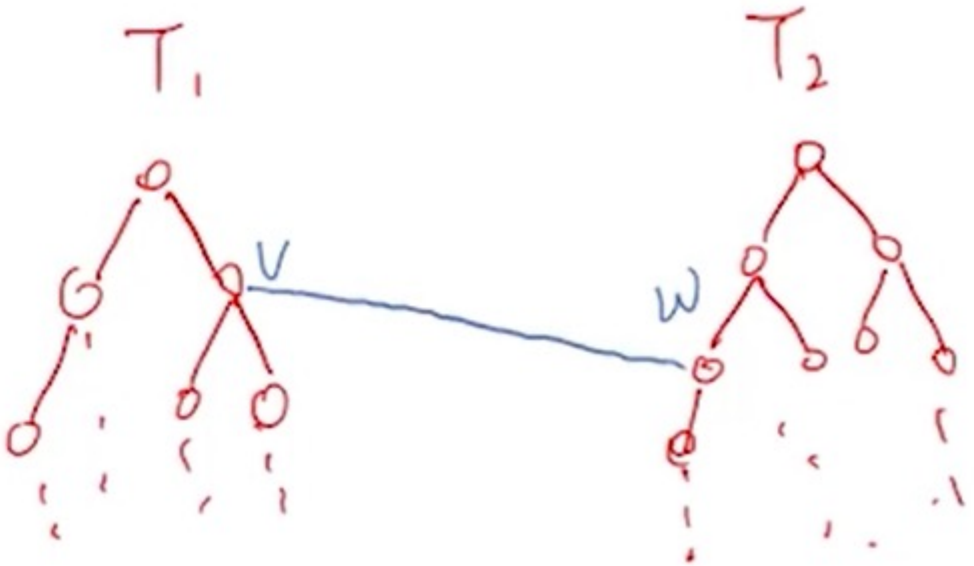
- **Proof**:
  - Assume $(v, w) \in E(G)$, $v \in T\_1, w \in T\_2$
  - VISIT(s) $\left\{ \begin{array}{l} \text{1. A single run of VISIT generates a tree} \\ \text{2. All nodes reachable from s will be visited} \end{array} \right.$
  - With 1 and 2, we have $w \in T\_1 \Rightarrow$ contradiction



Facts about Traversal Trees (Contd.)

- **Theorem**: Suppose we run algorithm traverse on G, resulting in a search forest F. Let $v,w \in V(G)$.

Then there can be no edges joining different trees of F.



Facts about Traversal Trees (Contd.)

- **Theorem**: Suppose we run algorithm traverse on G, resulting in a search forest F. Let $v,w \in V(G)$.

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.

- **Proof**.
  - Let $v \in T$ and s be the root of T.
  - Because w is reachable from v, v is reachable from s, then w is reachable from s
  - Then, w should be in the same tree as v.

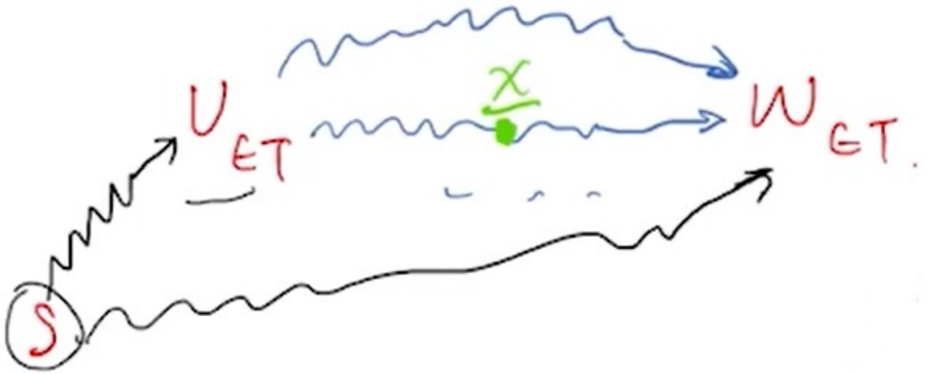
Facts about Traversal Trees (Contd.)

- **Theorem**: Suppose we run algorithm traverse on G, resulting in a search forest F. Let $v,w \in V(G)$.

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.

- **Proof.** For any node x in any path from v to w

1. $\mathrm{v}, \mathrm{w} \in T$
2. v is reachable from s the root of T, then x is reachable from s
3. By 2, $x \in T$



Complexity Analysis: General Graph Traversal

Algorithm 2 Traverse.

function TRAVERSE(digraph G)

  array $color[0..n-1]$

  array $pred[0..n-1]$

  for $u \in V(G)$ **do**

    $color[u] \leftarrow$ WHITE

  end for

  for $s \in V(G)$ **do**

    if $color[s] =$ WHITE **then**

      VISIT(s)

  return $pred$

Complexity Analysis: Visit(s)

Algorithm 1 Visit.

function VISIT(node s of digraph G)

    $color[s] \leftarrow$ Grey

    $pred[s] \leftarrow$ Null

    while there is a Grey node **do**

        choose a Grey node u

        if u has a WHITE (out-)neighbour **then**

            choose such a white (out-)neighbour v

            $color[v] \leftarrow$ Grey

            $pred[v] \leftarrow u$

        else

            $color[u] \leftarrow$ Black

Runtime Analysis of Traverse

- The initialization of the 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. In each loop:
  - The time taken in choosing grey nodes is $\Theta(1)$ each time.
  - 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.
    - If adjacency matrix is used, we need to scan the whole row, which takes $\Theta(n)$
    - If adjacency lists are used, we only need $\Theta(|L_i|)$ for finding white nodes in the adjacency list of node i.

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

Runtime Analysis of Traverse (Contd.)

- So, for simple selection rules and assuming a sparse input digraph, the adjacency list format seems preferable.

- If more complex selection rules are used, for example, rules that choose a single grey node $\Theta(n)$ time by scanning the whole list of grey nodes, then the running time is asymptotically $\Theta(n$^2) regardless of the data structure.

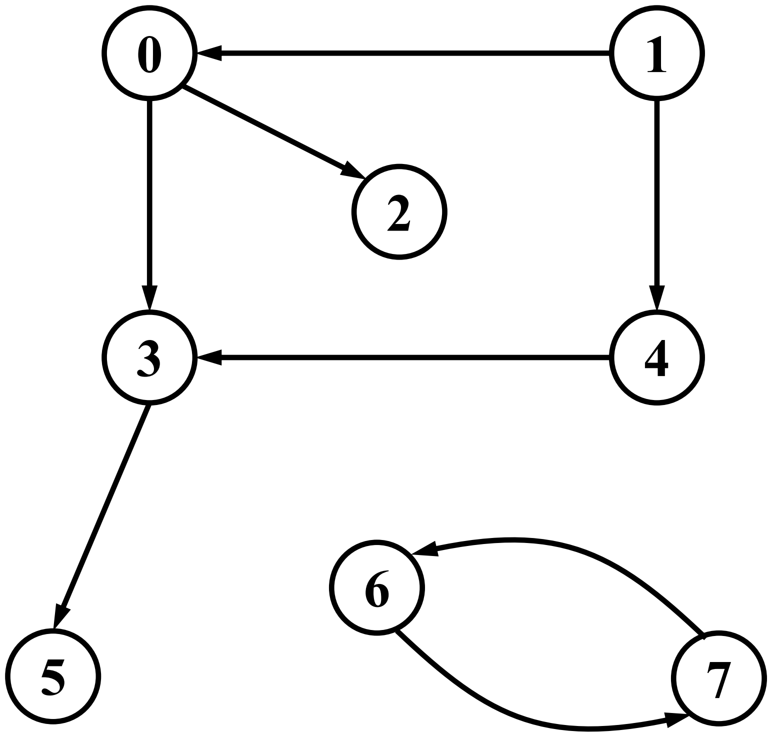
Graph Traversal

Algorithm 1 Visit.

function VISIT(node s of digraph G)
    $color[s] \leftarrow$ Grey
    $pred[s] \leftarrow$ Null
    while there is a Grey node **do**
        choose a Grey node u
        if u has a WHITE (out-)neighbour **then**
            choose such a (out-)neighbour v
            $color[v] \leftarrow$ Grey
            $pred[v] \leftarrow u$
        else
            $color[u] \leftarrow$ Black

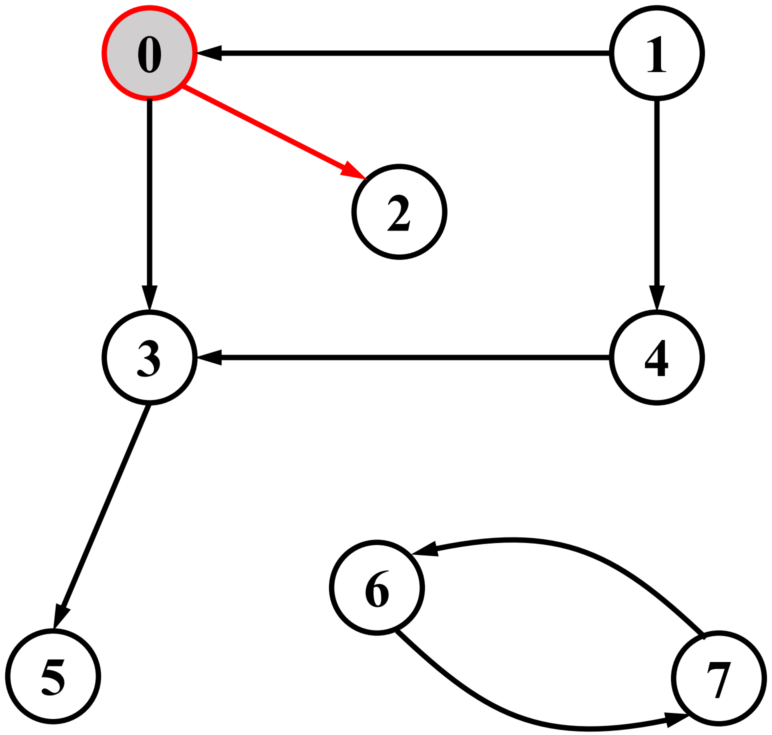
how to choose?

General graph traversal: example



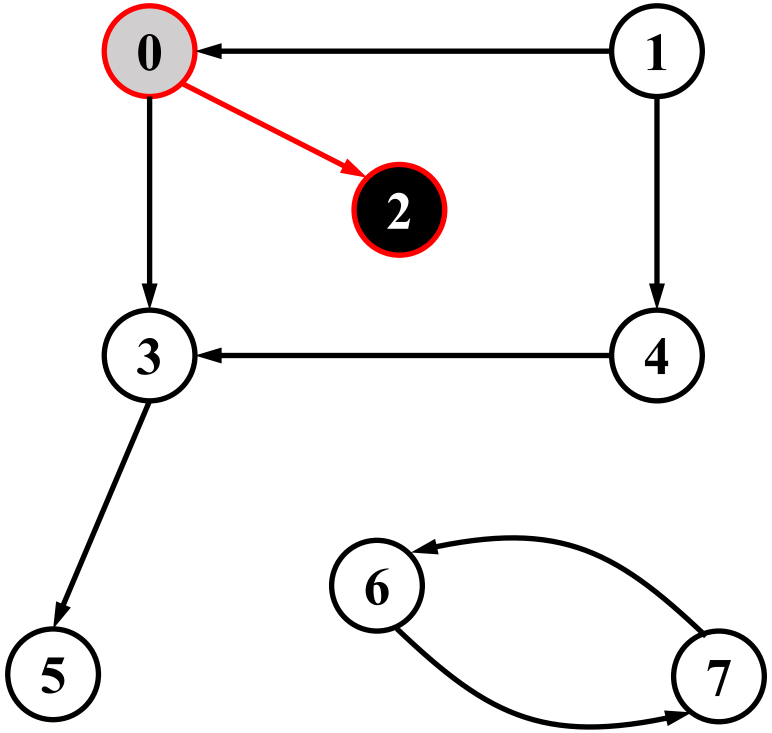
- Emmmm……

General graph traversal: example



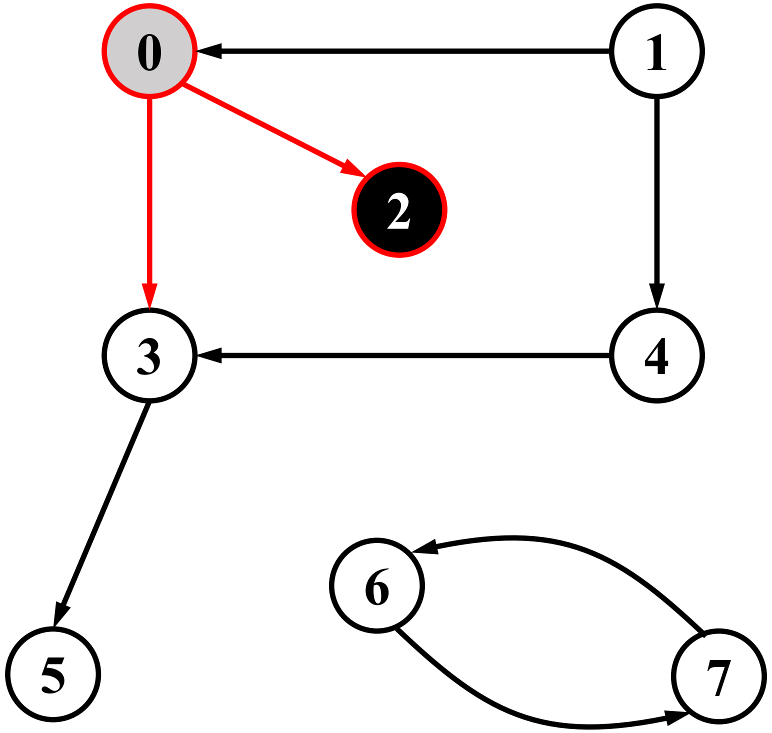
- Emmmm……

General graph traversal: example



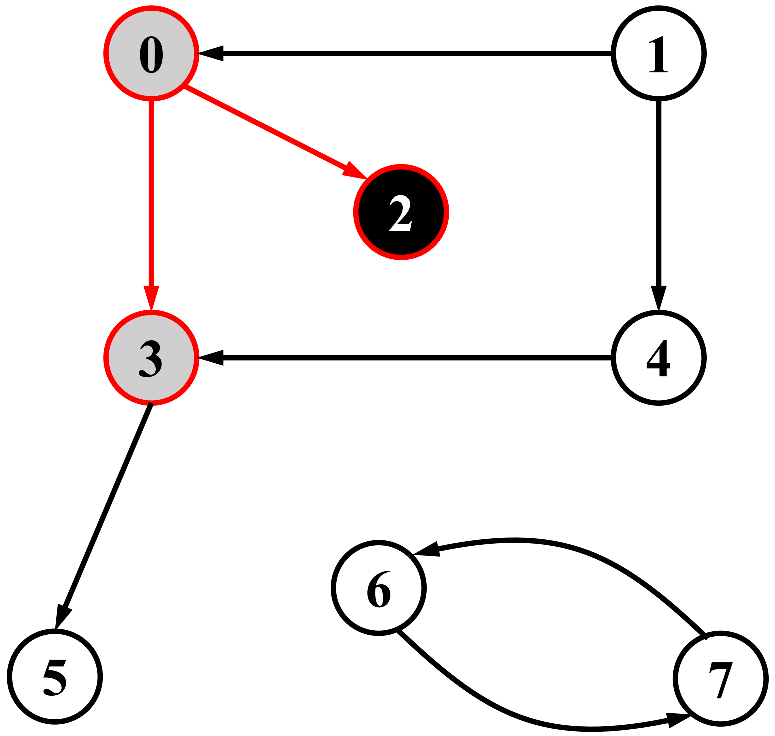
- Emmmm……

General graph traversal: example



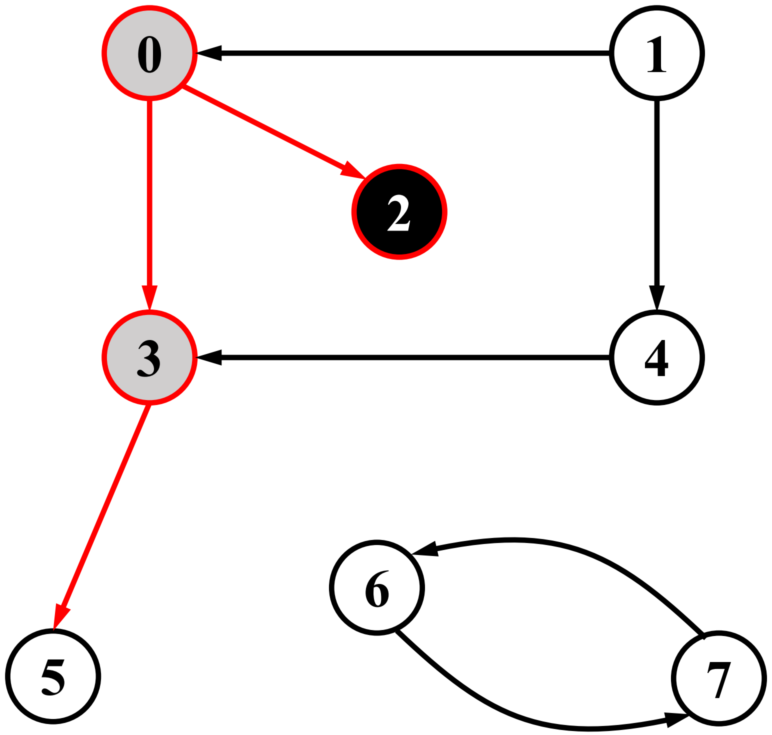
- Emmmm……

General graph traversal: example



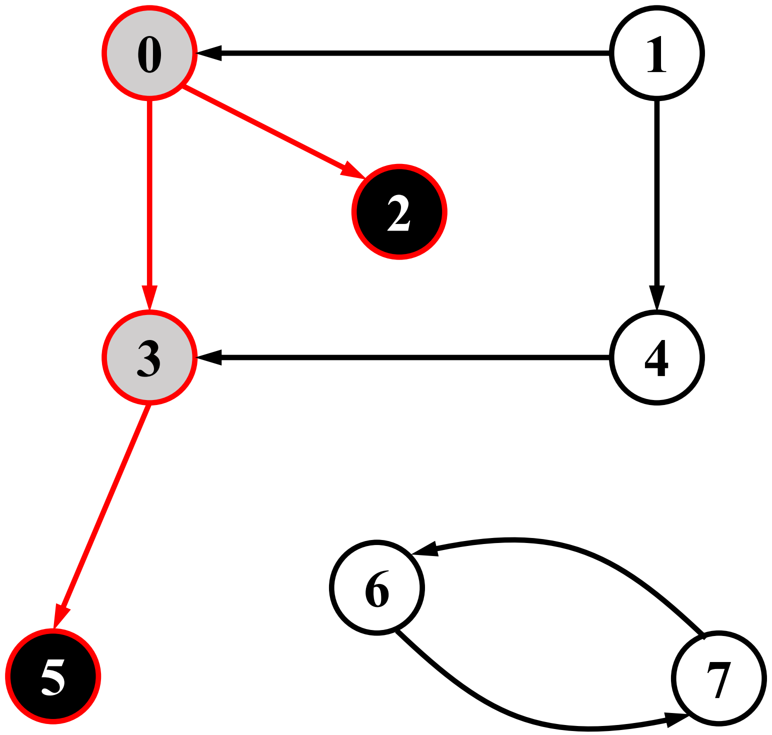
- Emmmm……

General graph traversal: example



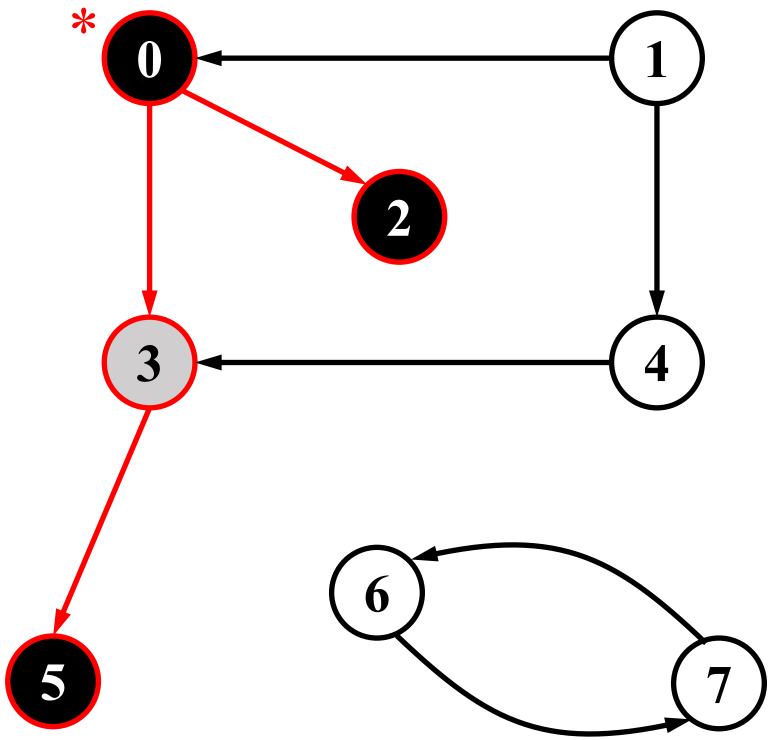
- Emmmm……

General graph traversal: example



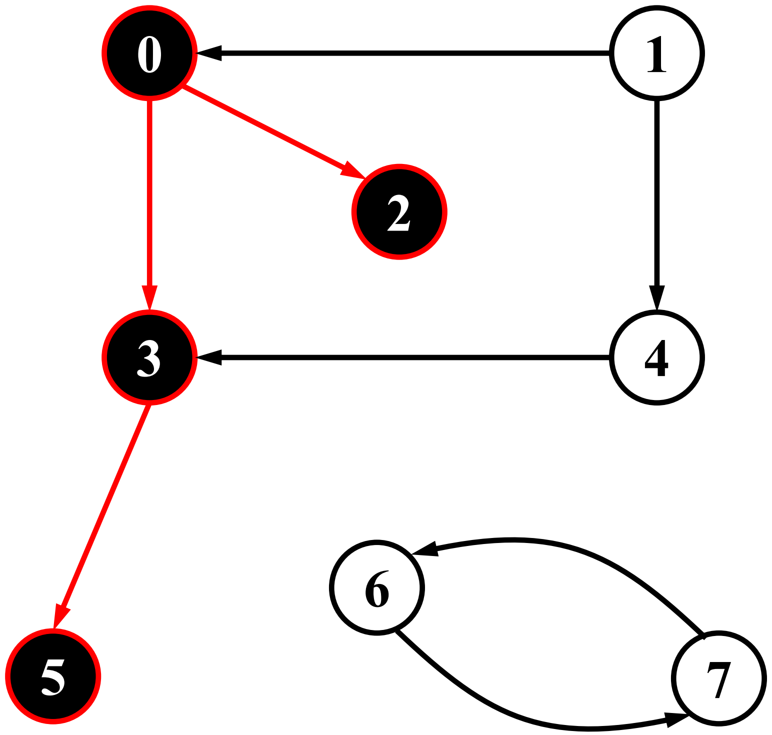
- Emmmm……

General graph traversal: example



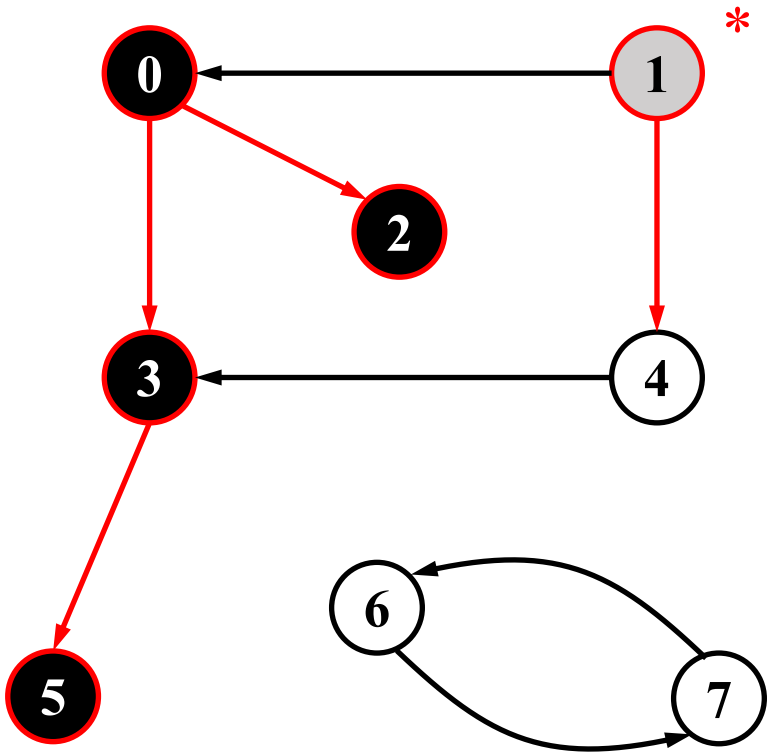
- Emmmm……

General graph traversal: example



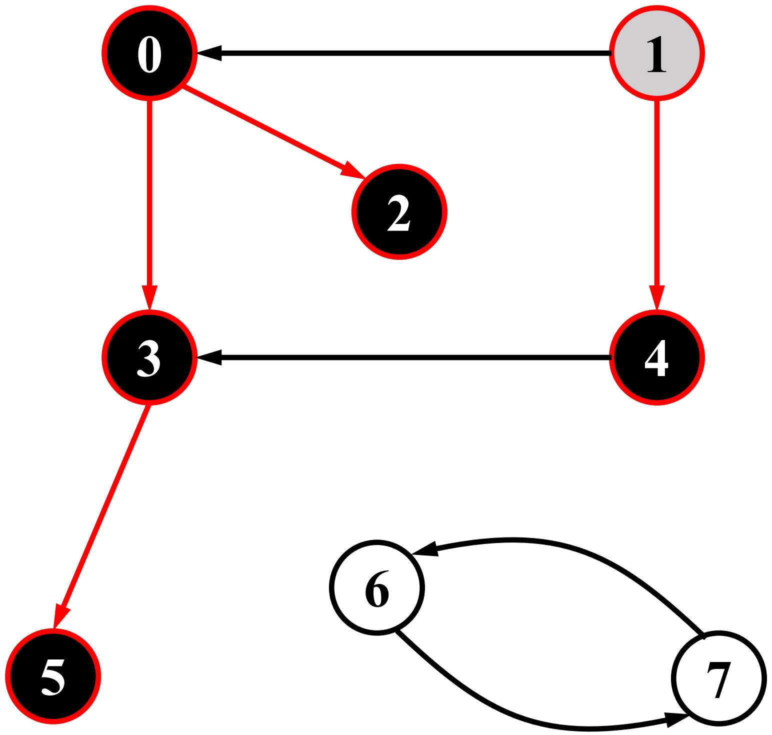
- Emmmm……

General graph traversal: example



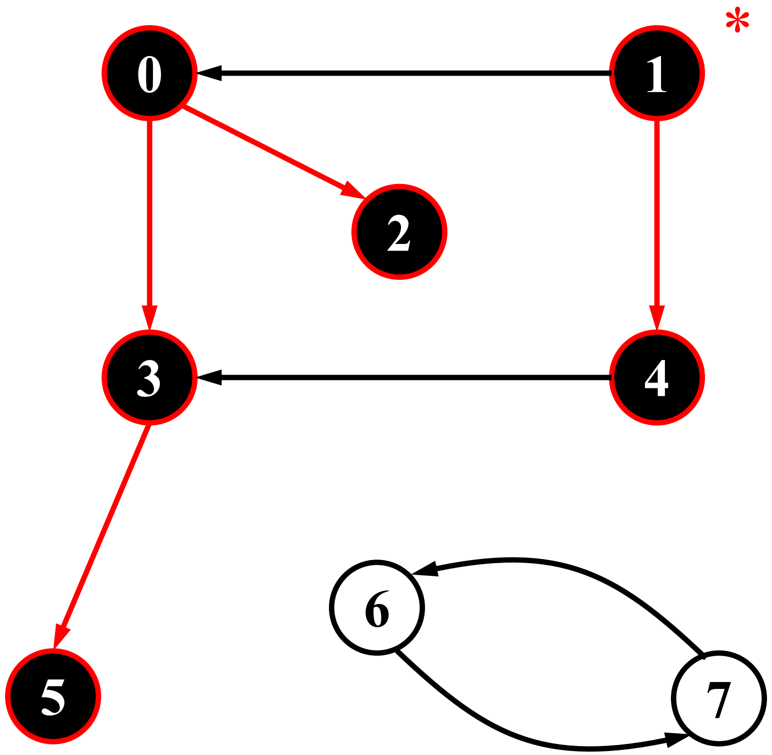
- Emmmm……

General graph traversal: example



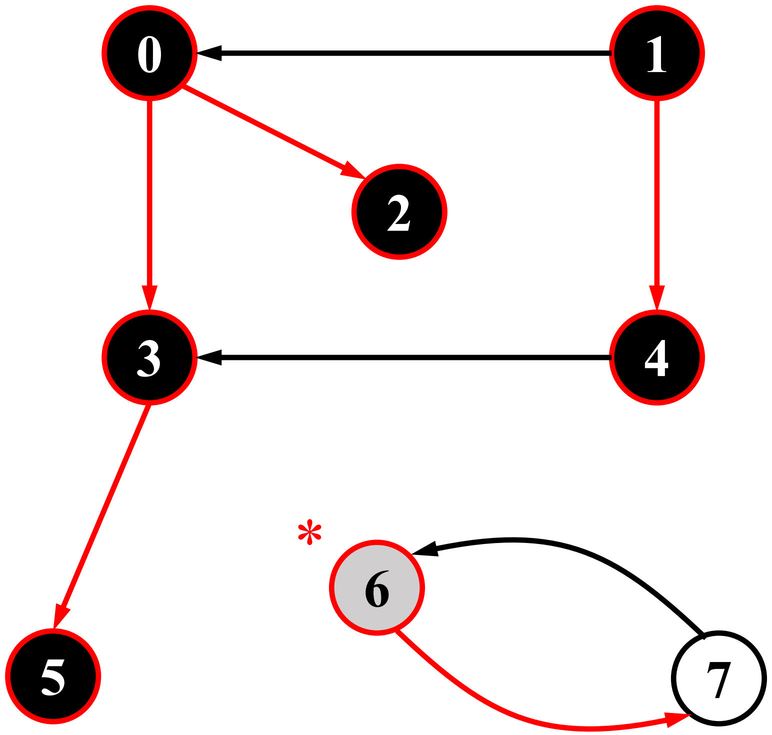
- Emmmm……

General graph traversal: example



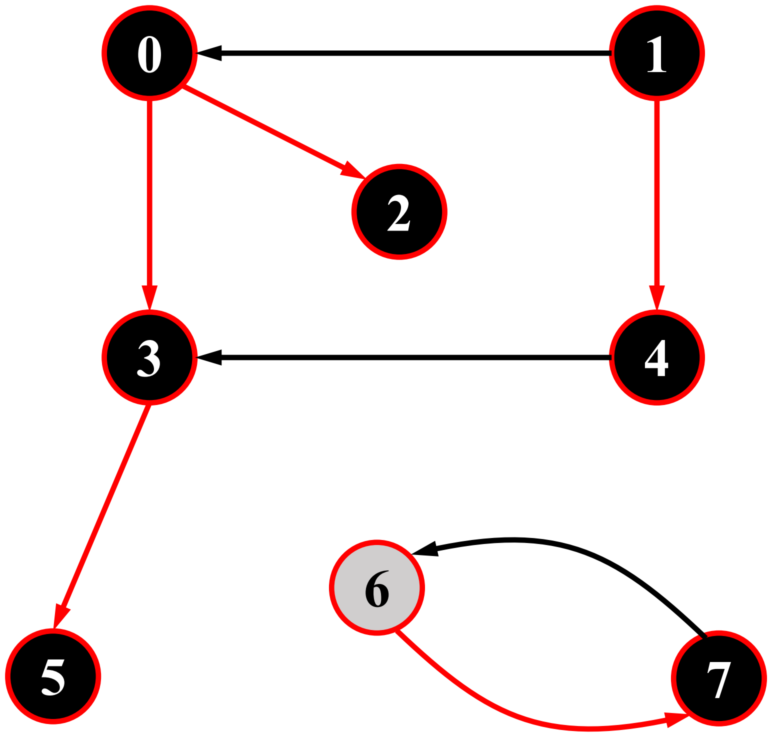
- Emmmm……

General graph traversal: example



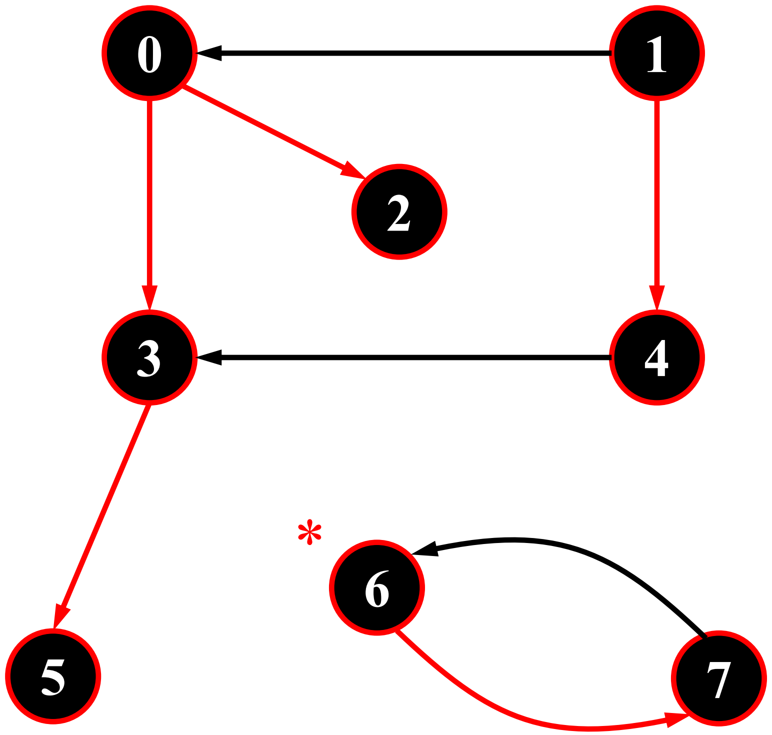
- Emmmm……

General graph traversal: example



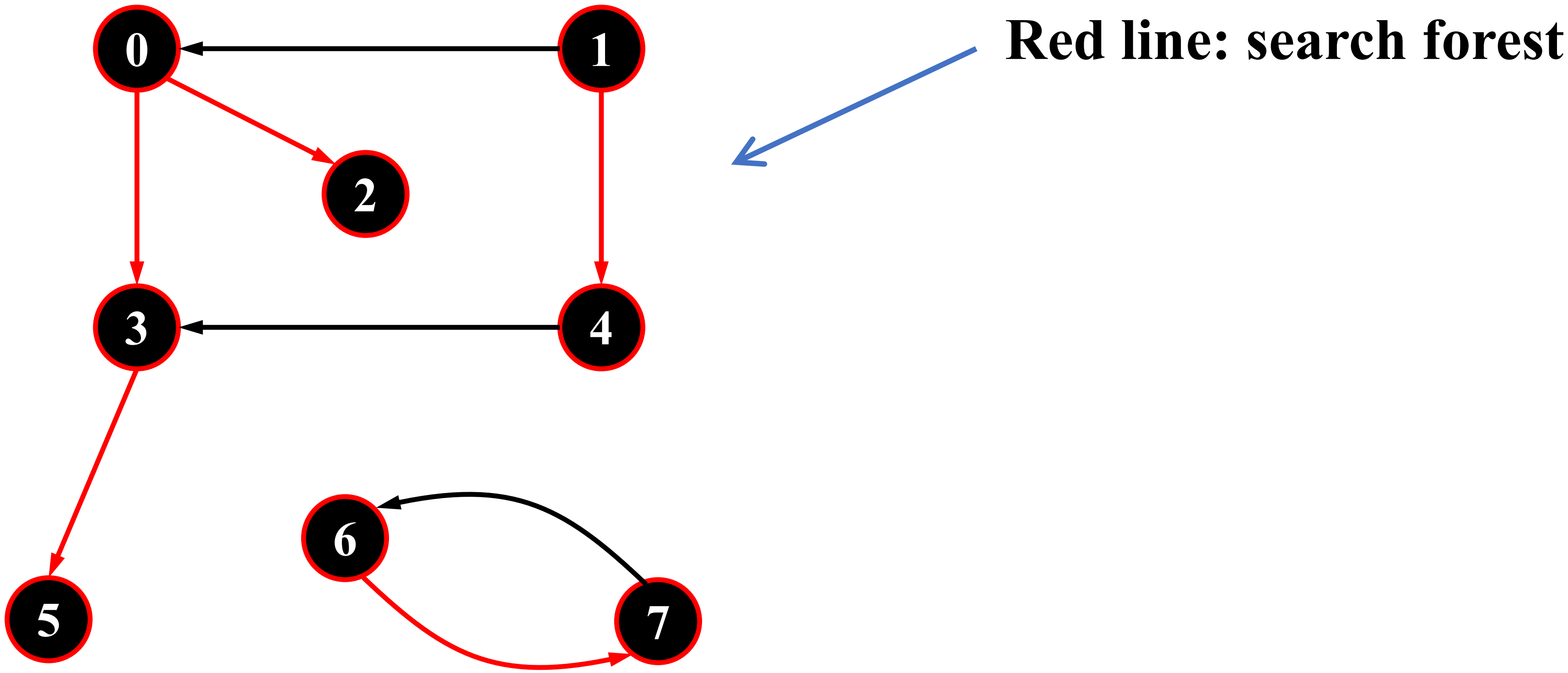
- Emmmm……

General graph traversal: example



- Emmmm……

General graph traversal: example



SUMMARY

- Graph Traversal Algorithm
- Facts about Traversal Trees
- Complexity Analysis
- Illustrative Example