COMP 250 - Lecture 32 : Graph traversal

→ Graph $G = (V, E)$

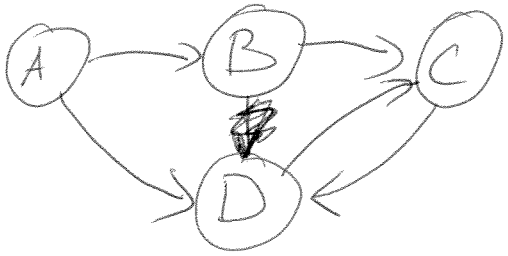
V: collections → E: for each vertex, collection of edges starting at the vertex

(also called adjacency list representation)

→ Adjacency matrix : mark edges by 1s

Are two vertices on the same path : reachability problem

can I reach B from A



$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} =$$

Path $A \longrightarrow C$

$A \xrightarrow{?} C$

?: edge

Paths of length 2

what paths of length 2 are there from A to another node?

$$= \begin{array}{c} \\ A \\ \\ \\ D \end{array}\begin{bmatrix} 0 & 0 & 1 & 0 \\ & & & \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Cycle: path that comes back to the same node

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 2 & 0 \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

2 paths of length 2 $A \to C$

$\sum_i a_{ij} a_{jk}$

Adjacency lists: is there a path => traversal

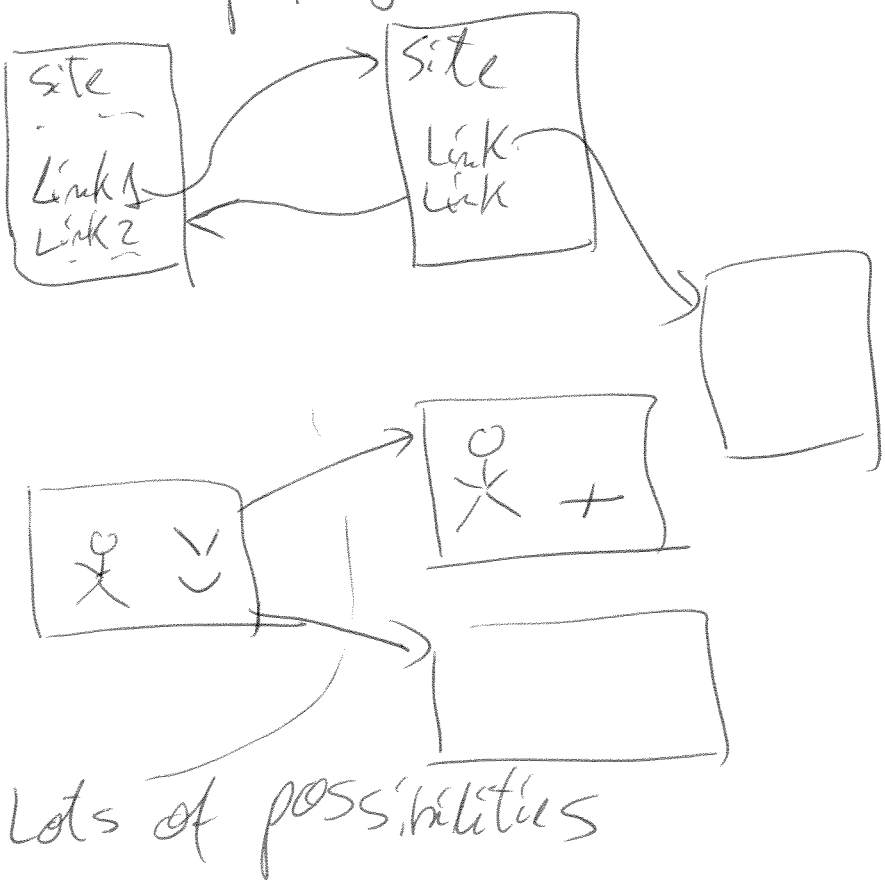
n vertices → n^2 data structure + matrix mult.

E.g if matrix is sparse

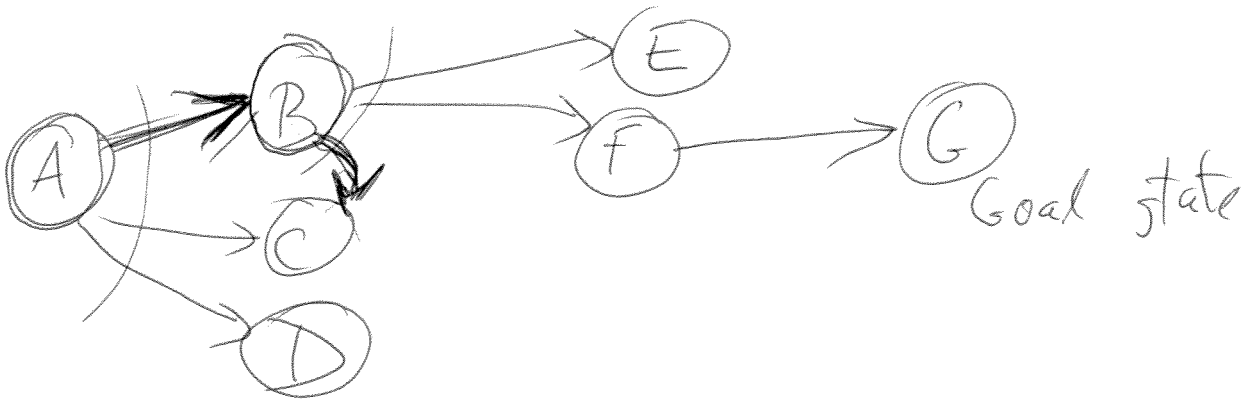
Graph traversal (possibly for large graphs)

↳ Web crawling

↳ Game playing search



Lots of possibilities



Pick <u>one</u> successor node, go there

Depth-first traversal; may get into dead ends; may need to "backtrack" to a previous state.

<u>Mark</u> where we've been

At a node/vertex → getNeighbors()

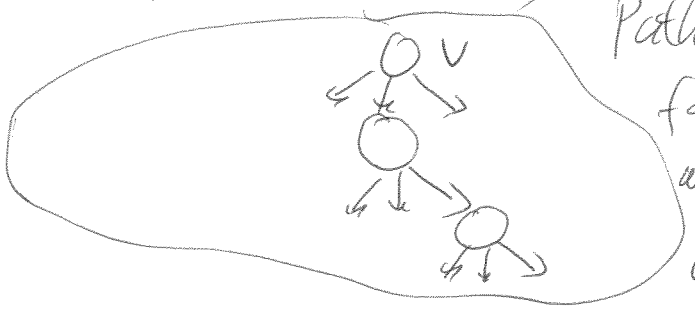
<u>Algorithm</u> DFS(G, v) start position

Initially all vertices v.visited = false

<u>Output</u>: Visit each vertex (at least) <u>once</u>

```
v.Process(); //eg print...
  v.setVisited(true); // each node-a boolean var to
                      mark if it has been seen
for u ∈ v.getNeighbors()
    if (!u.isVisited()) DFS(G, u)
```

Memory requirement

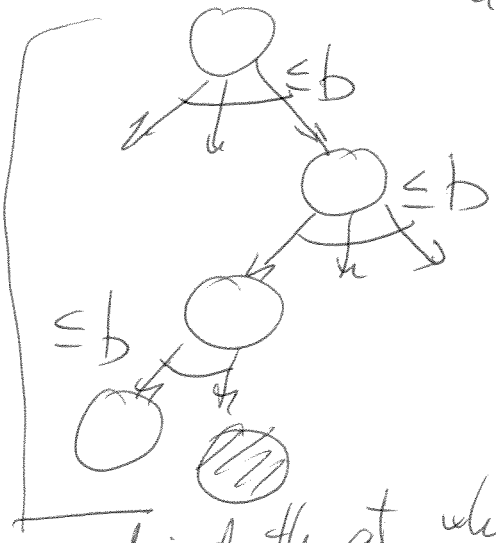


Path so far + alternatives on stack

Find some specific state (not see entire graph)

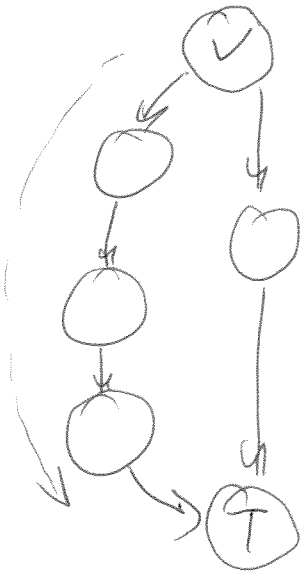
↳ Length of the path to the solution state

↳ Branching factor: how many successors can a node have?



d: depth at which sol is found

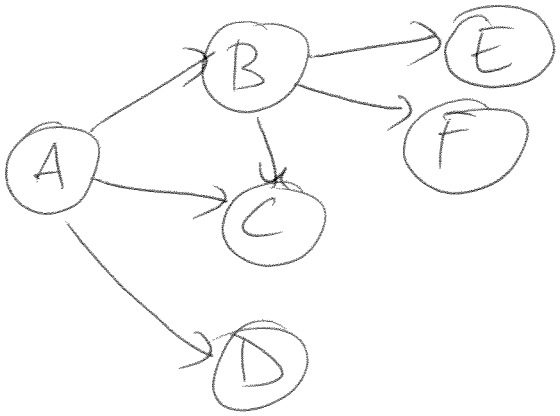
$\underbrace{b + b + b \cdots b}_{d} = b \cdot d$ Linear in b, d

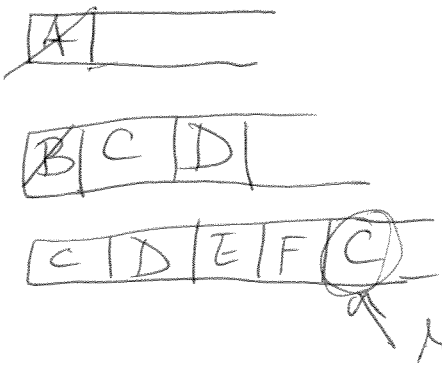


Find a path that is not of minimum length (aka suboptimal)

T: target state

Breadth-first search: not exploration, but optimization





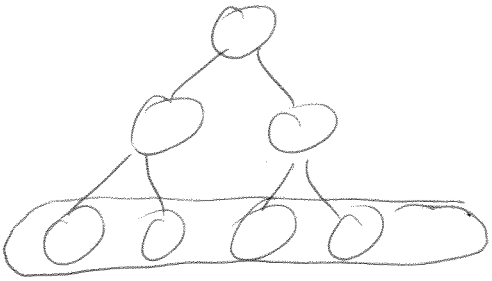
readable through a longer path

A → all successors → all their successors → ...

Frontier: nodes currently under consideration
(always look at all nodes that are k steps from A, before those at (k+1 steps)

Algorithm Iterative BFS (G, v)
// v = initial node
// Priority queue: enqueue nodes in order of distance from A; whenever we reach a node, we enqueue its successors

Queue (worst-case) may need to hold all nodes in graph
(see e.g. full trees; queue would need to hold last layer of nodes)



```
q ← new Queue()
v.setVisited(true)
q.enqueue(v)
while (!q.isEmpty())
    u ← q.dequeue()
    u.Process()
    for each s ∈ u.getNeighbors()
        q.enqueue(s) ← may want to check visited first
```