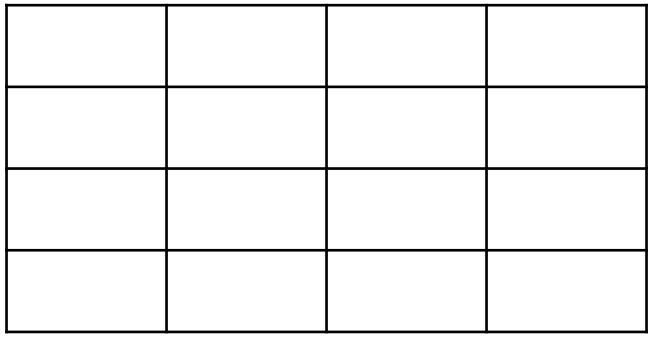
ECE 479/579: Principles of Artificial Intelligence

2.8.2022

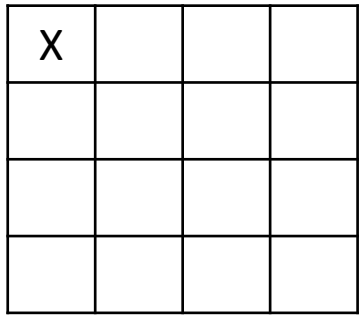
Backtracking: let's review the effectiveness of good control strategy

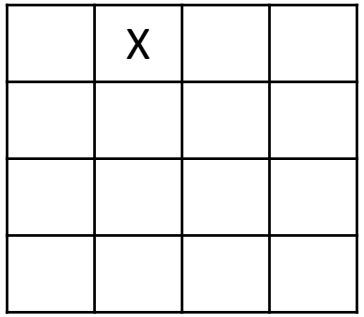
4-queens problem



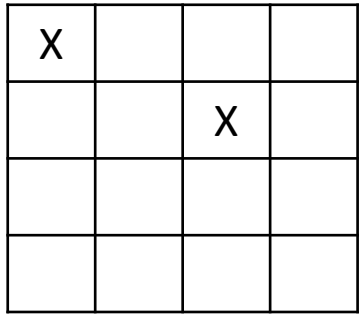
- Consider two approaches
  1) Just use basic rules: R_{ij}
     - $[R_{11} \quad R_{12} \quad R_{13} \quad R_{14}]$
  2) Use R_{ij} with rule ordering based on $diag(i,j)$
     - $[R_{12} \quad R_{13} \quad R_{11} \quad R_{14}]$

Q: How often will backtracking occur for both approaches?

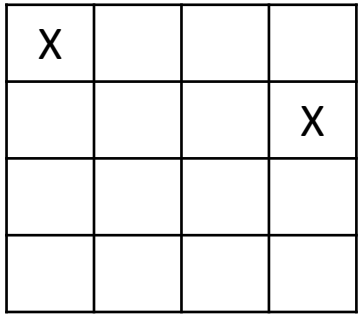




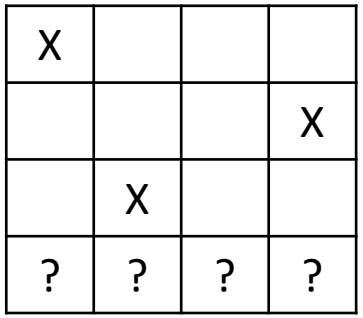
# of backtracking = 15 + 2



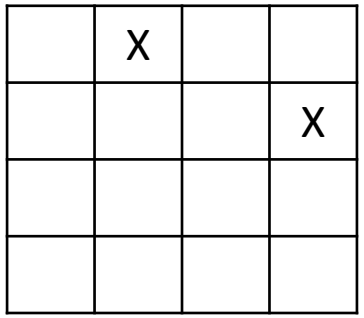
# of backtracking = 2



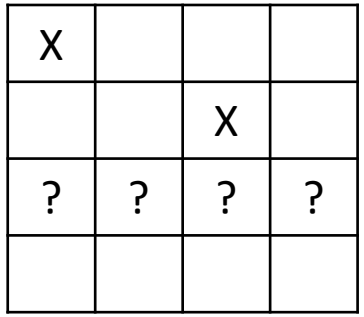
# of backtracking = 6 + 2



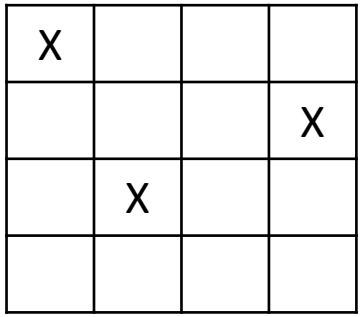
# of backtracking = 9 + 4



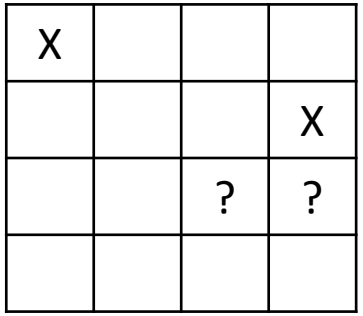
# of backtracking = 17 + 3



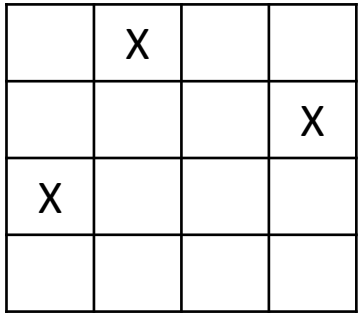
# of backtracking = 2 + 4

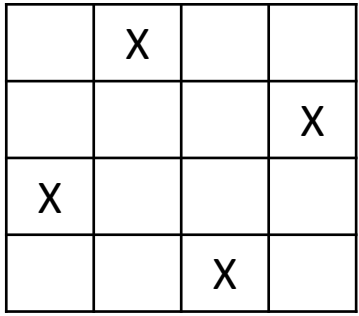


# of backtracking = 8 + 1

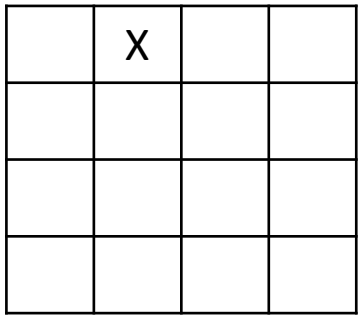


# of backtracking =13 + 2

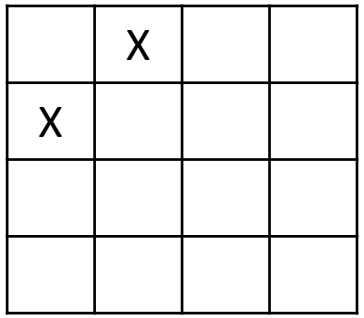




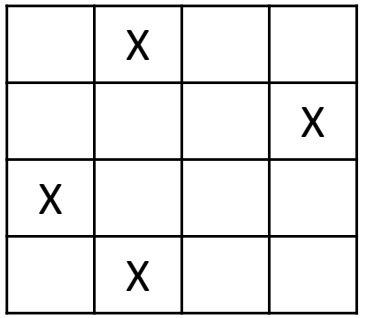
# of backtracking = 20 + 2



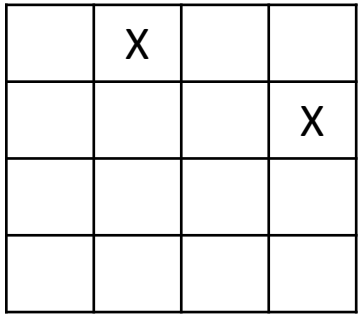
$[R_{12} \quad R_{13} \quad R_{11} \quad R_{14}]$



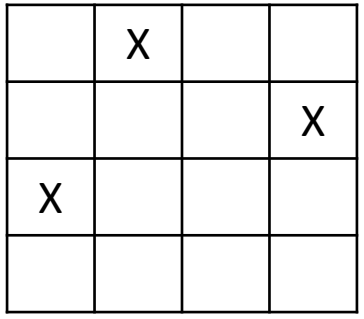
$[R_{12} \quad R_{13} \quad R_{11} \quad R_{14}]$
$[R_{21} \quad R_{24} \quad R_{22} \quad R_{23}]$



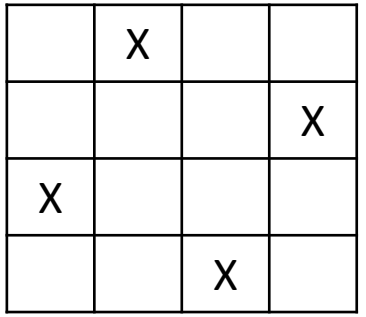
$[R_{12} \quad R_{13} \quad R_{11} \quad R_{14}]$
$[R_{21} \quad R_{24} \quad R_{22} \quad R_{23}]$
$[R_{31} \quad R_{34} \quad R_{32} \quad R_{33}]$
$[R_{42} \quad R_{43} \quad R_{41} \quad R_{44}]$



$[R_{12} \quad R_{13} \quad R_{11} \quad R_{14}]$
$[R_{21} \quad R_{24} \quad R_{22} \quad R_{23}]$



$[R_{12} \quad R_{13} \quad R_{11} \quad R_{14}]$
$[R_{21} \quad R_{24} \quad R_{22} \quad R_{23}]$
$[R_{31} \quad R_{34} \quad R_{32} \quad R_{33}]$

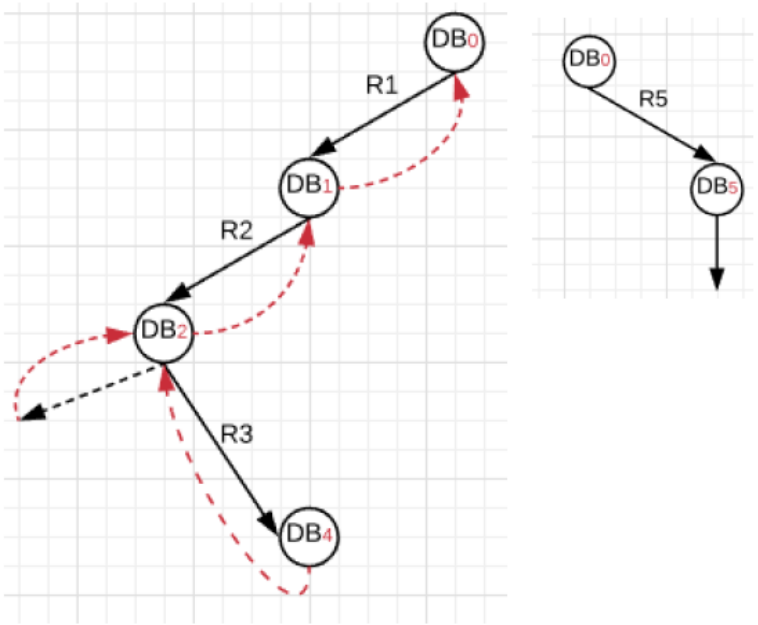


$[R_{12} \quad R_{13} \quad R_{11} \quad R_{14}]$
$[R_{21} \quad R_{24} \quad R_{22} \quad R_{23}]$
$[R_{31} \quad R_{34} \quad R_{32} \quad R_{33}]$
$[R_{42} \quad R_{43} \quad R_{41} \quad R_{44}]$

- The overall computational cost of an AI production system is the combined rule application cost and control strategy cost.
- Part of the art of designing efficient AI systems is deciding how to balance these two costs.

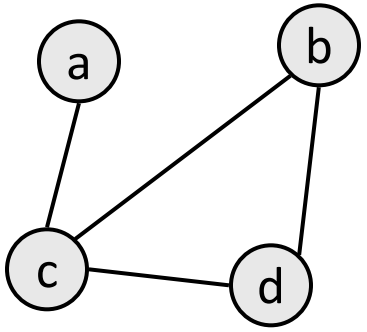
- The behavior of the control system as it makes rule selections can be regarded as a search process

  - Backtracking
    - The control system effectively forgets any trial paths that result in failures
      - cannot jump back to them to keep exploring them in a different way



  - **Graph search**

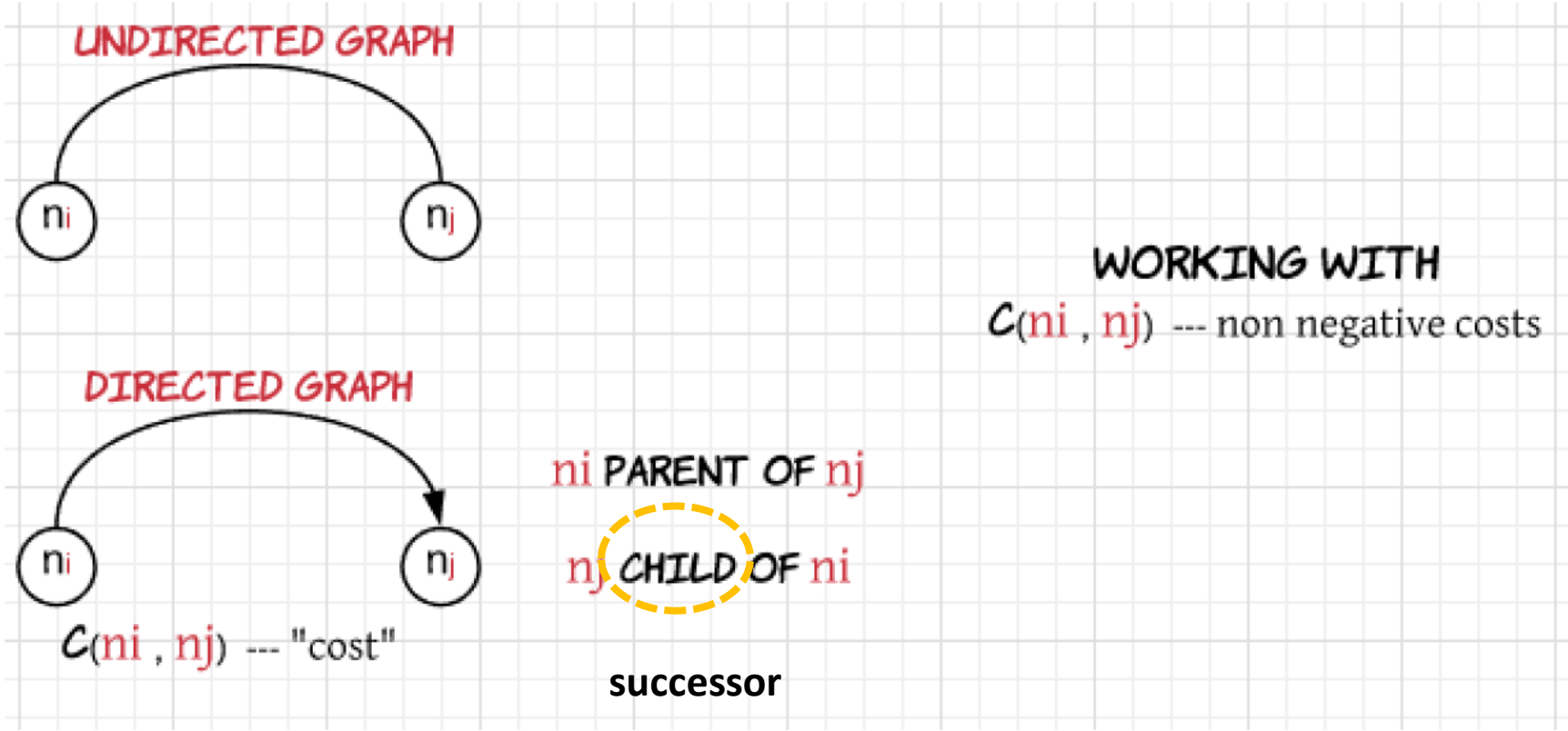
Graph notation



Graph (**G**): A data structure containing

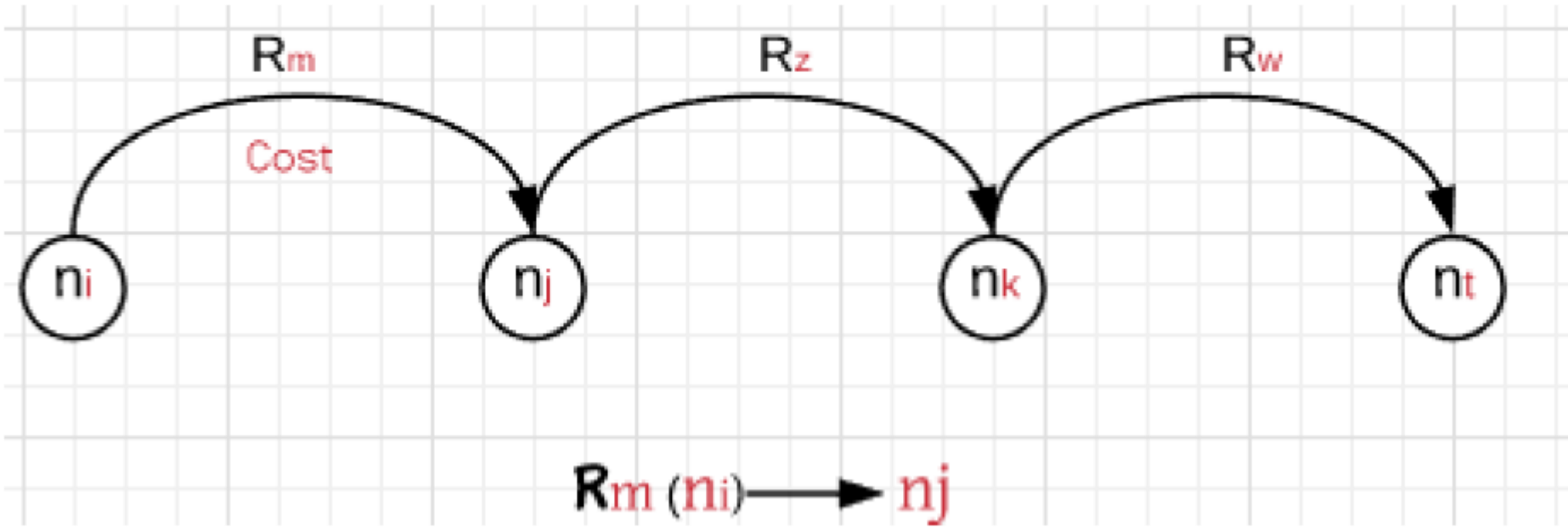
- A set of vertices (or nodes) (**V**)
- A set of edges (**E**) where an edge represents a connection between 2 vertices

$$G = \langle V, E \rangle$$



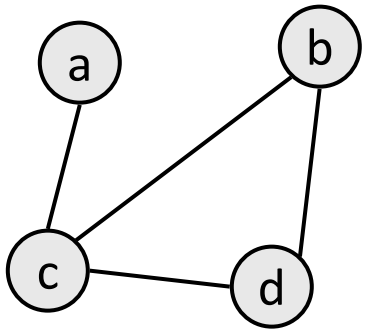
In the graphs that are of interest to us, a node can have only a finite number of successors (children)

- The nodes are labeled by databases
- The arcs are labeled by rules

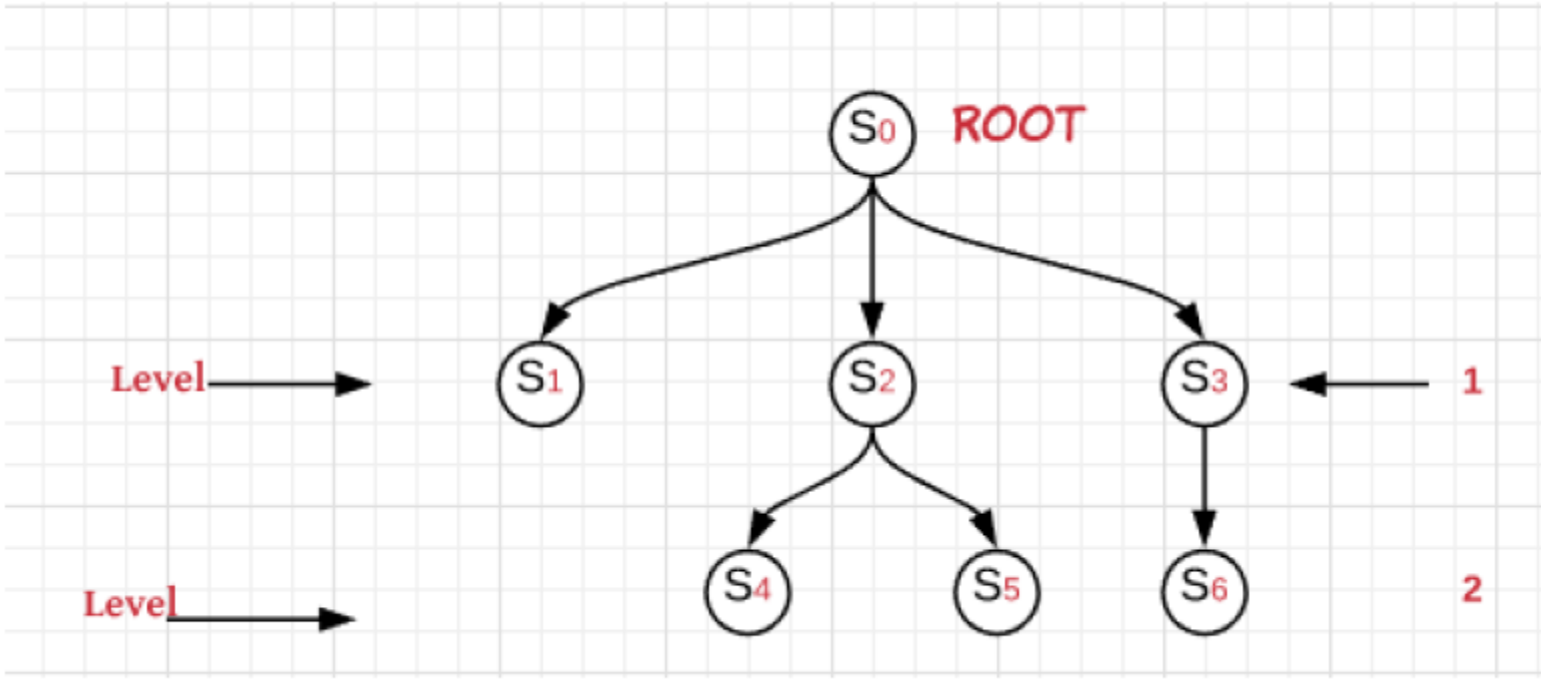


A **tree** is a special case of a graph in which each node has <u>at most one parent</u>

- A node in the tree having no parent is called a *root* node
- A node in the tree having no successors is called a *tip* (leaf) node



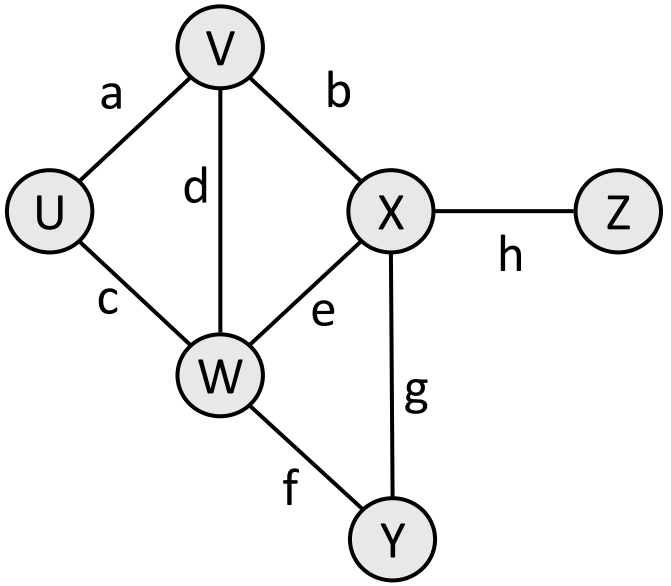
- The root node is of *depth* zero
- The depth of any other node in the tree is defined to be the depth of its parent plus 1



- Path: a path from vertex *a* to *b* is a sequence of edges that can be followed starting from *a* to reach *b*
  - *Example: a path from V to Z* – {V, X, Z} or {V, W, Y, X, Z}
- The **cost of a path** between two nodes is then the sum of the costs of all of the arcs connecting the nodes on the path. In some problems, we want to find that path having <u>minimal cost</u> between two nodes.

Searching for a path from one vertex to another

- Want any path (or want to know there is a path)
- Want to minimize a path length (i.e., # of edges)
- Want to minimize a path cost (i.e., sum of edge costs)



Procedure GRAPHSEARCH (Nils Nilsson)

1. Create a search graph, G, which consists only of the start node, s. s is placed on a list called OPEN.

2. Create a list called CLOSED. This list is initially empty.

3. LOOP: if Empty(OPEN), then exit with failure.

4. n <-- First(OPEN)
   OPEN = OPEN - {n}
   CLOSED = CLOSED + {n}

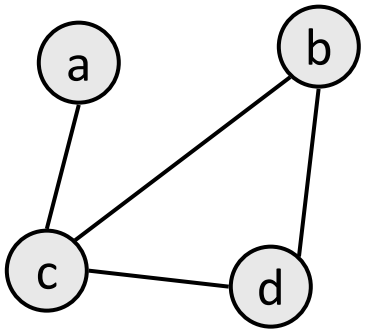
5. If Goal(n), exit successfully with the solution obtained by tracing a path along the pointers from n to s in G (pointers are established in step 7).

6. Expand node n, generating the set, M, of its successors and install them as successors of n in G

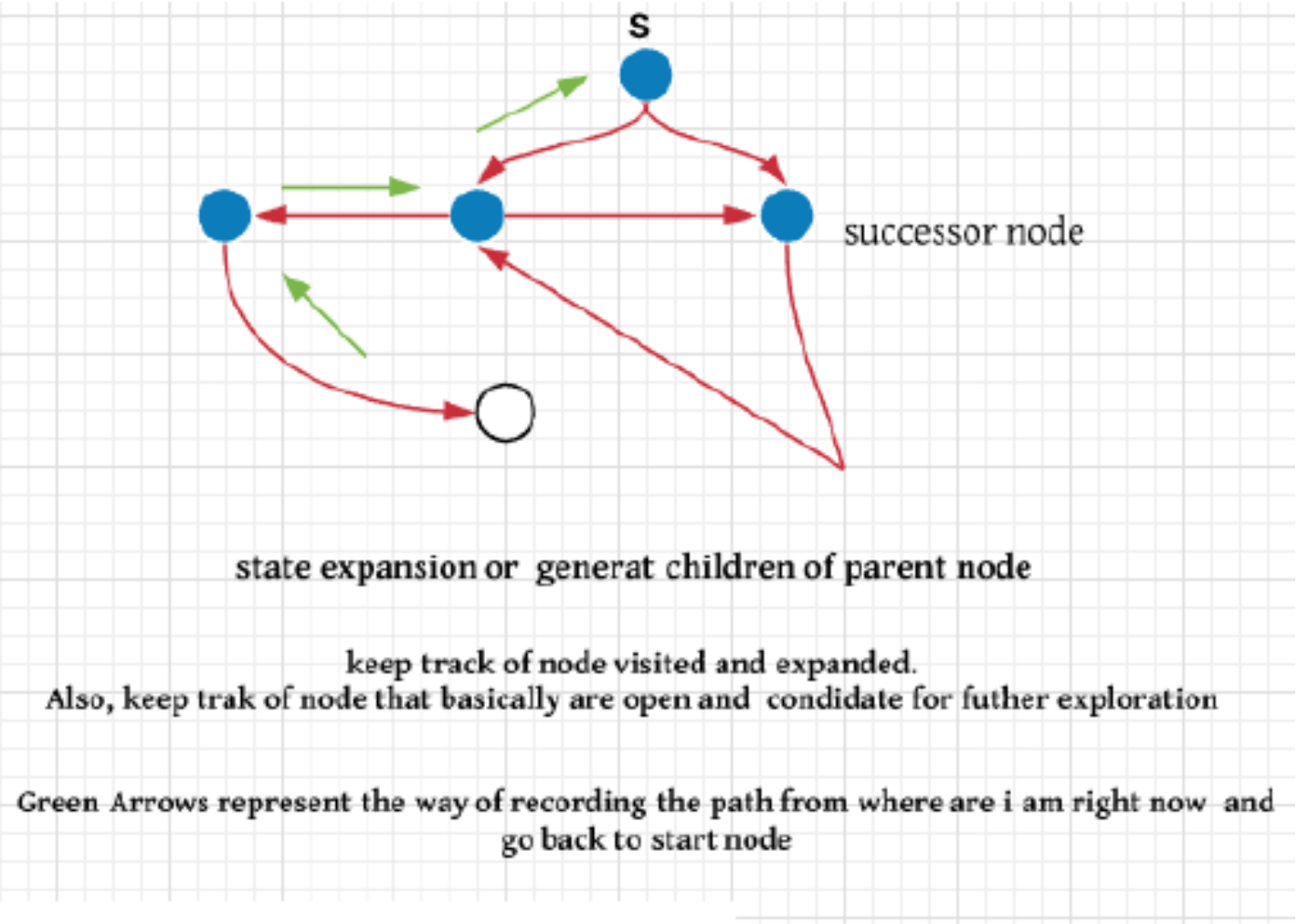
7. Establish a pointer to n from those members of M that were not already in G, i.e., on either CLOSED or OPEN. Add these members of M to OPEN.
   For each member of M that was already on OPEN or CLOSED, decide whether or not to redirect its pointer to n.
   For each member of M already on CLOSED, decide for each of its descendants in G whether or not to redirect its pointer.

8. Reorder the list OPEN, either according to an arbitrary scheme or according to heuristic merit.

9. End LOOP



It is a **general abstraction** for how to structure instances of specific algorithm that could search or build up and explicit portion of entire implicit search space as search graph and capture what would constitute solution from start node to goal node with some criteria of optimization and heuristic merit.



- The procedure generates an *explicit graph*, G, called the **search graph** and a subset, T, of G called the **search tree**.
- Each node (except s) in G has a pointer directed to just one of its parents in G, which defines its unique parent in T
- a single distinguished path to any node is defined by T
- the nodes on OPEN are those (tip) nodes of the search tree that have not yet been selected for expansion
- The nodes on CLOSED are either tip nodes selected for expansion that generated no successors in the search graph or non-tip nodes (i.e., already expanded) of the search tree.

We want to find that path having minimal cost between two nodes.

- When the search process generates a node that it had generated before, it finds *a (perhaps better) path to it other than the one already recorded in the search tree*.
- We desire that the search tree preserve the least costly path found so far from s to any of its nodes.

Assume edge cost of 1

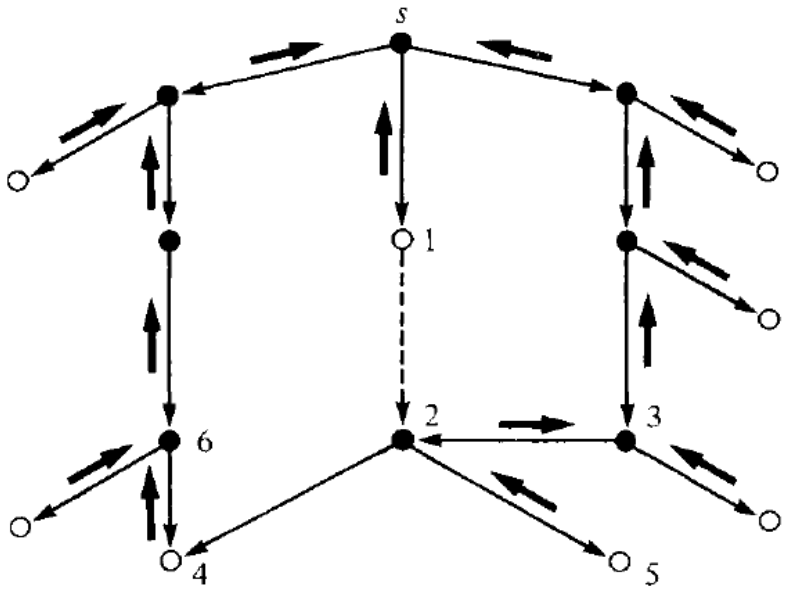


Fig. 2.4 A search graph and search tree before expanding node 1.

To arrive at node 2, path cost = 4

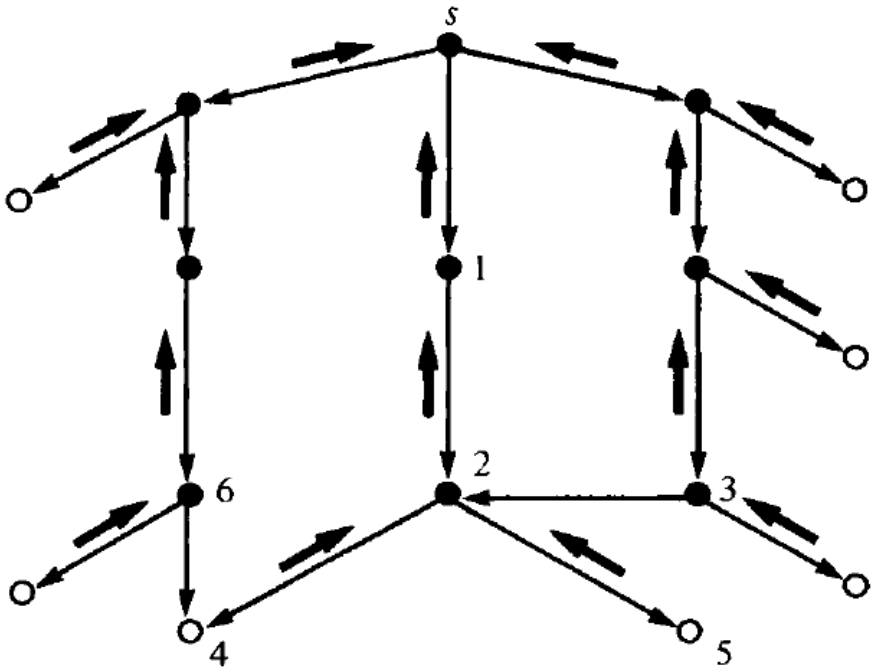
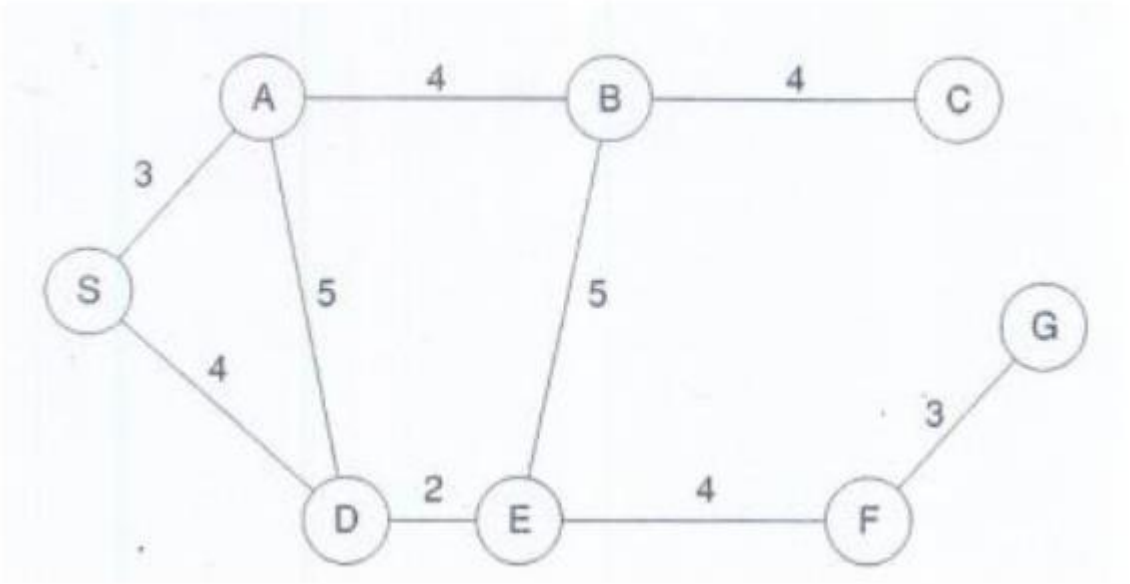


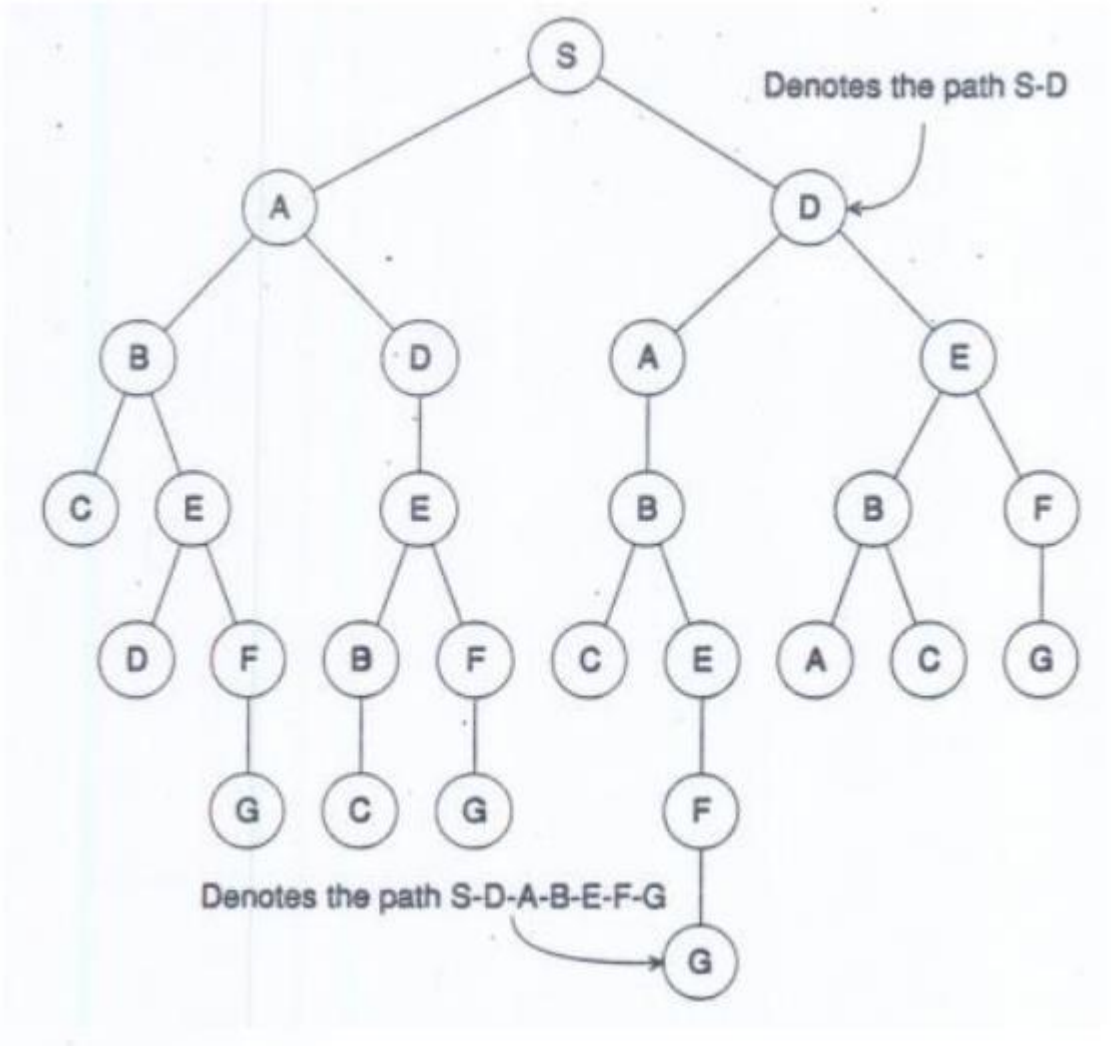
Fig. 2.5 A search graph and search tree after expanding node 1.

After expanding node 1, path cost = 2 to arrive at node 2

Basic search

- Find any path between S and G
- Complete explicit search tree generated from that graph connected between the cities
- Classical method called any path.
  - not imposing optimization criteria





Depth-first search (DFS)

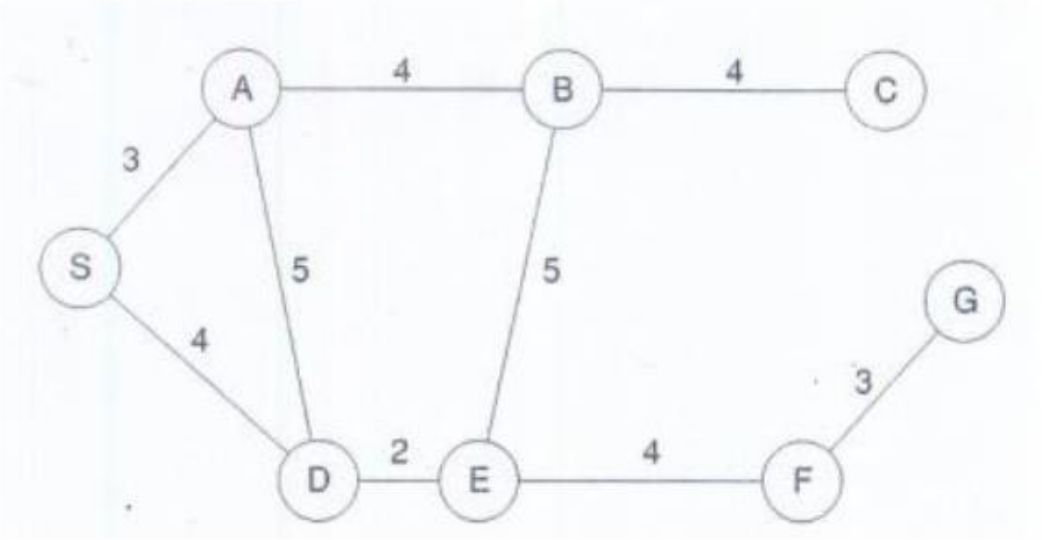
Finds a path between two vertices by exploring each possible path as far as possible before backtracking

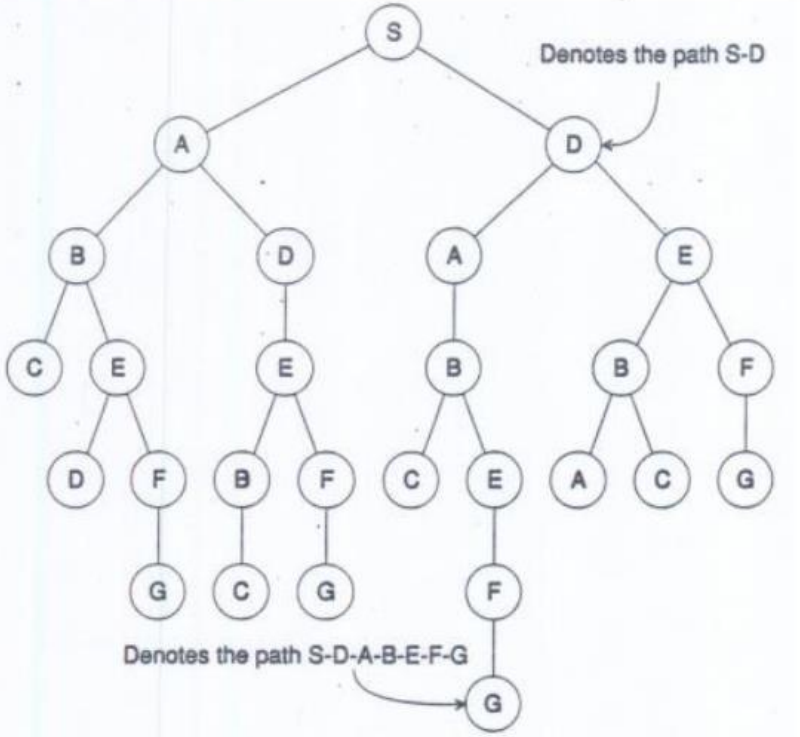
- Often implemented recursively
- The ordering the nodes on OPEN
  - The deepest node in the search tree is always selected for expansion
  - Using a LIFO queue, **new paths go to the front of the queue**, and **new paths get expanded first**

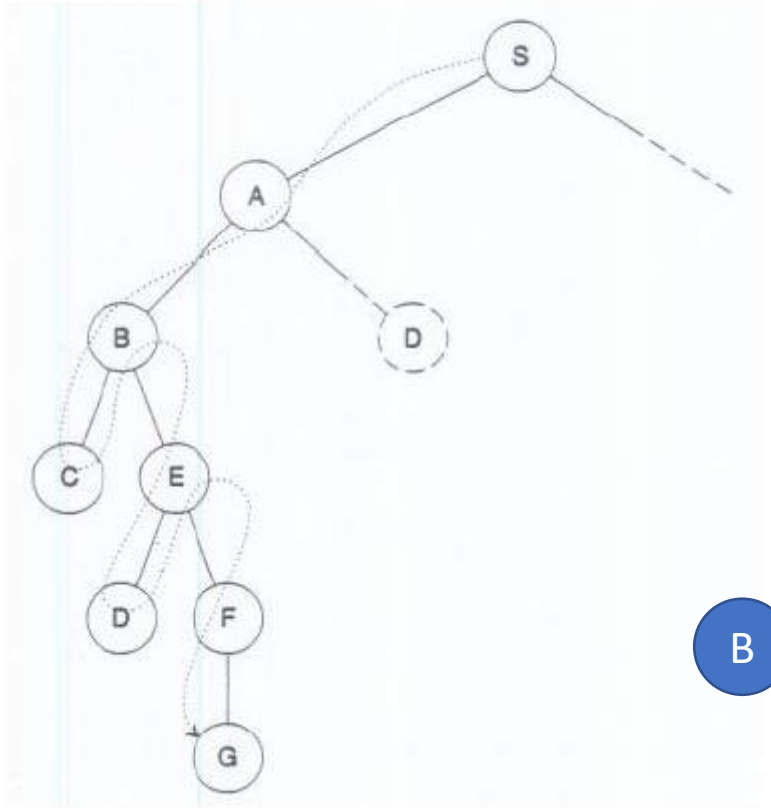
To conduct a depth-first search,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - Reject all new paths with loops.
  - Add the new paths, if any, to the *front* of the queue.
- If the goal node is found, announce success; otherwise, announce failure.

▷ Add the new paths, if any, to the *front* of the queue.



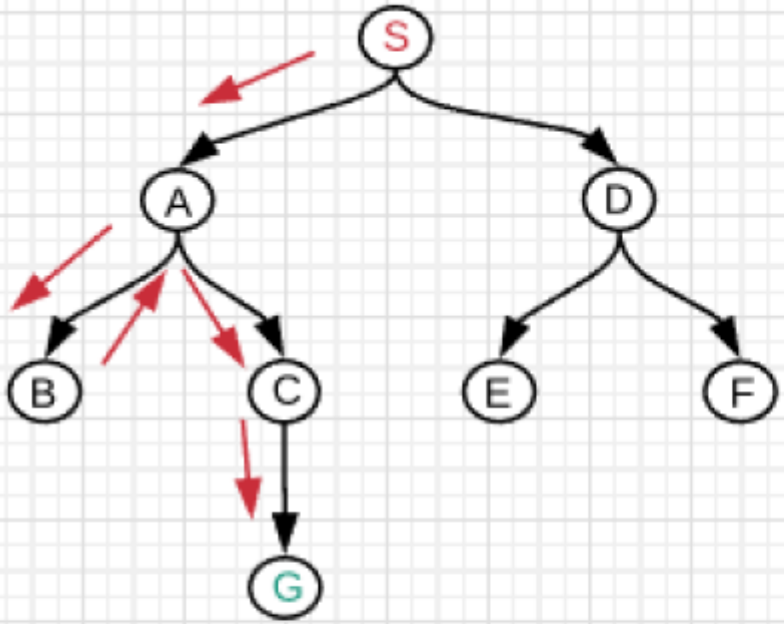




Q={S}

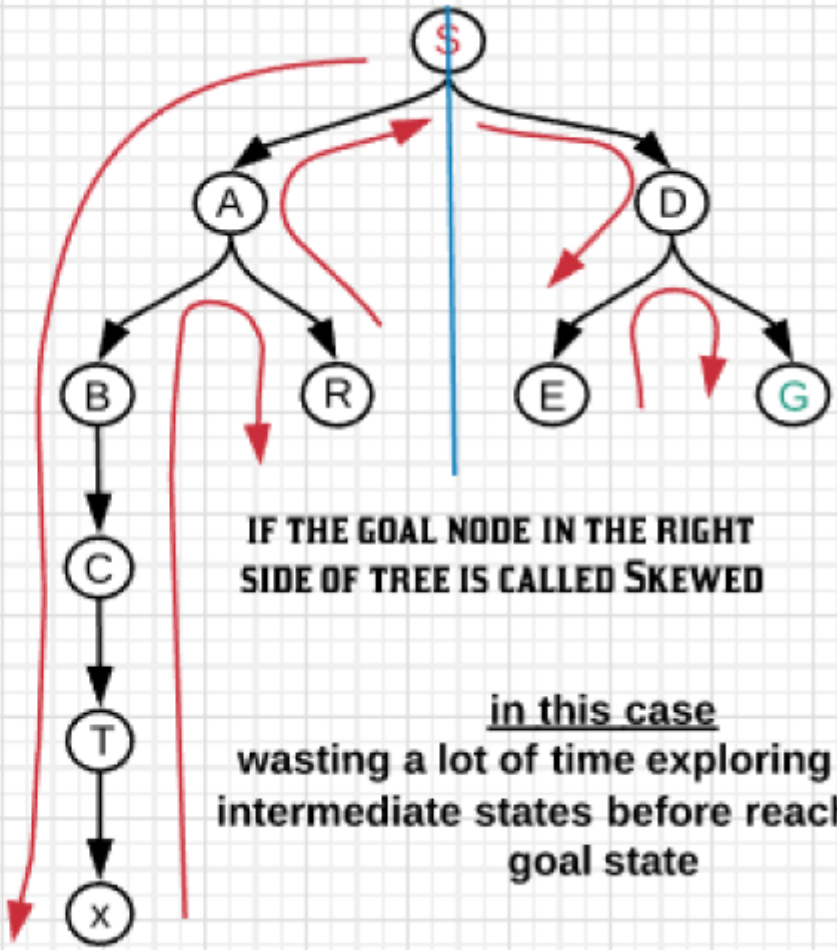
Q={SA, SD}

Q={SAB, SAD, SD}



Do guarantee to find the solution?

Yes



IF THE GOAL NODE IN THE RIGHT SIDE OF TREE IS CALLED SKEWED

in this case

wasting a lot of time exploring a lot of intermediate states before reaching the goal state

Breadth-first search (BFS)

Finds a path between two vertices by taking one step down all paths and then immediately backtracking

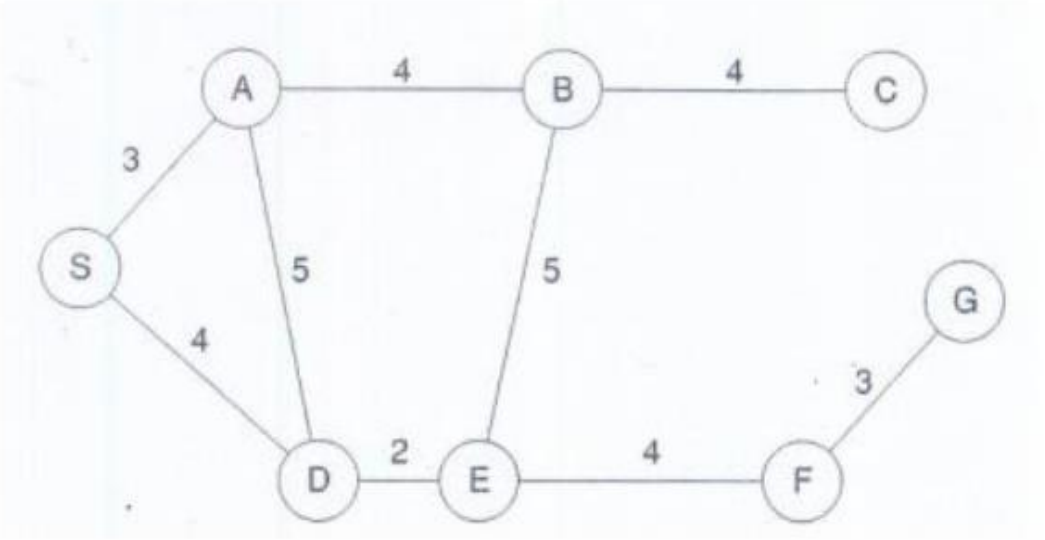
- Often implemented by maintaining a queue of vertices to visit

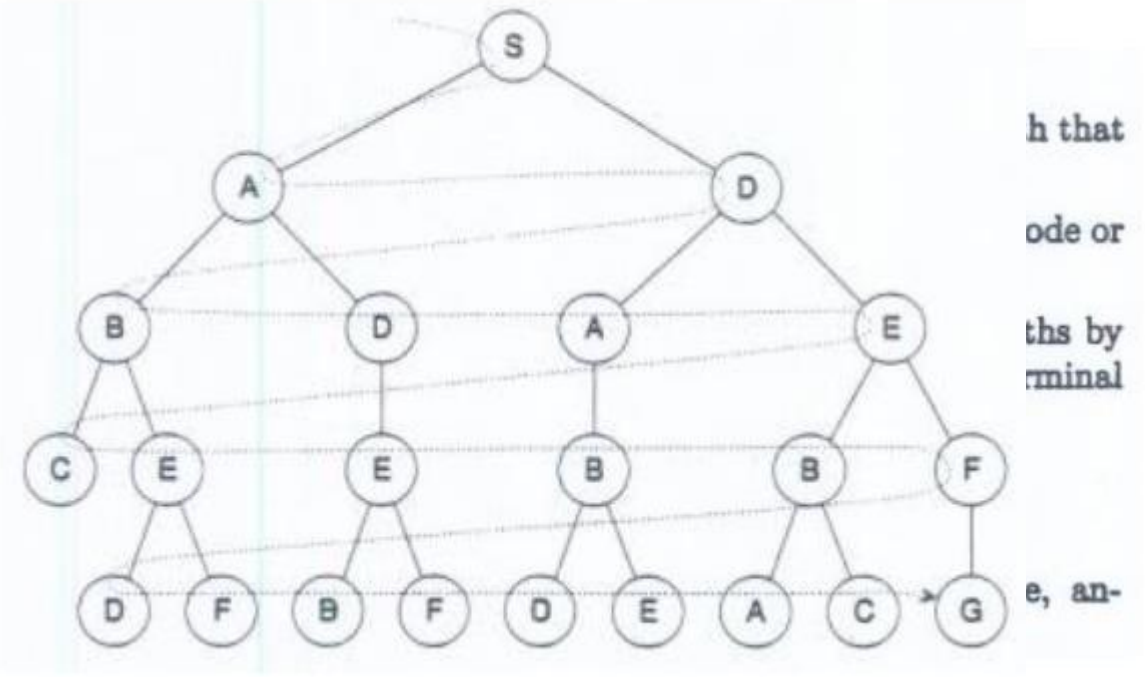
- The ordering the nodes on OPEN
  - Using a FIFO queue, **new paths go to the back of the queue**, and **old paths get expanded first**

To conduct a breadth-first search,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - Reject all new paths with loops.
  - Add the new paths, if any, to the *back* of the queue.
- If the goal node is found, announce success; otherwise, announce failure.

▷ Add the new paths, if any, to the *back* of the queue.





h that

ode or

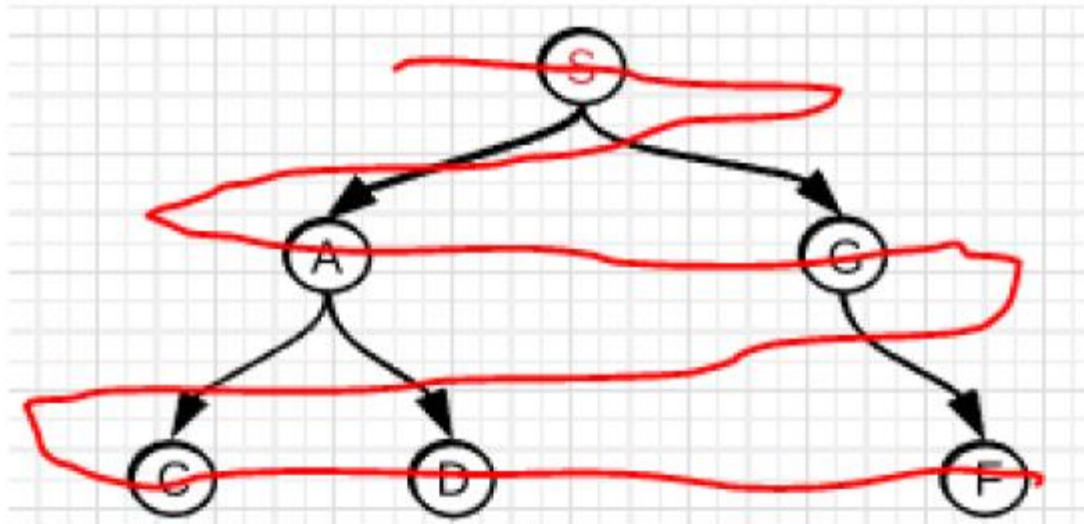
ths by
rminal

e, an-

Q={S}

Q={SA, SD}

Q={SD, SAB, SAD}



Advantage

if the goal not deep.
the goal might be in the firt
sweep,second or third and may hit
the goal