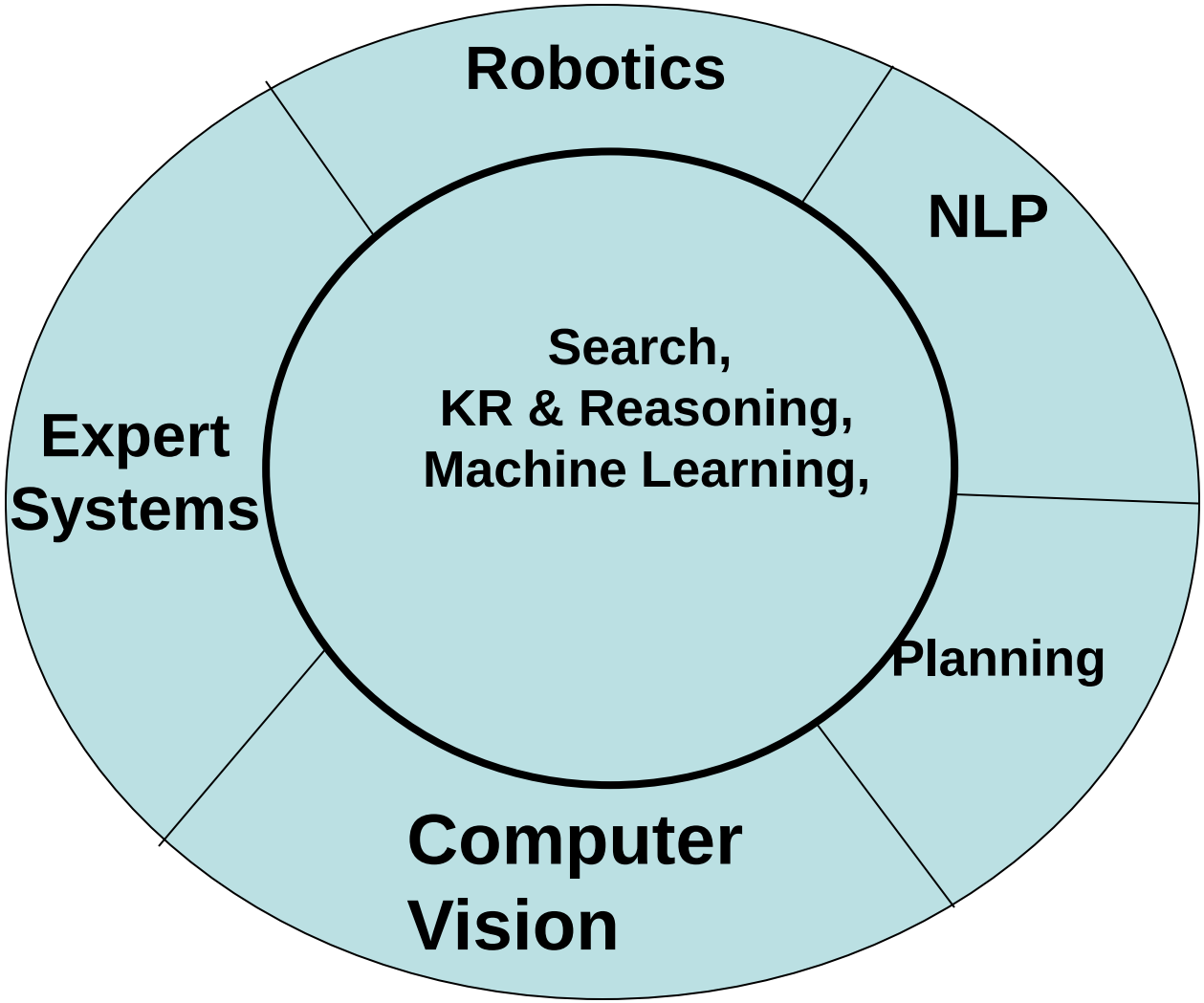
CS 561/571: Artificial Intelligence

Search

Outline

- Search Problems
- Search trees and state space graphs
- Uninformed search
  - Depth-first, Breadth-first, Uniform cost
  - Search graphs
- Informed search
  - Greedy search, A* search
  - Heuristics, admissibility
- Local search and optimization
  - Hill-climbing
  - Simulated annealing
  - Genetic algorithms

Disciplines which form the core of AI- inner circle
Fields which draw from these disciplines- outer circle.

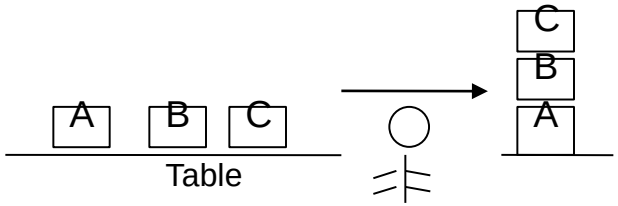


Search: Everywhere

Deciding on a sequence of actions to get to a goal

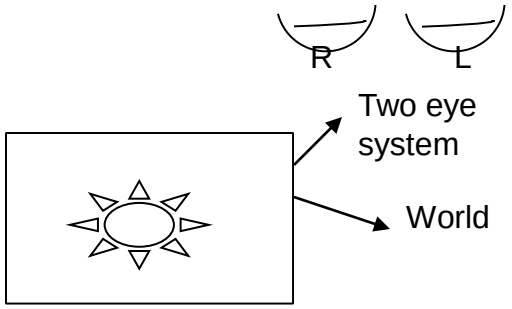
Planning

- (a) which block to *pick*, (b) which to *stack*, (c) which to *unstack*, (d) whether to *stack* a block or (e) whether to *unstack* an already stacked block. These options have to be searched in order to arrive at the right sequence of actions



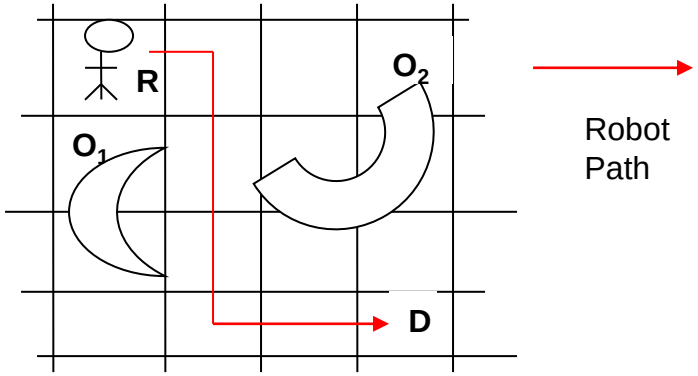
Vision

- A search needs to be carried out to find which point in the image of *L* corresponds to which point in *R*. Naively carried out, this can become an *O(n2)* process where *n* is the number of points in the retinal images.



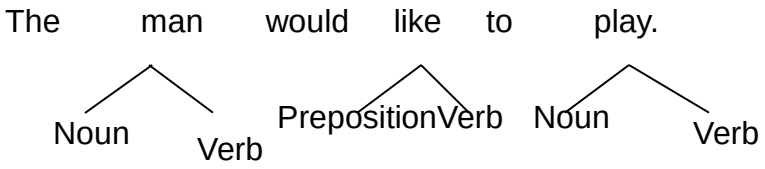
Robot Path Planning

- searching amongst the options of moving ***L***eft, ***R***ight, ***U***p or ***D***own. Additionally, each movement has an associated cost representing the relative difficulty of each movement. The search then will have to find the *optimal*, *i.e.*, the *least cost* path.



Natural Language Processing

- search among many combinations of parts of speech on the way to deciphering the meaning. This applies to every level of processing- *syntax, semantics, pragmatics* and *discourse*.



Expert Systems

Search among rules, many of which can apply to a situation:

If-conditions

the infection is primary-bacteremia
AND the site of the culture is one of the sterile sites
AND the suspected portal of entry is the gastrointestinal tract

THEN

there is suggestive evidence (0.7) that infection is bacteroid

(from MYCIN)

MYCIN *was an early expert system designed to identify bacteria causing severe infections, such as bacteremia and meningitis, and to recommend antibiotics*

Is Search Intelligent?

- **IBM's Deep Blue**
  - Beats Chess Master Kasparov in 1997, 2 wins, 1 loss, 3 draws
  - By Feng-Hsiung Hsu, Thomas Anantharaman, and Murray Campbell
  - Massively parallel, RS/6000 SP Thin P2SC-based system with 30-nodes, with each node containing a 120 MHz P2SC microprocessors for a total of 30, enhanced with 480 special purpose VLSI chess chips
  - Looks ahead at the consequences of its actions
  - Brute force, but simplified search space
  - Evaluates about 200 million positions per second
  - Minimax search

Details From Wikipedia

Is Search Intelligent?

When people express the opinion that human grandmasters do not examine 200,000,000 move sequences per second, I ask them, ``How do you know?'' The answer is usually that human grandmasters are not *aware* of searching this number of positions, or *are* aware of searching many fewer. But almost everything that goes on in our minds we are unaware of. I tend to agree that grandmasters are not searching the way Deep Blue does, but whatever they are doing would, if implemented on a computer, seem equally ``blind.'' Suppose most of their skill comes from an ability to compare the current position against 10,000 positions they've studied. (There is some evidence that this is at least partly true.) We call their behaviour insightful because they are unaware of the details; the right position among the 10,000 ``just occurs to them.'' If a computer does it, the trick will be revealed; we will see how laboriously it checks the 10,000 positions. Still, if the unconscious version yields intelligent results, and the explicit algorithmic version yields essentially the same results, then they will be intelligent, too.

ftp://ftp.cs.yale.edu/pub/mcdermott/papers/deepblue.txt

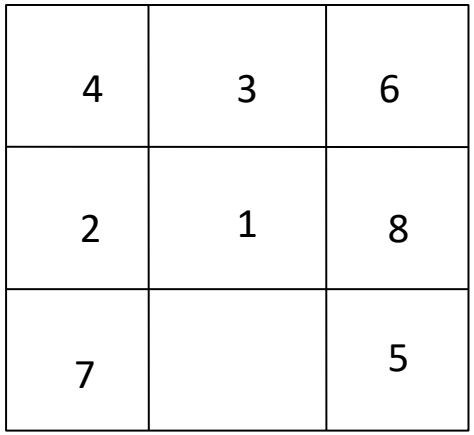
How Intelligent is Deep Blue?

Drew McDermott, Yale University

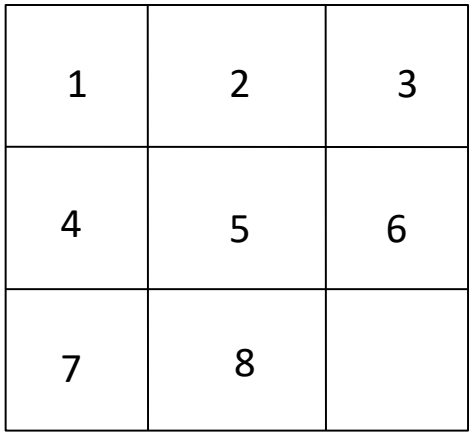
Search building blocks

- State Space: Graph of states (Express constraints and parameters of the problem)
- Operators : Transformations applied to the states
- Successor function: for a state x, it returns a set of <*action, successor*> ordered pairs, where each action is one of the legal actions in state *x* and each successor is a state that can be reached from *x* by applying the action
- Start state : S_0 (Search starts from here)
- Goal state : {*G*} - Search terminates here
- Cost : Effort involved in using an operator
- Optimal path : Least cost path

Examples (8 – puzzle)



S



G

Tile movement represented as the movement of the blank space.

Operators:

L : Blank moves left

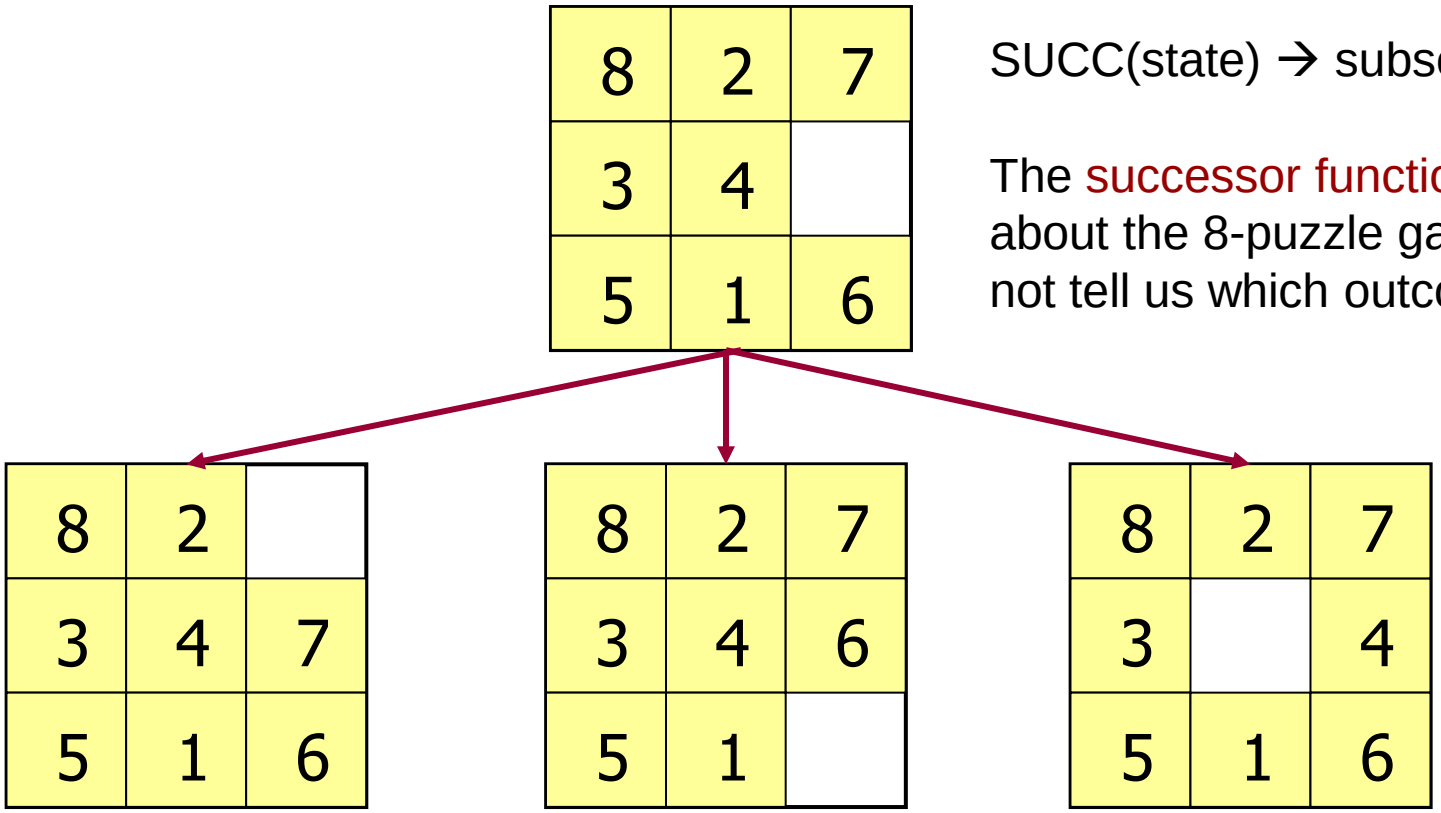
R : Blank moves right

U : Blank moves up

D : Blank moves down

$$C(L) = C(R) = C(U) = C(D) = 1$$

8-Puzzle: Successor Function



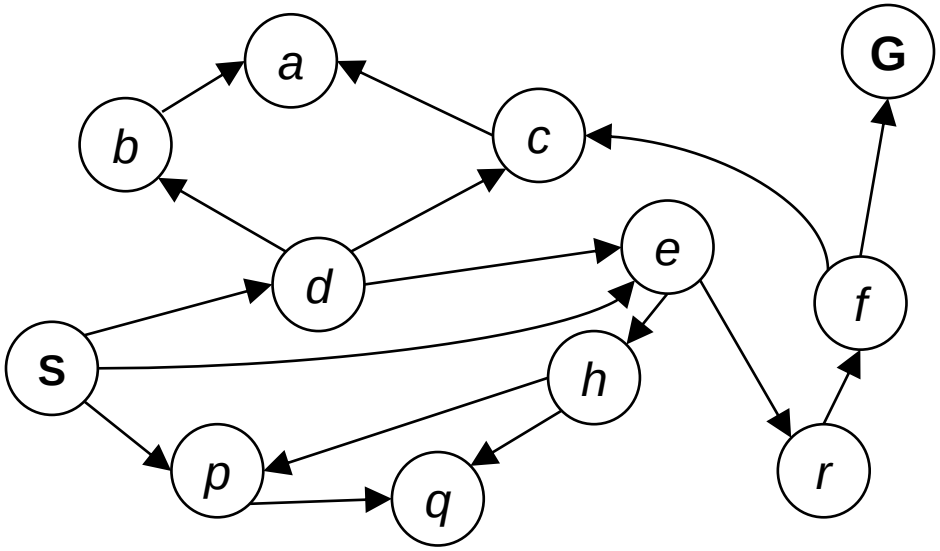
SUCC(state) → subset of states

The successor function is knowledge about the 8-puzzle game, but it does not tell us which outcome to use

Search is about the exploration of alternatives

State Space Graphs

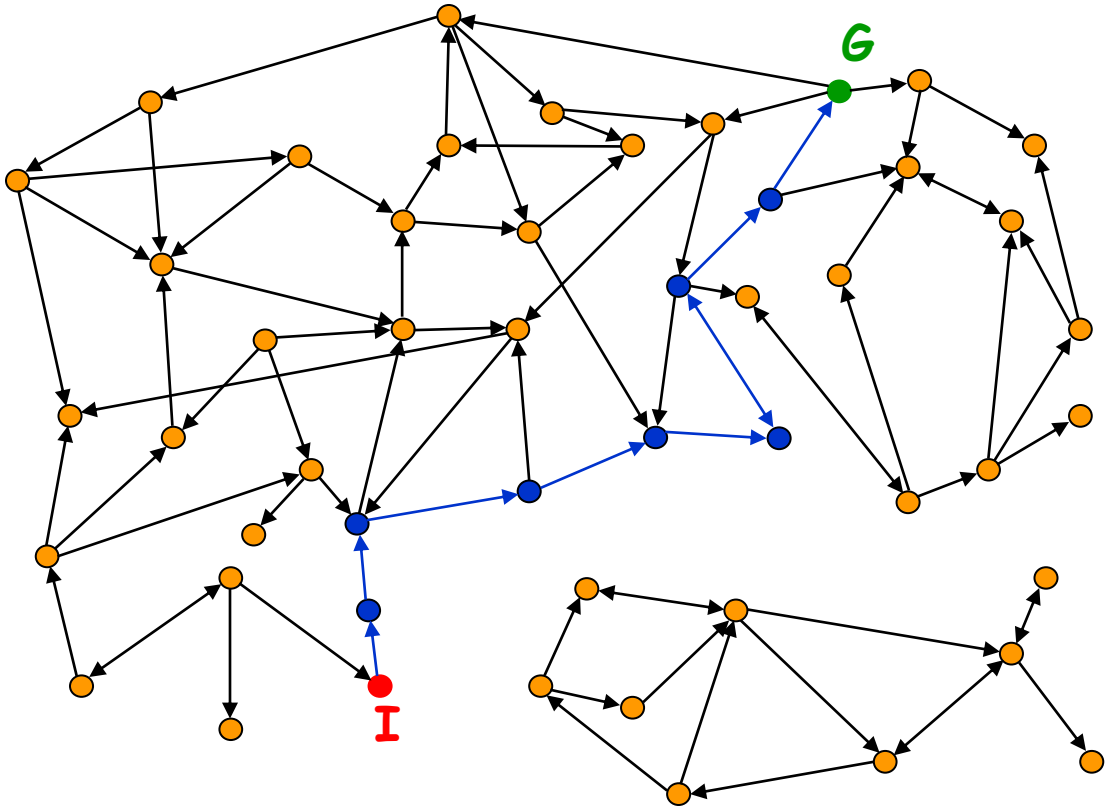
- State space graph: A mathematical representation of a search problem
  - For every search problem, there's a corresponding state space graph
  - The successor function is represented by arcs



- This can be large or infinite, so we won't create it in memory

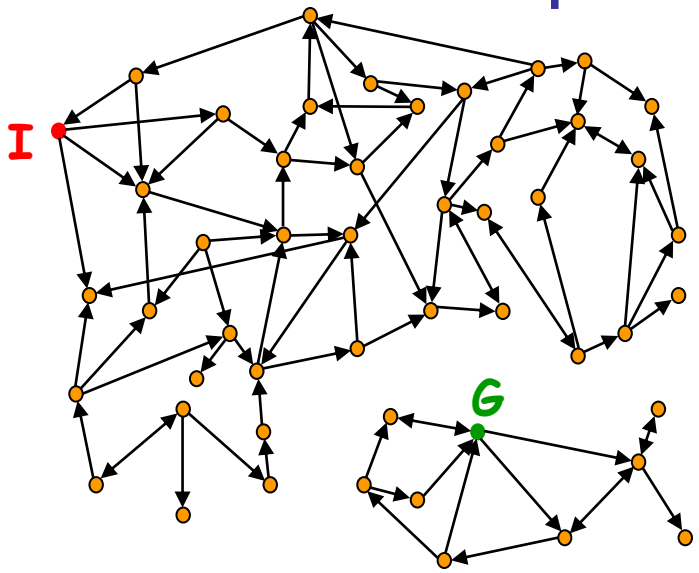
Solution to the Search Problem

- A solution is a path connecting the initial node to a goal node (any one)

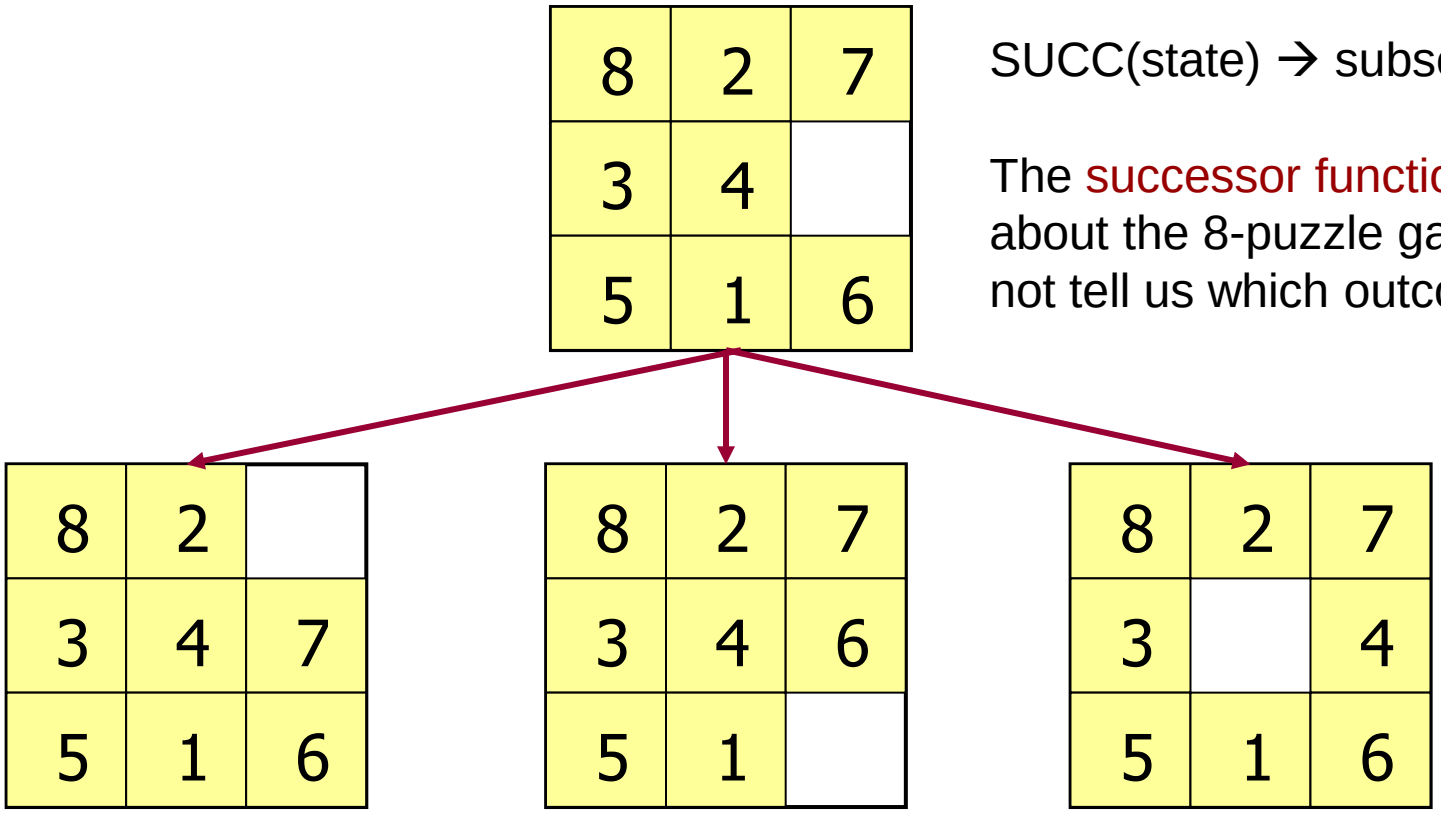


Solution to the Search Problem

- A solution is a path connecting the initial node to a goal node (any one)
- The cost of a path is the sum of the arc costs along this path
- An optimal solution is a solution path of minimum cost
- There might be no solution !



8-Puzzle: Successor Function



SUCC(state) → subset of states

The successor function is knowledge about the 8-puzzle game, but it does not tell us which outcome to use

Search is about the exploration of alternatives

How big is the state space of the (n^2-1)-puzzle?

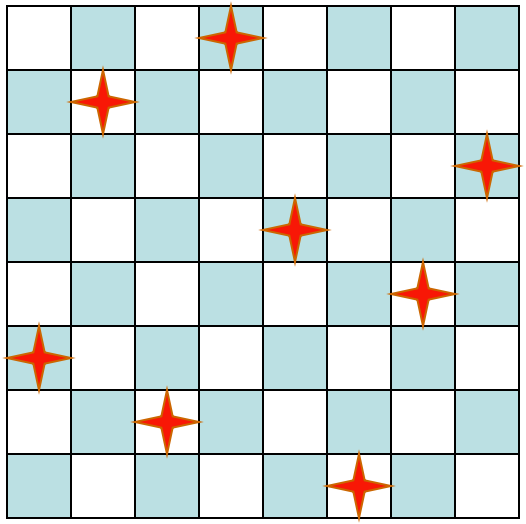
- 8-puzzle → 9! = 362,880 states
- 15-puzzle → 16! ~ 2.09 x 10^{13} states
- 24-puzzle → 25! ~ 10^{25} states

Only half of these states are reachable from any given state !

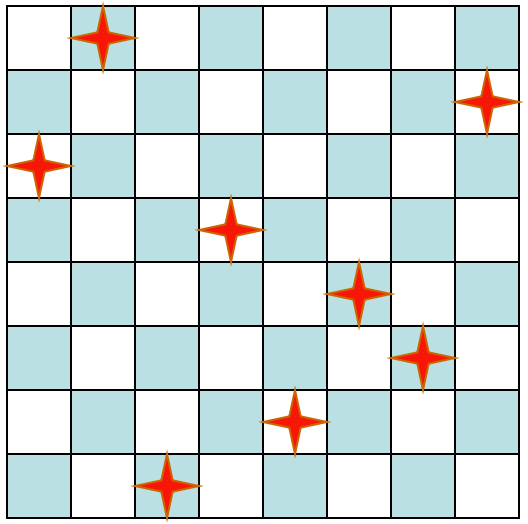
we don't know in advance !

8-Queens Problem

Place 8 queens in a chessboard so that no two queens are in the same row, column, or diagonal

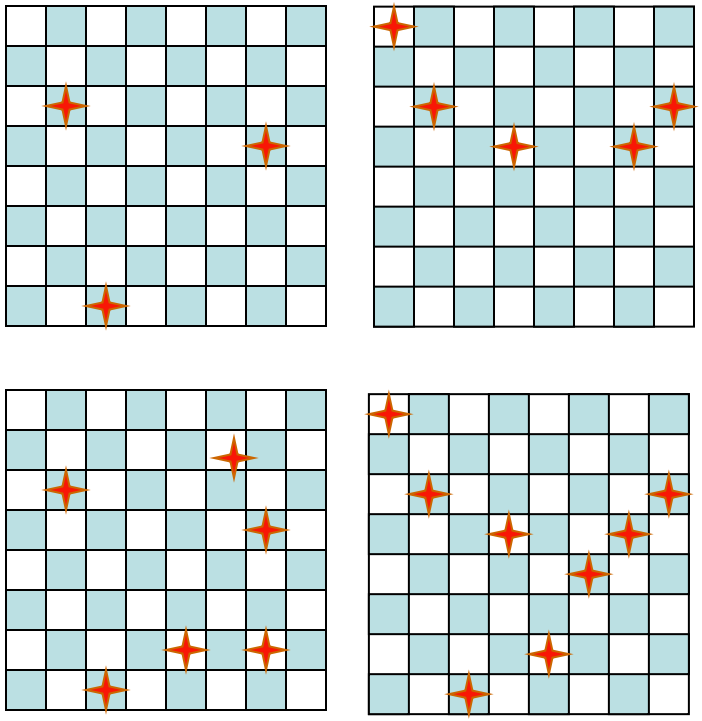


A solution



Not a solution

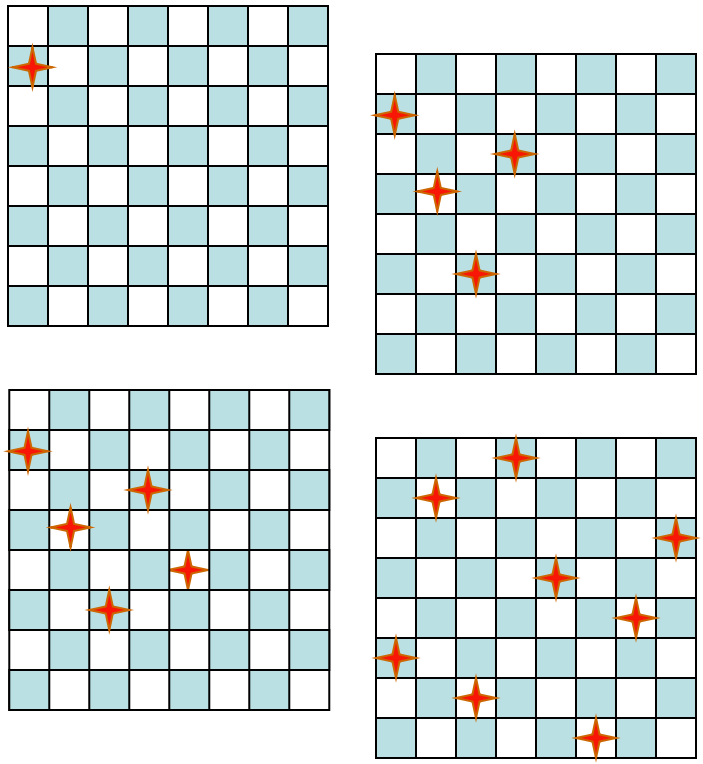
Formulation #1



- **States:** all arrangements of 0, 1, 2, ..., 8 queens on the board
- **Initial state:** 0 queens on the board
- **Successor function:** each of the successors is obtained by adding one queen in an empty square
- **Arc cost:** irrelevant
- **Goal test:** 8 queens are on the board, with no queens attacking each other

→ ~ 64x63x...x57 ~ 3×10^{14} states

Formulation #2



→ 2,057 states

- **States:** all arrangements of k = 0, 1, 2, ..., 8 queens in *the k leftmost columns with no two queens attacking each other*
- **Initial state:** 0 queens on the board
- **Successor function:** each successor is obtained by adding one queen in any square that is not attacked by any queen already in the board, in the leftmost empty column
- **Arc cost:** irrelevant
- **Goal test:** 8 queens are on the board

Representation of nodes

- State: the state in the state space
- Parent: node in the search tree that generated the node
- Action: action applied to the parent to generate the node
- Path-cost: cost of the path from the initial state to the node
- Depth: # steps along the path from the initial state

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
```

- Important ideas:
  - Frontier (aka fringe)
  - Expansion
  - Exploration strategy
- Main question: which frontier nodes to explore? (*search strategy*)

Implementation: general tree search

- Implemented as a queue
  - *MAKE-QUEUE (element, …):* creates a queue with given elements
  - *EMPTY?(queue):* returns true only if there are no more elements in the queue
  - *FIRST (queue):* return the first element in the queue
  - *REMOVE-FRONT (queue):* returns FIRST (queue) and removes from the queue
  - *INSERT (element, queue):* inserts an element into the queue and returns the resulting queue
  - *INSERT-ALL (elements, queue)*: inserts a set of elements into the queue and returns the resulting queue

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](STATE[node]) then return SOLUTION(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
```

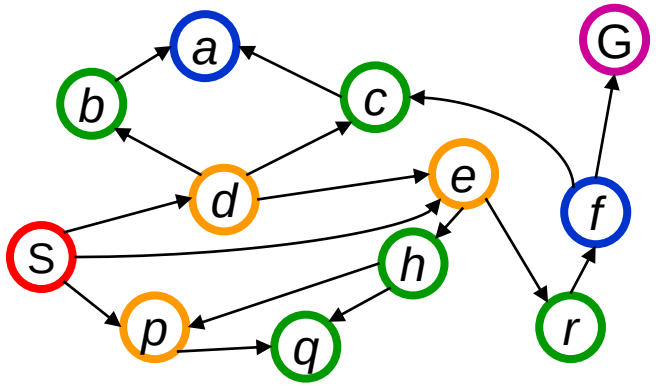
Uninformed vs. informed search

- Uninformed search (or, blind search)
  - Uses only the information available in the problem definition
  - Search through the space of possible solutions
  - Uses no knowledge about which path is likely to be best
  - E.g. BFS, DFS, uniform cost etc.
- Informed search (or, heuristic search)
  - Knows whether a non-goal state is more promising
  - E.g. Greedy search, A* Search etc.

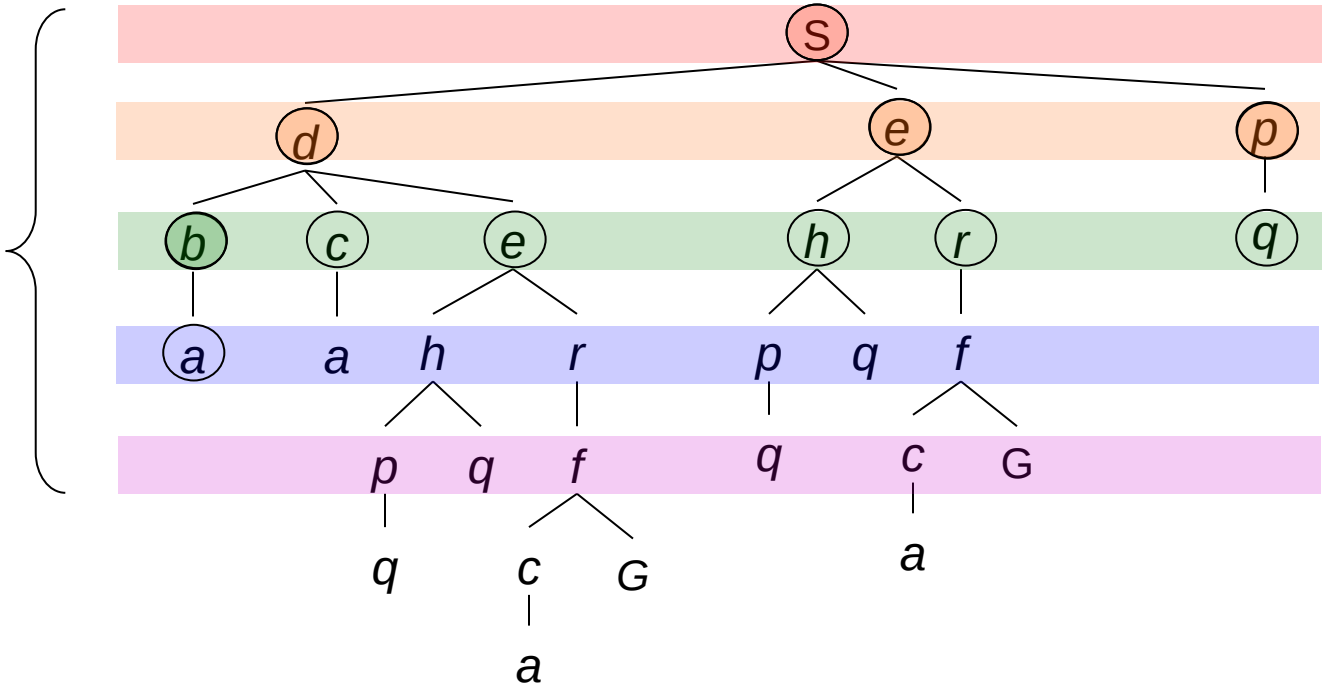
Breadth First Search

Strategy: expand shallowest node first

Implementation: Fringe is a FIFO queue



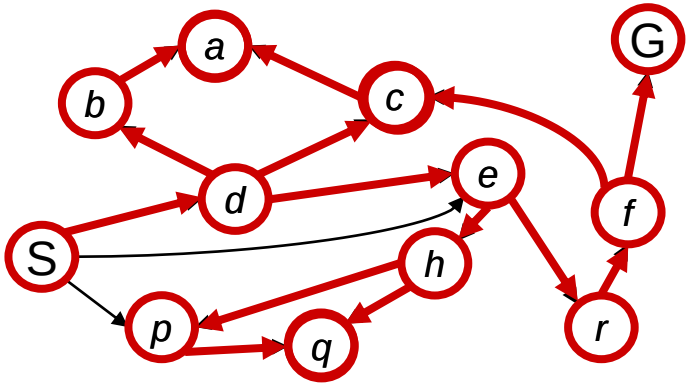
Search Tiers

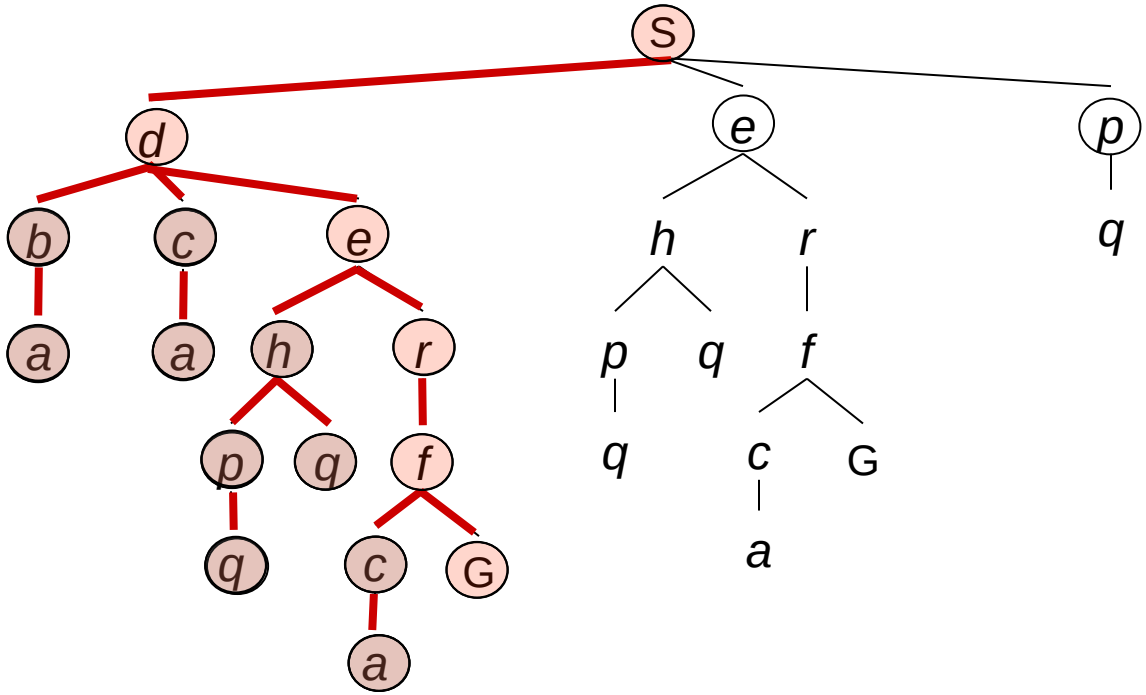


Depth First Search

Strategy: expand deepest node first

Implementation: Frontier is a LIFO stack





Comparisons

- When will BFS outperform DFS?
- When will DFS outperform BFS?

Search Algorithm Properties

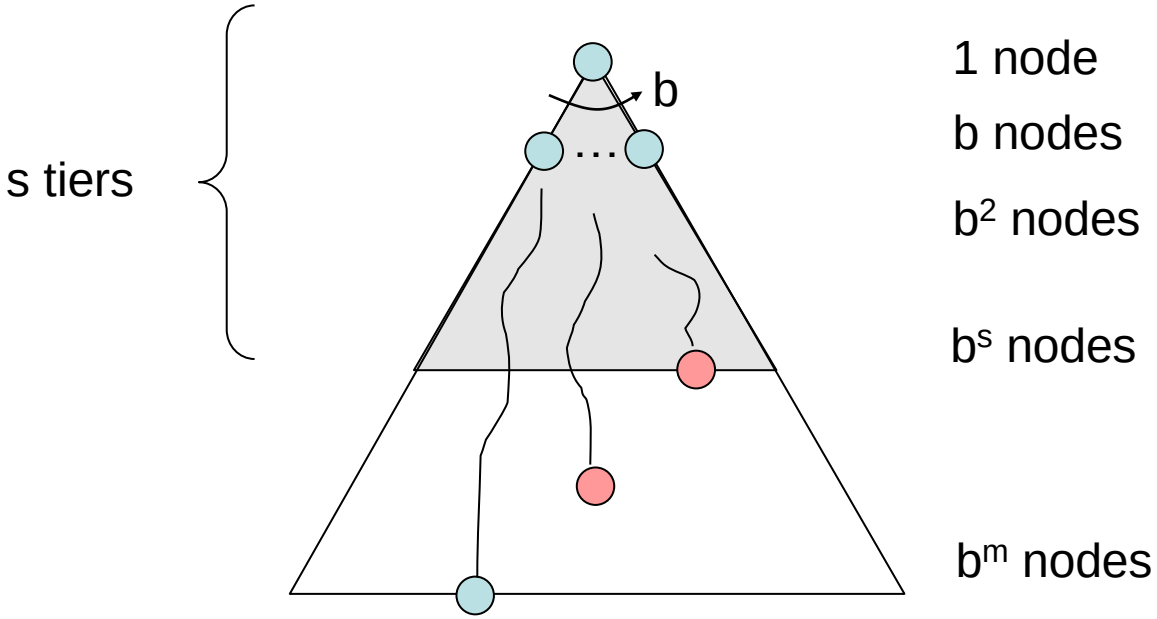
- Complete? Guaranteed to find a solution if one exists?
- Optimal? Guaranteed to find the least cost path?
- Time complexity? How long it takes to find a solution?
- Space complexity? How much memory required to perform search?

Variables:

n	Number of states in the problem
b	The average branching factor B (the average number of successors)
C^*	Cost of least cost solution
s	Depth of the shallowest solution
m	Max depth of the search tree

BFS

Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y	N	$O(b^{m+1})$	$O(bm)$
BFS		Y	N*	$O(b^{s+1})$	$O(b^{s+1})$



BFS

- When complete?
  - Shallowest goal node is at *d*, a finite depth
  - Will eventually find after expanding all the shallower nodes
    - *b* (branching factor) should be finite
- When optimal? (*when all step costs are equal*)
  - If path cost is a non-decreasing function of the depth of the node

Why BFS hard?

- Huge requirement of time and space !

- Time? $b+b^2+b^3+\ldots+b^s + b(b^s-1) = O(b^{s+1})$
- Space? $O(b^{s+1})$ (keeps every node in memory)

- Space is the bigger problem (more than time)

Why BFS hard?

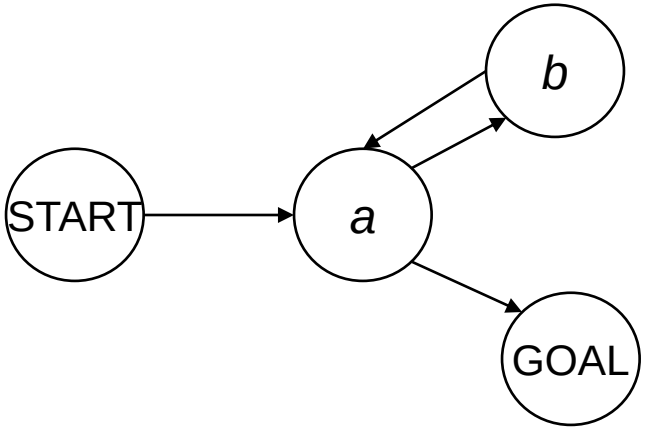
Assuming b=10, 10,000 nodes/sec, 1000 bytes/node

Depth of Solution	Nodes to Expand	Time	Memory
2	1100	.11 seconds	1 megabytes
4	111,100	11 seconds	106 megabytes
8	10^9	28 hours	1 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Un-informed search is not good for problems having exponential complexity

DFS

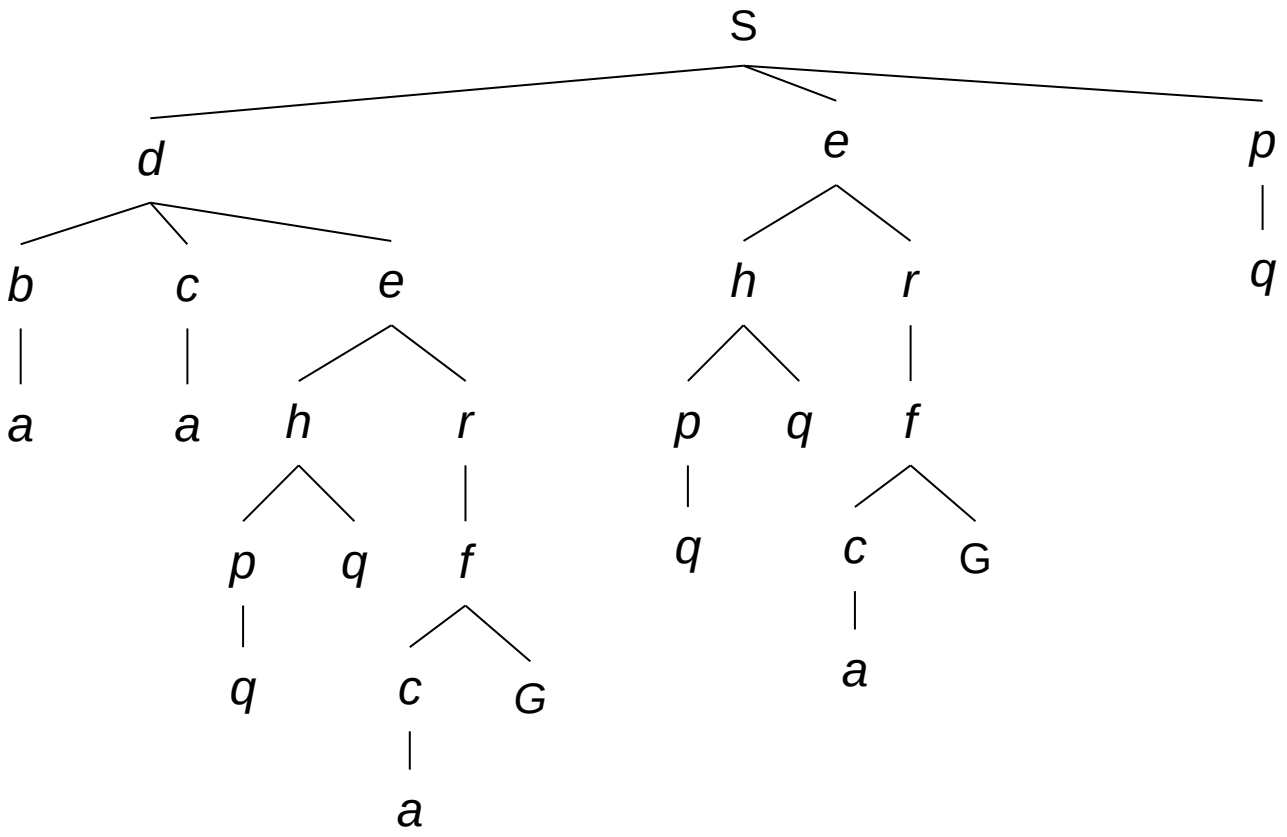
Algorithm		Complete	Optimal	Time	Space
DFS	Depth First Search	N	N	Infinite	Infinite



- Infinite paths make DFS incomplete…
- How can we fix this? (*checking of cycles!*)

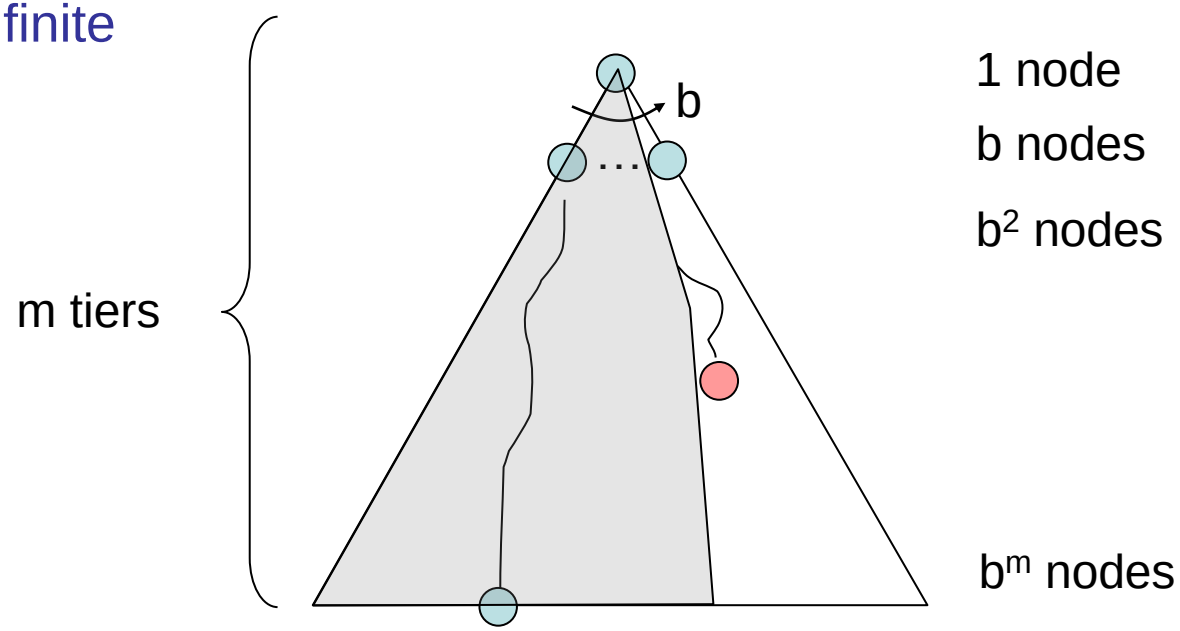
DFS

- Can make wrong choice and can go down infinite length
- Alternative choice would lead to a solution near the root
- Not optimal!



DFS

- With cycle checking and graph search (*avoid repeated state and redundant state*), DFS is complete
- d: finite



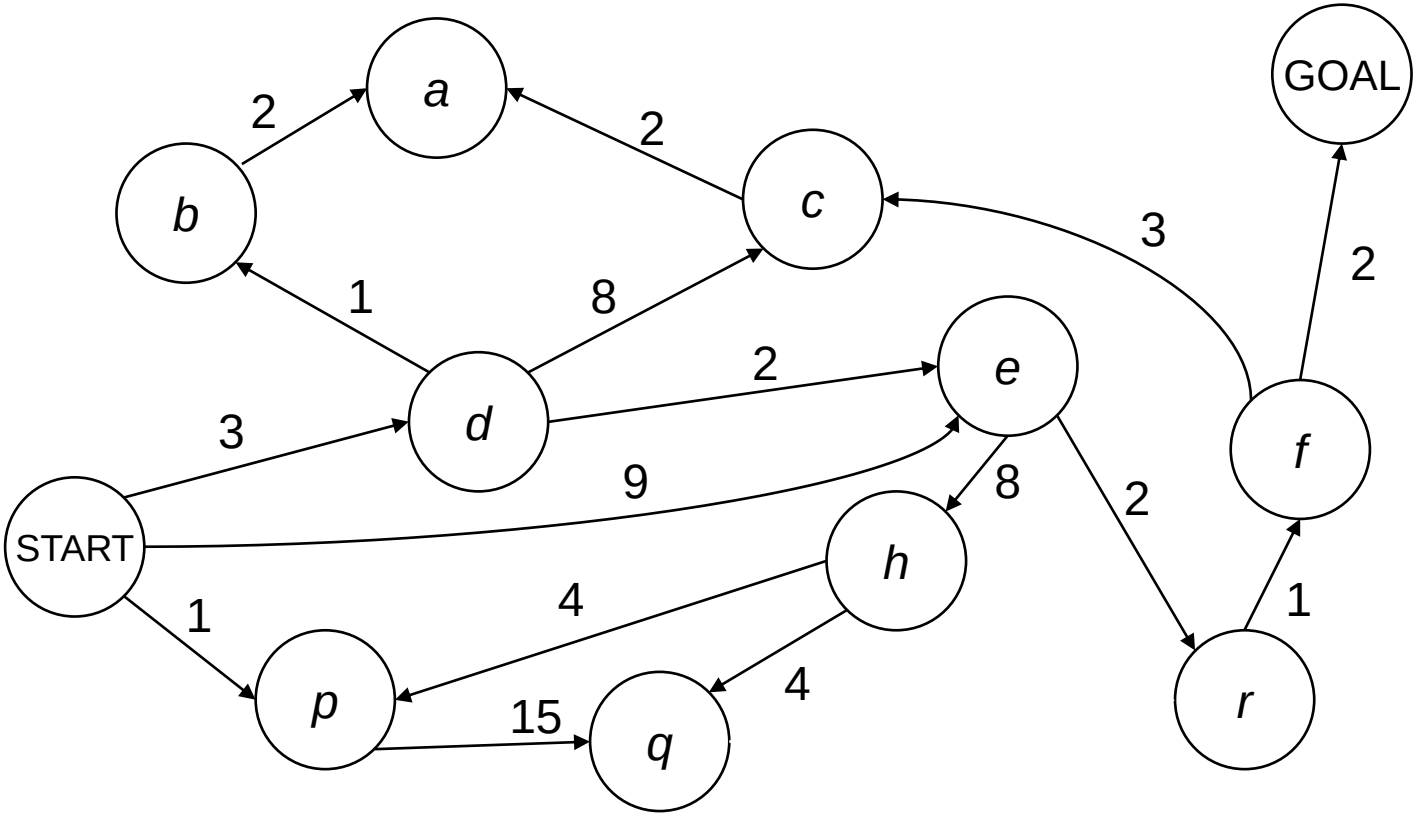
Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y	N	$O(b^{m+1})$	$O(bm)$

- When is DFS optimal? (*iterative deepening search*)

DFS with Backtracking

- Generate only one successor
- Each partially expanded node remembers which successor to explore next
- Undo modification when going back to generate next successor
- Space =O(m)
- Techniques are suitable for problems with large state descriptions (robotic path assembly)

Costs on Actions



Notice that BFS finds the shortest path in terms of number of transitions. It does not find the least-cost path.

We will quickly cover an algorithm which does find the least-cost path.

Uniform Cost Search

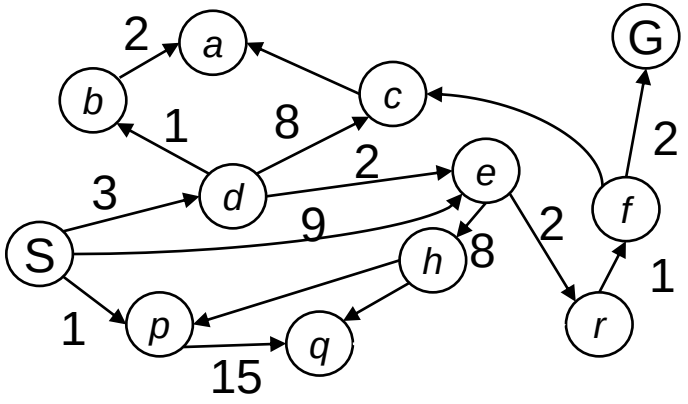
BFS: all step costs are equal

UCS: optimal with any step costs

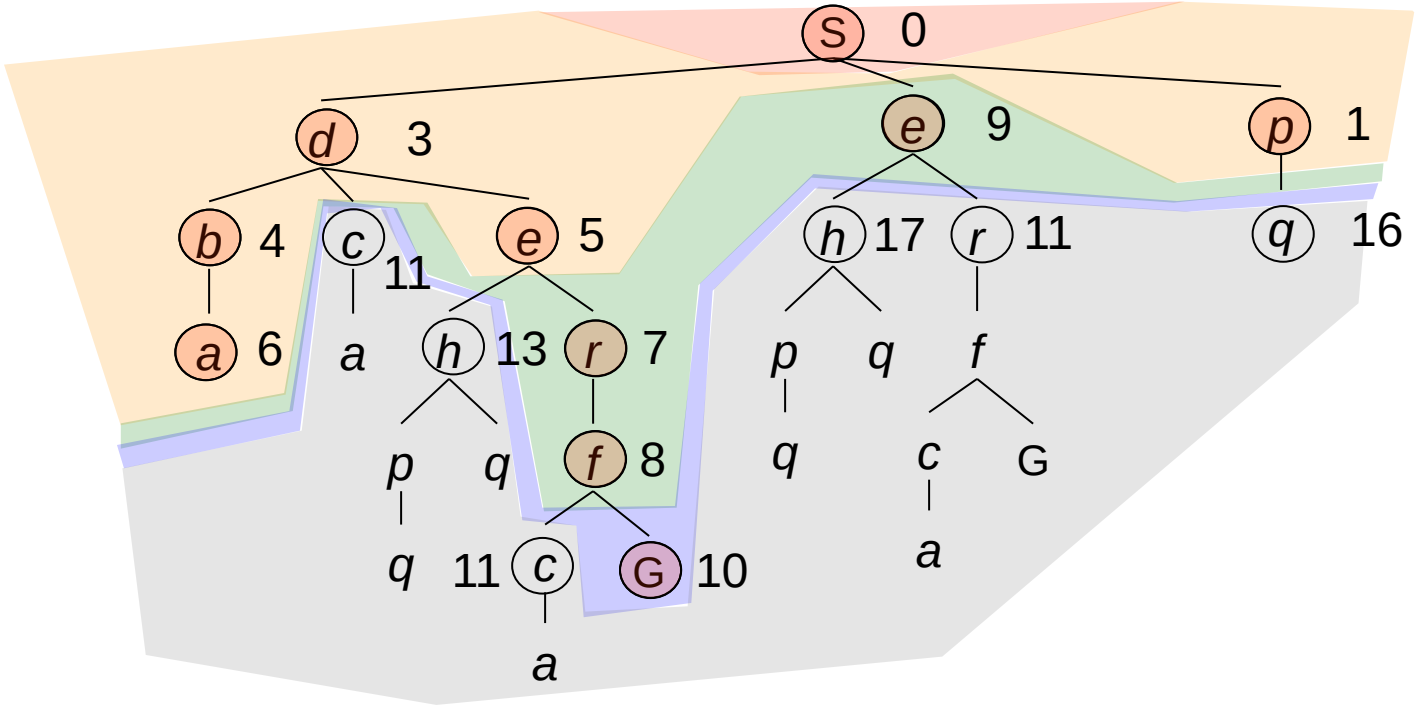
Expand cheapest node first:

Frontier is a priority queue

What about zero-cost action?



Cost contours



Uniform Cost Search

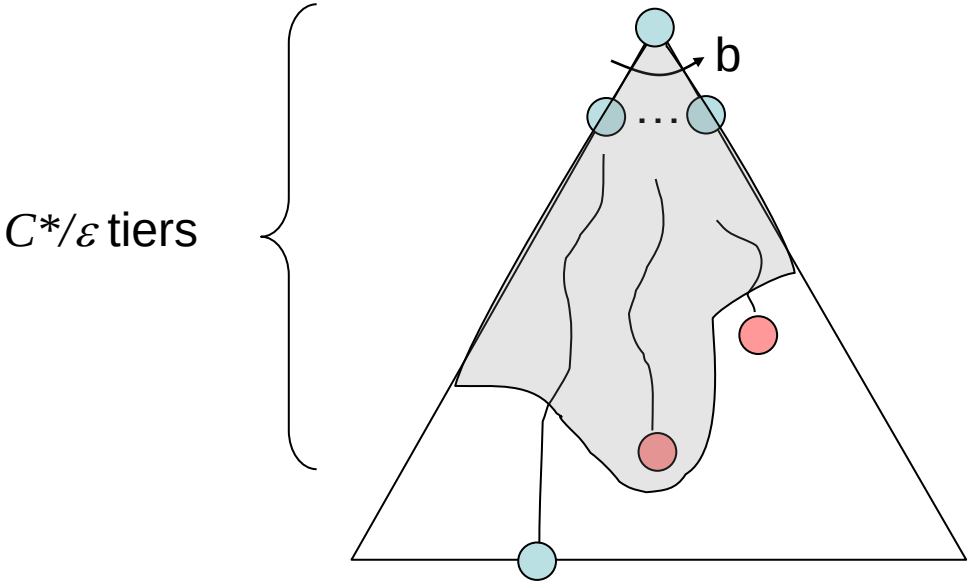
- Zero-cost action: leads back to the same state (*infinite loop!*)
- Cost must be greater than some small positive constant
  - *Sufficient for optimality and guarantees completeness*
  - Cost always increases as we proceed
- Algorithm always expands nodes in order of increasing path cost

Thus, first goal node always yields the optimal solution!

- *Guided by path costs rather than depths*
  - *Complexity can't be characterized by d and b*

Uniform Cost Search

Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y	N	$O(b^{m+1})$	$O(bm)$
BFS		Y	N	$O(b^{s+1})$	$O(b^{s})$
UCS		Y*	Y	$O(b^{C*/\varepsilon})$	$O(b^{C*/\varepsilon})$



** UCS can fail if actions can get arbitrarily cheap*

Depth-limited search

- Problem of unbounded trees
  - depth-first search with depth limit *L*, i.e., nodes at depth L treated to have no successors

```
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 SOLUTION(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
```

Depth limited search

- Incomplete if $L<d$ (d: depth of the solution)
- Time complexity = $O(b^L)$
- Space complexity = $O(bL)$
- For L=∞, depth-limited search=depth-first search

  Choice of L is important !!
- Route-finding problem
  - For 20 cities, L=19 is a possible choice (if there is any feasible solution)
  - But, with 9 (a, better depth limit) steps can be reached!
    - Diameter of search space (***not possible to predict until we solve the problem***)

Iterative Deepening Search

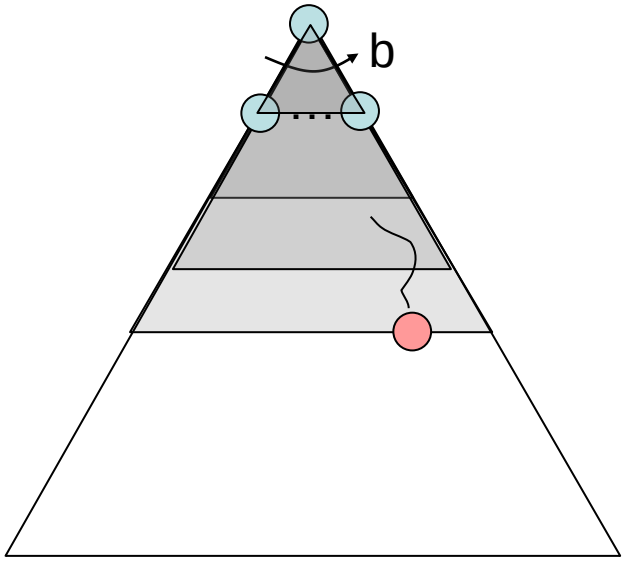
- General strategy often used in combination with DFS
  - Finds best depth limit
- Combines benefits of both BFS and DFS
  - Like DFS, memory is modest: $O(bd)$
  - Like BFS, complete when b is finite
  - Like BFS, optimal when path cost is an increasing function of the depth of a node

Iterative Deepening

Iterative deepening uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less.
2. If "1" failed, do a DFS which only searches paths of length 2 or less.
3. If "2" failed, do a DFS which only searches paths of length 3 or less.

….and so on.



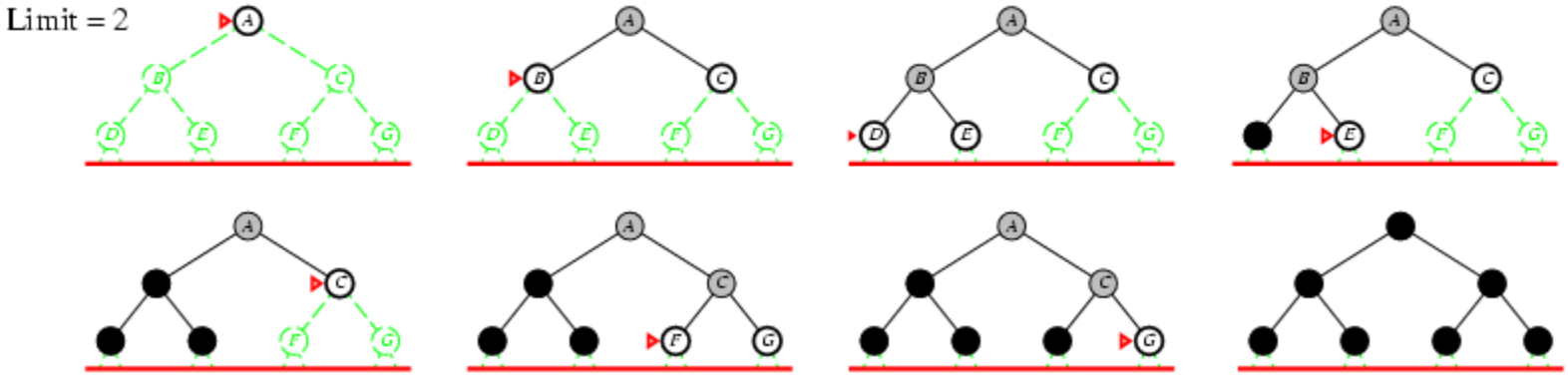
Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y	N	$O(b^{m+1})$	$O(bm)$
BFS		Y	N*	$O(b^{s+1})$	$O(b^{s})$
ID		Y	N*	$O(b^{s+1})$	$O(bs)$

Iterative deepening search / =0

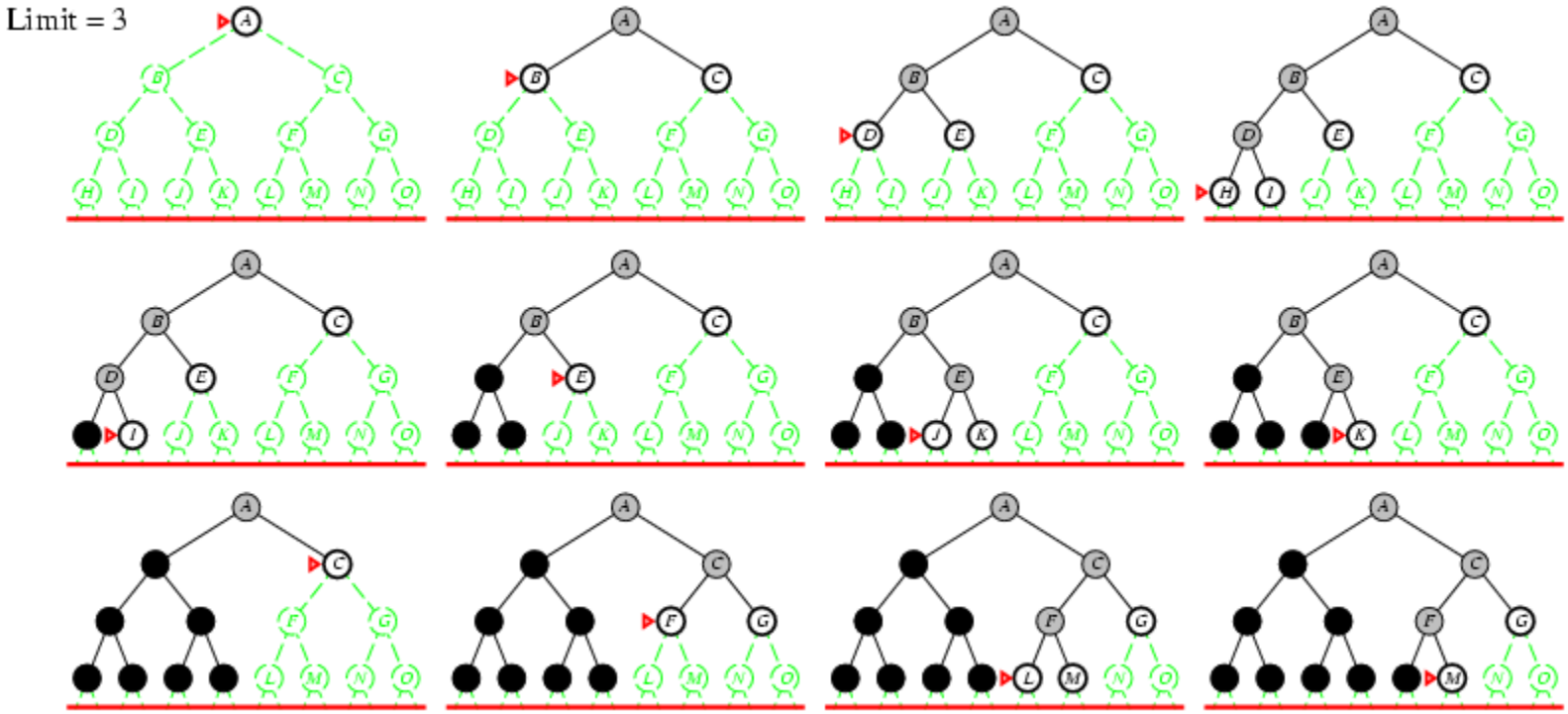
Limit = 0

Iterative deepening search / =1

Iterative deepening search l =2



Iterative deepening search / =3



Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

Iterative deepening search

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

$$N_{IDS} = d\, b^1 + (d-1)\, b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For *b = 10*, *d = 5*,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}450$
  - $N_{BFS} = 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 + 999{,}990 = 1{,}111{,}100$

- Overhead = (123,450 - 111,111)/111,111 = 11% (approx.)

Properties of iterative deepening search

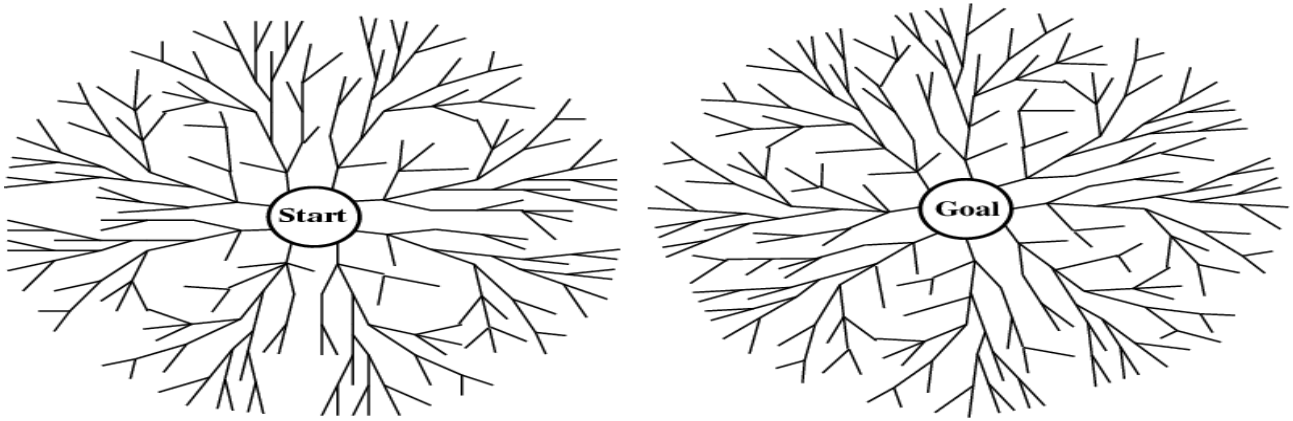
- Complete? Yes
- Time? $d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1
- Wasteful as states generated multiple times
- Not very costly
  - With the same branching factor at each level, most of the nodes are at the bottom
  - Upper level nodes don't cost much

When to use what?

- **Depth-First Search:**
  - Many solutions exist
  - Know (or have a good estimate of) the depth of solution
- **Breadth-First Search**:
  - Some solutions are known to be shallow
- **Uniform-Cost Search**:
  - Actions have varying costs
  - Least cost solution is the required

  This is the only uninformed search that worries about costs
- **Iterative-Deepening Search**:
  - Large search space and depth of the solution is not known

Bi-directional search



- Alternate searching from the start state toward the goal and from the goal state toward the start
- Stop when the frontiers intersect
- Works well only when there are unique start and goal states and when actions are reversible
- Can lead to finding a solution more quickly

Bi-directional search

- Motivation: $b^{d/2} + b^{d/2} < b^d$
- E.g. if solution depth =6 and each direction runs BFS
  - Worst case: two searches meet when each has expanded all the nodes at 3
  - For b=10,

    $b+b^2+b^3 = 10+100+1000=1110 *2=2220$

    $b^2+b^3+b^{4+} b^5+b^6+b^7= 111,111,00$
- One tree kept in memory to check membership
- Space complexity = $O(b^{d/2})$
- Time complexity = $O(b^{d/2})$

Bi-directional search

- Complete
  - If b is finite
  - If both directions use BFS (*check why not DFS?*)
- Optimal
  - If step costs are all identical
  - If both directions use BFS