Uninformed/Blind Search

BBM 405 – Fundamentals of Artificial Intelligence

Pinar Duygulu

Hacettepe University

Slides are mostly adapted from AIMA, MIT Open Courseware,

Manuela Veloso(CMU) and

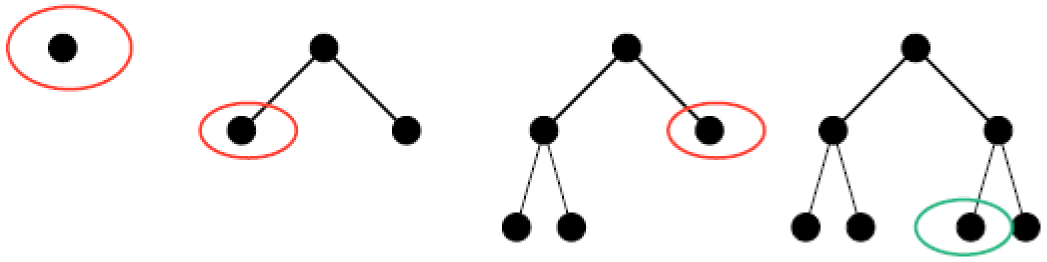
Svetlana Lazebnik (UIUC)

Uninformed search strategies

- A search strategy is defined by picking the order of node expansion
- Uninformed search (blind search) strategies use only the information available in the problem definition
- **Can only distinguish goal states from non-goal state**
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Breadth First Search

```
**function** BREADTH-FIRST-SEARCH (*problem*) **returns** a solution or failure
  **return** GENERAL-SEARCH (*problem*, ENQUEUE-AT-END)
```

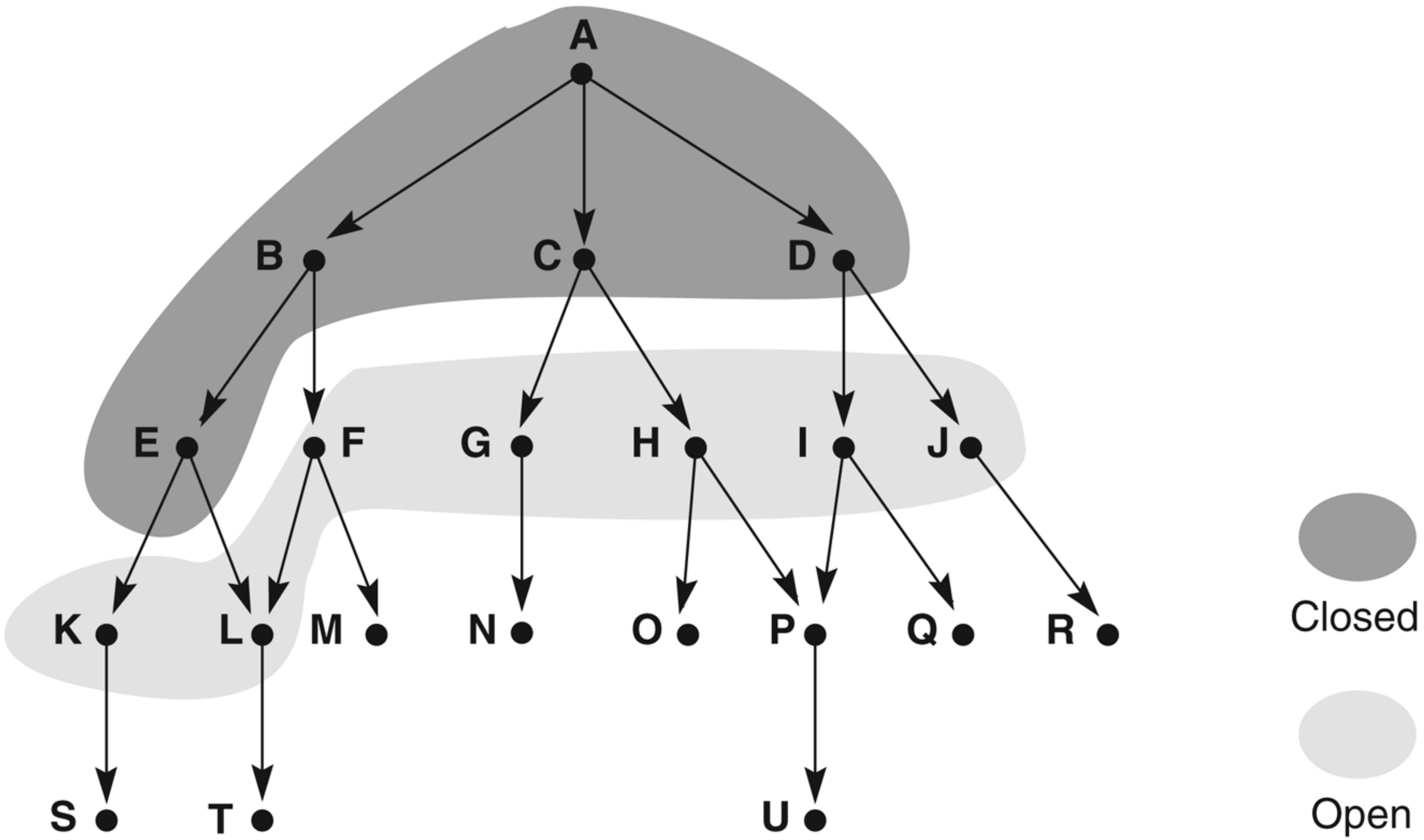


Breadth-first search tree after 0,1,2 and 3 node expansions

Expand: red nodes; Goal: green node

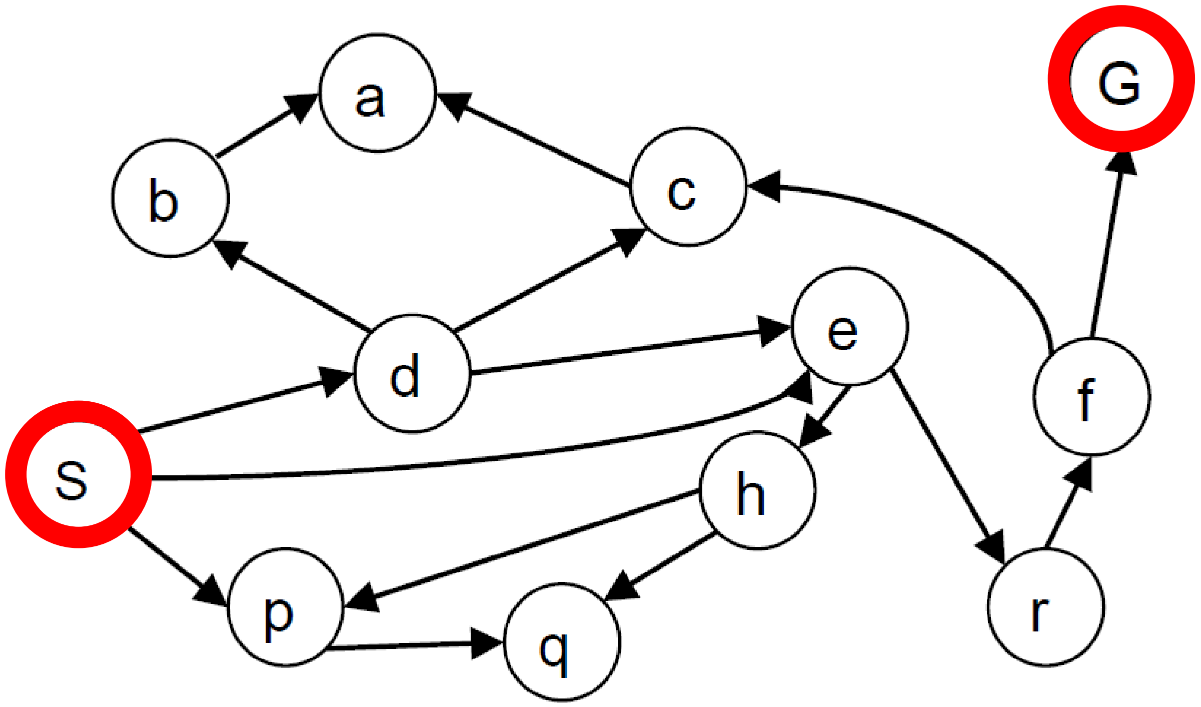
In this example, b = 2, and d = 2

Breadth-first search



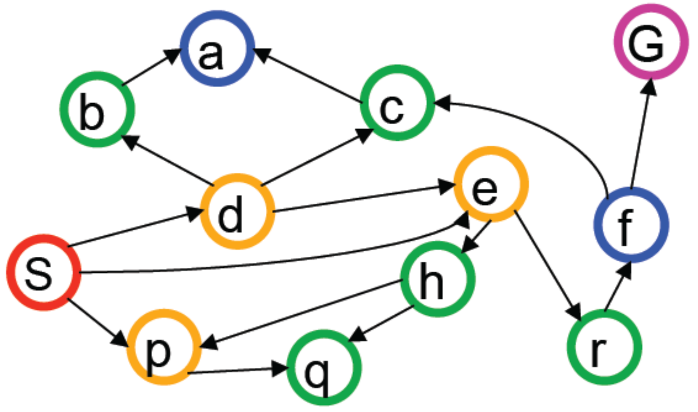
Breadth-first search

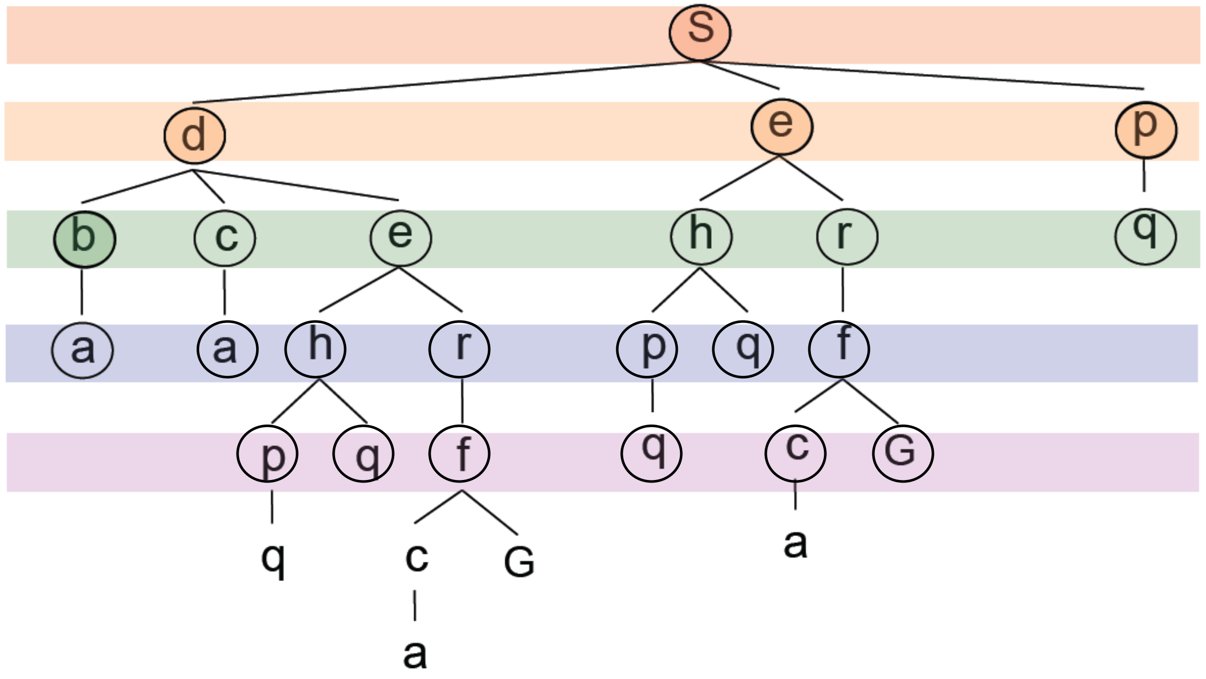
- Expand shallowest unexpanded node
- Implementation: *frontier* is a FIFO queue



Breadth-first search

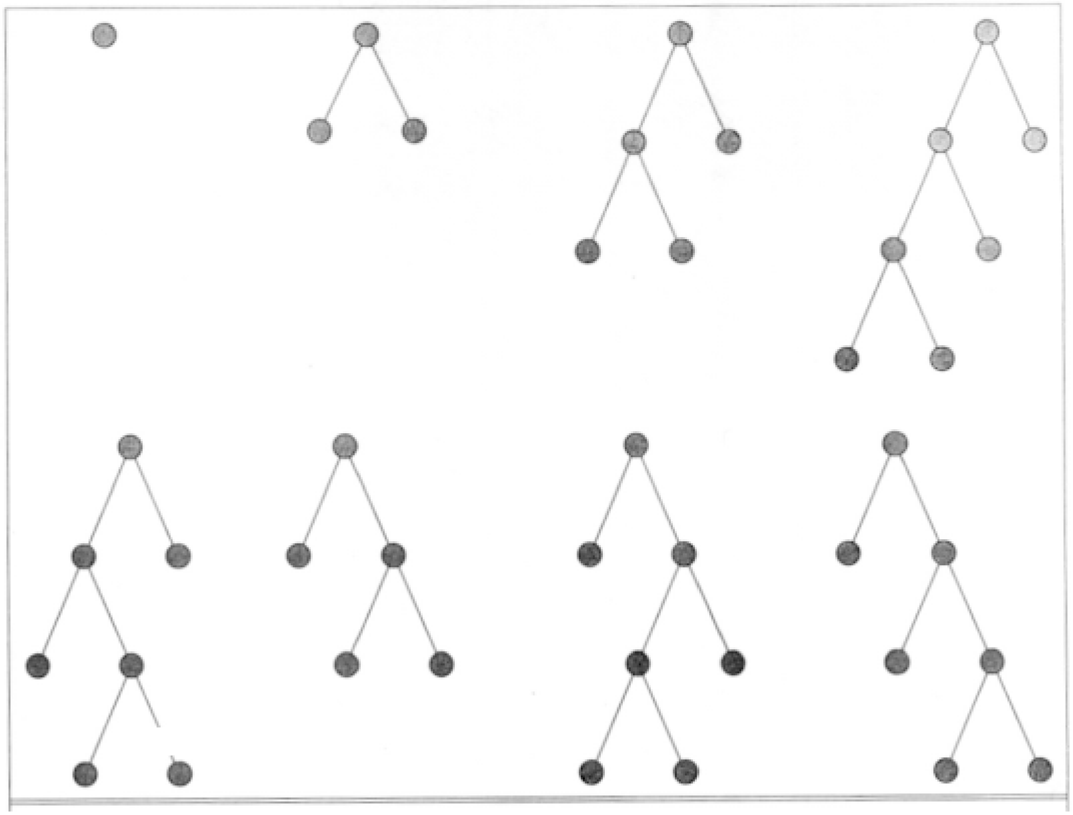
- Expansion order: (S,d,e,p,b,c,e,h,r,q,a,a, h,r,p,q,f,p,q,f,q,c,G)



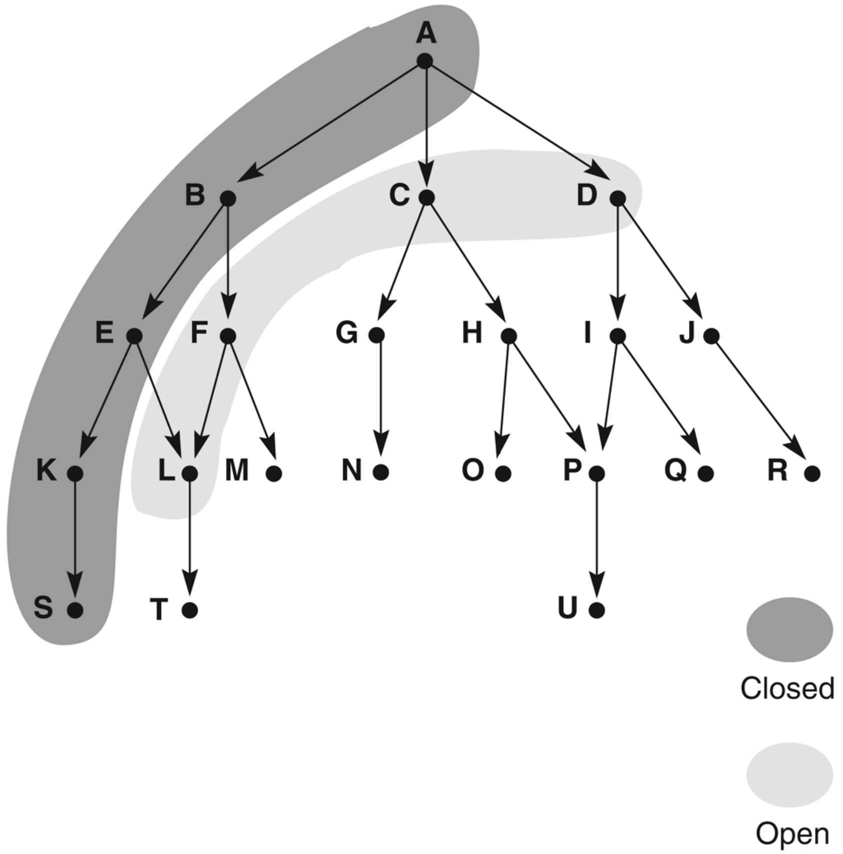


```
function DEPTH-FIRST-SEARCH (problem) returns a solution or failure
    GENERAL-SEARCH (problem, ENQUEUE-AT-FRONT)
```

Alternatively can use a recursive implementation.

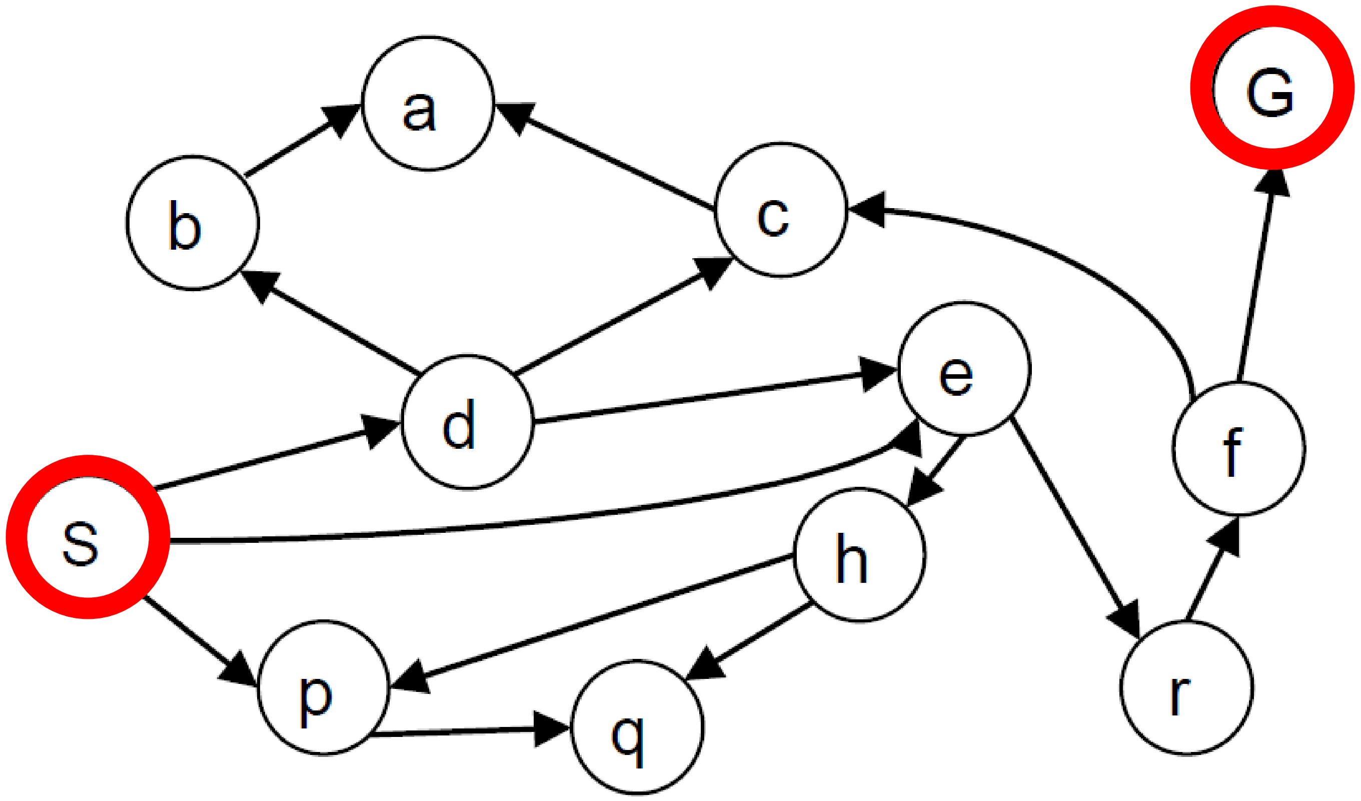


Depth-first search



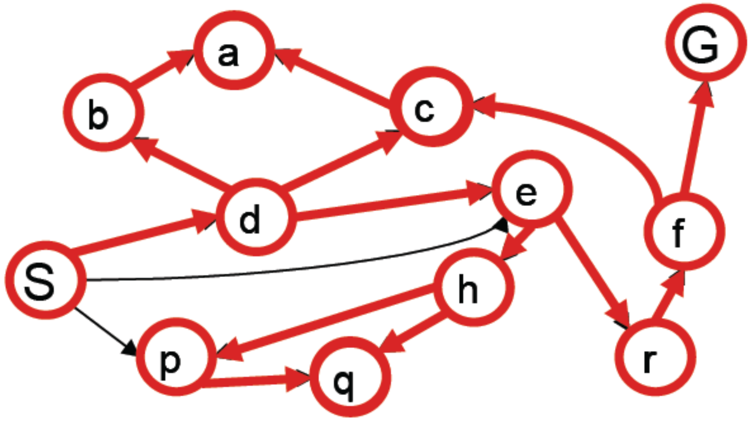
Depth-first search

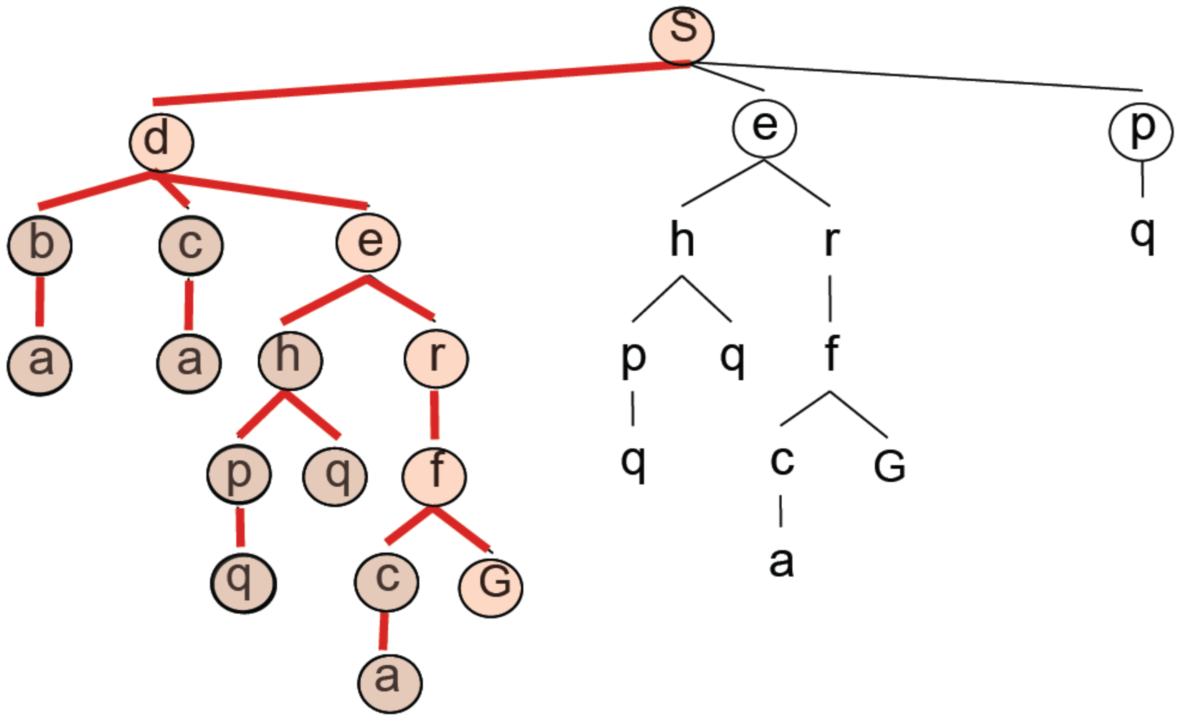
- Expand deepest unexpanded node
- Implementation: *frontier* is a LIFO queue



Depth-first search

- Expansion order: (S,d,b,a,c,a,e,h,p,q,q, r,f,c,a,G)





Analysis of Search strategies

- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - 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 ∞)

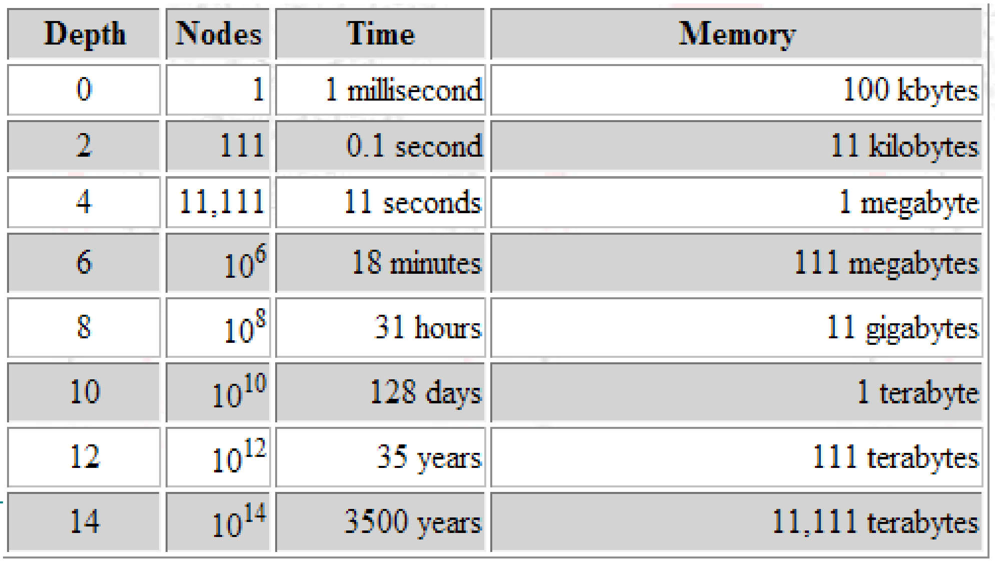
Properties of breadth-first search

- Complete? Yes (if *b* is finite)
- Time?

  Number of nodes in a *b*-ary tree of depth *d*:

  (*d* is the depth of the optimal solution)

  $1+b+b^2+b^3+\ldots+b^d = O(b^d)$
- Space? $O(b^d)$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)
- Space is the bigger problem (more than time)



Depth	Nodes	Time	Memory
0	1	1 millisecond	100 kbytes
2	111	0.1 second	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
    - → complete in finite spaces
- Time?

  Could be the time to reach a solution at maximum depth *m*: $O(b^m)$

  Terrible if *m* is much larger than *d*

  But if there are lots of solutions, may be much faster than BFS
- Space? *O(bm)*, i.e., linear space!
- Optimal? No - – returns the first solution it finds

Depth-limited search

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

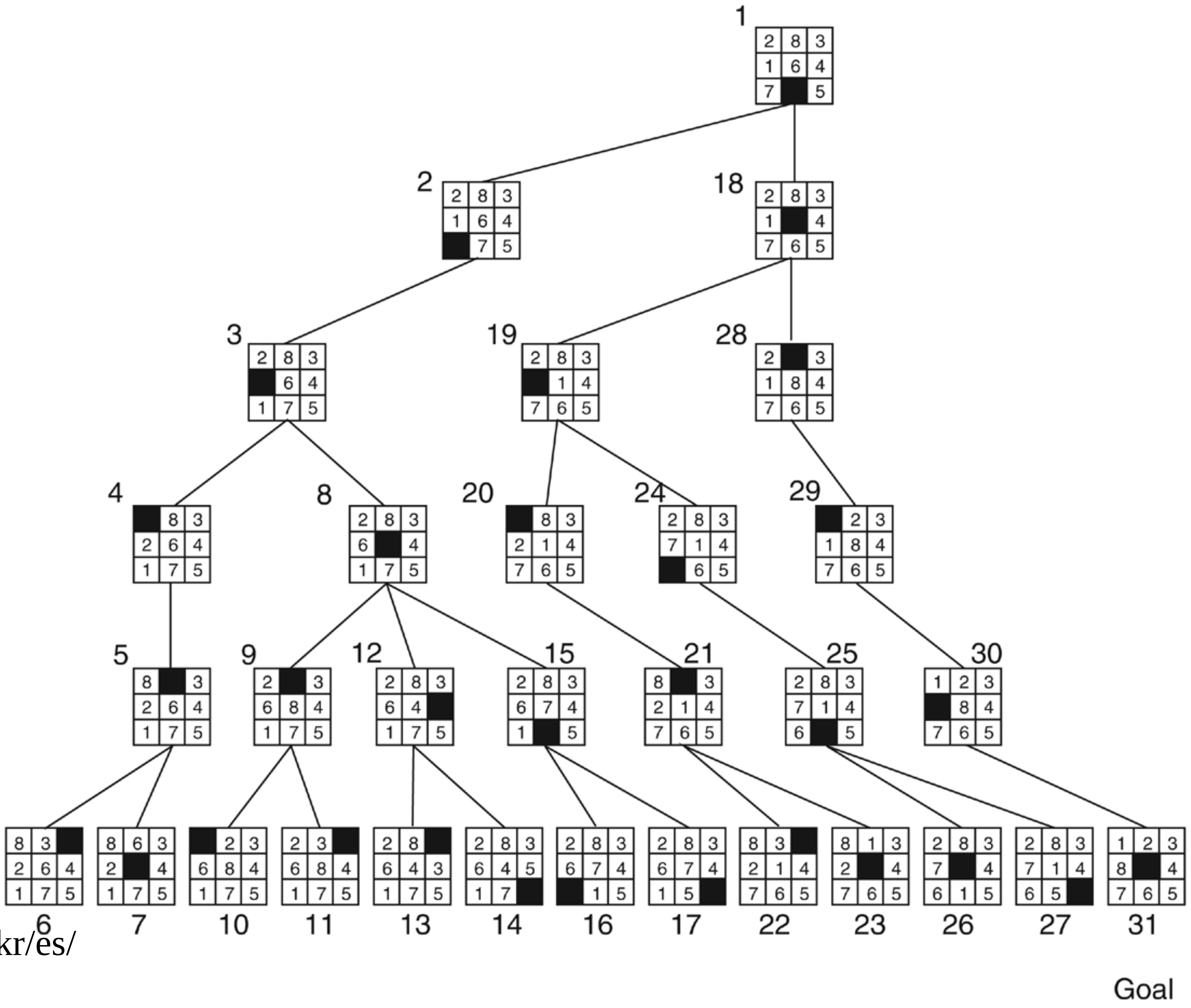
- Recursive implementation:

```
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 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
```

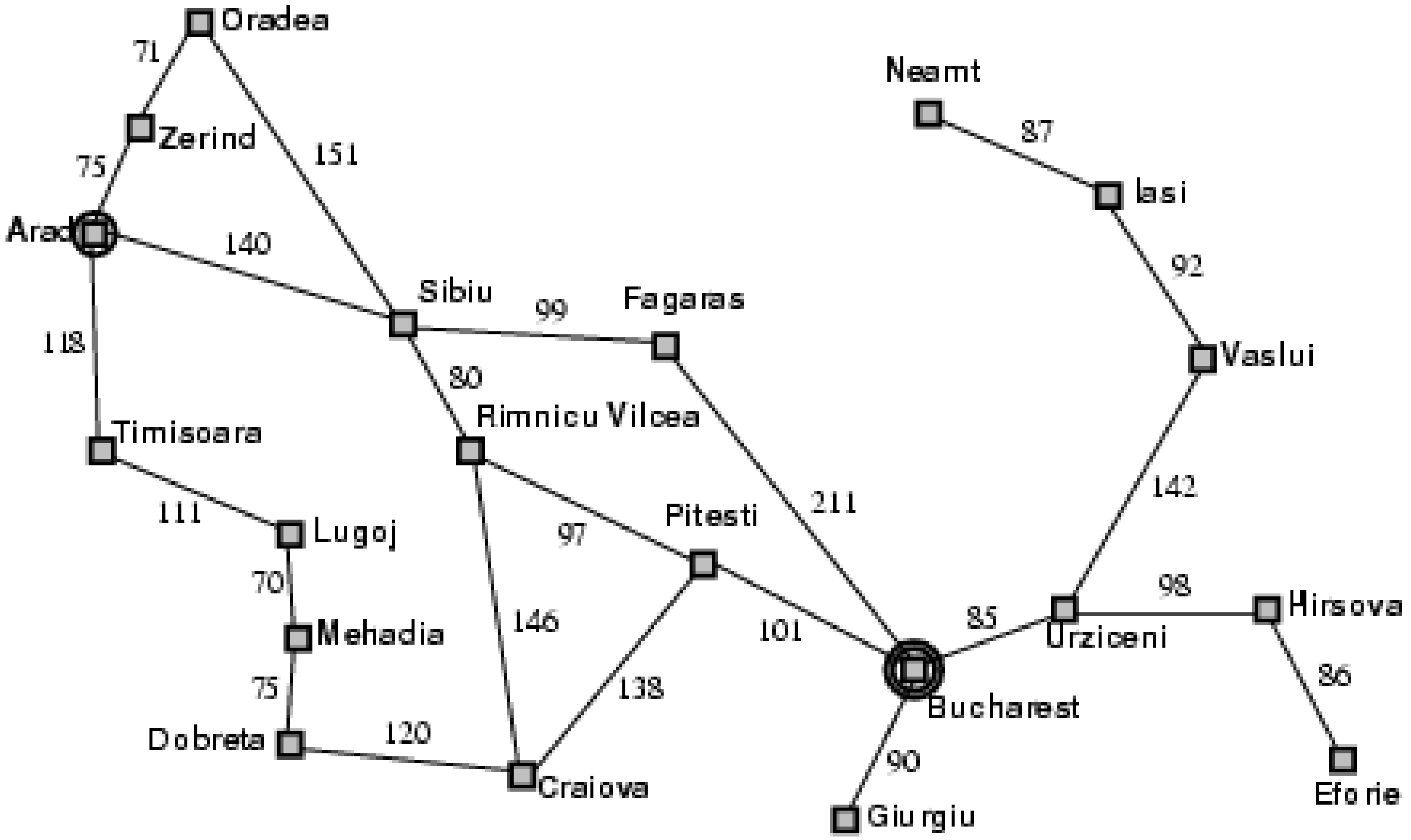
Depth limited search

Depth-first search of the 8-puzzle with a depth bound of 5.



Taken from http://iis.kaist.ac.kr/es/

Example: Romania



Depth limited search

- Like depth-first search, except:
  - Depth limit in the expand functin

L = depth limit

Complete if L ≥ d (d is the depth of the shallowest goal)

Not optimal (even if one continues the search after the first solution has been found, because an optimal solution may not be within the depth limit L)

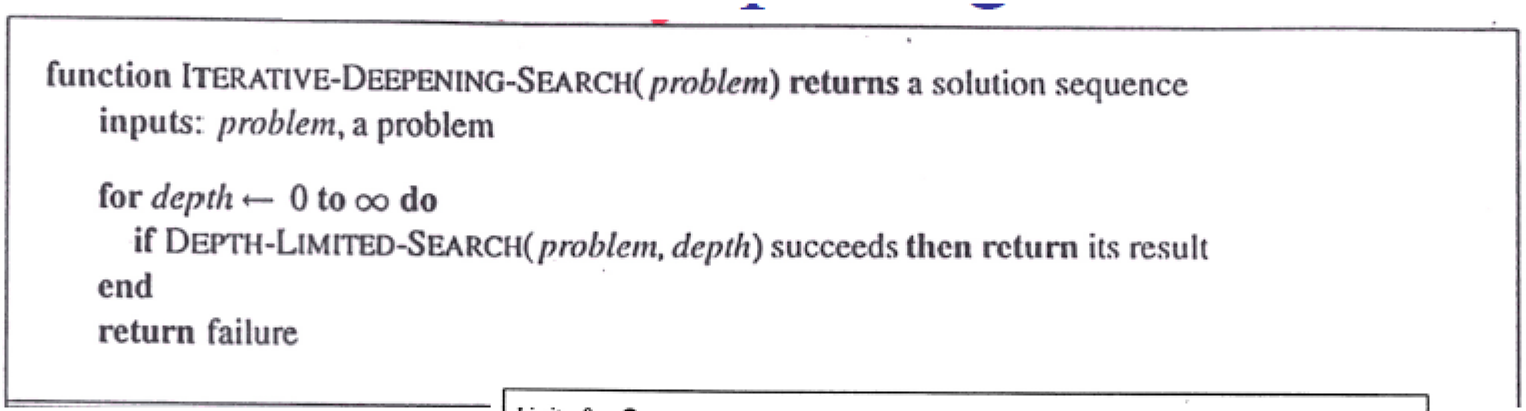
$O(b^L)$ time

O(bL) space

Iterative deepening search

- Use DFS as a subroutine
  1. Check the root
  2. Do a DFS searching for a path of length 1
  3. If there is no path of length 1, do a DFS searching for a path of length 2
  4. If there is no path of length 2, do a DFS searching for a path of length 3…

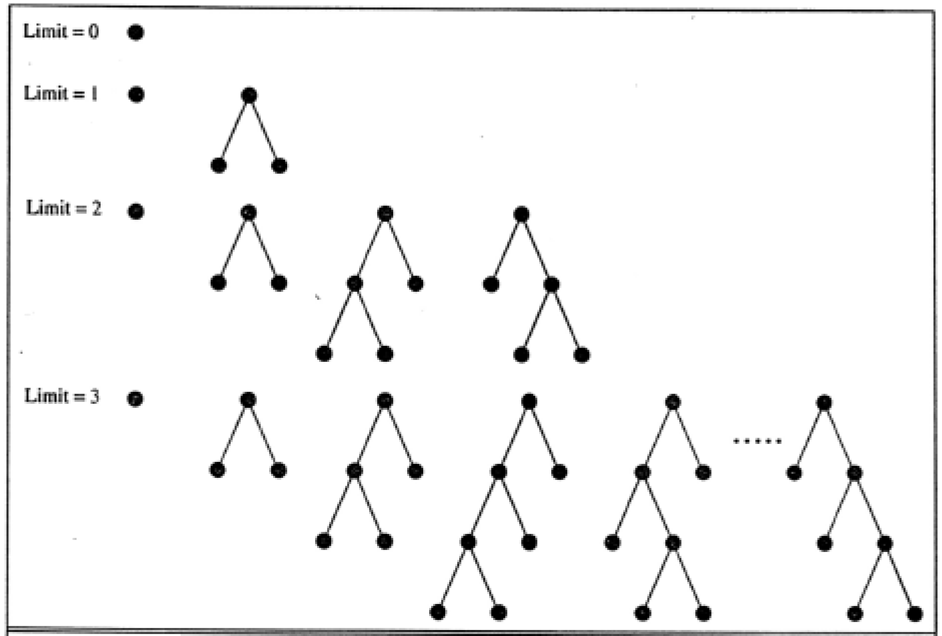
Iterative deepening search



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

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

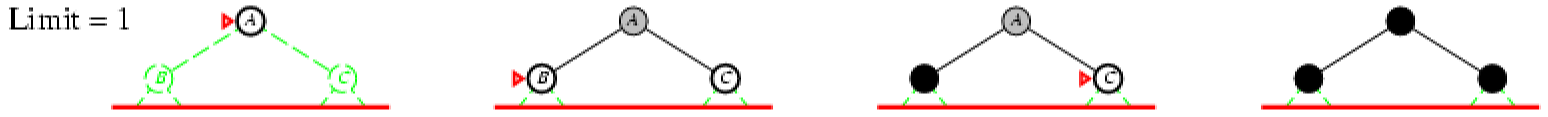
Depth-first search for each depth.



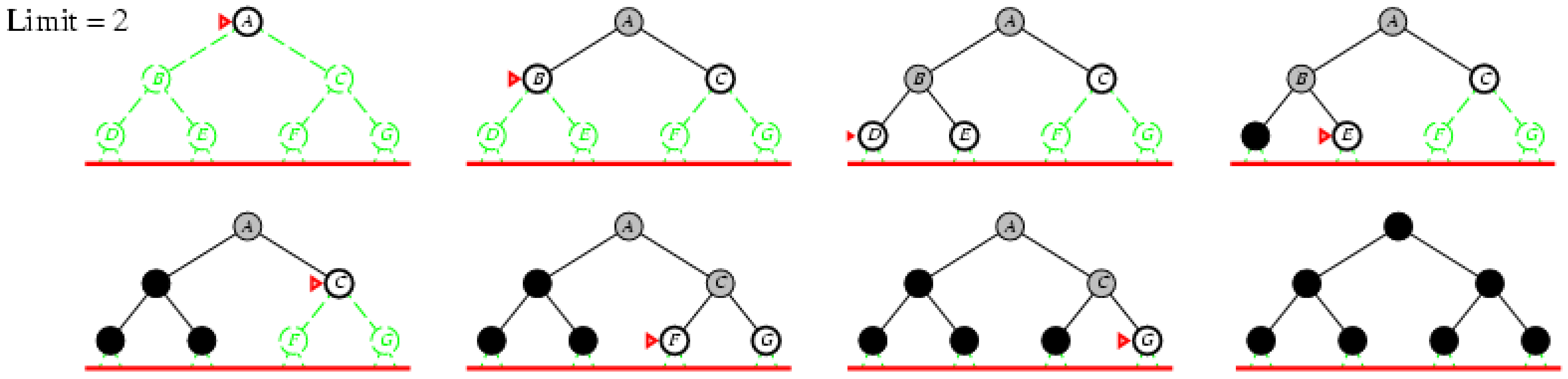
Four iterations of iterative deepening search on a binary tree.

Iterative deepening search *l* =0

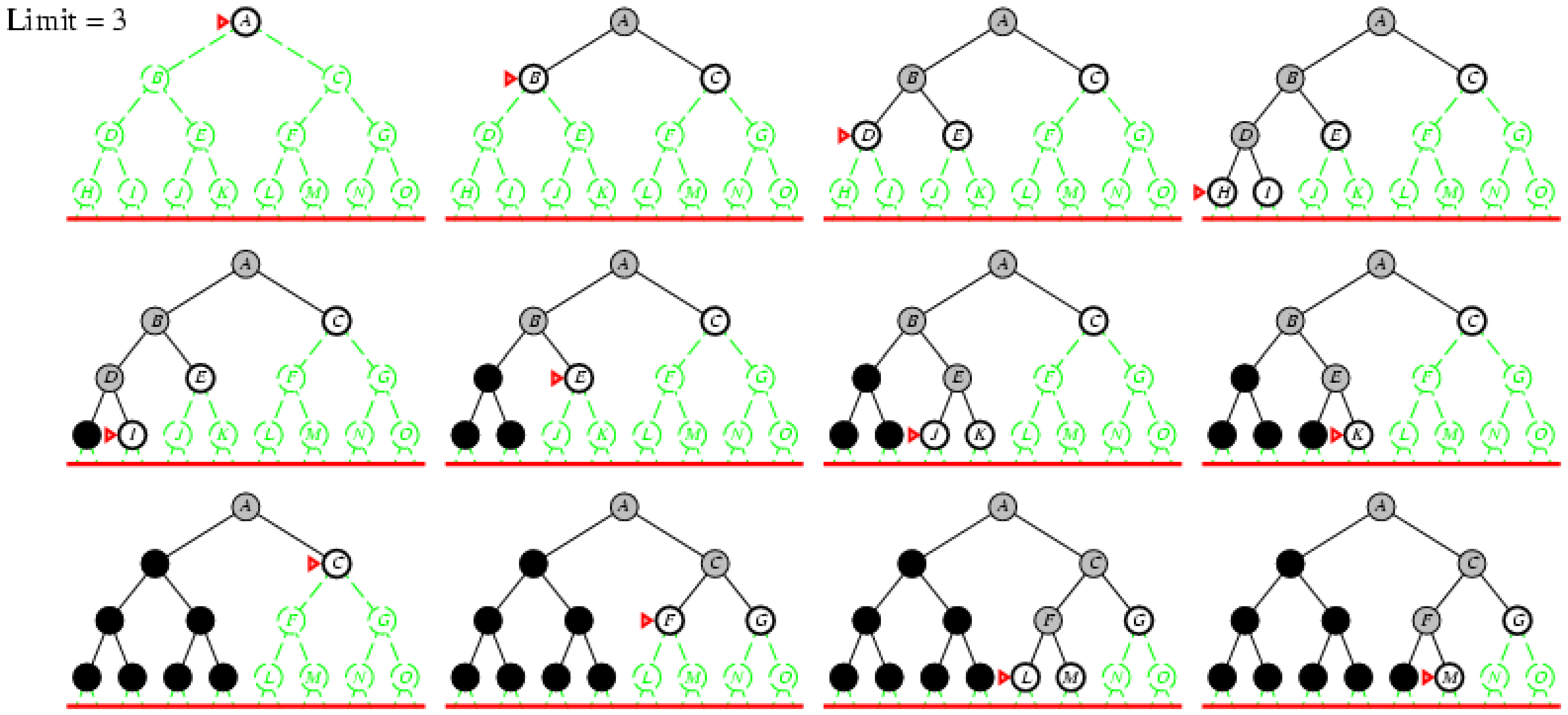
Iterative deepening search *l* =1



Iterative deepening search *l* =2



Iterative deepening search *l* =3



Iterative deepening search

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

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

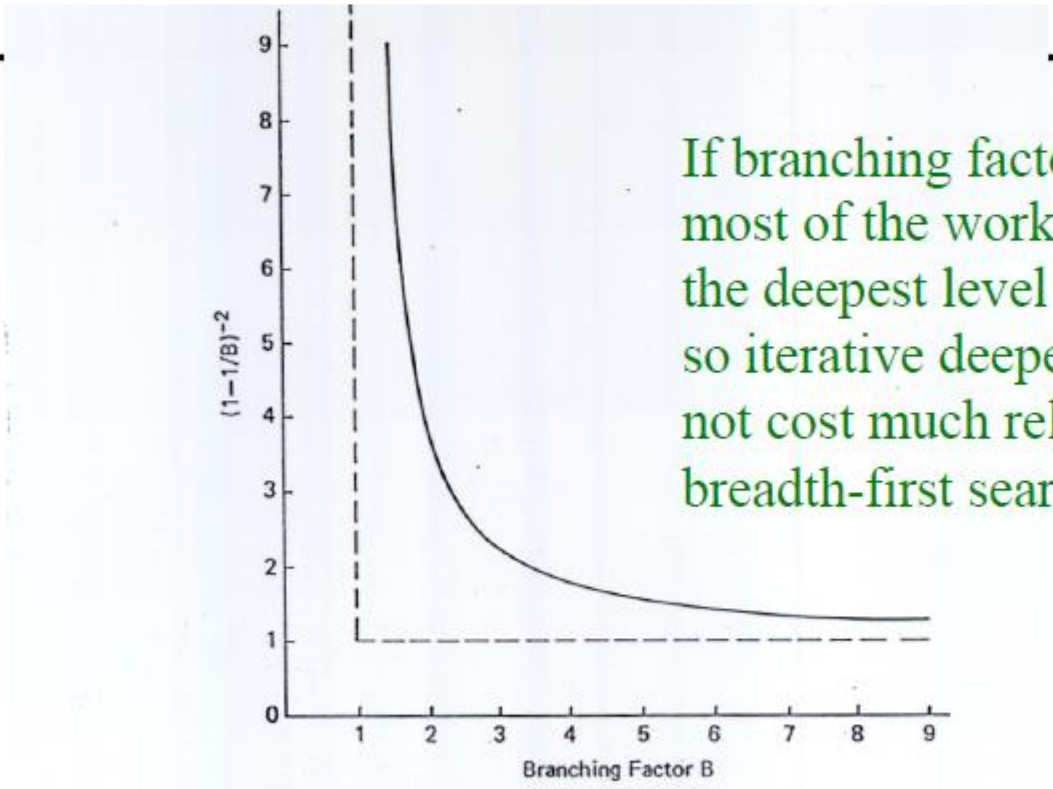
- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

$$N_{IDS} = (d+1)b^0 + d\ b\text{^}^1 + (d-1)b\text{^}^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For *b = 10*, *d = 5*,
  - N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111
  - N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456

- Overhead = (123,456 - 111,111)/111,111 = 11%

Iterative deepening search



If branching factor is *large*, most of the work is done at the deepest level of search, so iterative deepening does not cost much relatively to breadth-first search.

Graph of branching factor vs. constant coefficient as search depth goes to infinity.

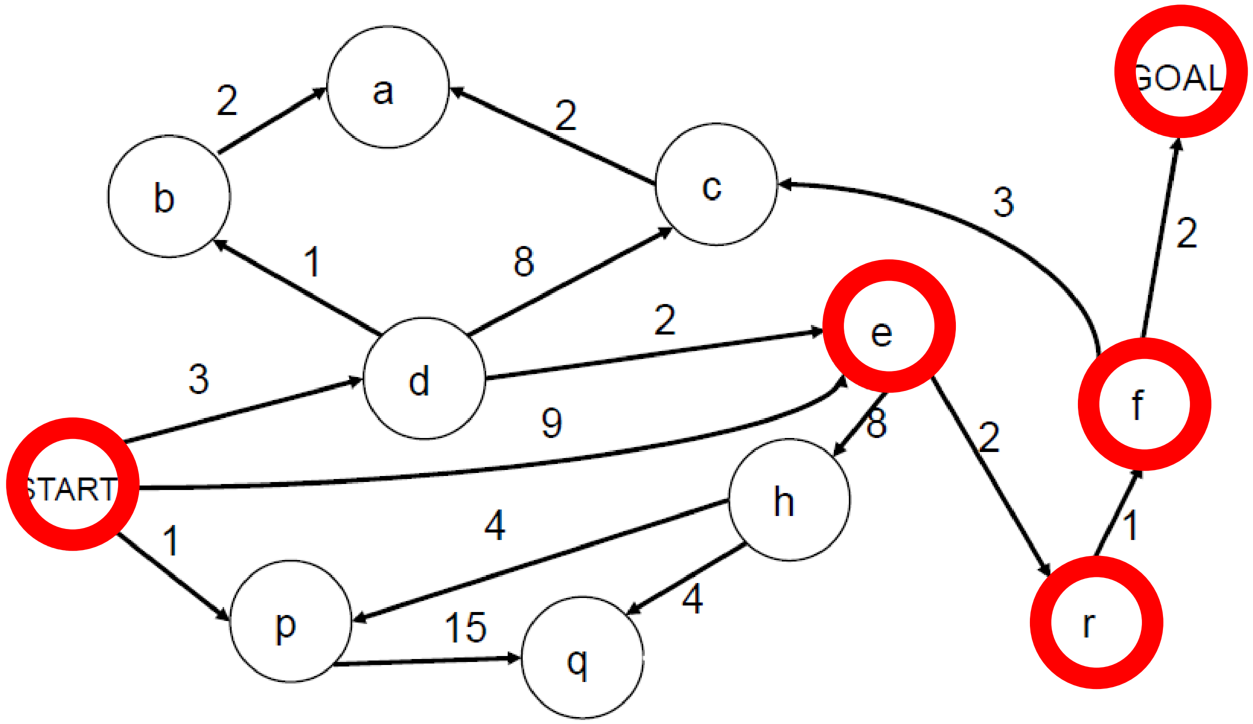
Conclusion:

- Iterative deepening is preferred when search space is large and depth of (optimal) solution is unknown
- Not preferred if branching factor is tiny (near 1)

Properties of iterative deepening search

- Complete? Yes
- Time? $(d+1)b^0 + d\, b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

Search with varying step costs



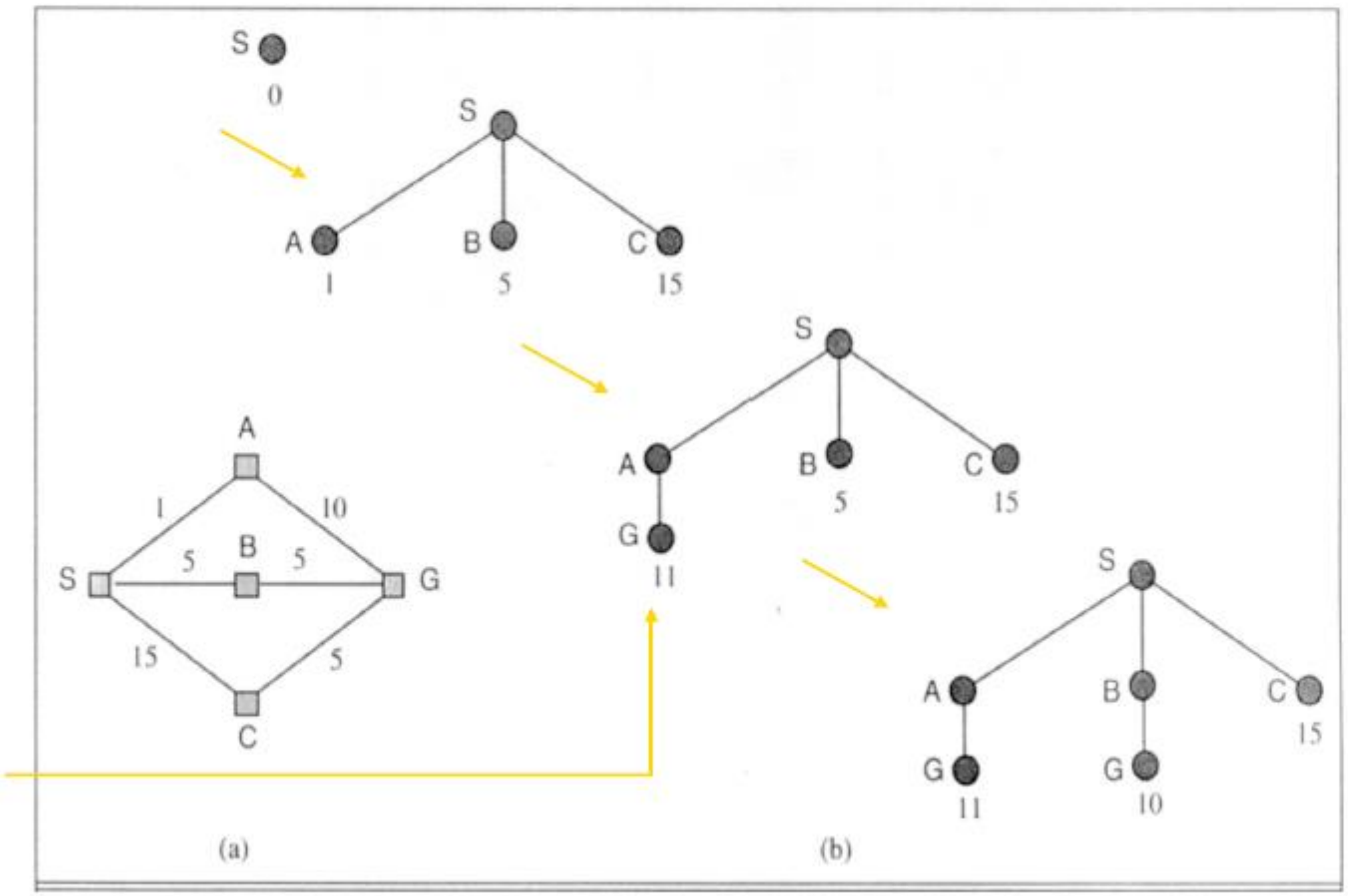
- BFS finds the path with the fewest steps, but does not always find the cheapest path

Uniform-cost search

- For each frontier node, save the total cost of the path from the initial state to that node
- Expand the frontier node with the lowest path cost
- Implementation: *frontier* is a priority queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- Equivalent to Dijkstra’s algorithm in general

Uniform-cost search

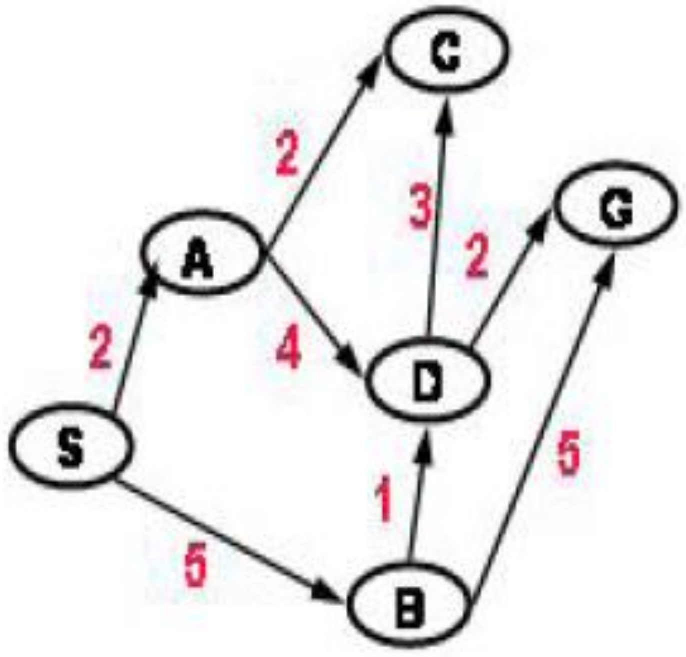
Insert nodes onto open list in ascending order of **cost** of path to root.



Taken from http://www.cs.nott.ac.uk/~gxk/courses/g5aiai/003blindsearches/blind_searches.htm

Uniform cost search

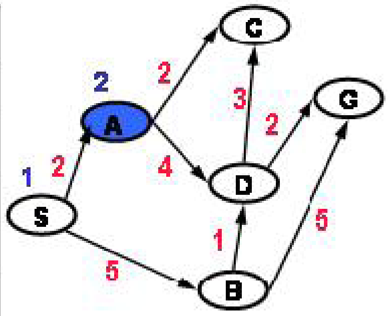
- Each link has a length or cost (which is always greater than 0)
- We want shortest or least cost path



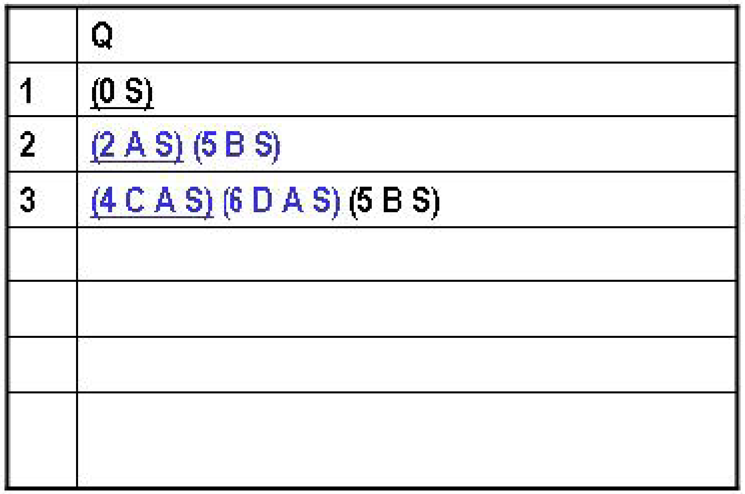
Total path cost:	
(S A C)	4
(S B D G)	8
(S A D C)	9

Uniform cost search

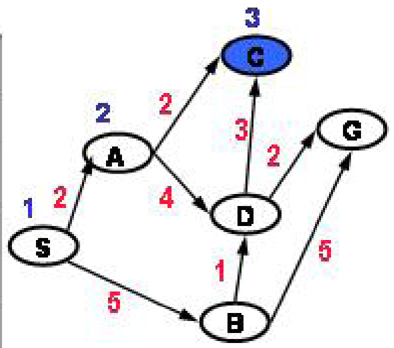
	Q
1	(0 S)
2	(2 A S) (5 B S)



Uniform cost search

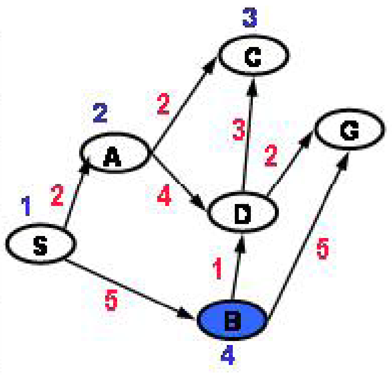


	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)

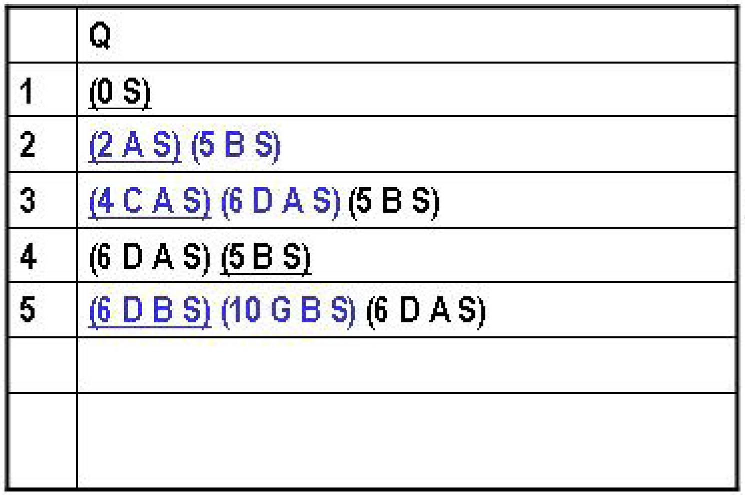


Uniform cost search

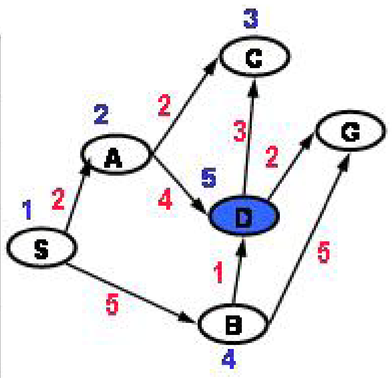
	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)



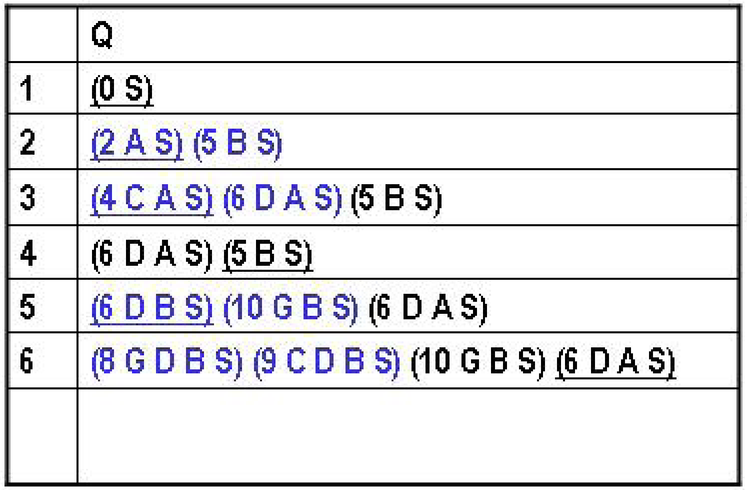
Uniform cost search



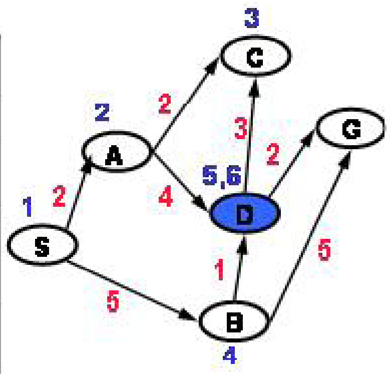
	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)
5	(6 D B S) (10 G B S) (6 D A S)



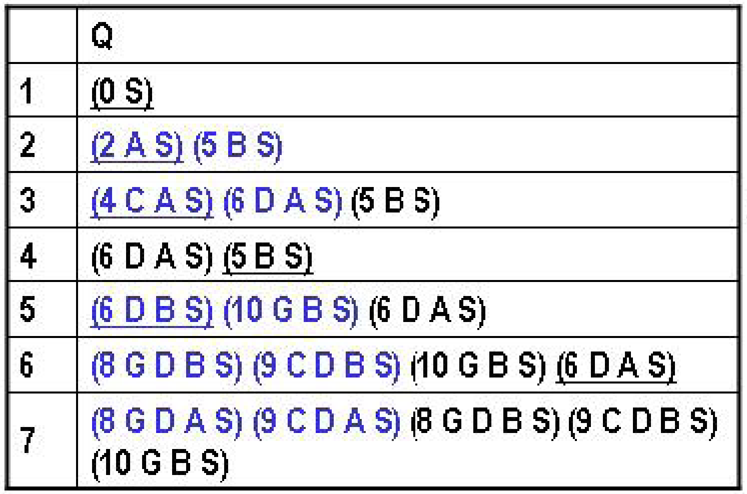
Uniform cost search



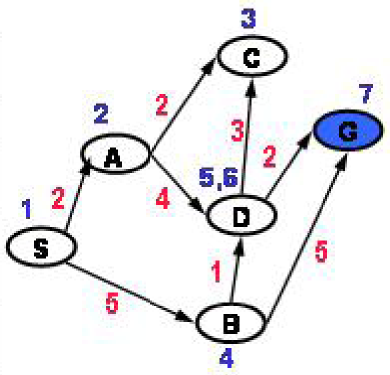
	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)
5	(6 D B S) (10 G B S) (6 D A S)
6	(8 G D B S) (9 C D B S) (10 G B S) (6 D A S)



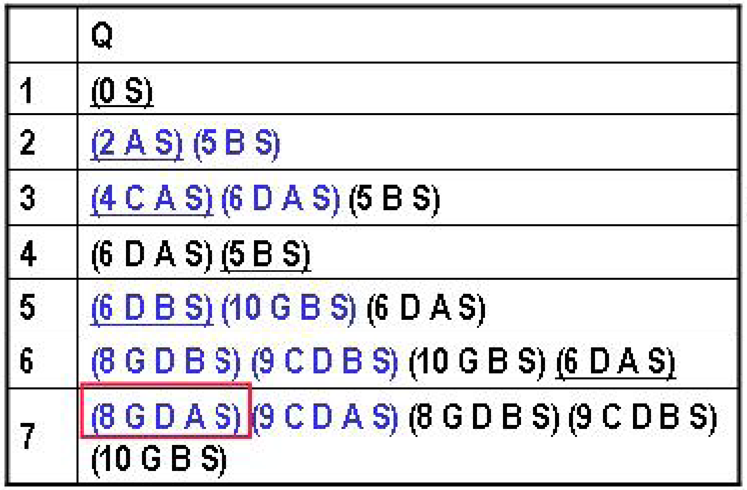
Uniform cost search



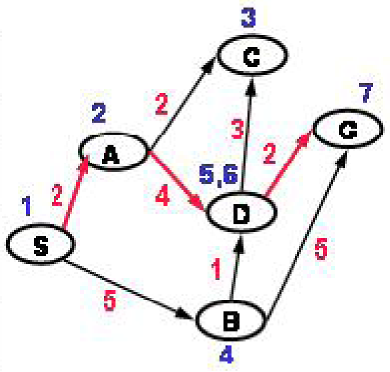
	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)
5	(6 D B S) (10 G B S) (6 D A S)
6	(8 G D B S) (9 C D B S) (10 G B S) (6 D A S)
7	(8 G D A S) (9 C D A S) (8 G D B S) (9 C D B S) (10 G B S)



Uniform cost search



	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)

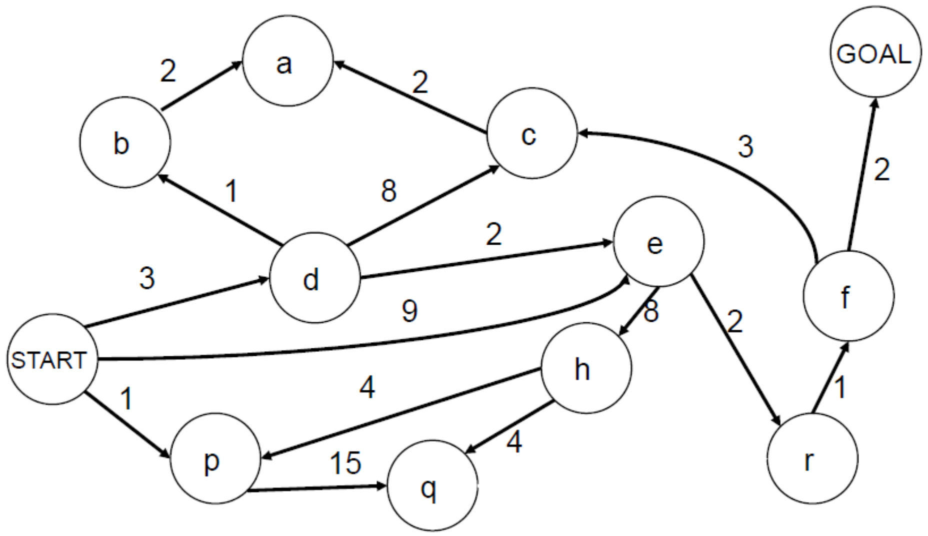


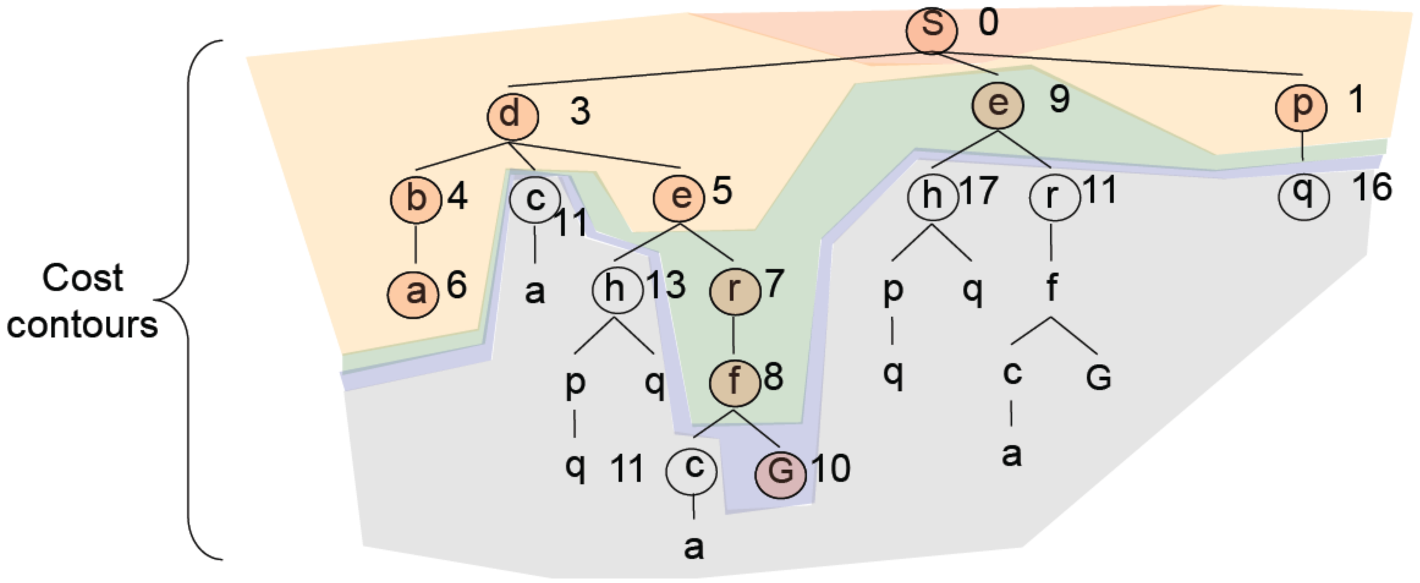
Why not stop on the first goal

- When doing Uniform Cost, it is not correct to stop the search when the first path to a goal is generated, that is, when a node whose state is a goal is added to Q.
- We must wait until such a path is pulled off the Q and tested in step 3. It is only at this point that we are sure it is the shortest path to a goal since there are no other shorter paths that remain unexpanded.
- This contrasts with the Any Path searches where the choice of where to test for a goal was a matter of convenience and efficiency, not correctness.
- In the previous example, a path to G was generated at step 5, but it was a different, shorter, path at step 7 that we accepted.

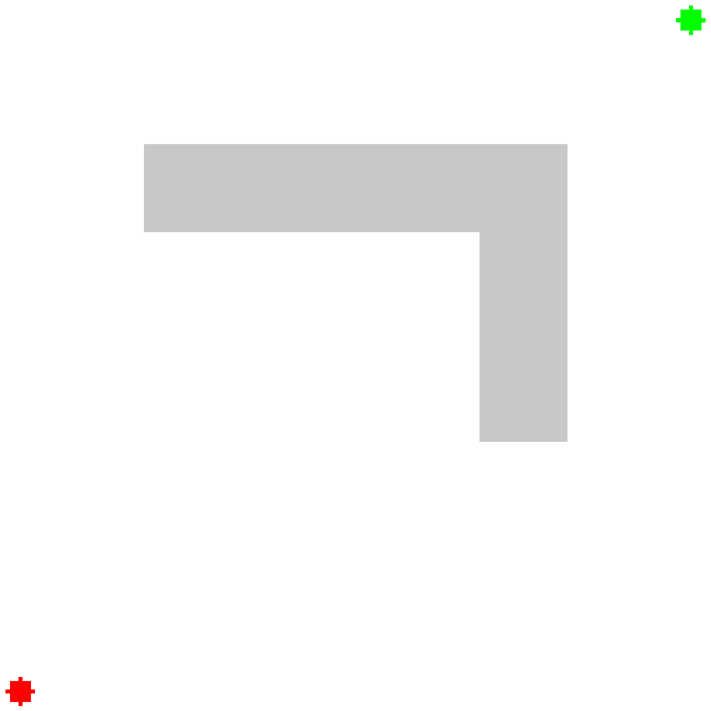
Uniform-cost search example

- Expansion order: (S,p,d,b,e,a,r,f,e,G)





Another example of uniform-cost search



Wikipedia

Properties of uniform-cost search

- Expand least-cost unexpanded node
- Implementation:
  - *fringe* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal

- **Complete?**

  Yes, if step cost is greater than some positive constant ε (we don't want infinite sequences of steps that have a finite total cost)

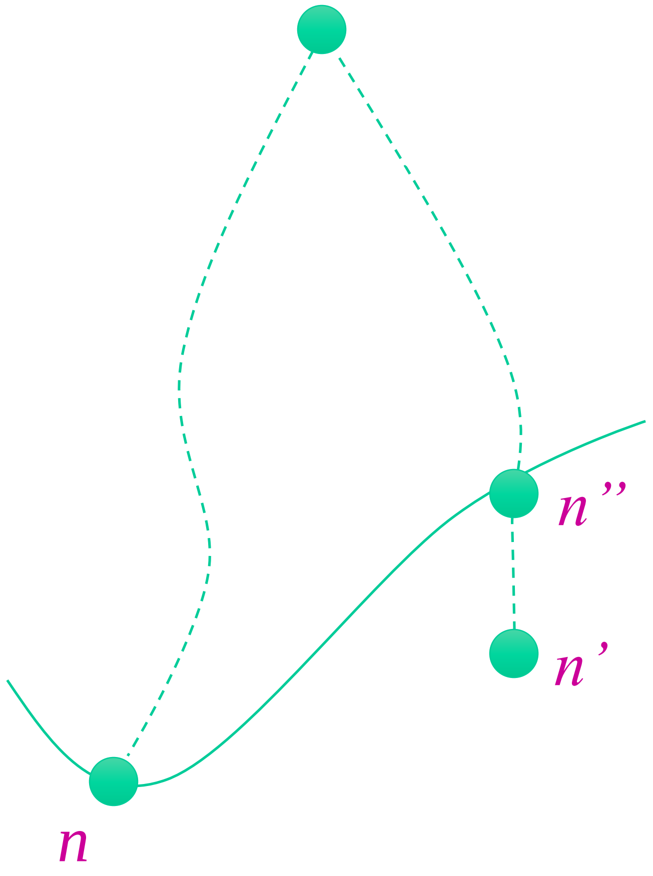
- **Optimal?**

  Yes

Optimality of uniform-cost search

- **Graph separation property**: every path from the initial state to an unexplored state has to pass through a state on the frontier
  - Proved inductively

- Optimality of UCS: proof by contradiction
  - Suppose UCS terminates at goal state n with path cost $g(n)$ but there exists another goal state n' with $g(n') < g(n)$
  - By the graph separation property, there must exist a node n'' on the frontier that is on the optimal path to n'
  - But because $g(n'') \leq g(n') < g(n)$, n'' should have been expanded first!



Uniform-cost search

- Complete? Yes, if step cost $\geq \varepsilon$
- Time? # of nodes with *pathn cost* $g \leq$ cost of optimal solution,

  $O(b^{ceiling(C^*/\,\varepsilon)})$ where C^* is the cost of the optimal solution
- This can be greater than $O(b^d)$: the search can explore long paths consisting of small steps before exploring shorter paths consisting of larger steps
- Space? # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\,\varepsilon)})$
- Optimal? Yes – nodes expanded in increasing order of $g(n)$

Review: Uninformed search strategies

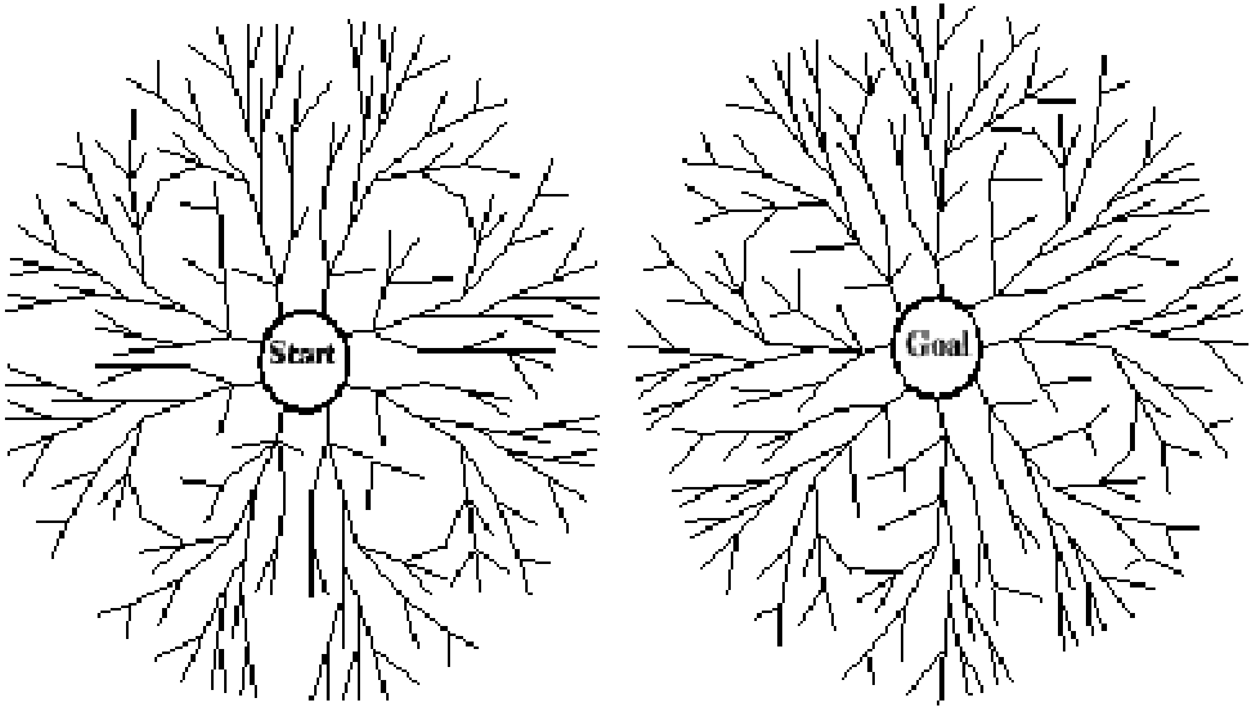
Algorithm	Complete?	Optimal?	Time complexity	Space complexity
BFS	Yes	If all step costs are equal	$O(b^d)$	$O(b^d)$
DFS	No	No	$O(b^m)$	$O(bm)$
IDS	Yes	If all step costs are equal	$O(b^d)$	$O(bd)$
UCS	Yes	Yes	Number of nodes with $g(n) \leq C^*$	

b:	maximum branching factor of the search tree
d:	depth of the optimal solution
m:	maximum length of any path in the state space
C*:	cost of optimal solution
g(n):	cost of path from start state to node n

Bidirectional search

- Run two simultaneous searches – one forward from the initial state, and the other backward from the goal, stopping when two searches meet in the middle

$b^{d/2} + b^{d/2} < b^d$



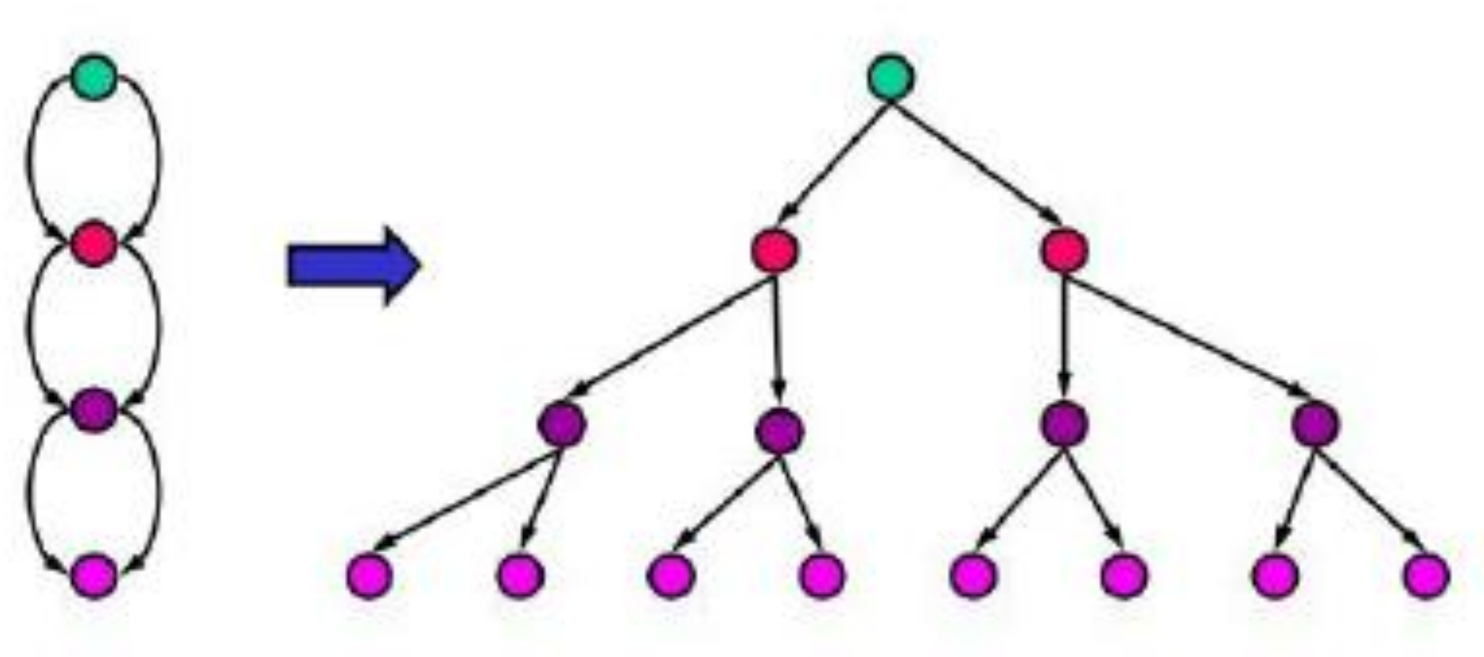
For b = 10, d= 6 , BFS 1,111, 111 nodes, BS 2,222 nodes

Bi-directional Search discussion

- Need to have operators that calculate **predecessors**.
- Need a way to check that **states meet (are the same)**
- Efficient way to check when searches meet: hash table
- Can use IDS or BFS or DFS in each half
- Optimal, complete, $O(b^{d/2})$ time.
- Still $O(b^{d/2})$ space (even with iterative deepening) because the nodes of at least one of the searches have to be stored to check matches.

Repeated states

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



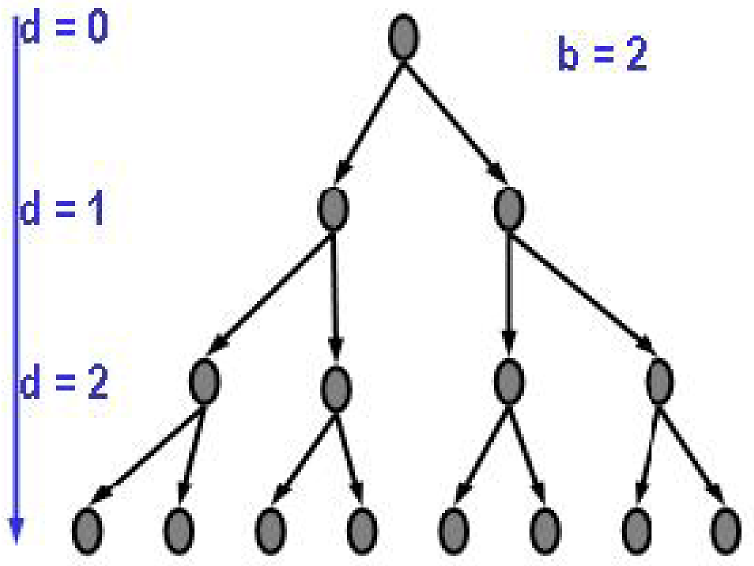
Handling repeated states

- Do not return to the state just you came from
- Do not create paths with cycles in them
- Do not generate any state that was ever generated before

Worst Case Running Time

Max Time $\propto$ Max #Visited

- The number of states in the search space may be exponential in some "depth" parameter, e.g. number of actions in a plan, number of moves in a game.
- All the searches, with or without visited list, may have to visit each state at least once, in the worst case.
- So, all searches will have worst case running times that are at least proportional to the total number of states and therefore exponential in the "depth" parameter.



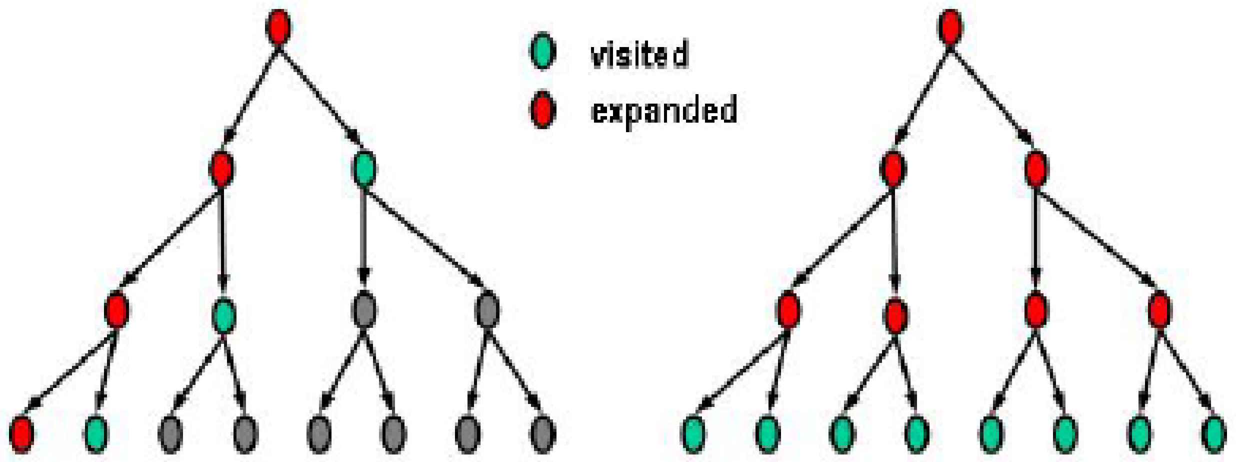
d is depth
b is branching factor

$b^d < (b^{d+1} - 1) / (b - 1) < b^{d+1}$

states in tree

Worst Case Space

Max Q size = Max (#Visited – #Expanded)

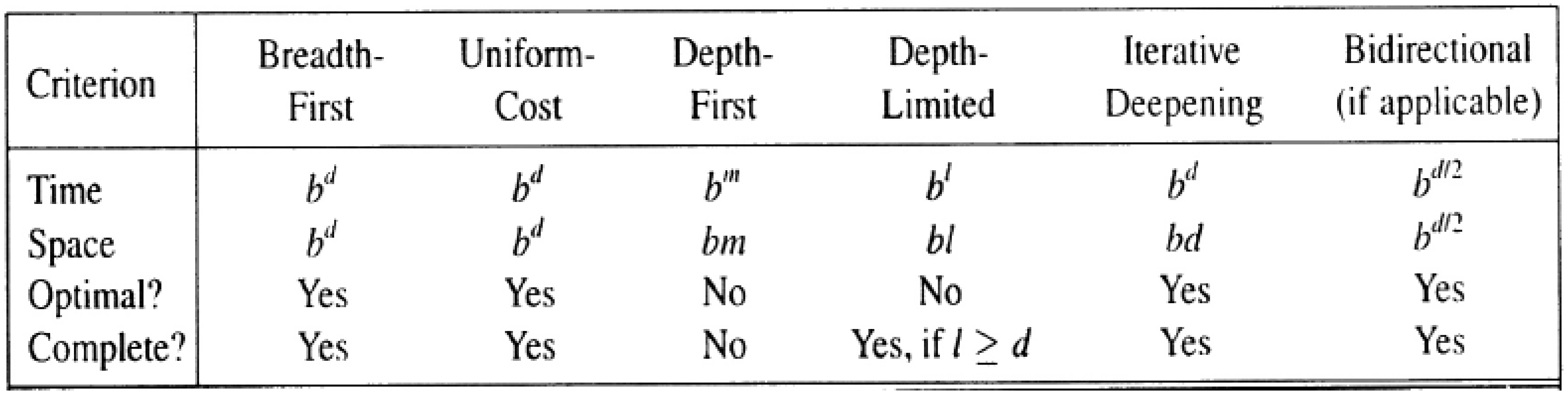


Depth First max Q size
$(b-1)d \approx bd$

Breadth First max Q size
b^d

Summary

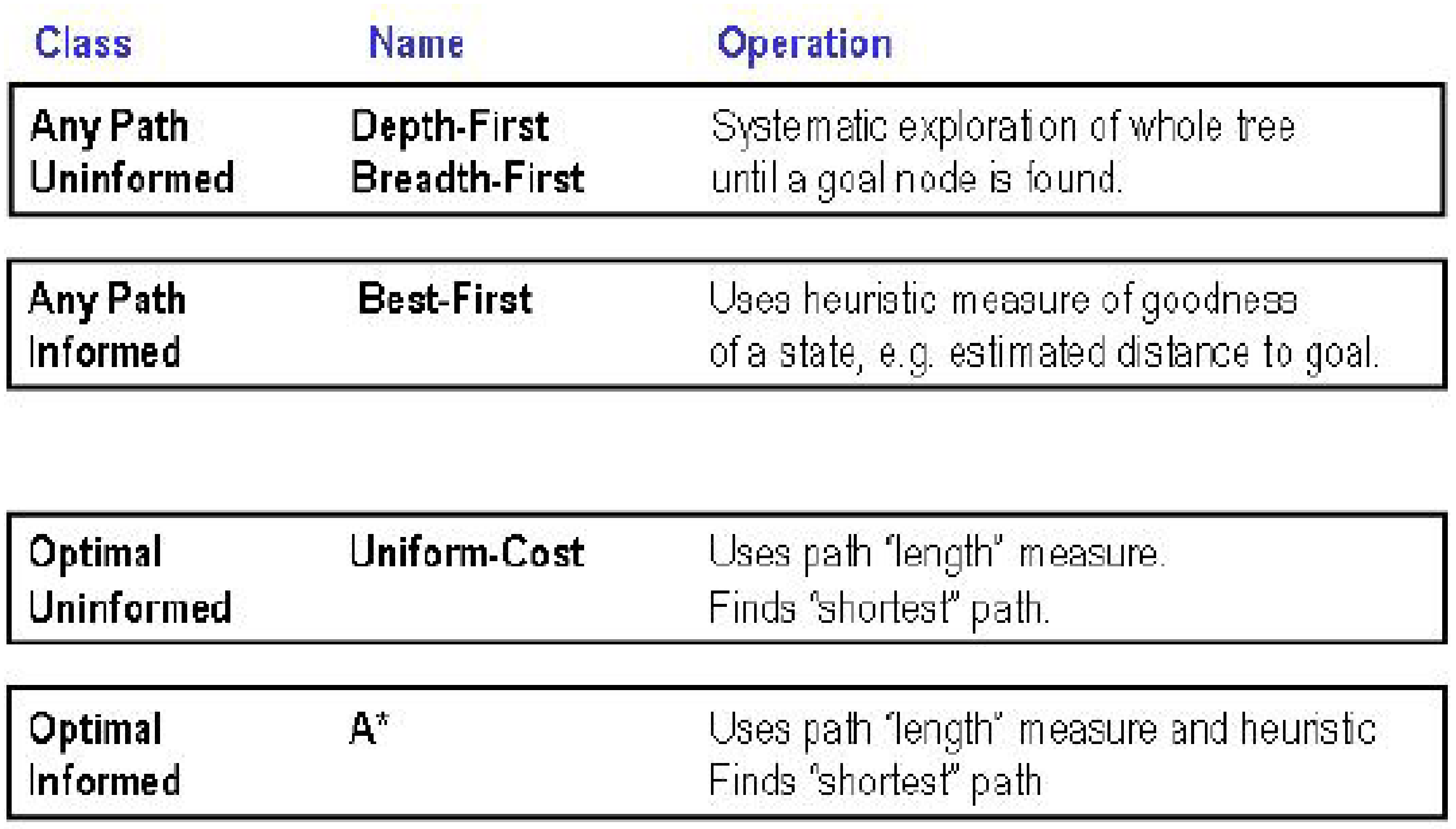
- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms



Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

b = branching factor
d = depth of shallowest goal state
m = maximum depth of the search tree
l = depth limit of the algorithm

Classes of Search



Class	Name	Operation
Any Path Uninformed	Depth-First Breadth-First	Systematic exploration of whole tree until a goal node is found.
Any Path Informed	Best-First	Uses heuristic measure of goodness of a state, e.g. estimated distance to goal.
Optimal Uninformed	Uniform-Cost	Uses path "length" measure. Finds "shortest" path.
Optimal Informed	A*	Uses path "length" measure and heuristic Finds "shortest" path