Uninformed Search

- **Rational agents need to *perform sequences of actions in order to achieve goals.***

- **Intelligent behavior can be generated by having a look-up table or reactive policy that tells the agent what to do in every circumstance, but**:

- Such a table or policy is difficult to build
- All contingencies must be anticipated

- **A more general approach is for the agent to have knowledge of the world and how its actions affect it and be able to simulate execution of actions in an internal model of the world in order to determine a sequence of actions that will accomplish its goals.**

- **This is the general task of problem solving and is typically performed by searching through an internally modelled space of world states**

Problem Solving and Search

Search is one of the most powerful approaches to problem solving in AI

- Search is a universal problem solving mechanism that
  - Systematically explores the **alternatives**
  - Finds the sequence of steps **towards a solution**

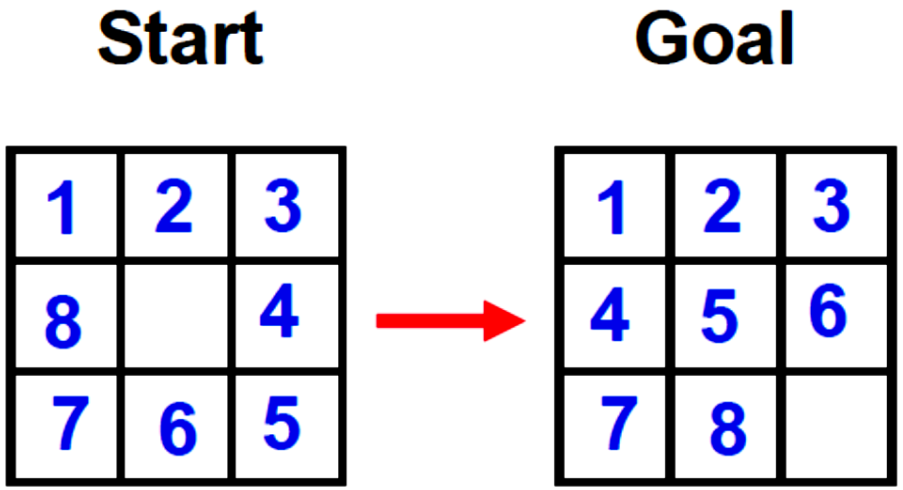
- Problem Space Hypothesis All goal-oriented symbolic computing activities occur in a problem space
  - Search in a problem space is claimed to be a completely general model of intelligence

Example Search Problems

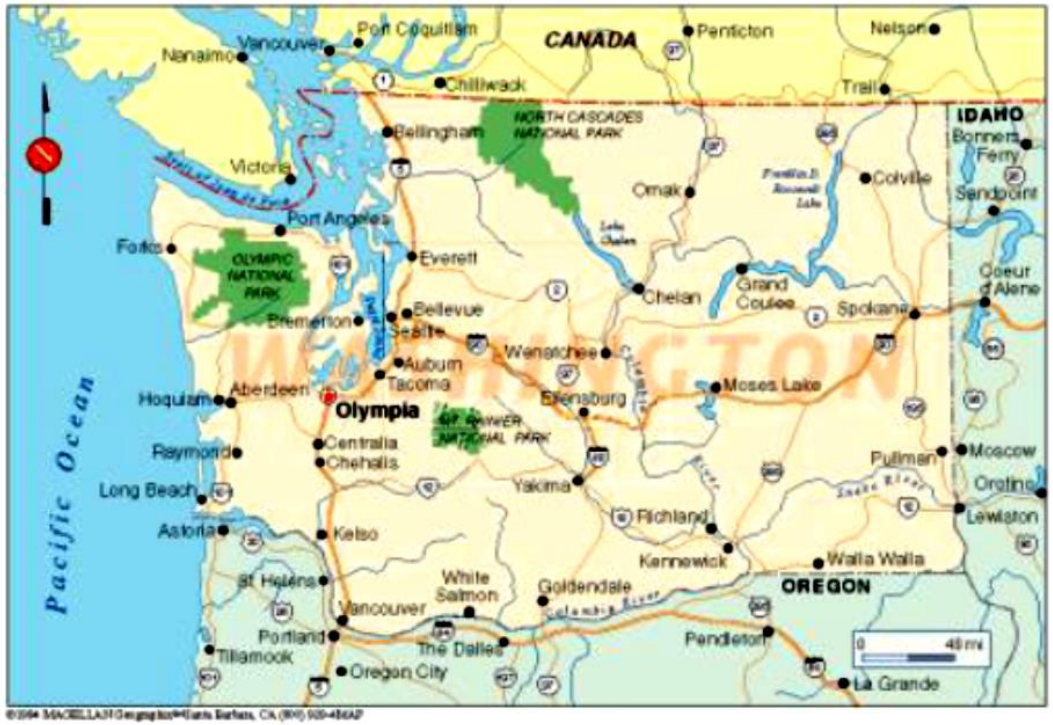
- 8-puzzle
- Tower of Hanoi
- Missionaries and Cannibals
- Water Jug
- Vacuum World
- Wumpus World
- Block World
- Travelling Salesperson
- Maze
- Crossword Puzzle
- Crypt-arithmetic
- Wheel of Fortune
- Chess, Bridge, etc

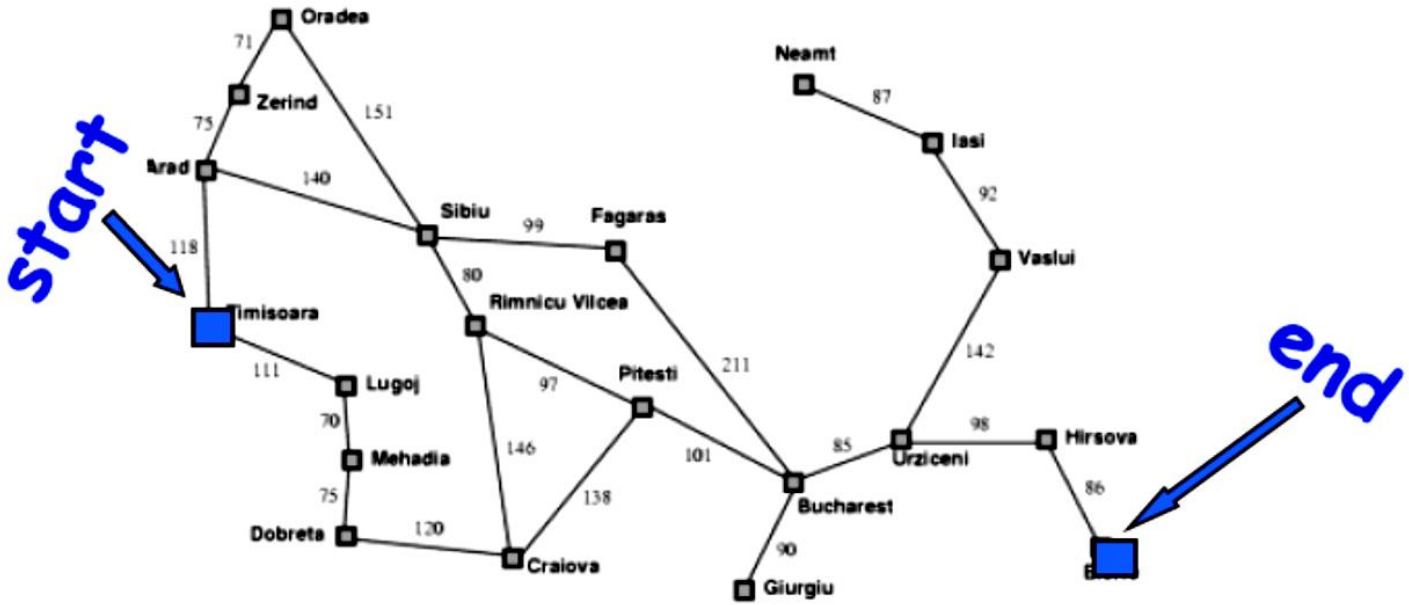
Problem Solving using Search: Examples

Example 1: The 8-puzzle

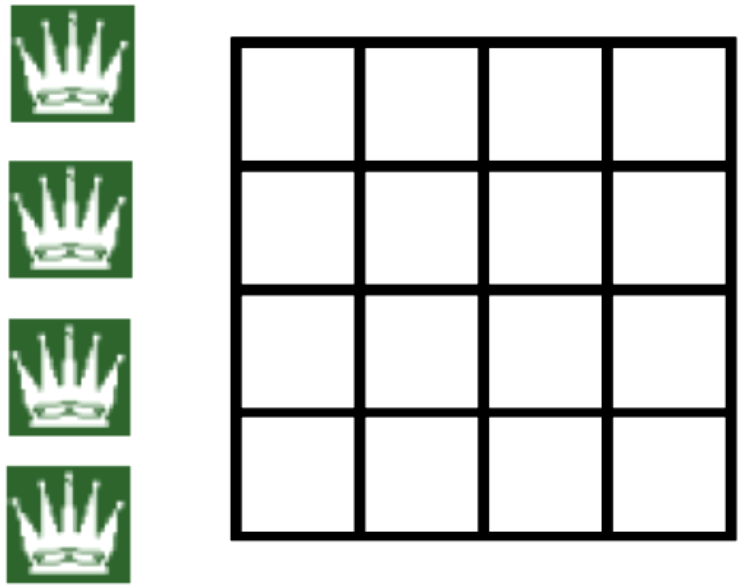


Example 2:
Route Planning





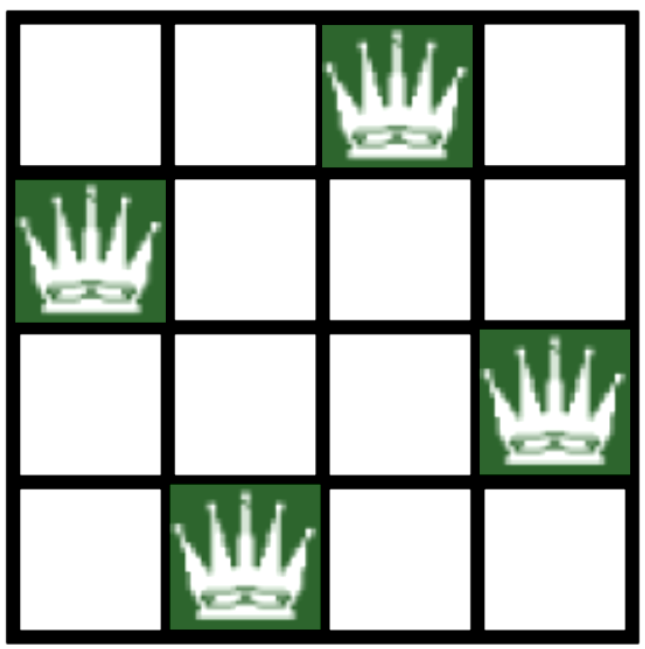
Example 3: N Queens

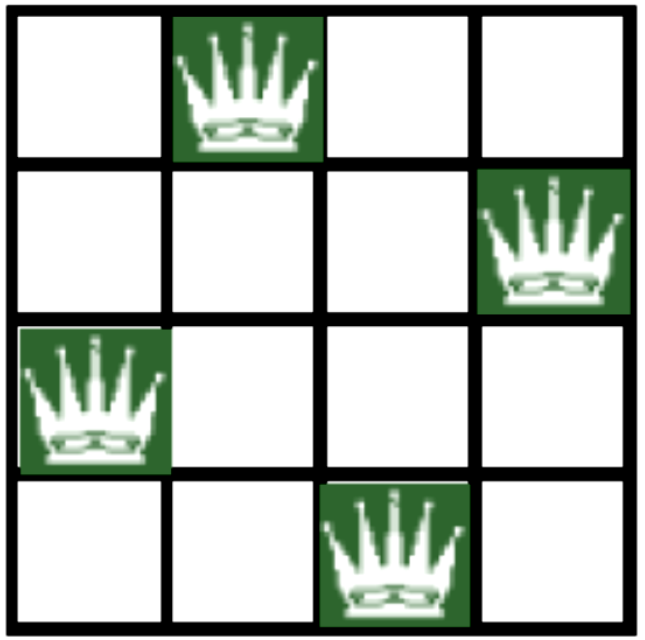


4 Queens problem

(Place queens such that no queen attacks any other)

Example 3: 4 Queens





4 Queens

Example 4:
Vacuum World

- Let the world consist of only **2 locations** - Left and Right Box
- Each location may contain **dirt**
- The agent may be in **either box**
- There are **8 possible states**
- The agent can have **3 possible actions** - Left, Right and Suck

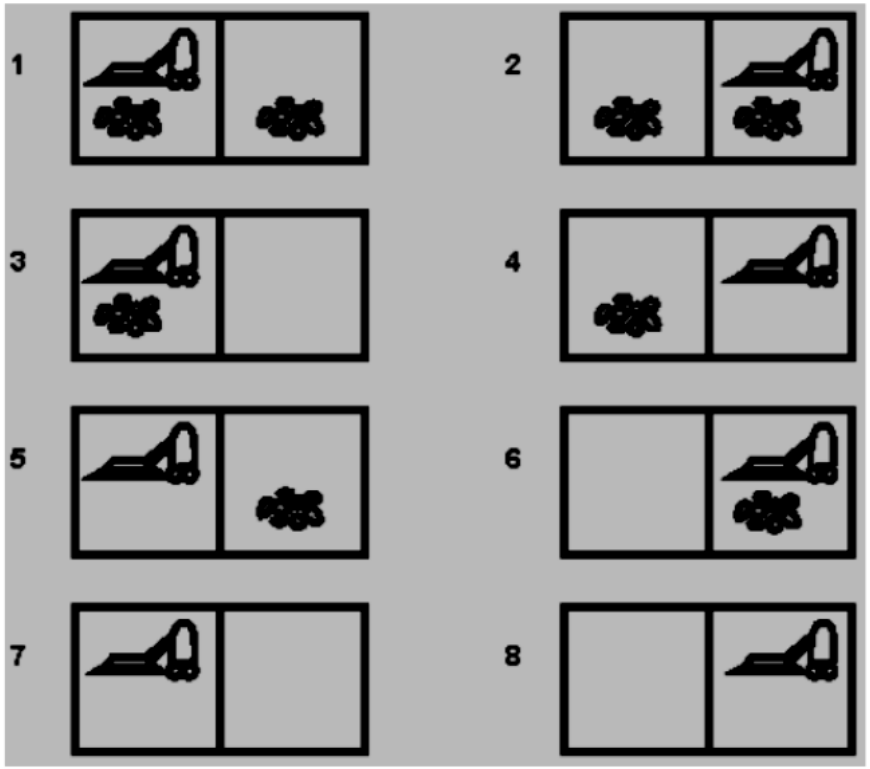


Fig.4.1: The 8 possible states of a Vacuum World

Search thru
Problem Space / State Space

General problem:

Find a path from a *start state* to a *goal state* given:

- **A goal test: Tests if a given state is a goal state**
- **A successor function (transition model): Given a state and action, generate *successor* state**

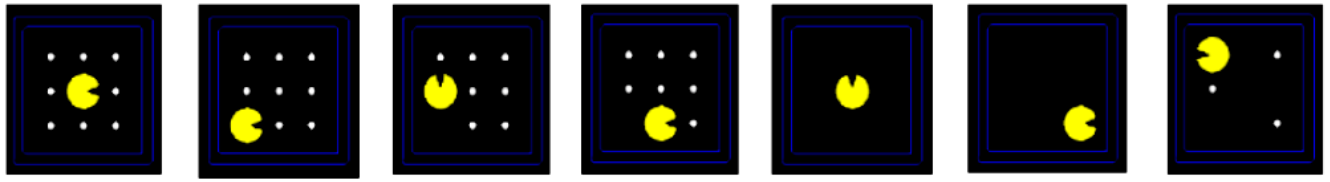
Variants:

- **Find any path *vs.* a least-cost path (if each step has a different cost i.e. a "step-cost")**
- **Goal is completely specified, task is to find a path or least-cost path**
  - **Route planning**
- **Path doesn't matter, only finding the goal state**
  - **8 puzzle, N queens, Rubik's cube**

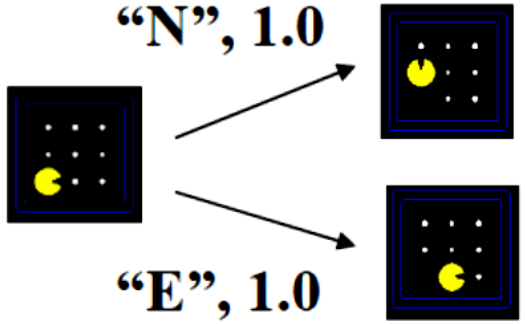
Example: Simplified Pac Man

Input:

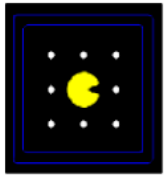
- **State space**



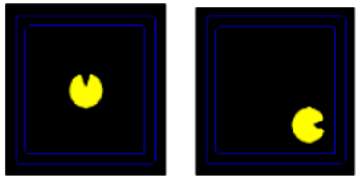
- **Successor function**



- **Start state**

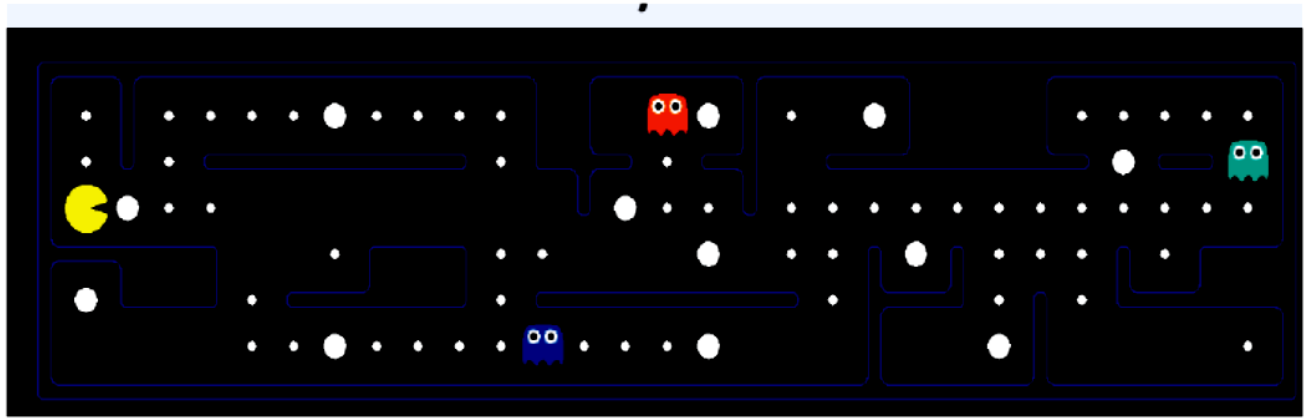


- **Goal test**



What is in State Space

- A world state includes every details of the environment



- A search state includes only details needed for planning

Problem: Pathing

States: {x,y} locations

Actions: NSEW moves

Successor: update location

Goal: is (x,y) End?

Problem: Eat-all-dots

States: {(x,y), dot booleans}

Actions: NSEW moves

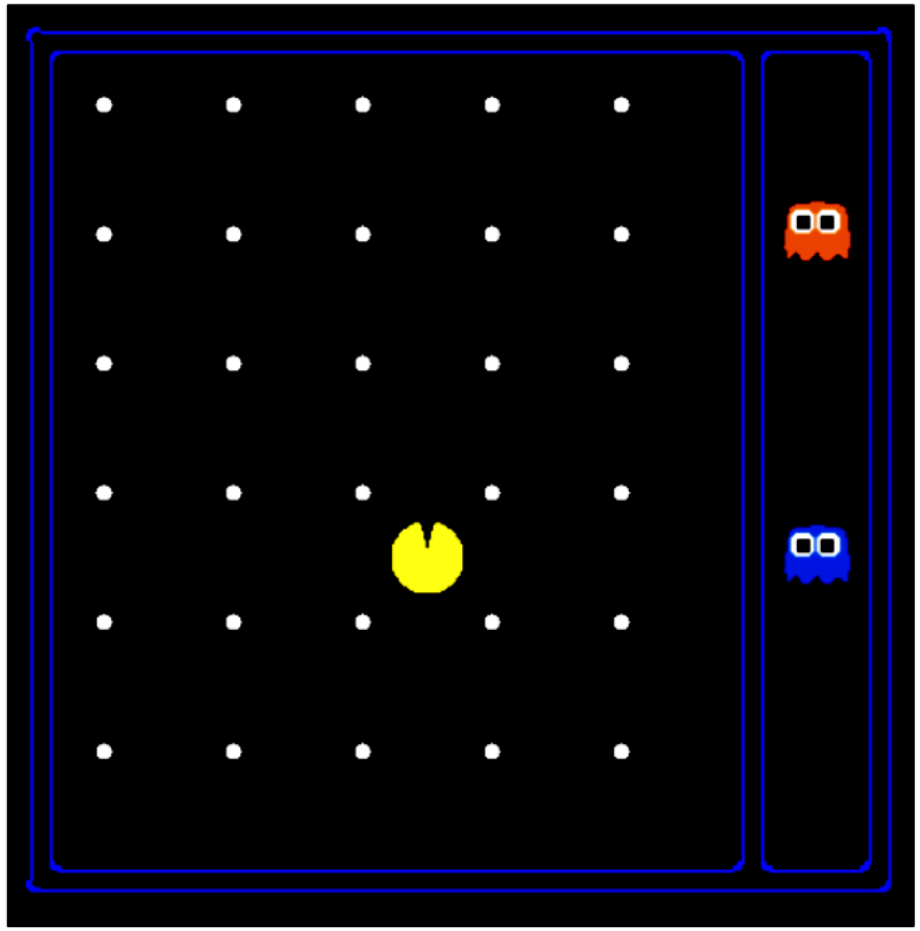
Successor: update location and dot boolean

Goal: dots all false?

State Space Sizes

- World states:

- Pacman positions:
  10 x 12 = 120
- Pacman facing:
  up, down, left, right
- Food Count: 30
- Ghost positions: 12



State Space Sizes

- How many?
- World State:

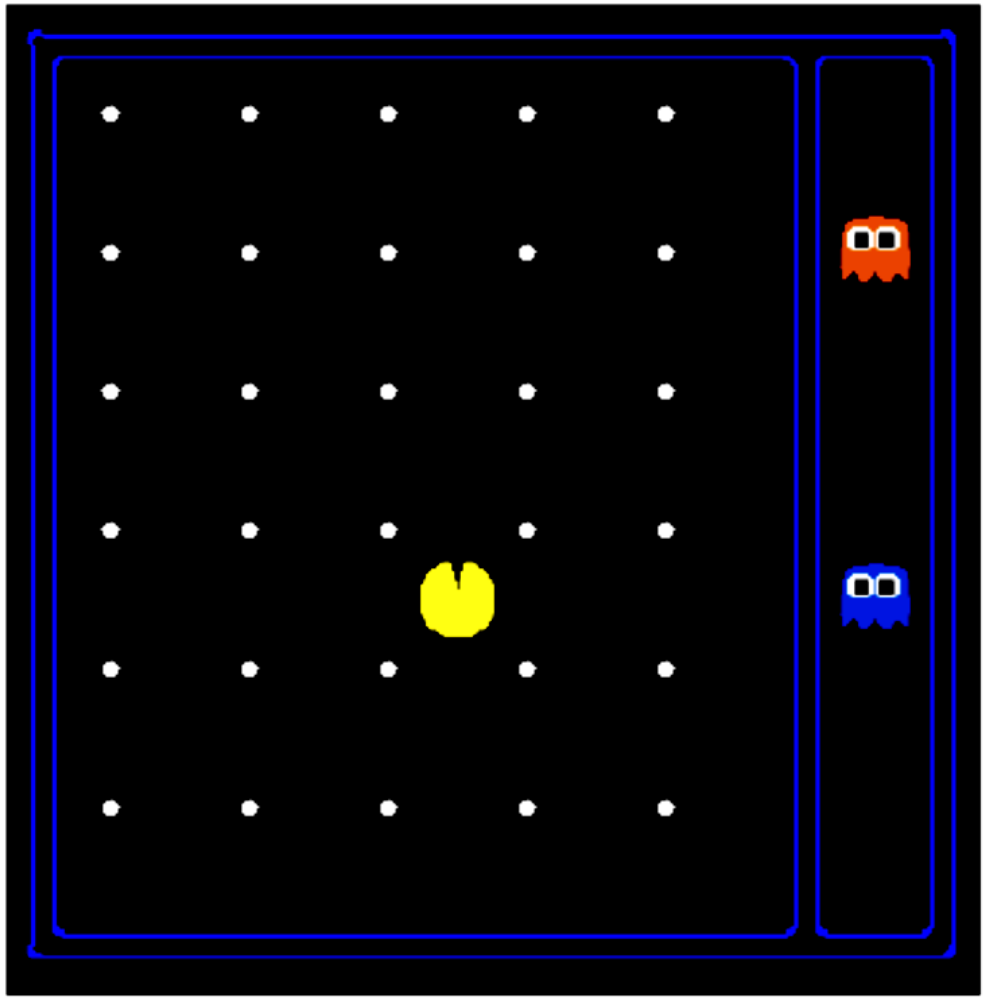
  $120*(2^{30})*(12^2)*4$

- States for Pathing:

  120

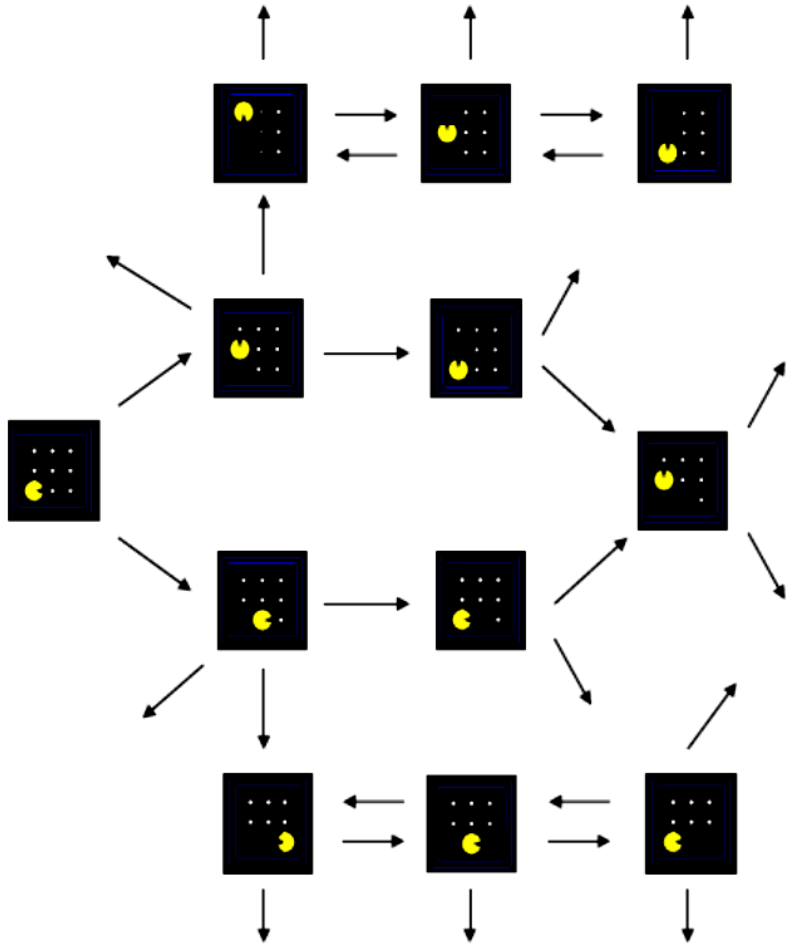
- States for eat-all-dots:

  $120*(2^{30})$

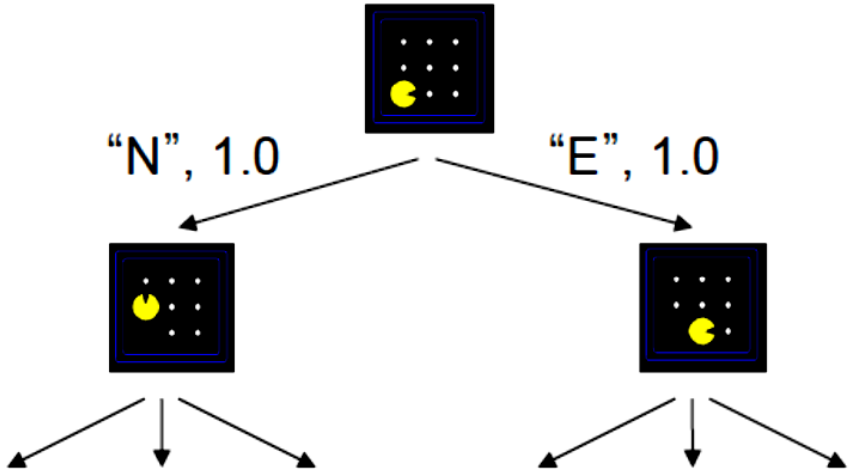


State Space Graphs

- State space graph:
  - Each node is a state
  - The successor function is represented by arcs
  - Edges may be labeled with costs
- We can rarely build this graph in memory (so we don't)

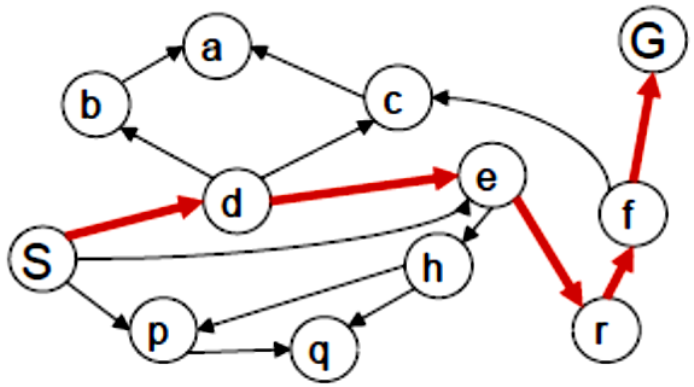


Search Trees



- A search tree:
  - Start state at the root node
  - Children correspond to successors
  - Nodes contain states, correspond to PLANS to those states
  - Edges are labeled with actions and costs
  - For most problems, we can never actually build the whole tree

State Graphs vs. Search Trees

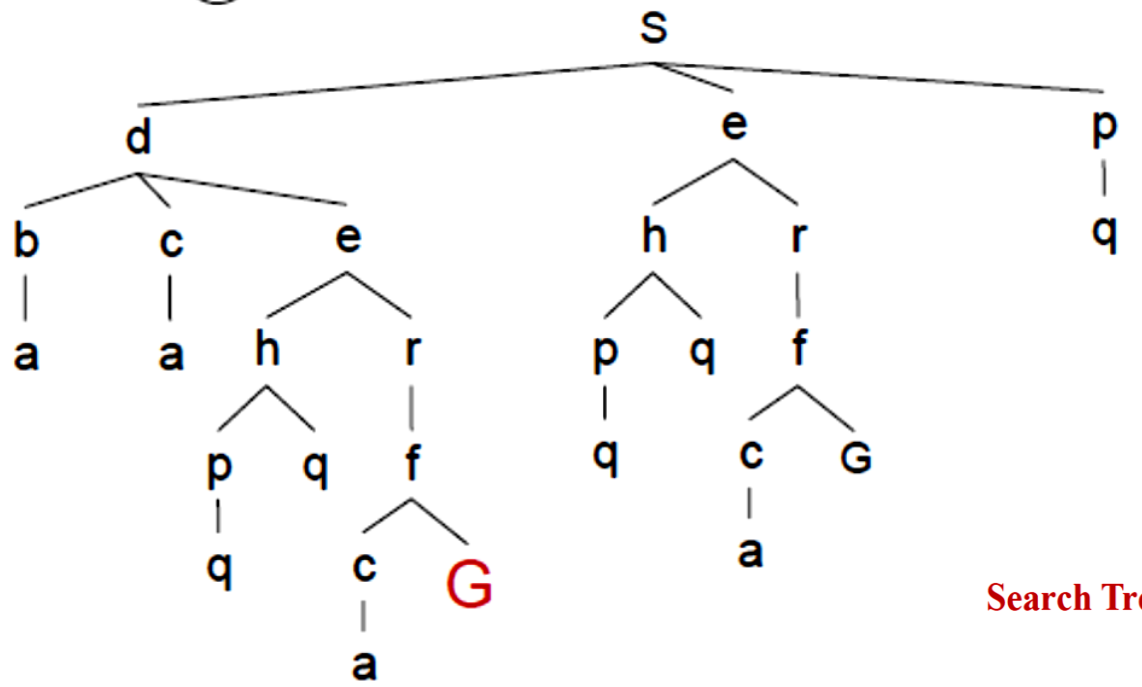


State Space/Graph

(graph of states with arrows pointing to successors)

Each NODE in in the search tree is an entire PATH in the problem graph.

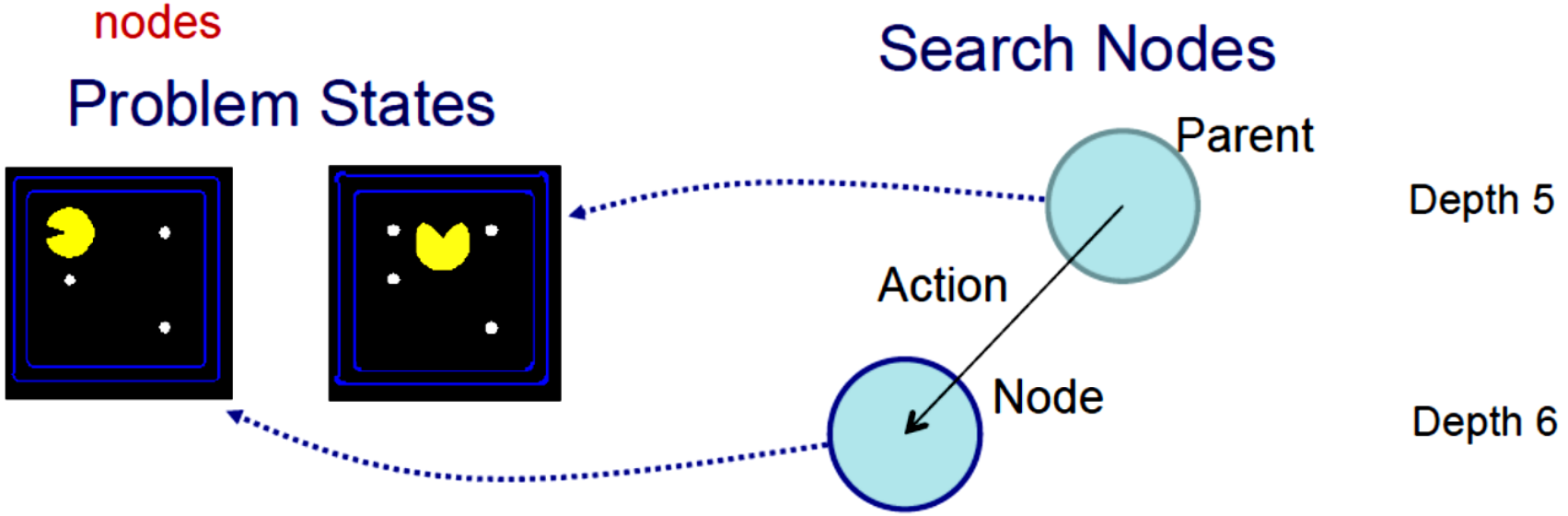
We construct both on demand – and we construct as little as possible.



Search Tree

States vs. Nodes

- Nodes in state space graphs are problem states
  - Represent an abstracted state of the world
  - Have successors, can be goal / non-goal, have multiple predecessors
- Nodes in search trees are plans
  - Represent a plan (sequence of actions) which results in the node's state
  - Have a problem state and one parent, a path length, a depth & a cost
  - The same problem state may be achieved by multiple search tree nodes



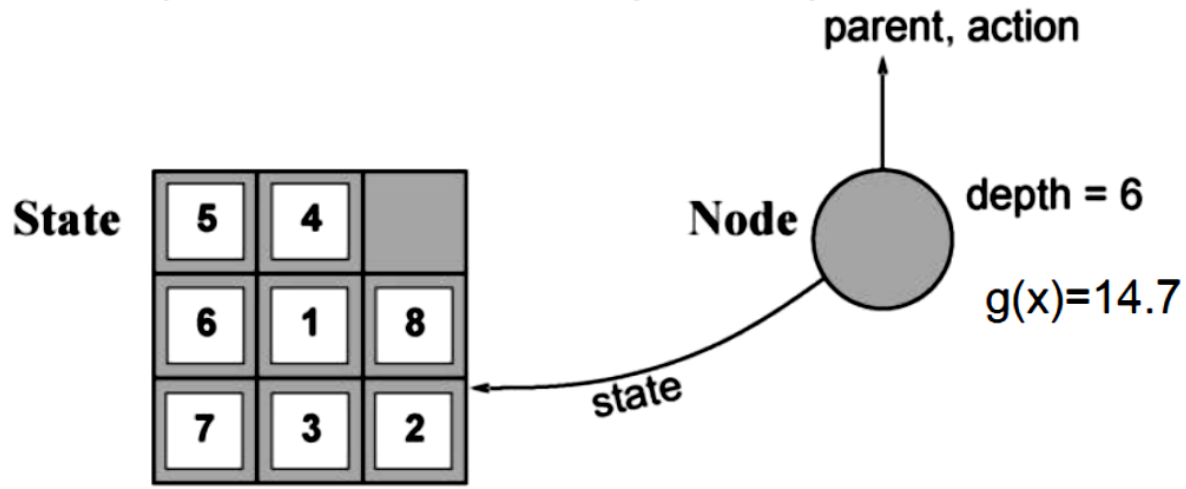
Implementation: states vs. nodes

A *state* is a (representation of) a physical configuration

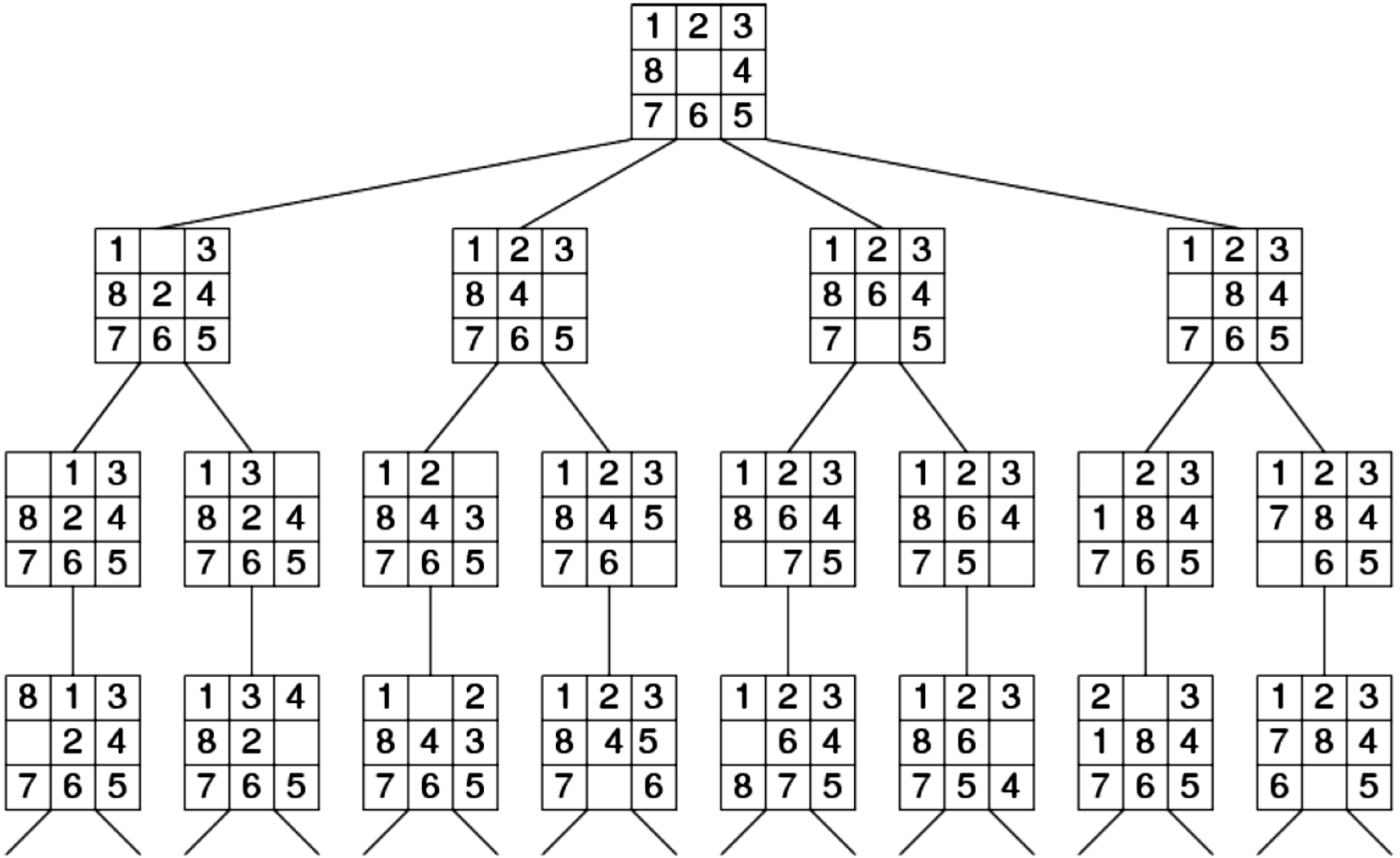
A *node* is a data structure constituting part of a search tree

includes *parent*, *children*, *depth*, *path cost* $g(x)$

States do not have parents, children, depth, or path cost!



Search Tree for 8-Puzzle:



Key Concepts

- Search problem:
  - States (configurations of the world)
  - Successor function; drawn as a graph
  - Start state and goal test

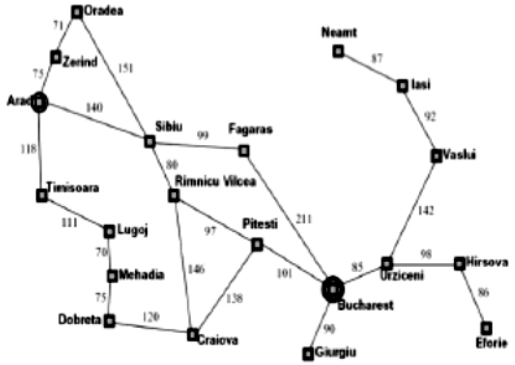
- Search tree:
  - Nodes: represent plans for reaching states
  - Plans have costs (sum of action costs)

Search thru a
Problem Space / State Space

- Input:
  - Set of states
  - Successor Function [and costs - default to 1.0]
  - Start state
  - Goal state [test]

- Output:
  - Path: start ⇒ a state satisfying goal test
  - [May require shortest path]
  - [Sometimes just need state passing test]

Building Search Trees



- Search:
  - Expand out possible plans
  - Maintain a fringe of unexpanded plans
  - Try to expand as few tree nodes as possible

General Tree Search

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

- Search Algorithms:
  - Systematically builds a search tree
  - Chooses an ordering of the fringe (unexplored nodes)
  - Main question: which fringe nodes to explore?

Implementation: general tree search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem] applied to STATE(node) succeeds return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Search strategies

A strategy is defined by picking the *order of node expansion*

Strategies are evaluated along the following dimensions:

- completeness—does it always find a solution if one exists?
- time complexity—number of nodes generated/expanded
- space complexity—maximum number of nodes in memory
- optimality—does it always find a least-cost solution?

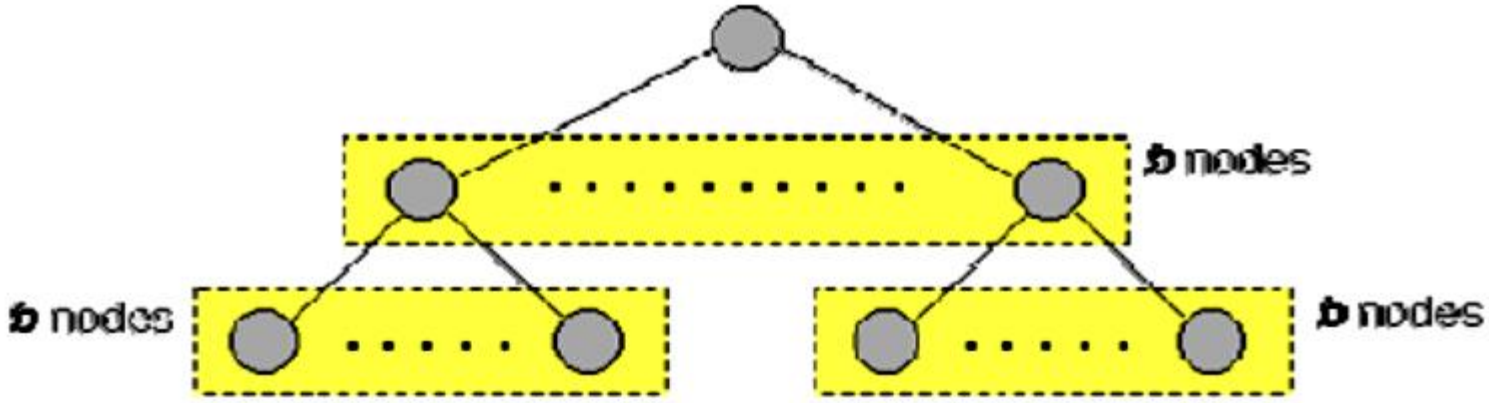
Time and space complexity are measured in terms of

- b—maximum branching factor of the search tree
- d—depth of the least-cost solution
- m—maximum depth of the state space (may be ∞)

Branching factor: The number of nodes that are expanded at any given node in a hypothetical search tree.

Assuming a branching factor of ***b***, the number of nodes that are expanded at a depth ***d*** is given by the formula $1 + b + b^2 + b^3 + \ldots + b^d$.

The branching factor is used for defining the time and space complexity of different strategies.



Search tree of depth ***d***=2 and branching factor ***b***

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

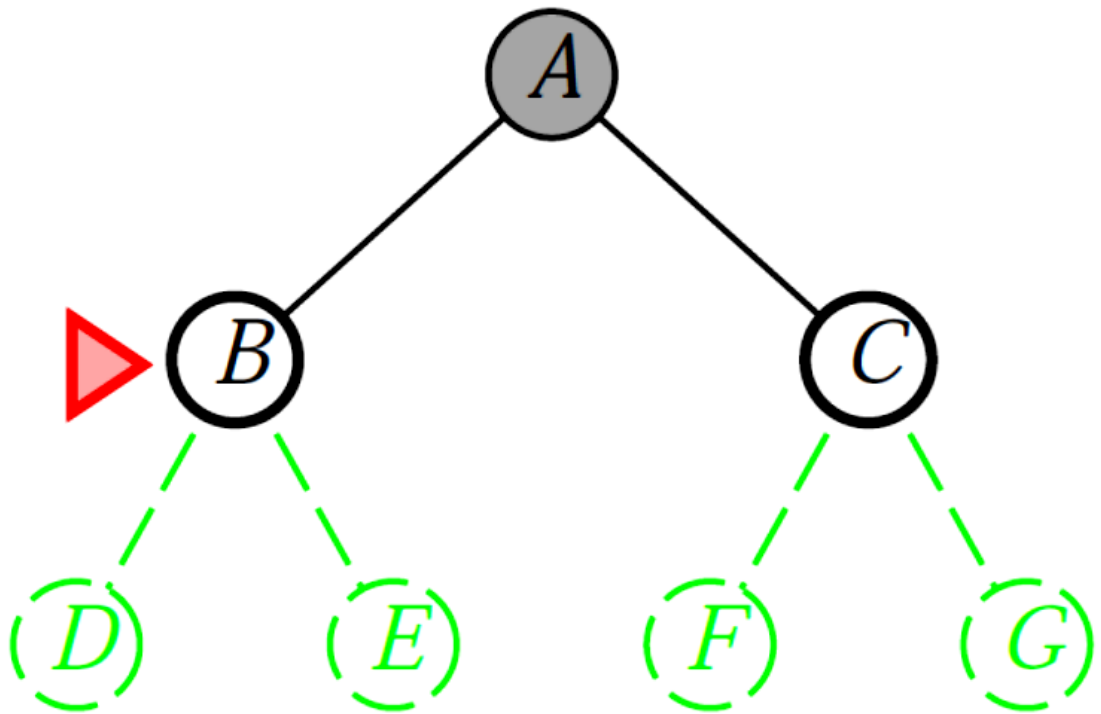
Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

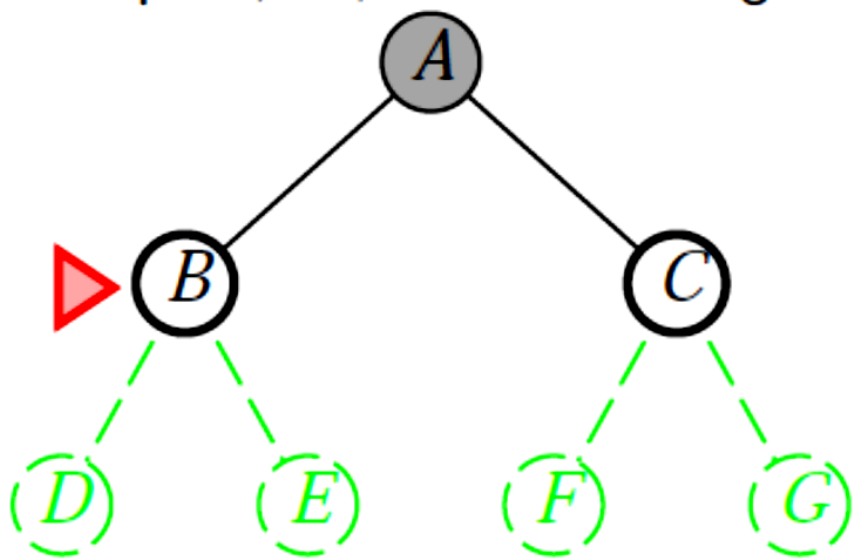


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

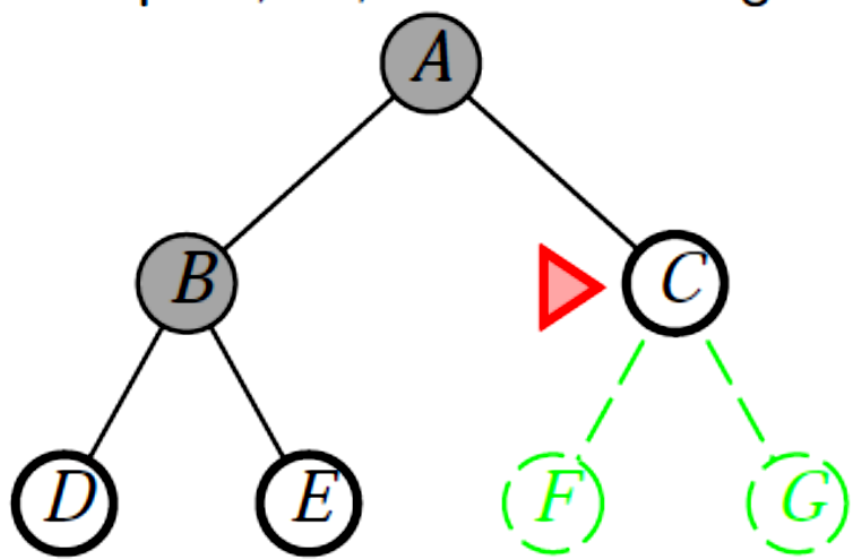


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

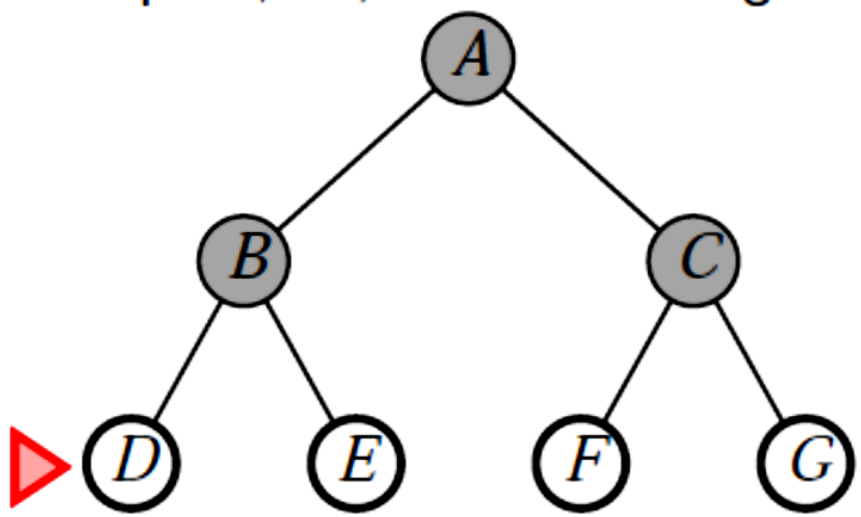


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end



Properties of breadth-first search

Complete??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? Yes (if cost $= 1$ per step); not optimal in general

Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs $=$ 8640GB.

Space and time are big problems for BFS.

Example: b = 10 with 1,000,000 nodes/sec, 1000 Bytes/node

d = 2 ➔ 110 nodes, 0.11 millisecs, 107KB

d = 4 ➔ 11,110 nodes, 11 millisecs, 10.6 MB

d = 8 ➔ 10^8 nodes, 2 minutes, 103 GB

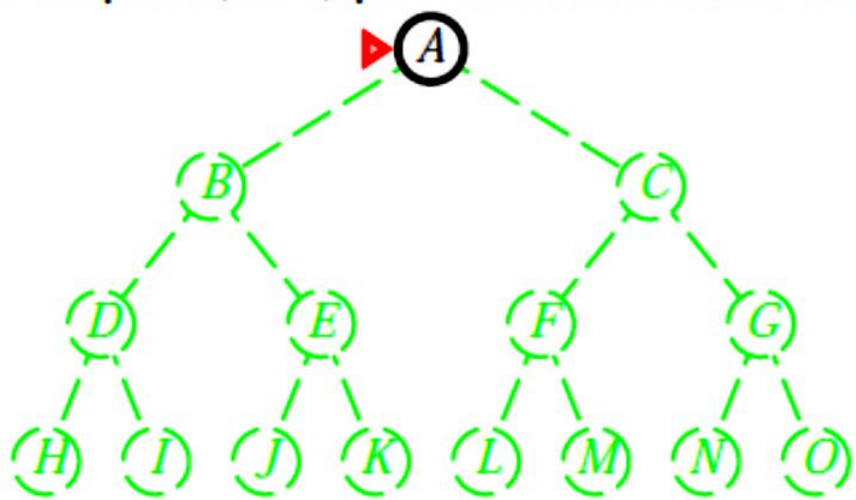
d = 16 ➔ 10^{16} nodes, 350 years, 10 EB (1 billion GB)

Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

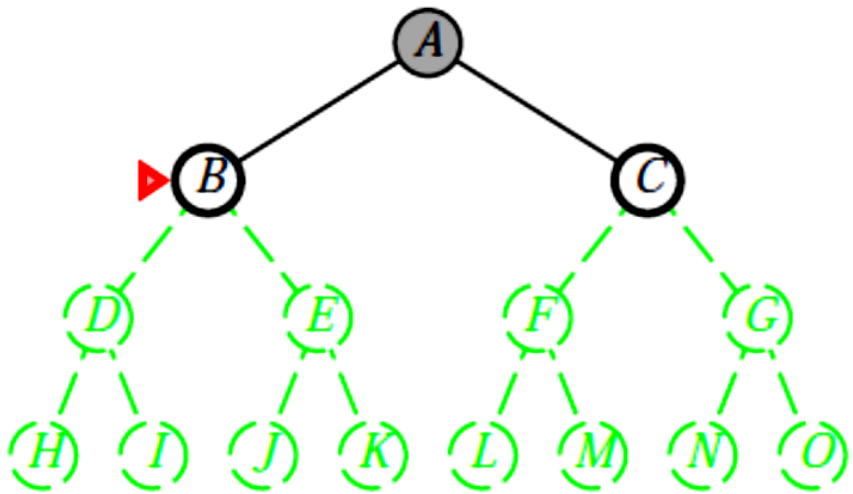


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

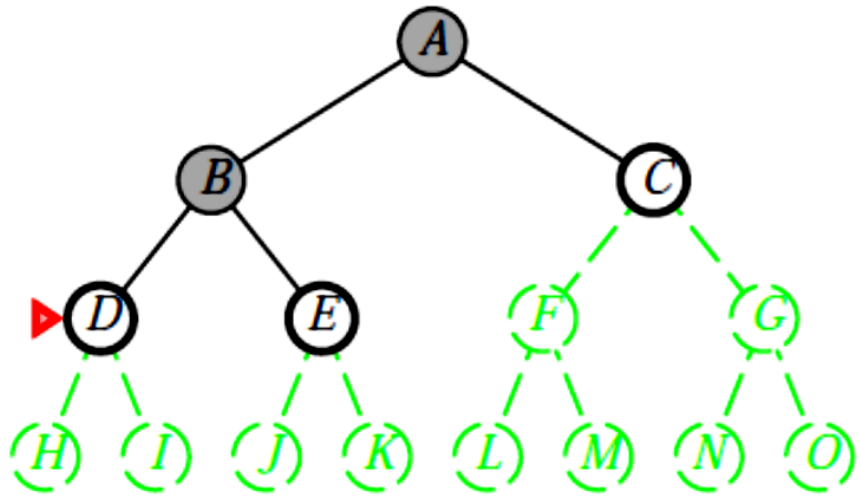


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

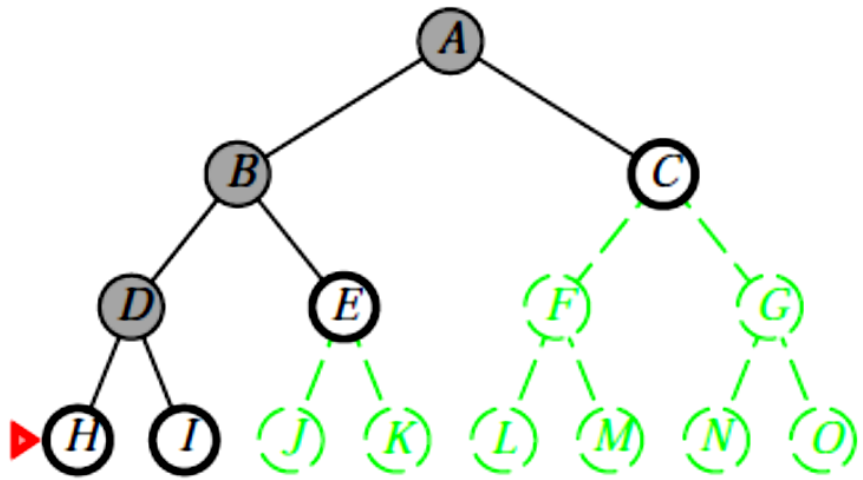


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

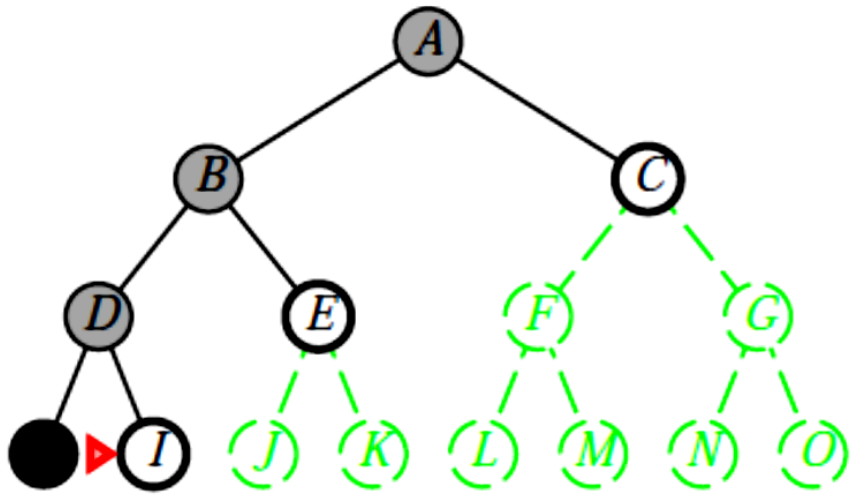


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

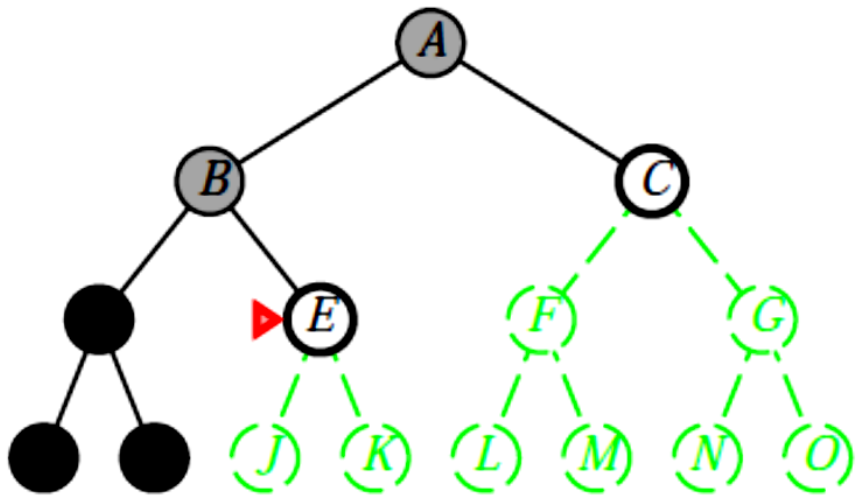


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

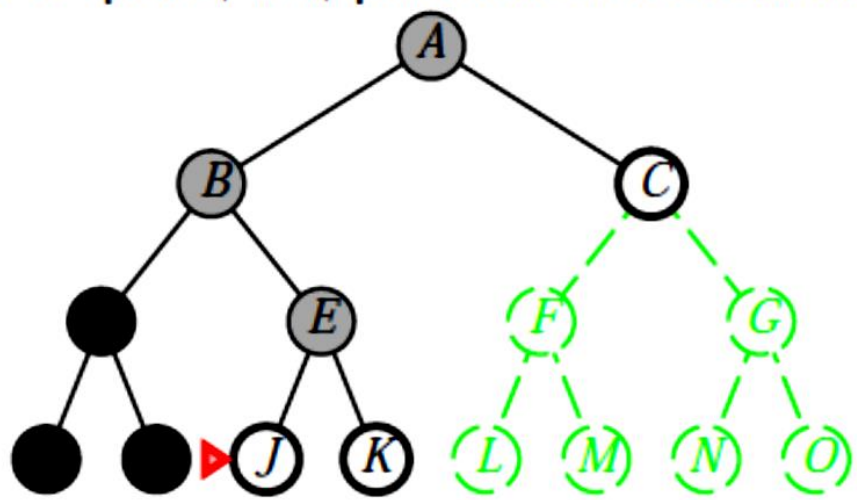


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

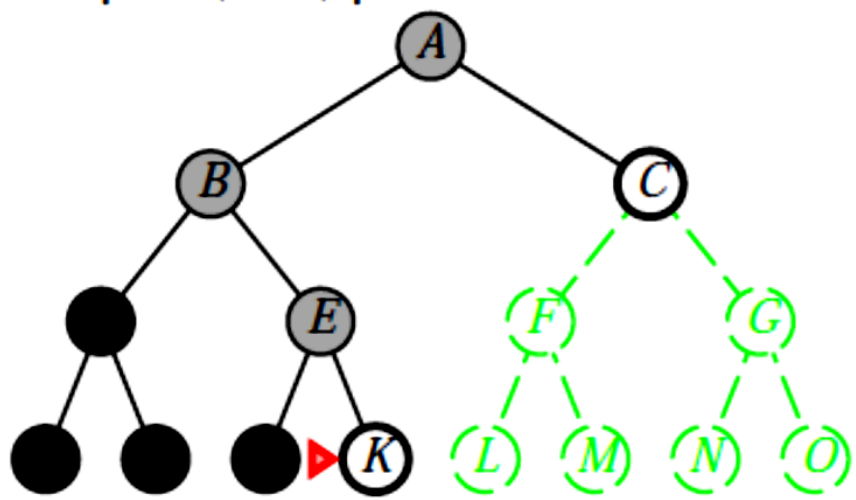


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

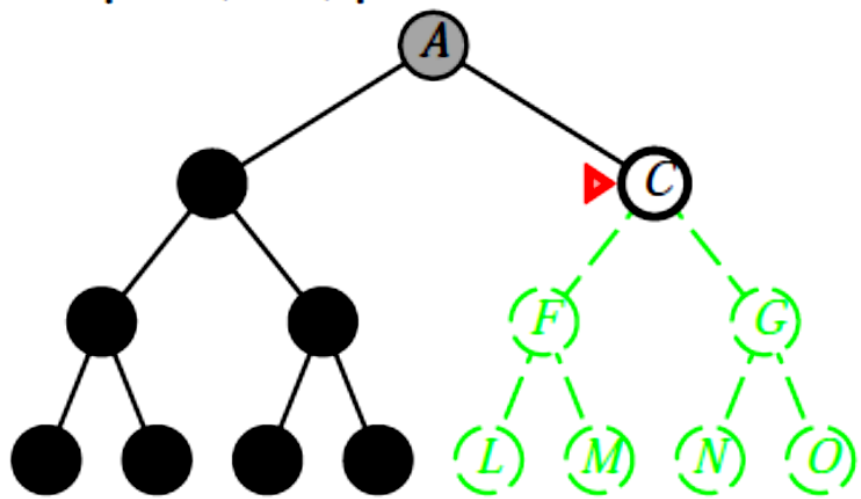


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

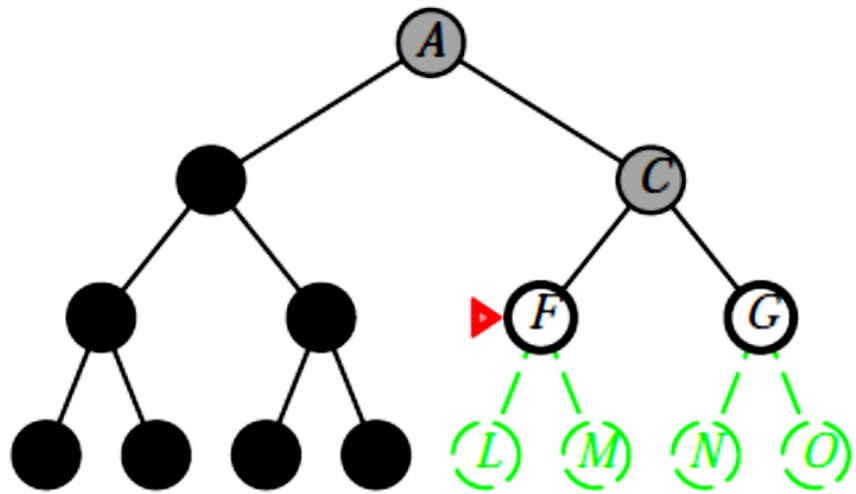


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

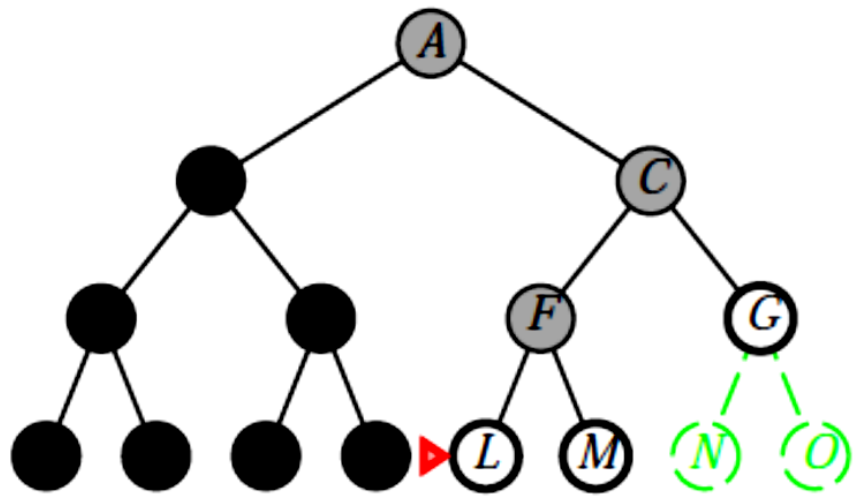


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

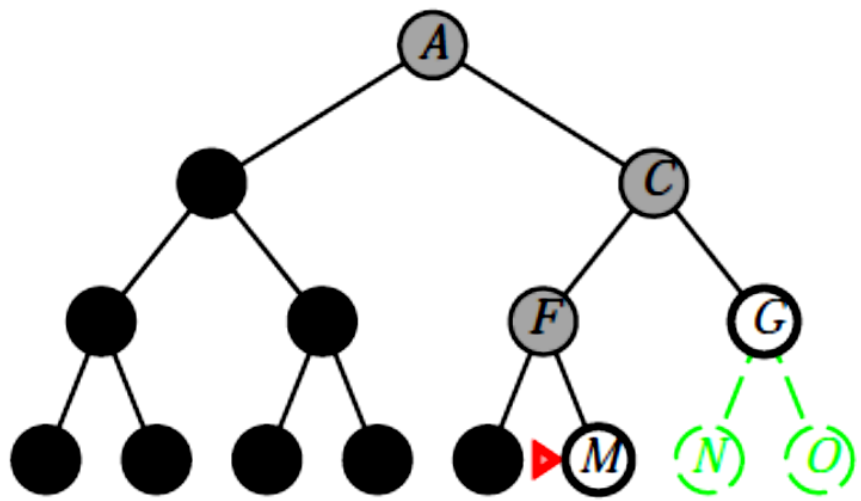


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front



Properties of depth-first search

Complete??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

Time??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, may be much faster than breadth-first

Space??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, may be much faster than breadth-first

Space?? $O(bm)$, i.e., linear space!

Optimal??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, may be much faster than breadth-first

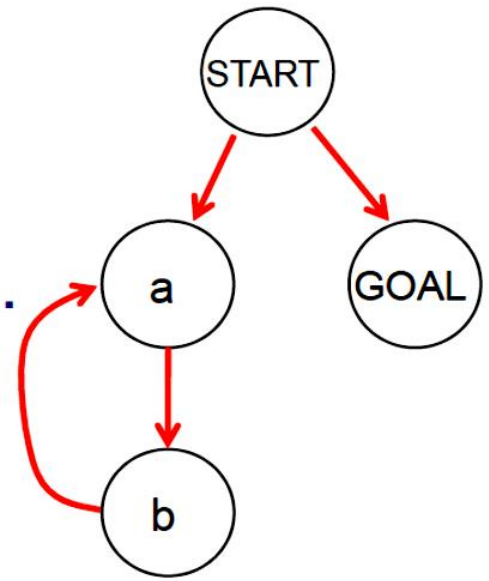
Space?? $O(bm)$, i.e., linear space!

Optimal?? No

DFS

Algorithm		Complete	Optimal	Time	Space
DFS	Depth First Search	No	No	Infinite	Infinite

- Infinite paths make DFS incomplete…
  - How can we fix this? and non-optimal
  - Check new nodes against path from S
- Infinite search spaces still a problem
  - If the left subtree has unbounded depth



Space cost is a big advantage of DFS over BFS.

Example: b = 10, 1000 Bytes/node

d = 16 → 156 KB instead of 10 EB (1 billion GB)

Depth-limited search

= depth-first search with depth limit l,
i.e., nodes at depth l have no successors

Recursive implementation:

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST(problem, STATE[node]) then return node
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

Iterative deepening search $l = 0$

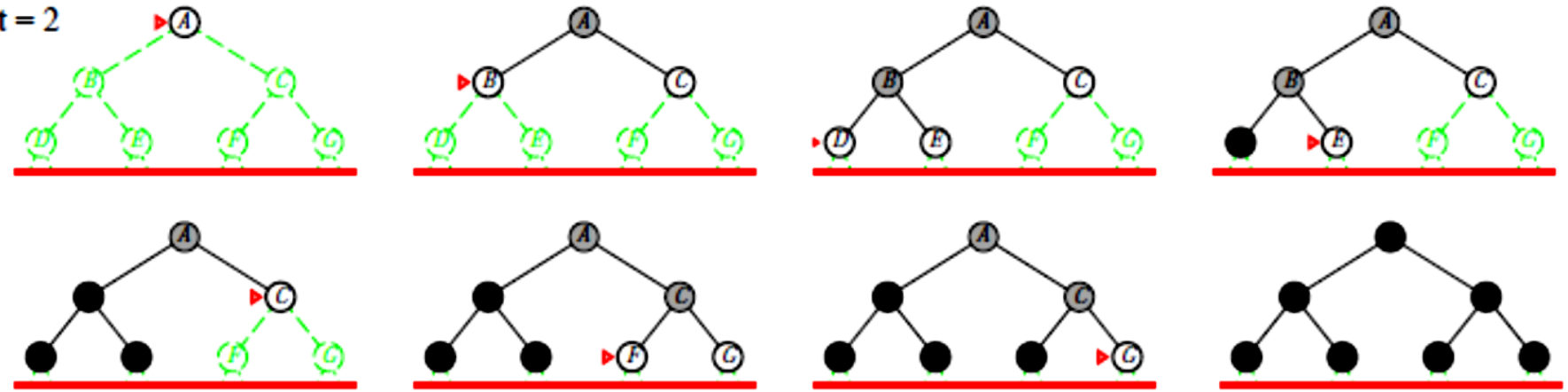
Limit = 0

Iterative deepening search $l = 1$

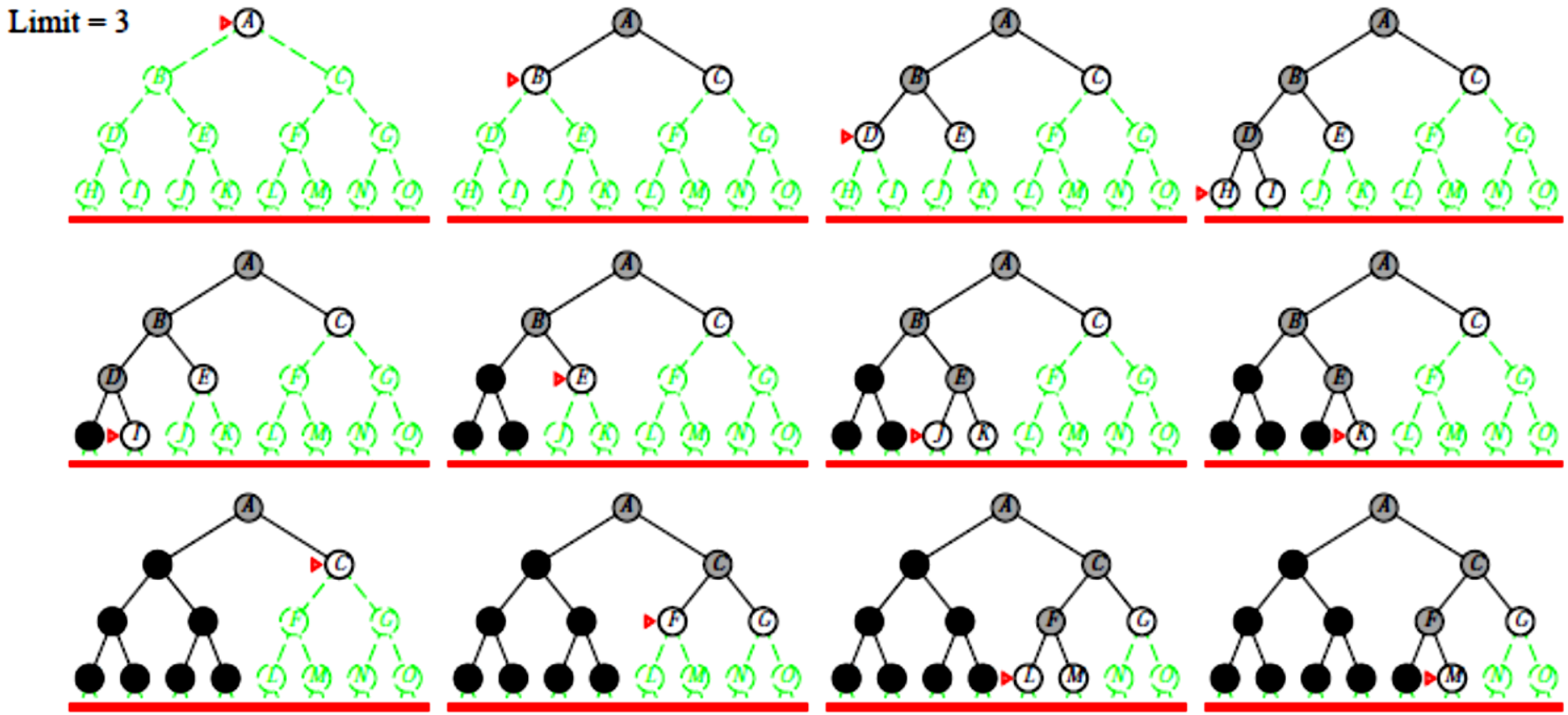
Limit = 1

Iterative deepening search $l = 2$

Limit = 2



Iterative deepening search $l = 3$



Properties of iterative deepening search

Complete??

Properties of iterative deepening search

Complete?? Yes

Time??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost $= 1$
Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

IDS does better because other nodes at depth d are not expanded

BFS can be modified to apply goal test when a node is **generated**

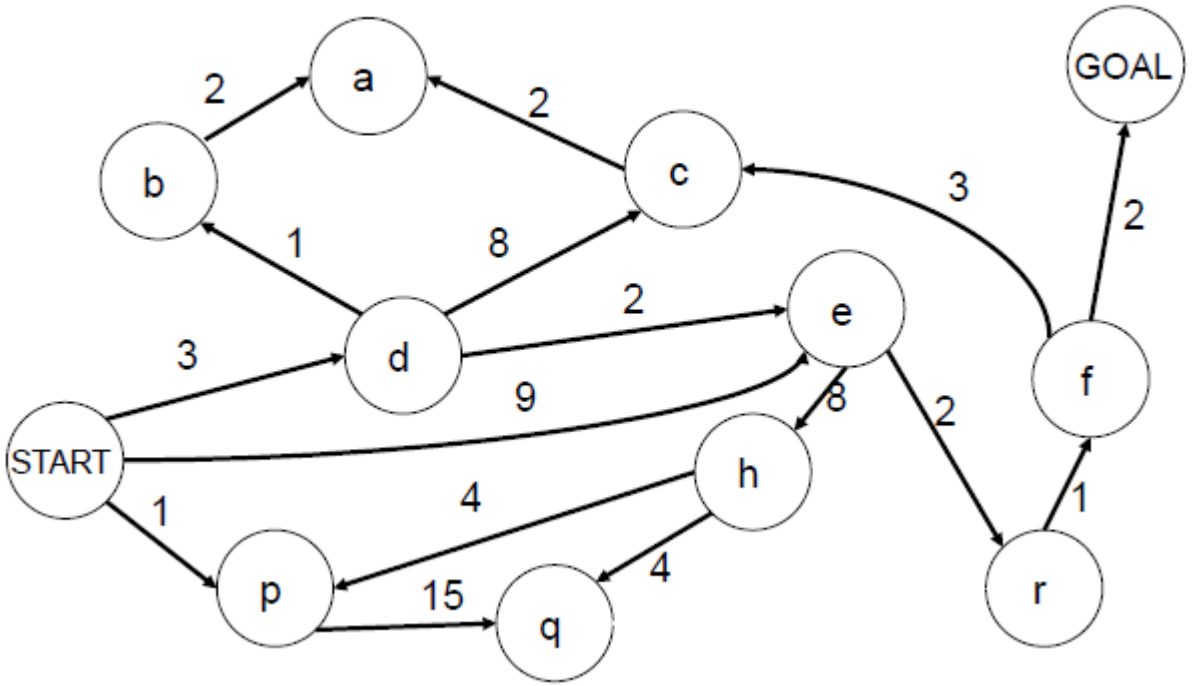
Summary of algorithms

Criterion	Breadth-First		Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*		No	Yes, if $l \geq d$	Yes
Time	b^{d+1}		b^m	b^l	b^d
Space	b^{d+1}		bm	bl	bd
Optimal?	Yes*		No	No	Yes*

What if the step costs are not equal?

Can we modify BFS to handle any step cost function?

Uniform Cost Search



- Generalization of Breadth-First Search
- Fringe List is a Priority Queue
- Cost function g(n) is associated with each node

Uniform Cost Search

- Generalization of Breadth-First Search
- Fringe List is a Priority Queue
- Cost function g(n) is associated with each node

```
Add initial state to priority queue
While queue not empty
        Node = head(queue)
        If goal?(node) then return node
        Add children of node to queue
```

Priority Queue Refresher

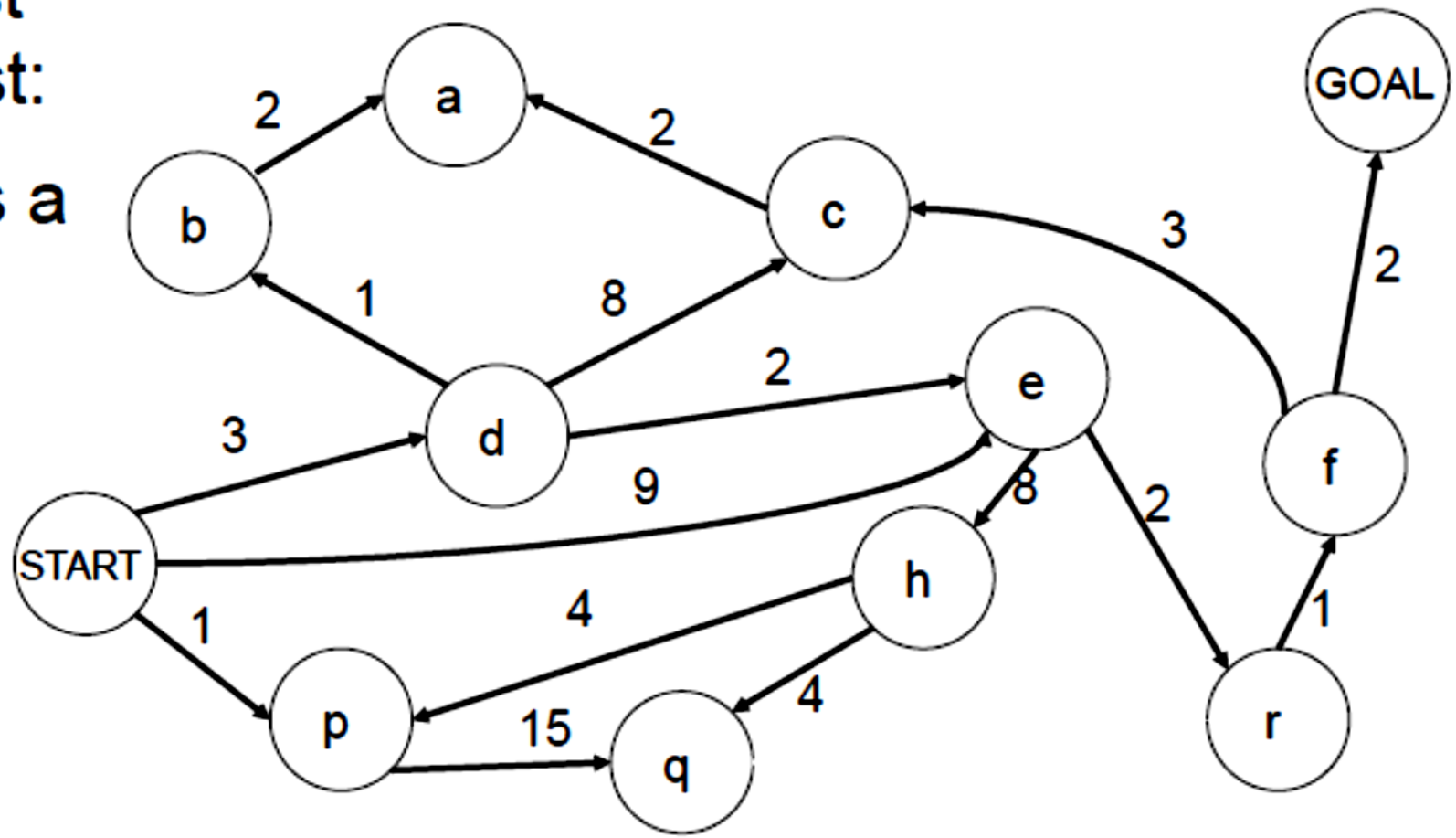
- A priority queue is a data structure in which you can insert and retrieve (key, value) pairs with the following operations:

pq.push(key, value)	inserts (key, value) into the queue.
pq.pop()	returns the key with the lowest value, and removes it from the queue.

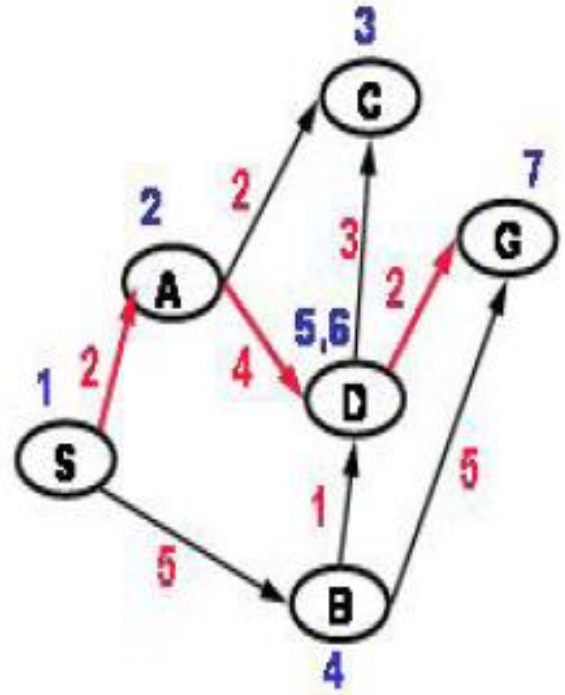
- Unlike a regular queue, insertions aren't constant time, usually O(log n)
- We'll need priority queues for cost-sensitive search methods

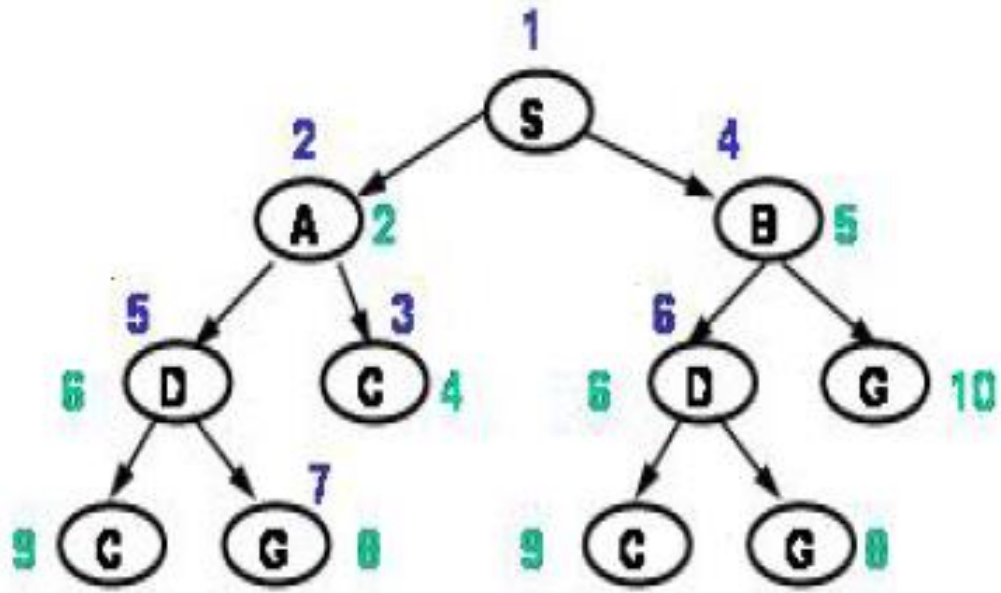
Expand cheapest node first:

Fringe is a priority queue



	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)
5	(6 D B S) (10 G B S) (6 D A S)
6	(8 G D B S) (9 C D B S) (10 G B S) (6 D A S)
7	(8 G D A S) (9 C D A S) (8 G D B S) (9 C D B S) (10 G B S)

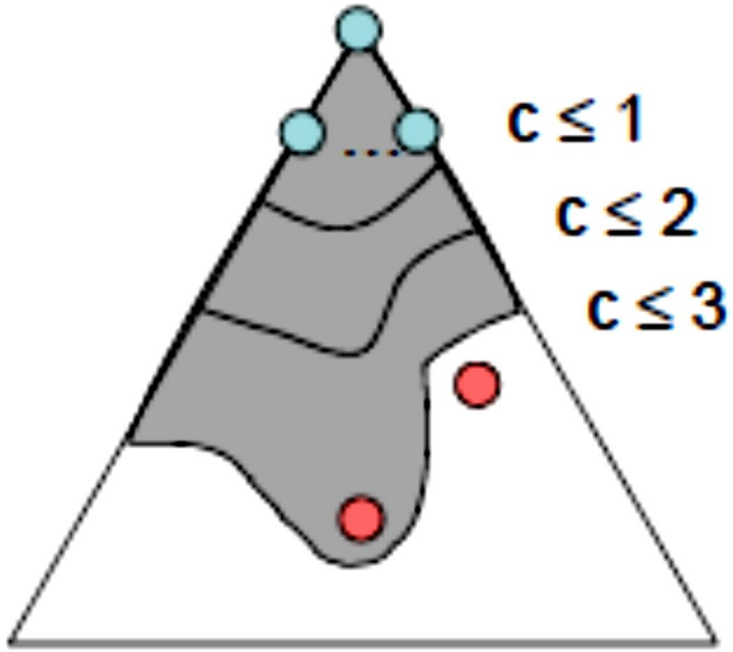




The n-tuple, n=1,2,3,4,.. is the plan associated with the node. It's the path from the root to the node in reverse order.

Underscore tuple is the node picked next from fringe.

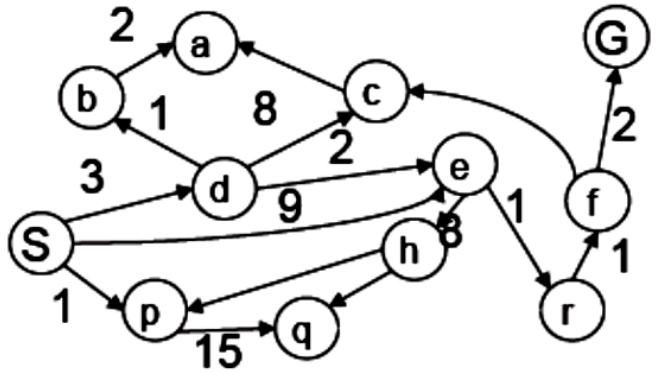
- Strategy: expand lowest path cost
- Nodes on paths of higher costs are examined
  only after examining all nodes on paths with lower costs
- Imposes monotonically increasing cost contours

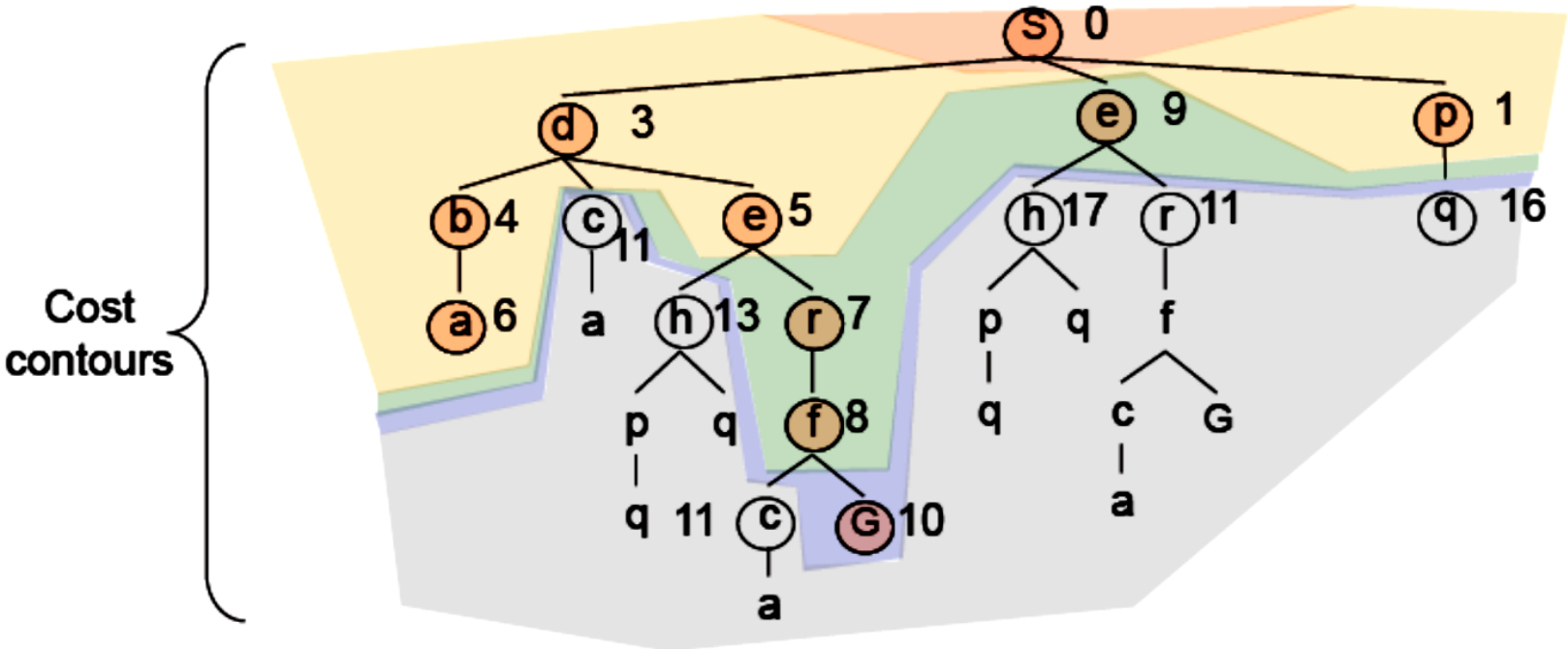


Uniform Cost Search

Expansion order:

(S,p,d,b,e,a,r,f,e,G)

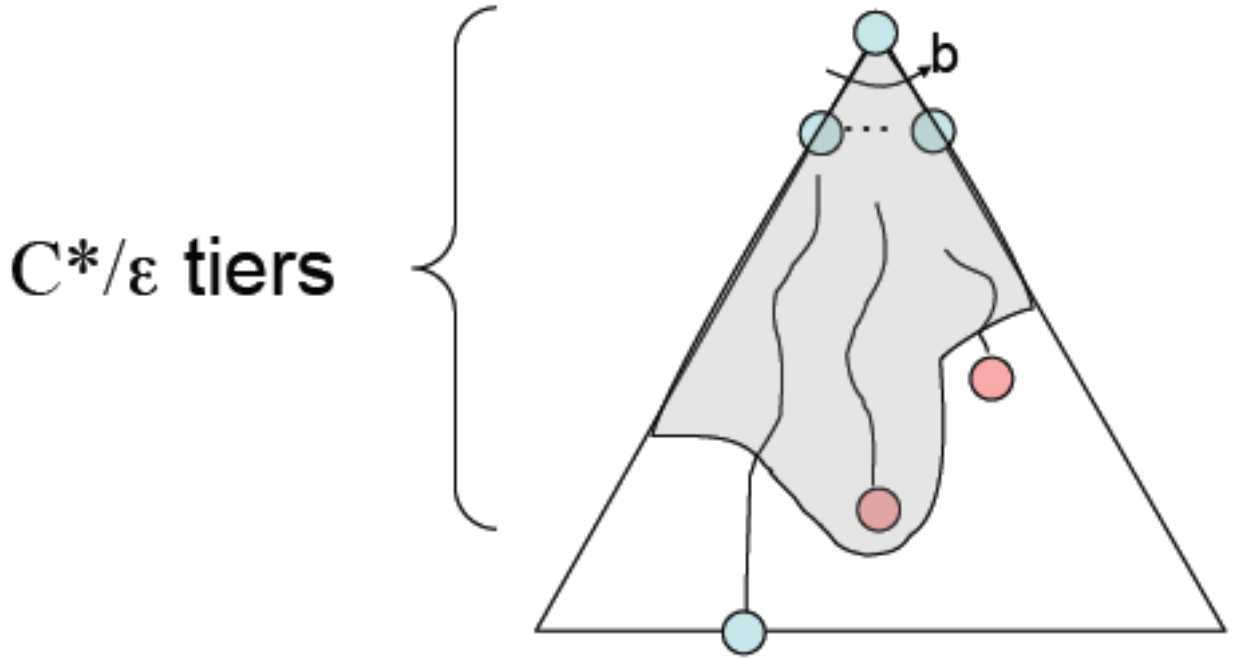




Contour costs <= 0, <=6, <=8, <=10.<=16,..

Uniform Cost Search as it sounds **searches** in branches which are more or less the same in **cost**

If that solution costs C^* and arcs cost at least ε , then the "effective depth" is roughly at most C^*/ε



Both Time and Space complexity: $O(b^{C^*/\varepsilon})$

Exercise: Proof of Optimality of Uniform Cost Search. Proof uses this property

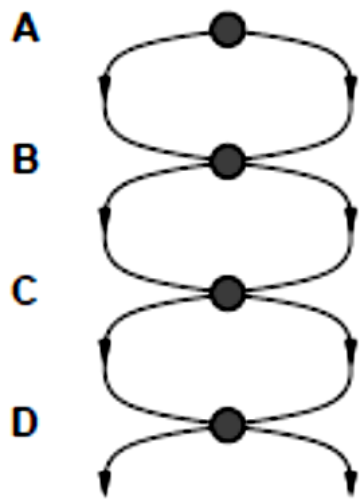
For every path p from start state to goal state there always exists a node in fringe list that is on path p.

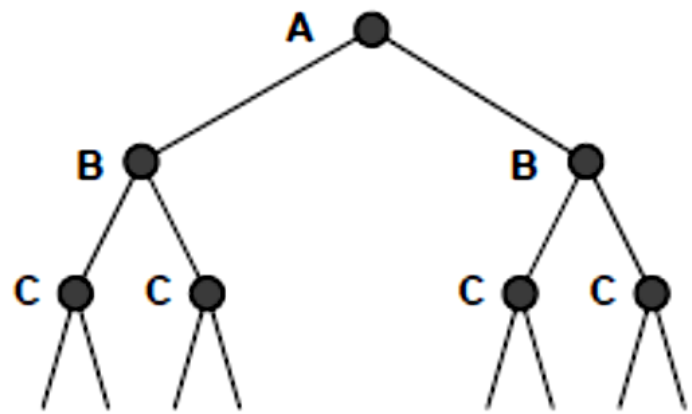
Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

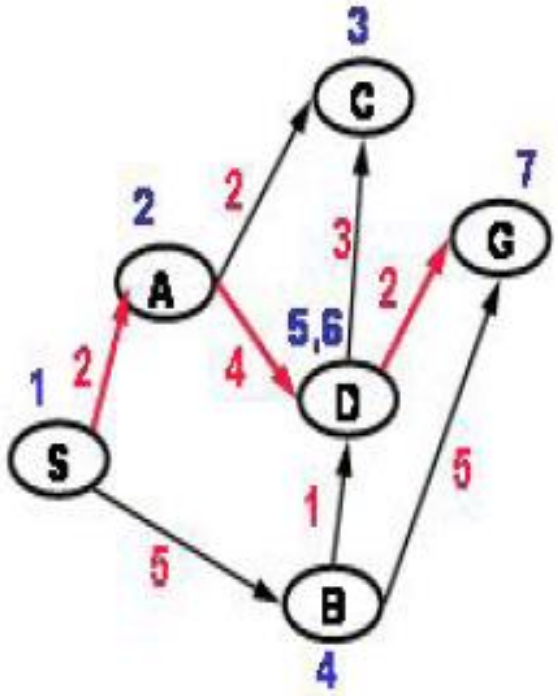
Repeated states

Failure to detect repeated states can turn a linear problem into an exponential one!

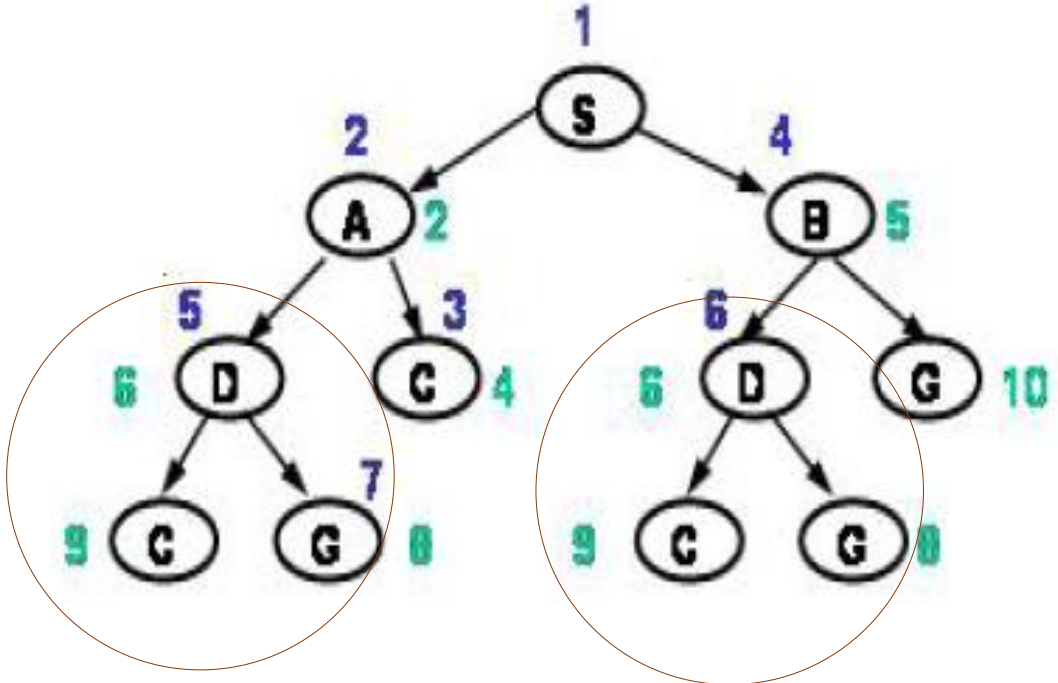




	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)
5	(6 D B S) (10 G B S) (6 D A S)
6	(8 G D B S) (9 C D B S) (10 G B S) (6 D A S)
7	(8 G D A S) (9 C D A S) (8 G D B S) (9 C D B S) (10 G B S)



Repeated subtrees are circled

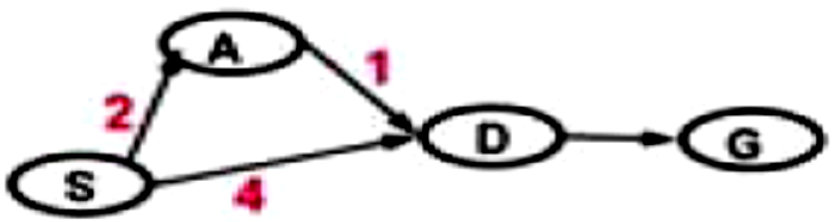


Use a Data Structure Visited as well as the usual Q

1. Initialize Q with start node S. Visited = {S}
2. If Q is empty, fail. Else pick a node N from Q //start of loop
3. If state(N) is Goal then exit loop with solution
4. Otherwise find all successors of state(N) not in Visited and add to Q and update the paths

Pros: Avoids repeated expansion of subtrees
No infinite paths because of loops
Cons: But does not give optimal solution. Example below:

- **Visited** : A state S is first visited when a path to S first gets added to the Fringe List Q. In general a state is said to have been visited if it has ever shown up in a search node in Q. The intuition is that we have briefly "visited" them to place them in Q, but we may have not yet generated their successors yet.



The path S-D-G is returned – not optimal

In addition to Visited we also maintain Expanded List - also known as Closed List.

- **Visited** : A state S is first visited when a path to S first gets added to the Fringe List Q. In general a state is said to have been visited if it has ever shown up in a search node in Q. The intuition is that we have briefly "visited" them to place them in Q, but we may have not yet generated their successors yet.

- **Expanded :** A state S is expanded when its successors have been generated and added to Q.

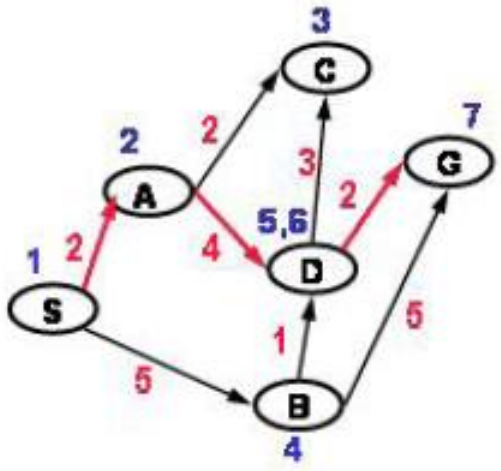
Graph Search Schema

Maintain a new Data Structure Closed in addition to Q.
Closed is also referred to as Expanded

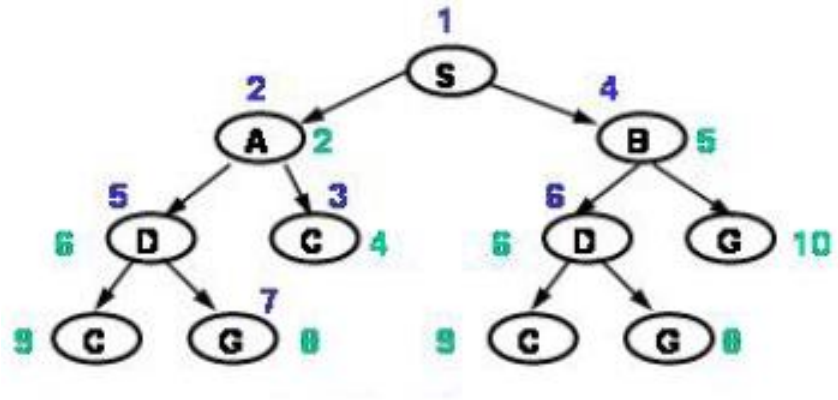
```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure

    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

For optimal solution we should only keep least cost paths in Q.

Optimization – Only insert into fringe only those successors not in closed list

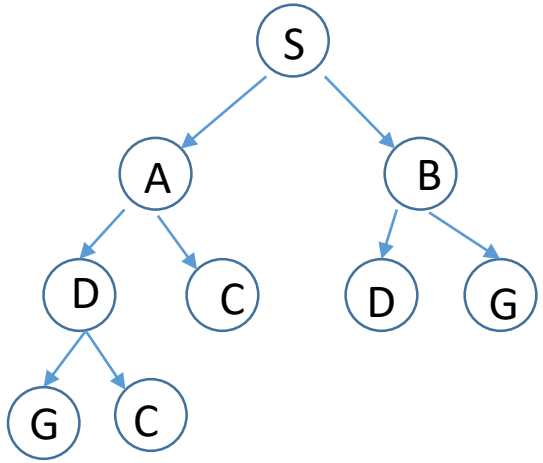


	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)
5	(6 D B S) (10 G B S) (6 D A S)
6	(8 G D B S) (9 C D B S) (10 G B S) (6 D A S)
7	(8 G D A S) (9 C D A S) (8 G D B S) (9 C D B S) (10 G B S)



Search Structure for Tree Search.
Subtree under D repeated

	Q	Expanded
1	(0 S)	
2	(2 A S) (5 B S)	S
3	(4 C A S) (6 D A S) (5 B S)	S,A
4	(6 D A S) (5 B S)	S,A,C
5	~~(6 D B S)~~ (10 G B S) (6 D A S)	S,A,C,B
6	(8 G D A S) ~~(9 C D A S) (10 G B S)~~	S,A,C,B,D



Search Structure for Graph Search
Subtree under D occurs only once