CS 3011: Artificial Intelligence

Solving Problems by Searching

Instructors: Dr. Durgesh Singh

CSE Discipline, PDPM IIITDM, Jabalpur -482005

Uninformed Search Strategies

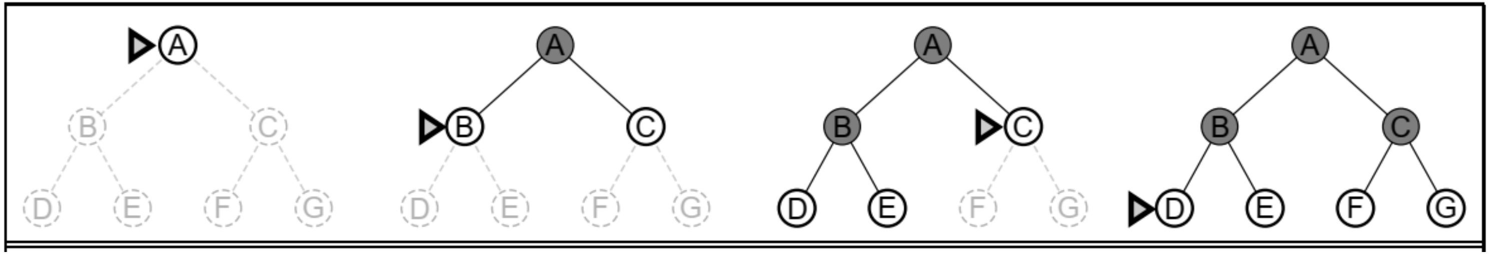
- It is also called **blind search**
- Uninformed search strategies use only the information available in the problem definition
- All they can do is generate successors and distinguish a goal state from a nongoal state.
    - Breadth-first search
    - Uniform-cost search
    - Depth-first search
    - Depth-limited search
    - Iterative deepening search

Breadth-first search

- Breadth-first search is a simple strategy in which the root node is expanded first
    - then all the successors of the root node are expanded next, then their successors, and so on
- In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
    - Expand shallowest unexpanded node
- Implementation:
  - This is achieved very simply by using a FIFO queue for the frontier.

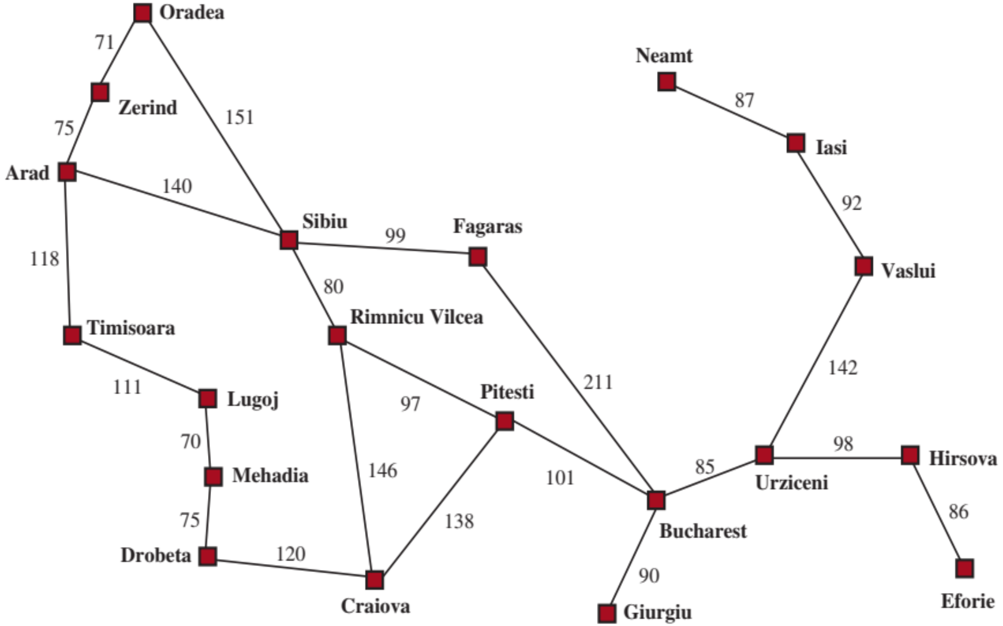
Breadth-first search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then do
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```



Breadth-first search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then do
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```



Properties of Breadth-first search

- Complete?
- Yes: if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes (provided the branching factor b is finite).
- Optimal?
- It is cost-optimal for problems where all actions have the same cost, but not for problems that don't have that property.
- Time and Space complexity?
- Imagine searching a uniform tree where every state has b successors. Suppose that the solution is at depth d. Then the total number of nodes generated is

$$1+b+b^2+b^3+\ldots+b^d = O(b^d)$$

- All the nodes remain in memory, so both time and space complexity are $O(b^d)$

Properties of Breadth-first search

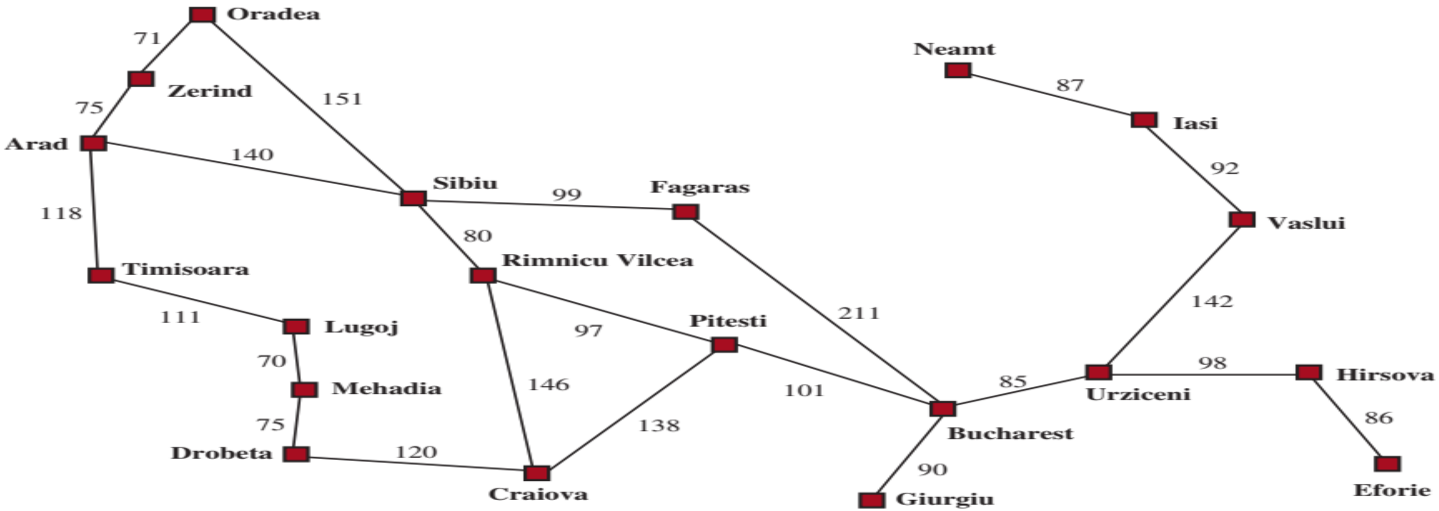
- An exponential complexity bound such as $O(b^d)$ is scary
- As a typical real-world example, consider a problem with branching factor b=10, processing speed 1 million nodes/second, and memory requirements of 1 Kbyte/node.

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

- The memory requirements are a bigger problem for breadth-first search than the execution time.
  - However, time is still an important factor. At depth d=14, even with infinite memory, the search would take 3.5 years.

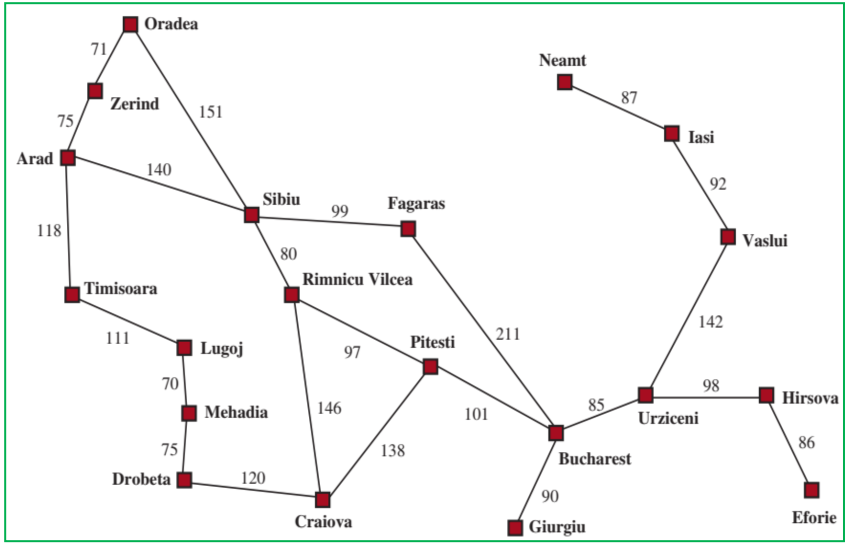
Uniform-cost search

- Instead of expanding the shallowest unexpanded node, Uniform-cost Search expands the node n with the lowest path cost g(n).
- This is done by storing the frontier as a priority queue ordered by g
- Uniform-cost Search does not care about the number of steps a path has, but about their total cost.



Uniform-cost search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node.STATE to explored
      // Print the contents of the Priority Queue (frontier)
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              frontier ← INSERT(child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```



Uniform-Cost Search

- Expand least-cost unexpanded node
- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost ≥ ε >0
- Time? $O(b^{1+floor(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
- Space? $O(b^{1+floor(C^*/\varepsilon)})$
- Optimal? Yes – given the condition of completeness – you always expand the node with lowest cost

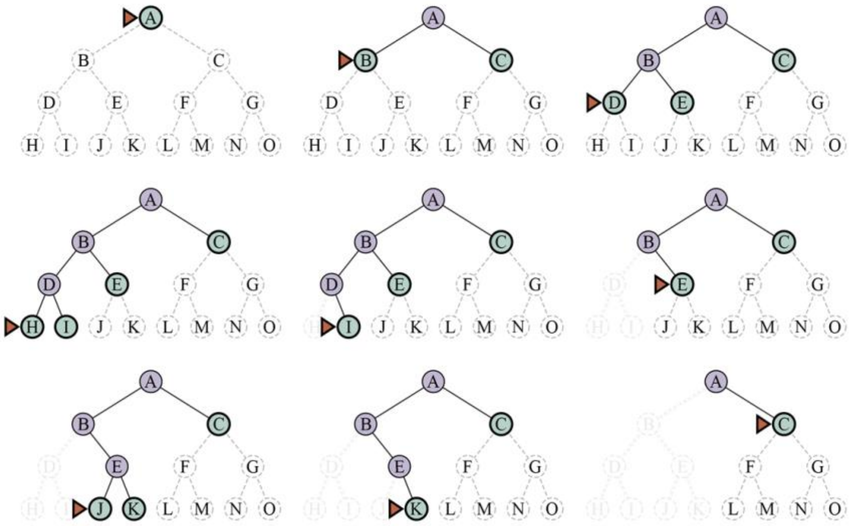
Depth-first search

- Depth-first search always expands the deepest node in the current frontier of the search tree.
- The algorithm starts at the initial states and explores as far as possible along each branch before backtracking.
- Implementation:
  - This is achieved very simply by using a LIFO data structure (i.e., Stack) for the frontier.

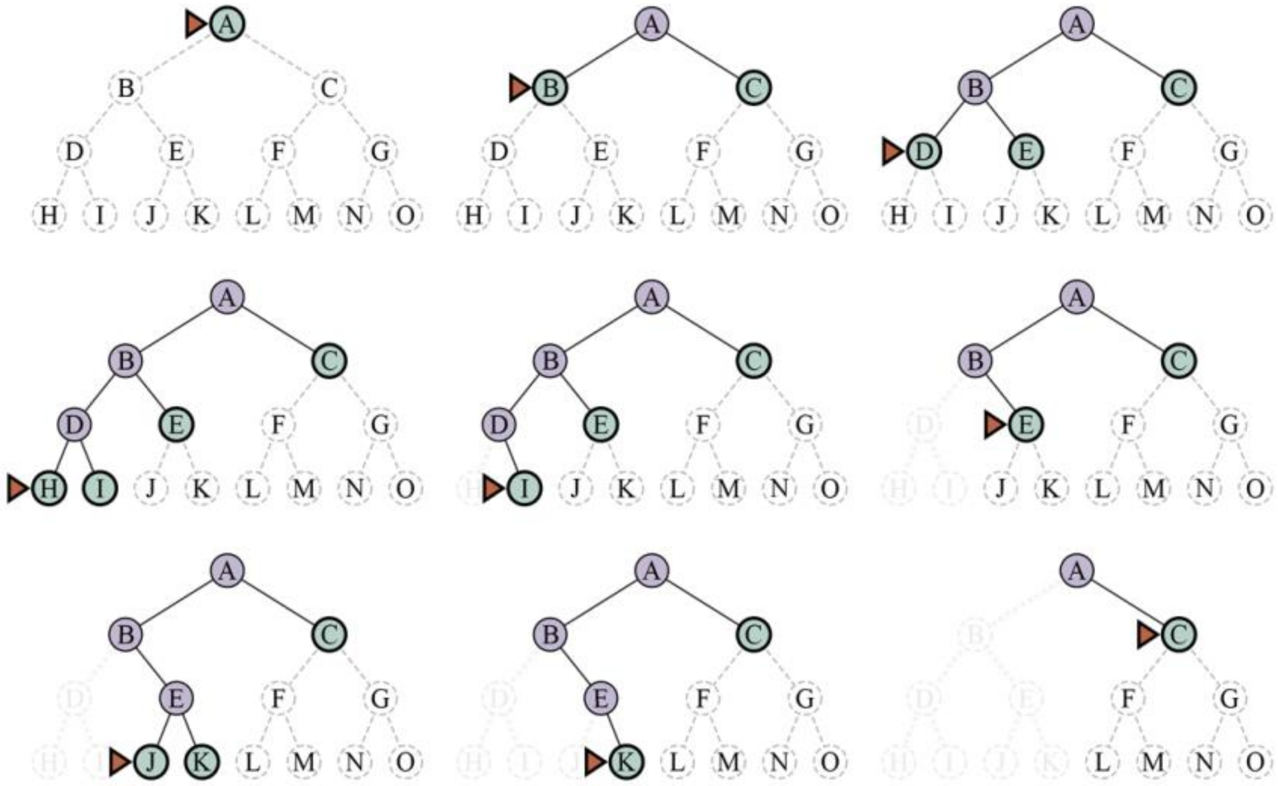
Depth-first search

What changes are required in this algorithm to make DFS?

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier



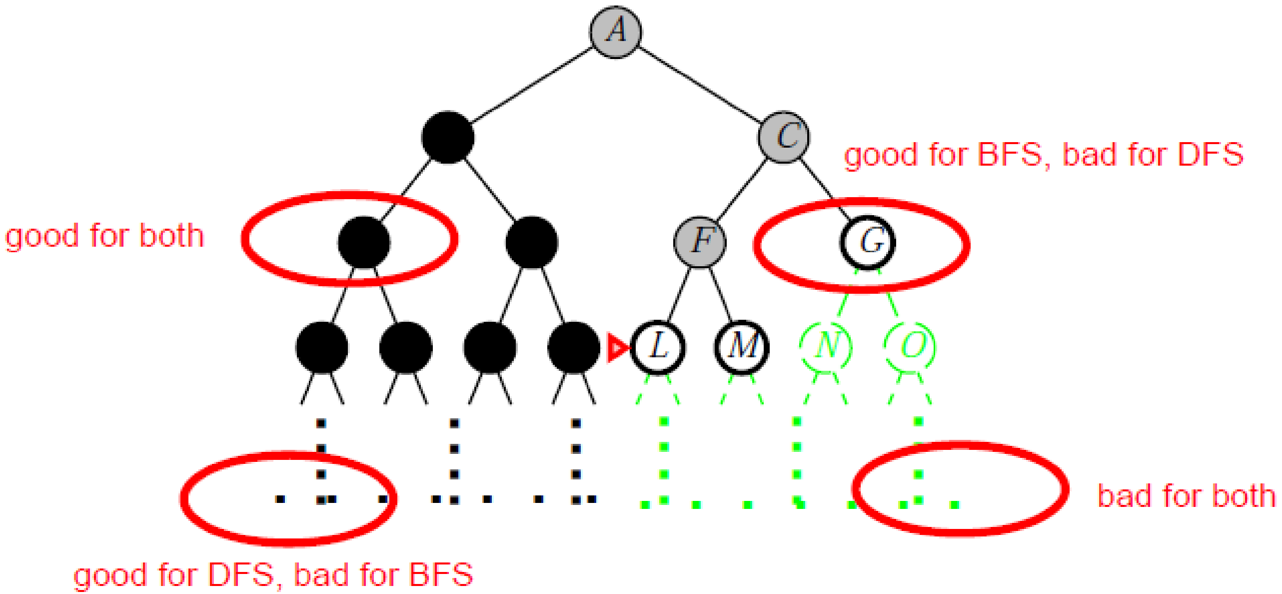
Depth-first search



Properties of Depth-first search

- Complete?
- No: fails in infinite-depth spaces
- 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! we only need to remember a single path + generated unexplored nodes.
- Optimal? No

BFS or DFS



A
C
good for BFS, bad for DFS
good for both
F
G
L
M
N
O
bad for both
good for DFS, bad for BFS

Depth-limited search

- Depth-first search with depth limit *l*, i.e., nodes at depth *l* have no successors
- Unfortunately, if we make a poor choice for the algorithm will fail to reach the solution

Depth-Limited Search

- Complete?
- NO. Why? (l < d)
- Time?
- $O(b^l)$
- Space?
- $O(bl)$
- Optimal?
- NO
- Depth-first is a special case of depth-limited with l being infinite.

Iterative deepening search

- Iterative deepening repeatedly applies depth limited search with increasing limits.
    - Trying all values of *l*: first 0, then 1, then 2, and so on—until either a solution is found, or the depth limited search returns the failure value rather than the cutoff value.

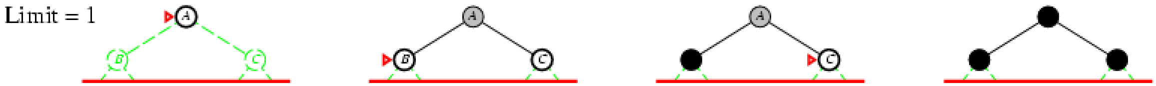
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

Here, Cutoff means there might be a solution at deeper depth than *l*

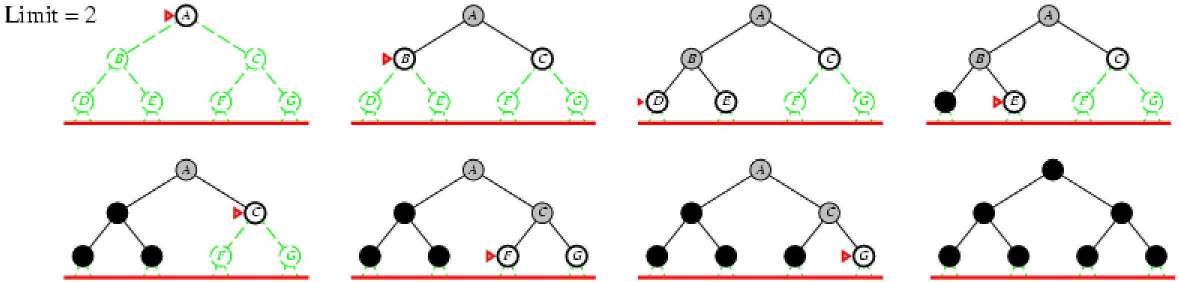
Iterative deepening search l =0

Limit = 0

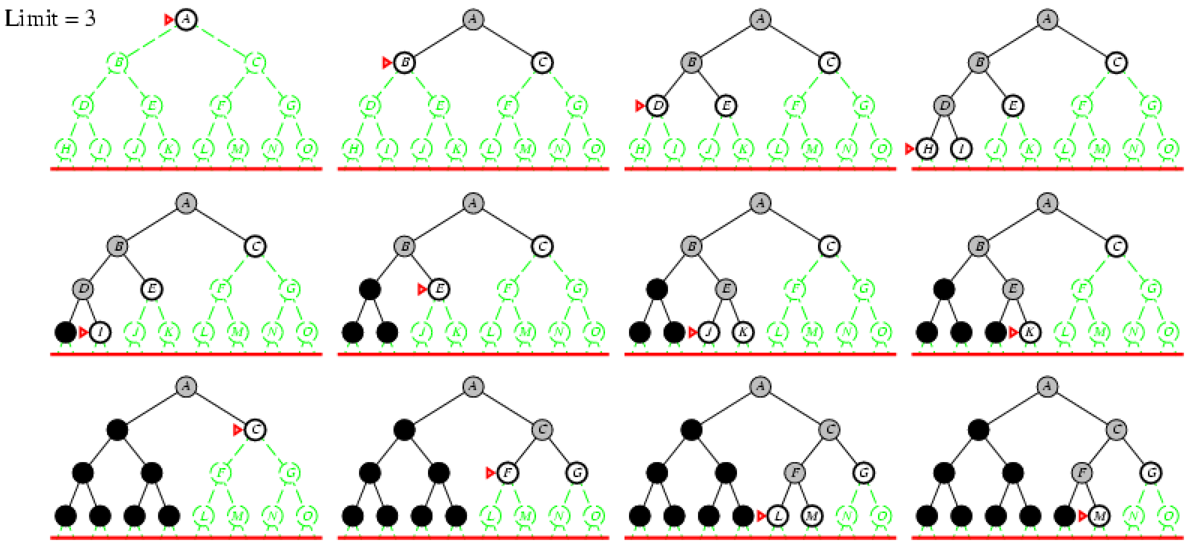
Iterative deepening search l =1



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*:

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

- For $b = 10$, $d = 5$,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$
  - Overhead = (123,456 - 111,111)/111,111 = 11%

Properties of iterative deepening search

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

Bidirectional search

- The idea behind bidirectional search is to run two simultaneous searches
  - one forward from the initial state and the other backward from the goal
  - stopping when the two searches meet in the middle
- Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found.

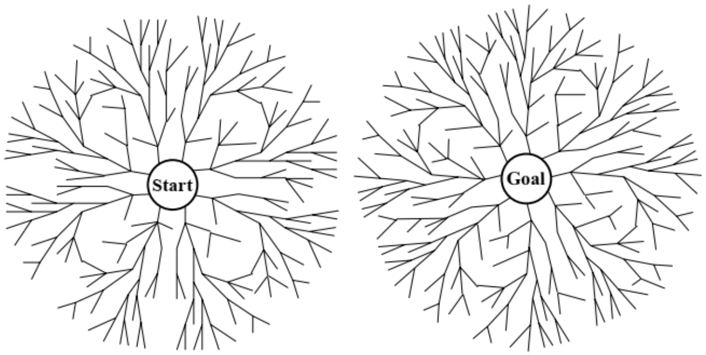
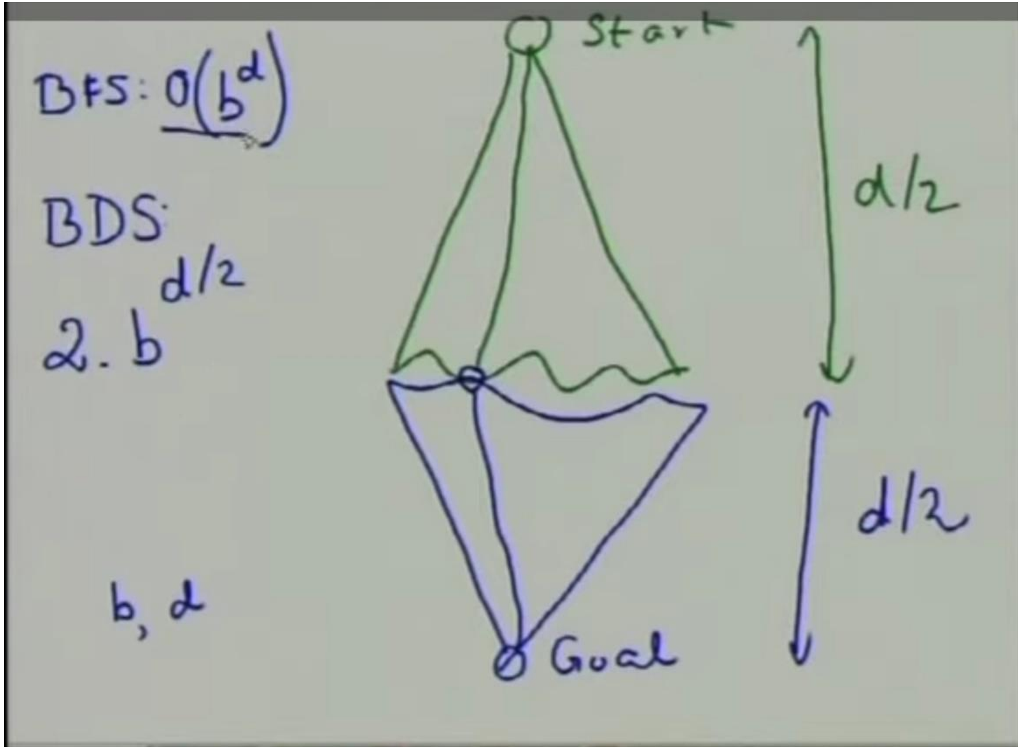


Fig: A schematic view of a bidirectional search that is about to succeed, when a branch from the start node meets a branch from the goal node.

Bidirectional search



Summary of Algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \varepsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.