Un-informed Search Strategies

Dr.Y.Sangeetha
Associate Professor

Un-informed Search Strategies

- Uninformed strategies means that they have no additional information about states beyond that provided in the problem definition (also called blind search algorithms)
- Breadth First Search
- Uniform cost Search
- Depth First Search (Depth Limited Search)
- Iterative Deepening Search
- Bidirectional Search

Breadth-first search

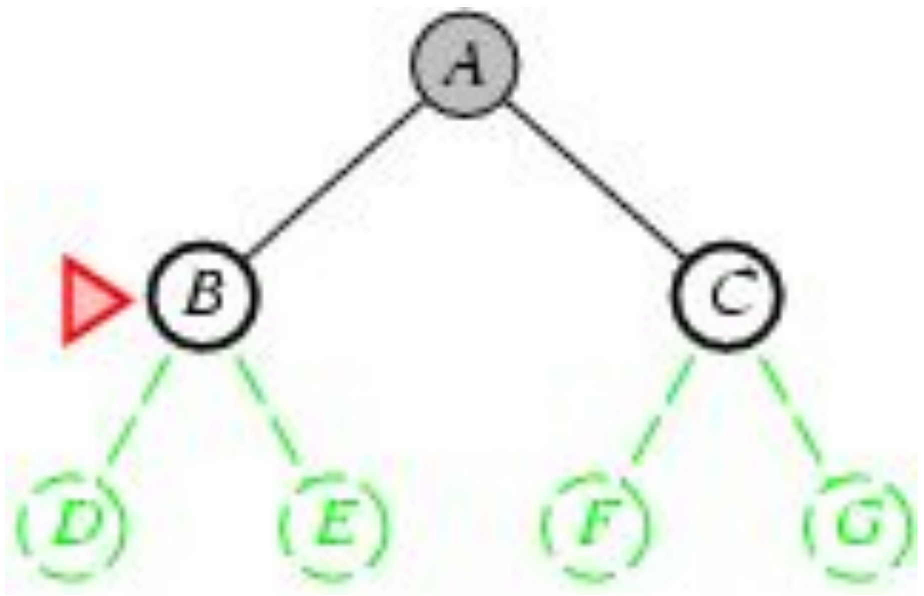
Expand shallowest unexpanded node

Implementation:

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

Expand:
fringe = [B,C]

Is B a goal state?



Breadth-first search

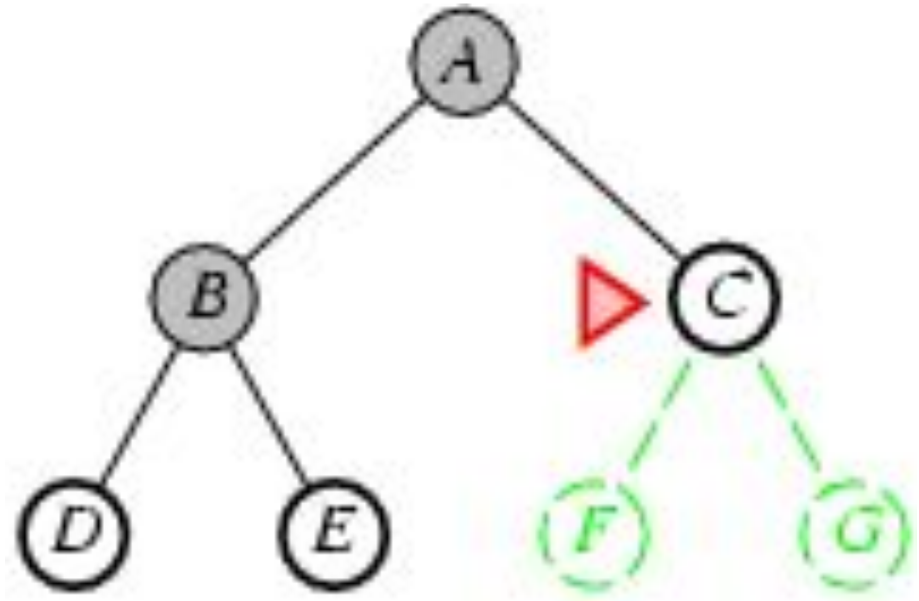
Expand shallowest unexpanded node

Implementation:

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

Expand:
fringe=[C,D,E]

Is C a goal state?



Breadth-first search

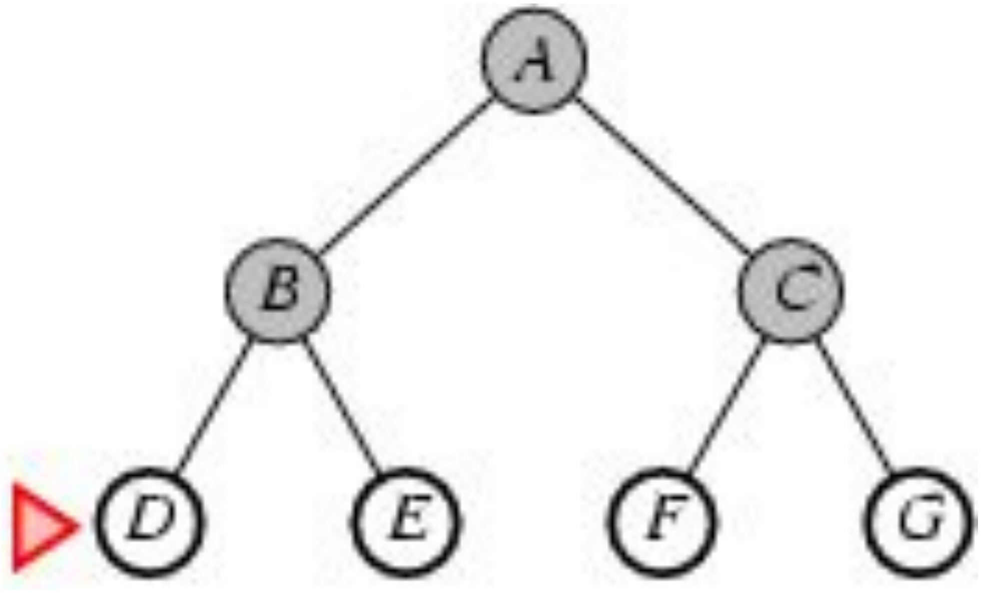
Expand shallowest unexpanded node

Implementation:

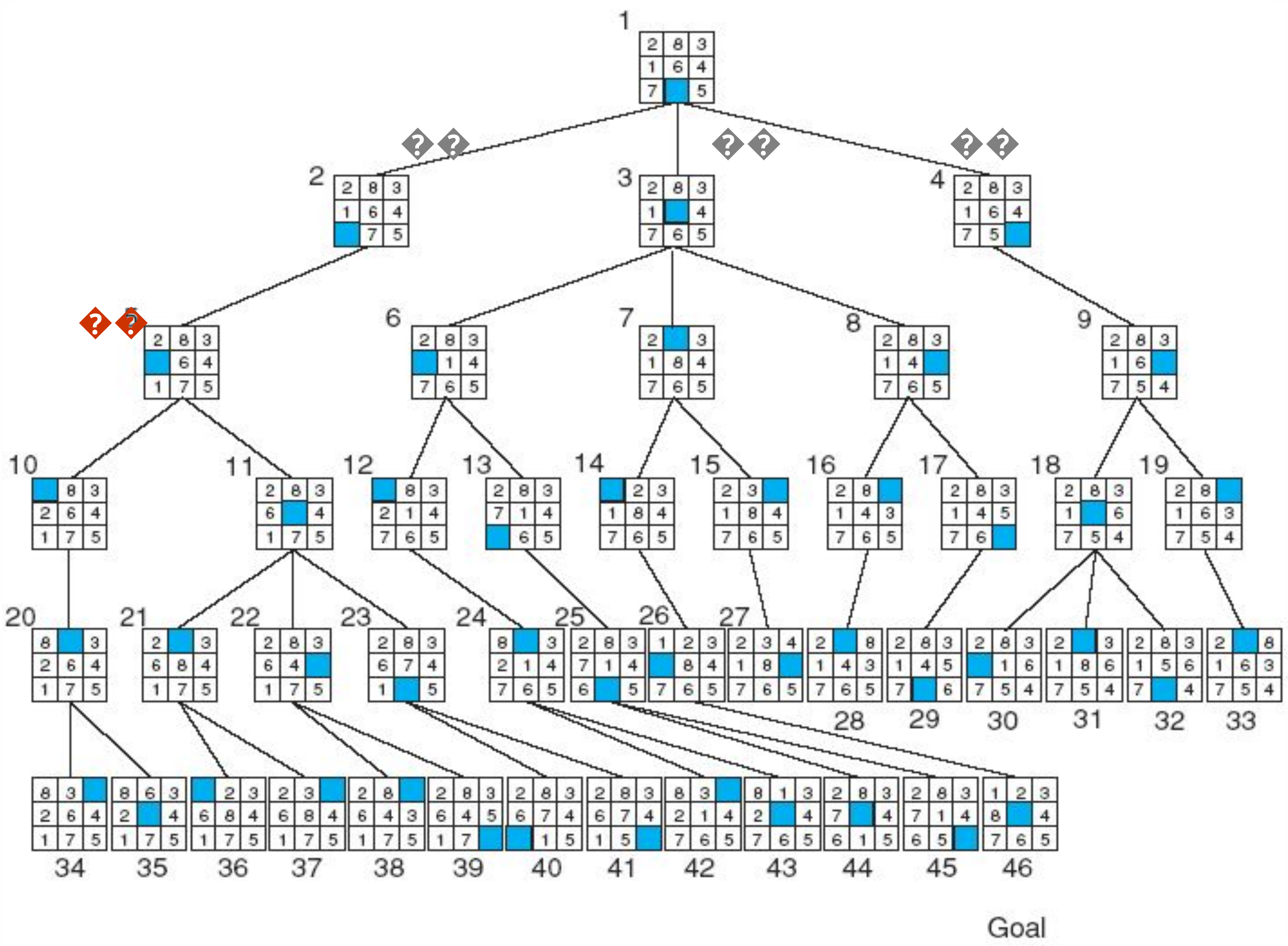
fringe is a FIFO queue, i.e., new successors go at end

Expand:
fringe=[D,E,F,G]

Is D a goal state?



Breadth-first search of the 8-puzzle



Properties of breadth-first search

Complete? Yes it always reaches goal (if *b* is finite)

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

(this is the number of nodes we generate)

Space? $O(b^d)$ (keeps every node in memory,

either in fringe or on a path to fringe).

Optimal? Yes (if we guarantee that deeper solutions are less optimal, e.g. step-cost=1).

Space is the bigger problem (more than time)

```
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
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Uniform-cost search

uniform-cost search (**UCS**) is a tree search algorithm used for traversing or searching a weighted tree. Can we guarantee optimality for any step cost?

Uniform-cost Search: Expand node with smallest path cost g(n).

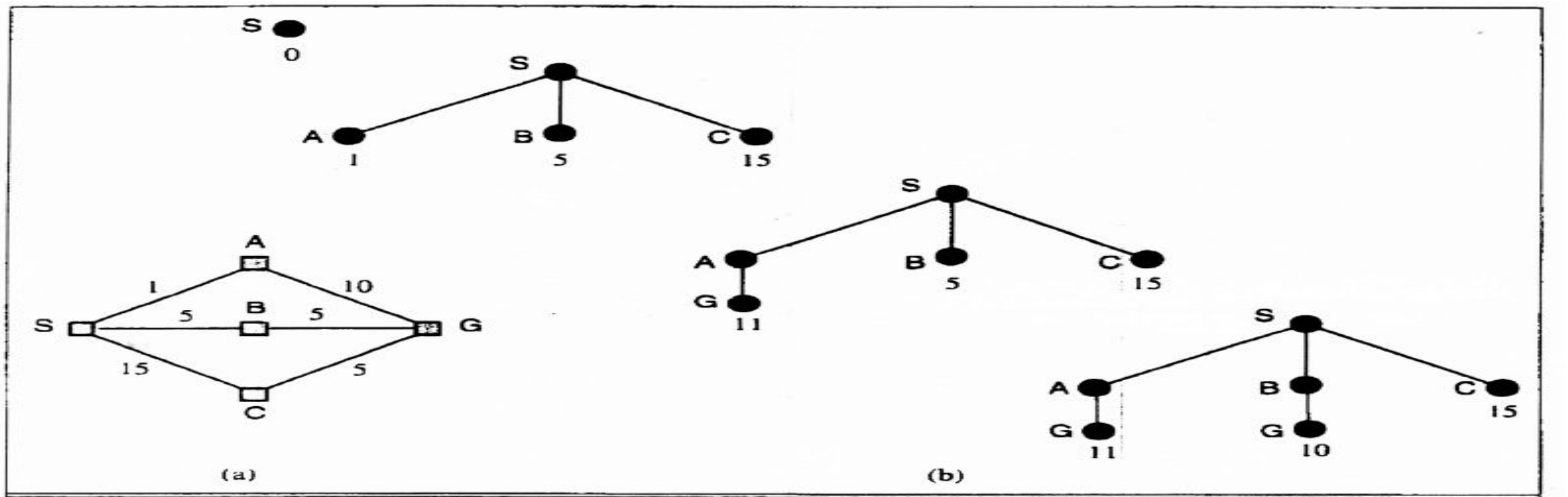


Figure 3.13 A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with $g(n)$. At the next step, the goal node with $g = 10$ will be selected.

At any given point in the execution, the algorithm never expands a node which has a cost greater than the cost of the shortest path in the graph.

Uniform cost search is implemented using priority queue

Uniform-cost search

Implementation *fringe* = queue ordered by path cost
Equivalent to breadth-first if all step costs all equal.

Complete? Yes, if step cost ≥ ε
(otherwise it can get stuck in infinite loops)

Time? # of nodes with *path cost* ≤ cost of optimal solution.

Space? # of nodes on paths with path cost ≤ cost of optimal solution.

Optimal? Yes, for any step cost ≥ ε

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

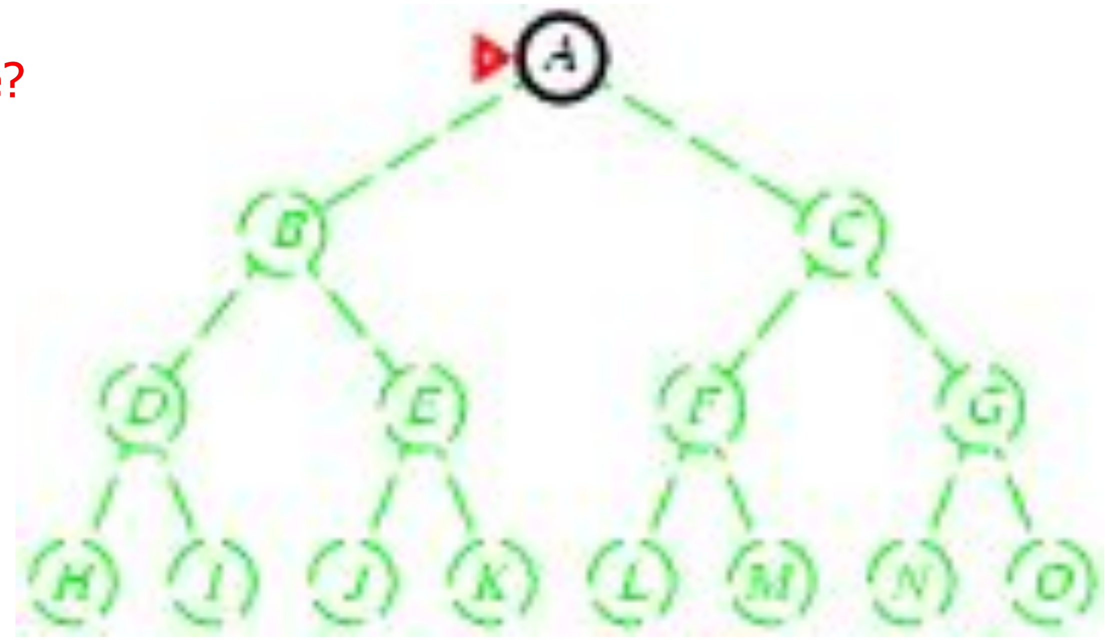
Depth-first search

Expand *deepest* unexpanded node

Implementation:

fringe = Last In First Out (LIPO) queue, i.e., put successors at front

Is A a goal state?



Depth-first search

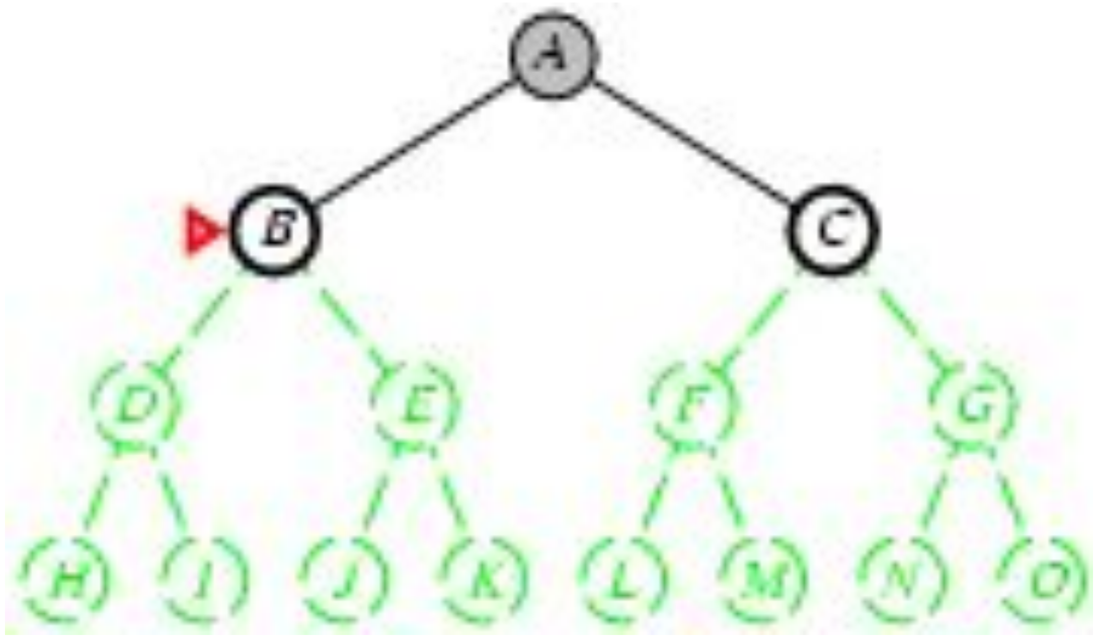
Expand deepest unexpanded node

Implementation:

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

queue=[B,C]

Is B a goal state?



Depth-first search

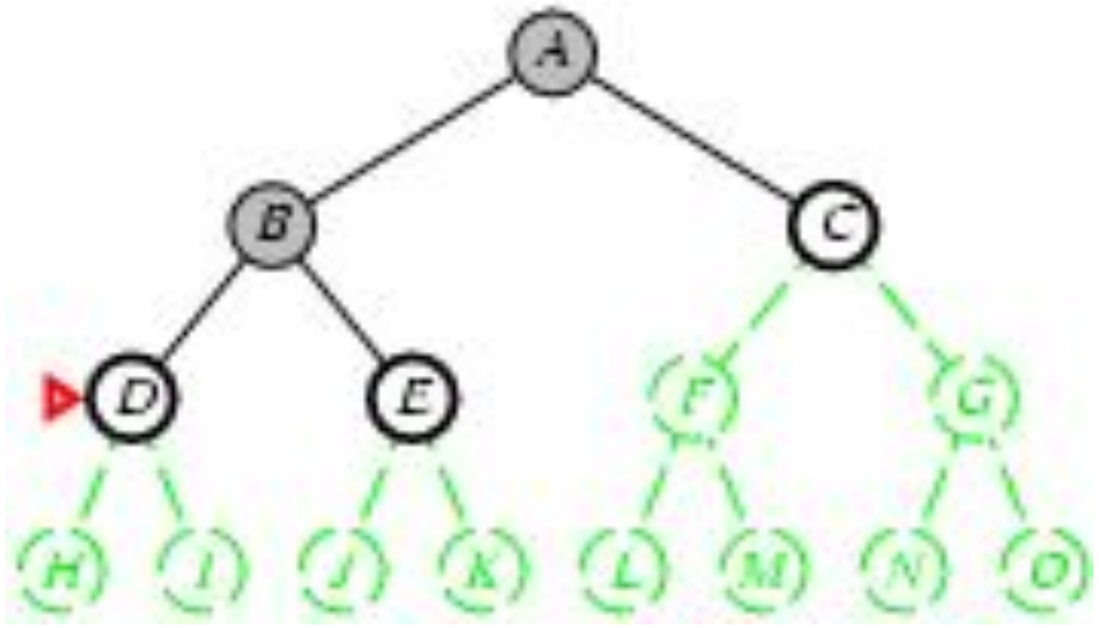
Expand deepest unexpanded node

Implementation:

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

queue=[D,E,C]

Is D = goal state?



Depth-first search

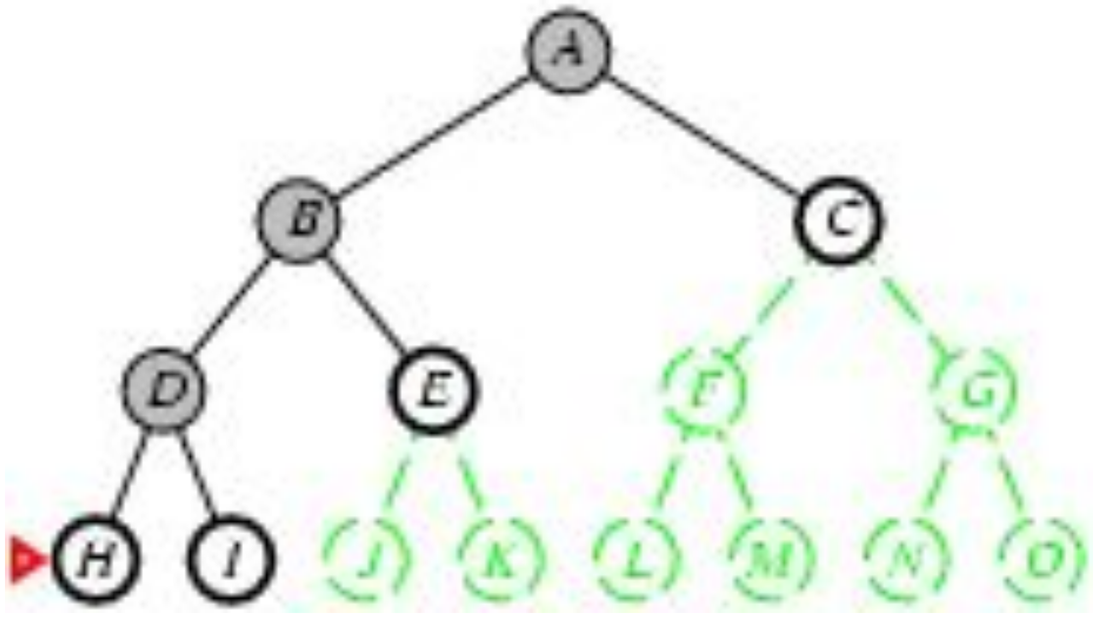
Expand deepest unexpanded node

Implementation:

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

queue=[H,I,E,C]

Is H = goal state?



Depth-first search

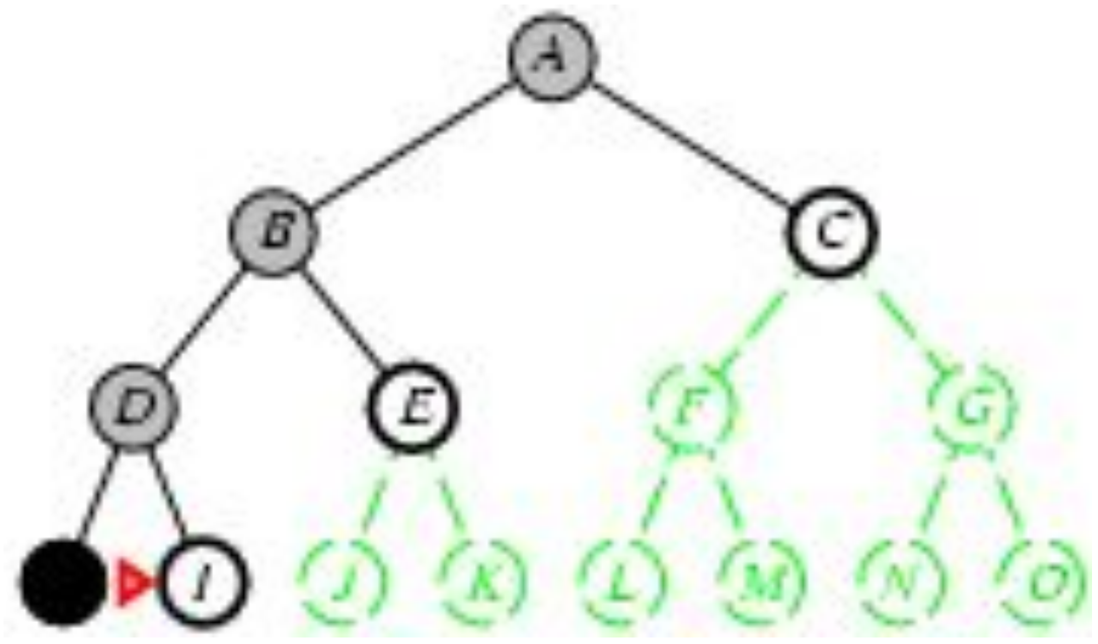
Expand deepest unexpanded node

Implementation:

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

queue=[I,E,C]

Is I = goal state?



Depth-first search

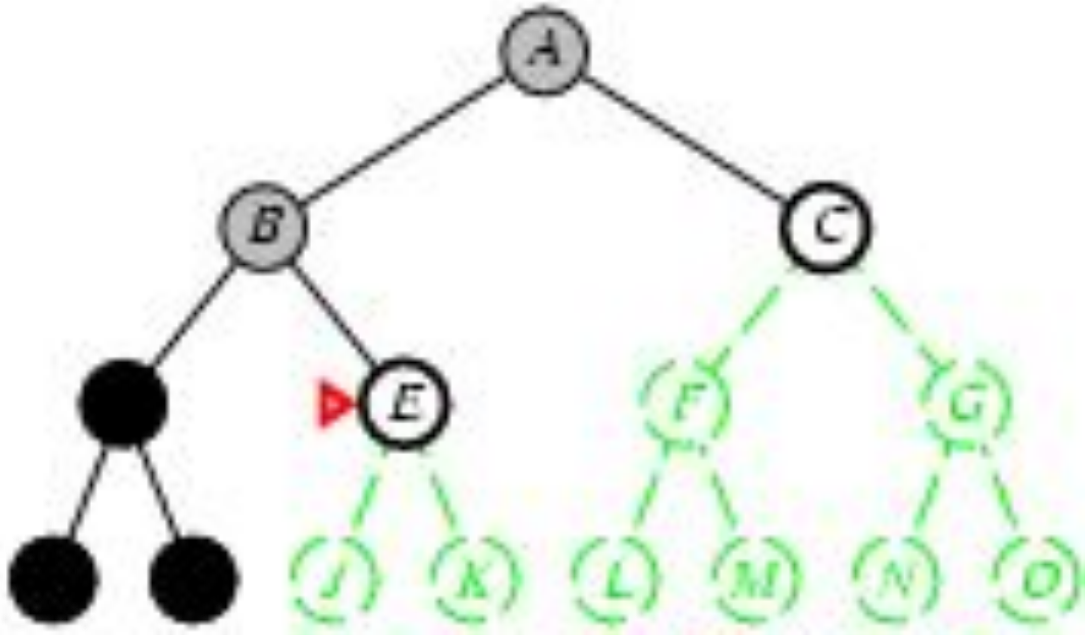
Expand deepest unexpanded node

Implementation:

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

queue=[E,C]

Is E = goal state?



Depth-first search

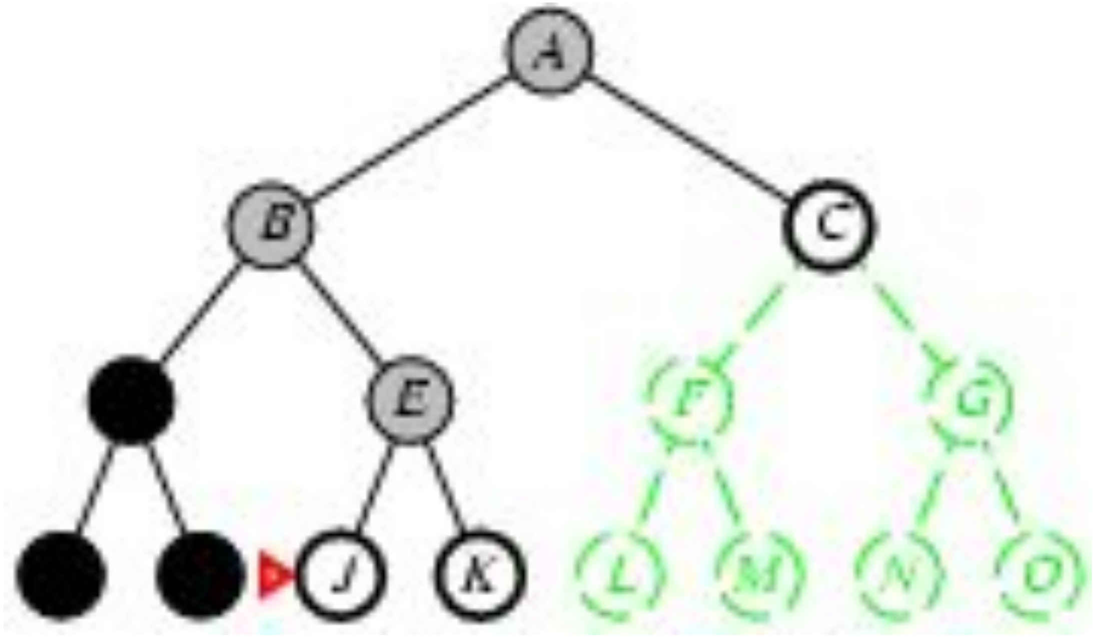
Expand deepest unexpanded node

Implementation:

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

queue=[J,K,C]

Is J = goal state?



Depth-first search

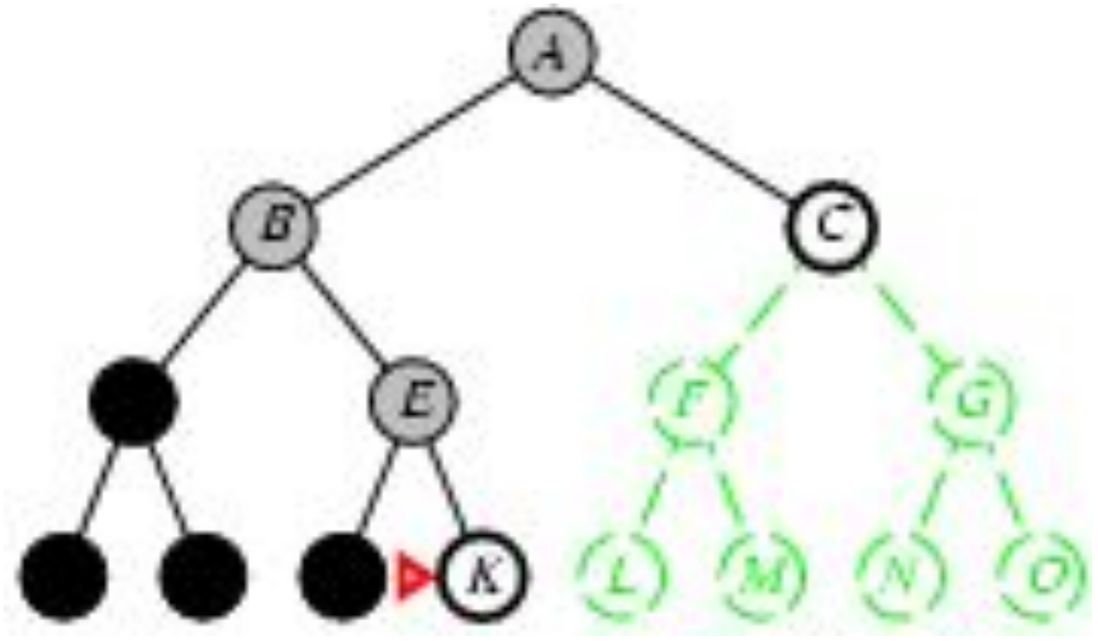
Expand deepest unexpanded node

Implementation:

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

queue=[K,C]

Is K = goal state?



Depth-first search

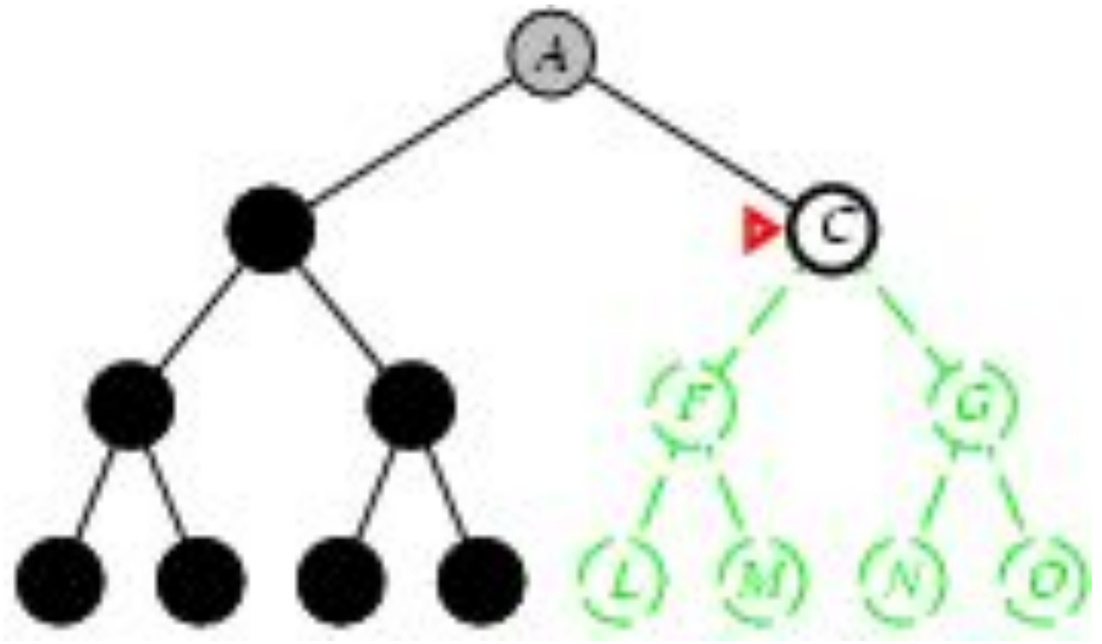
Expand deepest unexpanded node

Implementation:

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

queue=[C]

Is C = goal state?



Depth-first search

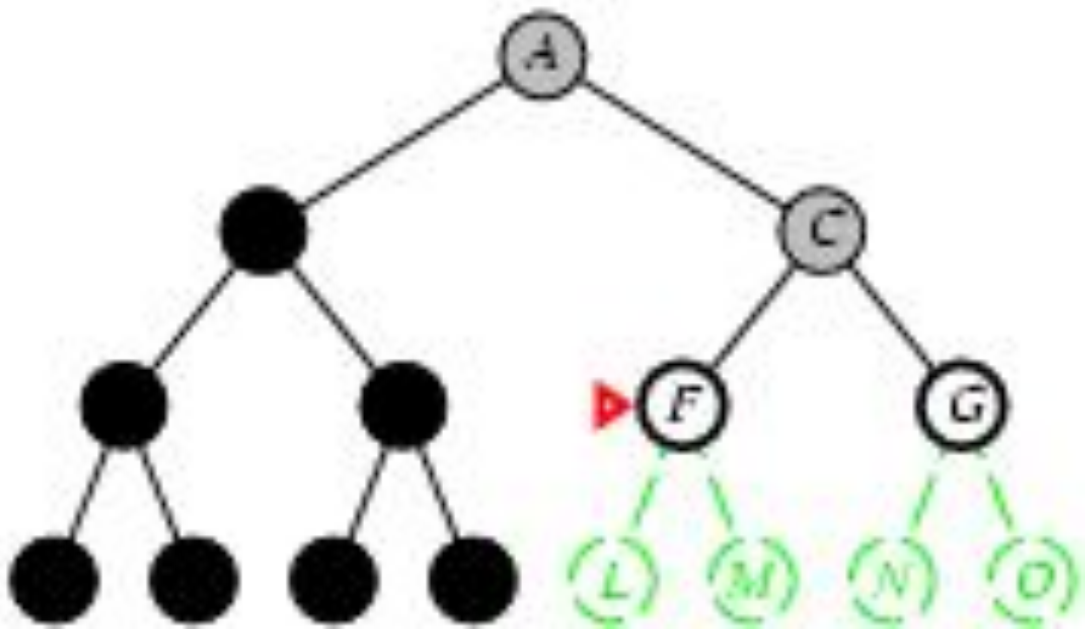
Expand deepest unexpanded node

Implementation:

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

queue=[F,G]

Is F = goal state?



Depth-first search

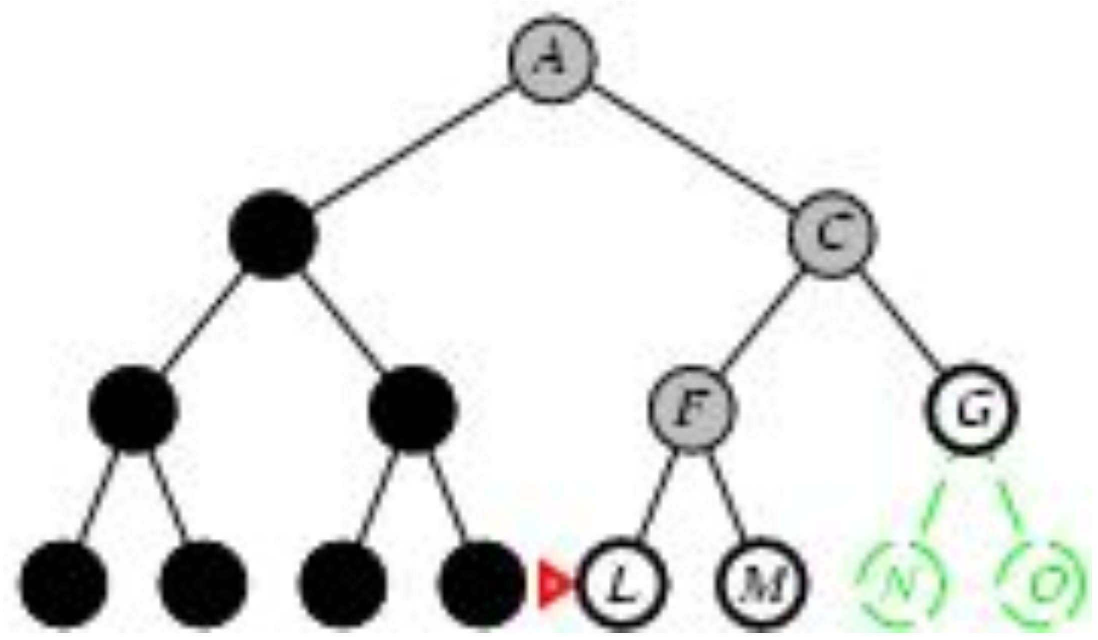
Expand deepest unexpanded node

Implementation:

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

queue=[L,M,G]

Is L = goal state?



Depth-first search

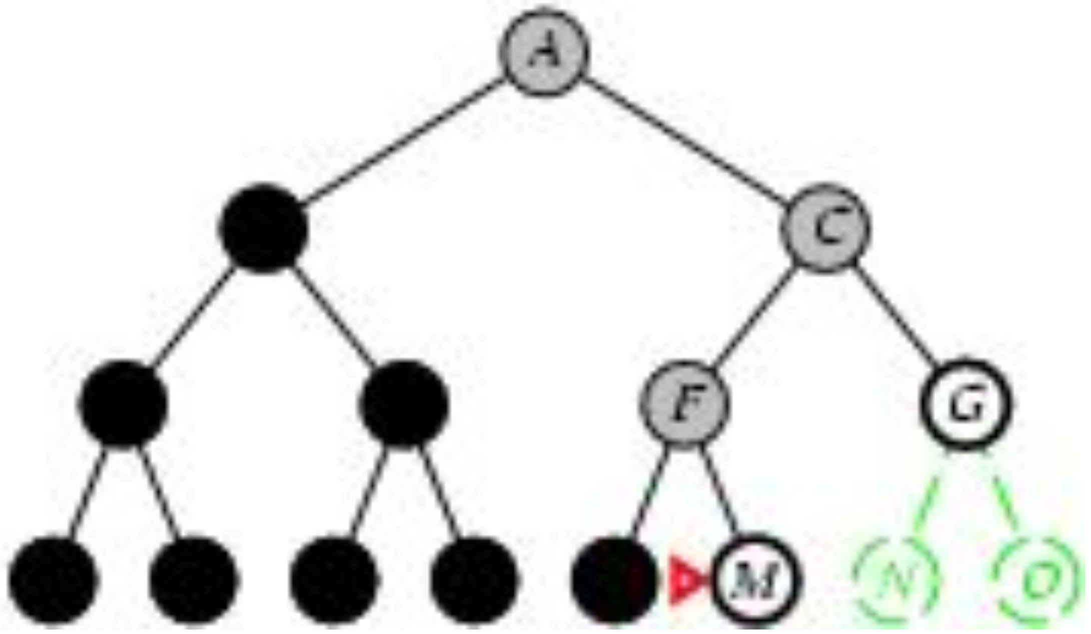
Expand deepest unexpanded node

Implementation:

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

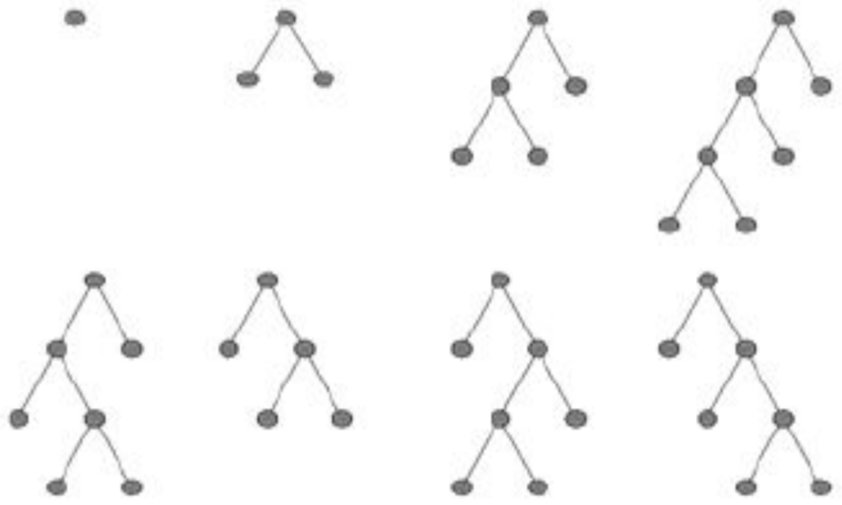
queue=[M,G]

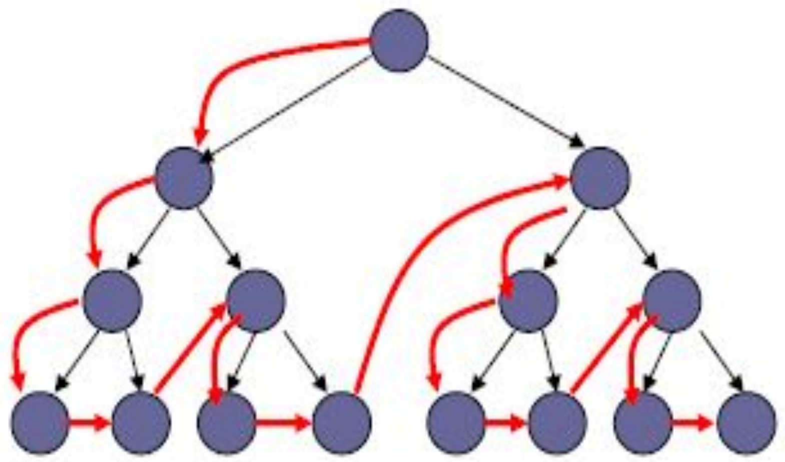
Is M = goal state?



Depth-first search (DFS)

- The deepest node is expanded first
- Backtrack when the path cannot be further expanded





Properties of depth-first search

Complete? No: fails in infinite-depth spaces

Can modify to avoid repeated states along path

Time? $O(b^m)$ with m=maximum depth

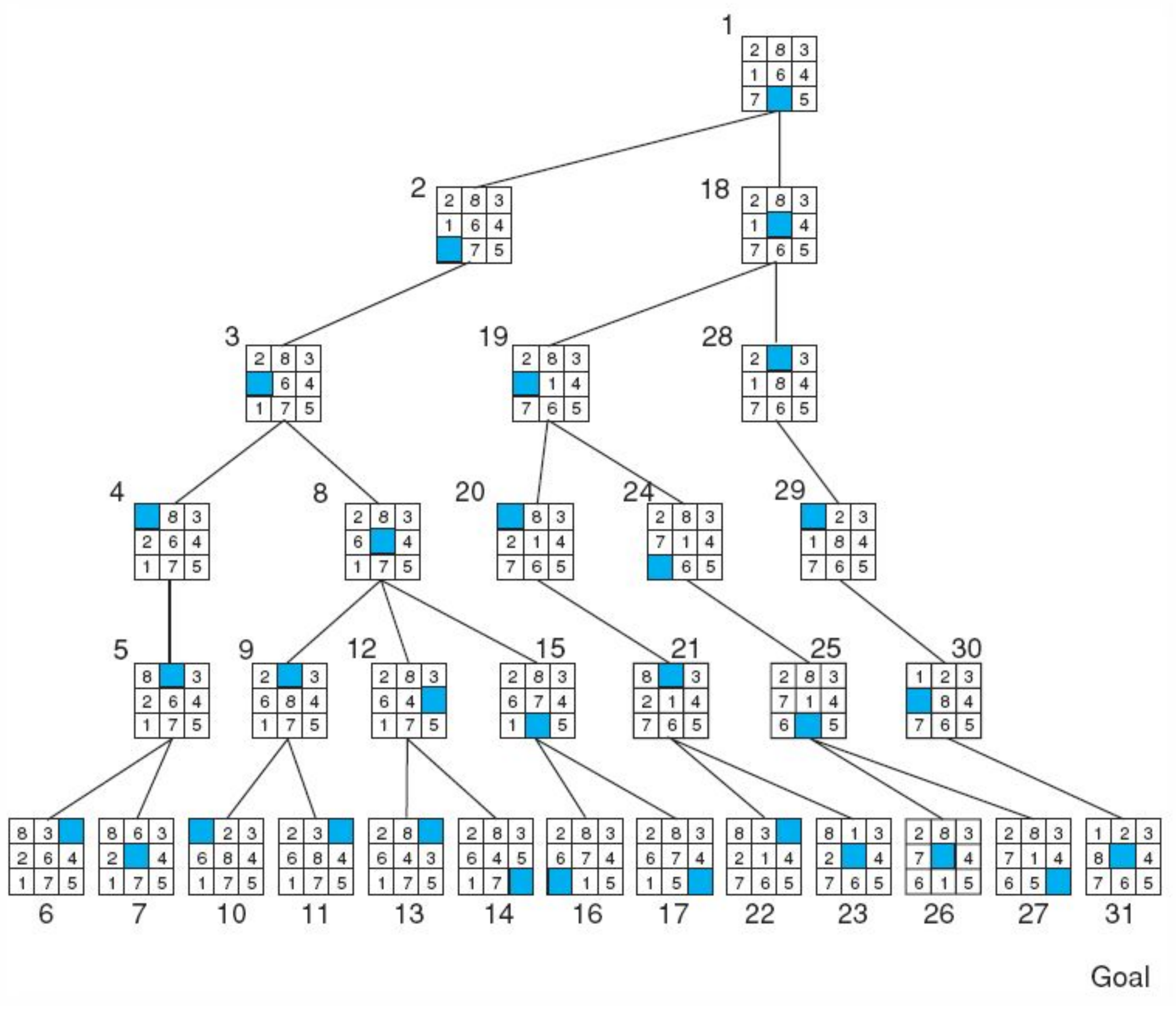
terrible if *m* is much larger than *d*

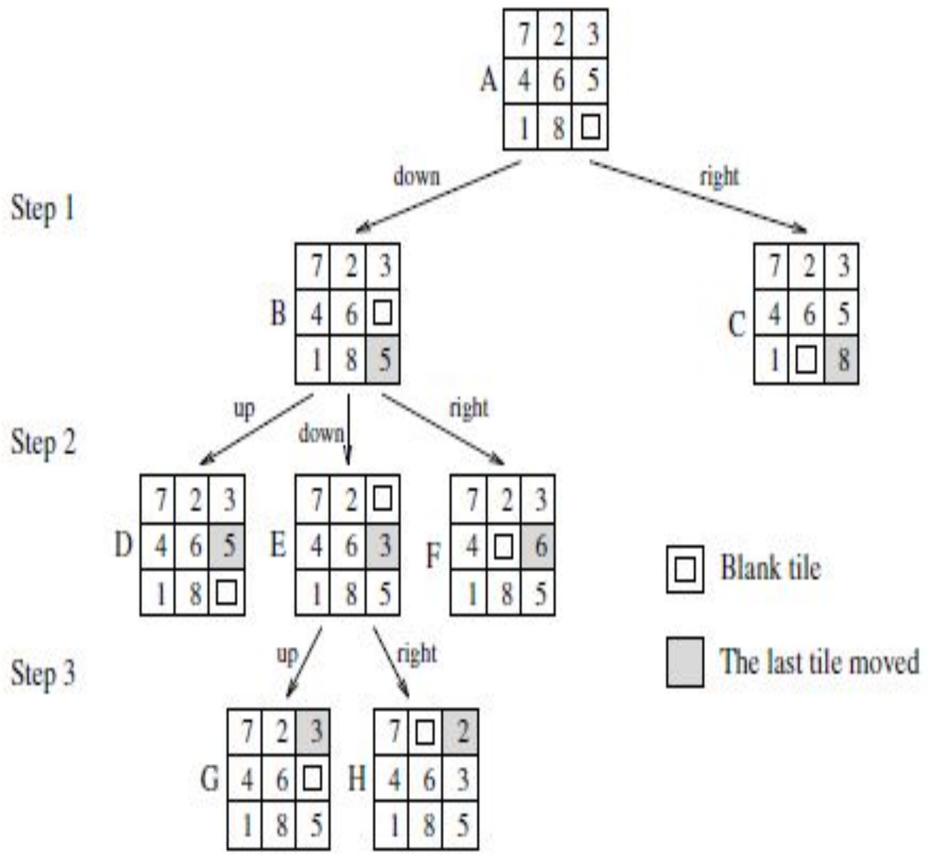
but if solutions are dense, may be much faster than breadth-first

Space? *O(bm),* i.e., linear space! (we only need to remember a single path + expanded unexplored nodes)

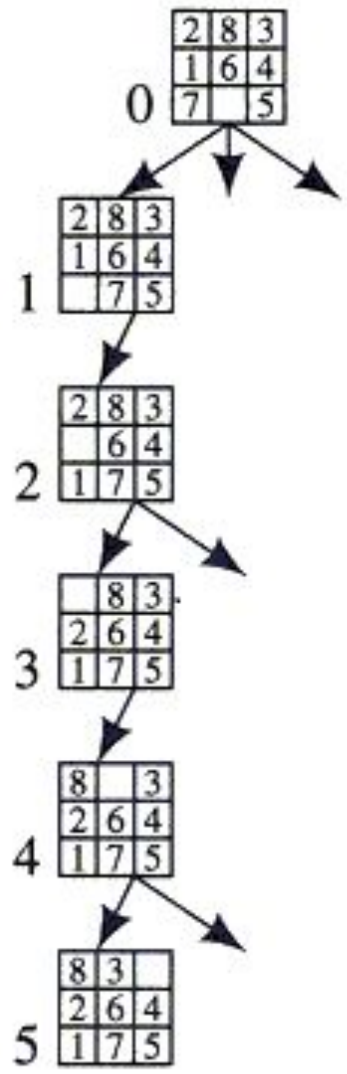
Optimal? No (It may find a non-optimal goal first)

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





Example 1



Example 2

Depth Limited search

Definition

Depth-limited search (DLS) is the variant of DFS with pre-specified depth limit. That is if depth limit is denoted by ℓ then no successor are added further to the node at the depth ℓ. The DLS overcomes the failure of DFS in infinite space and solve the infinite path problem.

- If ℓ is not properly chosen in the case when d is unknown, $\ell < d$, then DLS is incomplete that is if the goal is beyond the depth limit.
- The depth limit also reduces the scope of the search.
- It is also suboptimal, since the algorithm may find first path to the goal instead of shortest path.

Iterative deepening search

- To avoid the infinite depth problem of DFS, we can

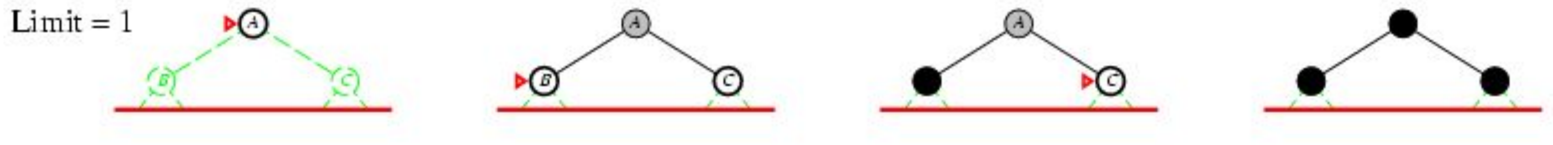
  decide to only search until depth L, i.e. we don’t expand beyond depth L.
- The idea is to do depth-limited DFS repeatedly, with an increasing depth limit, until a solution is found.

  IDDFS combines depth-first search's space-efficiency and breadth-first search's completeness

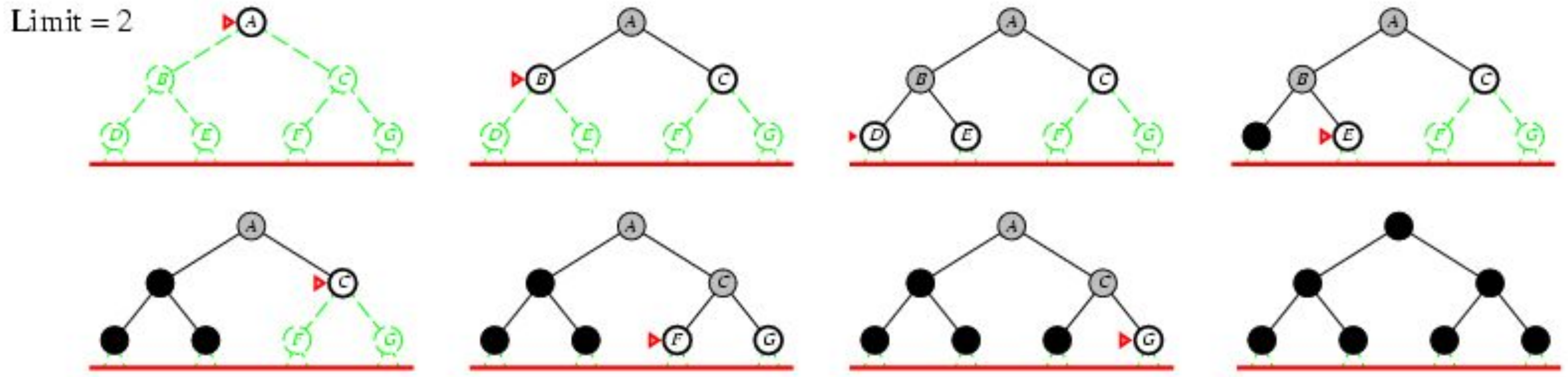
Iterative deepening search *L*=0

Limit = 0

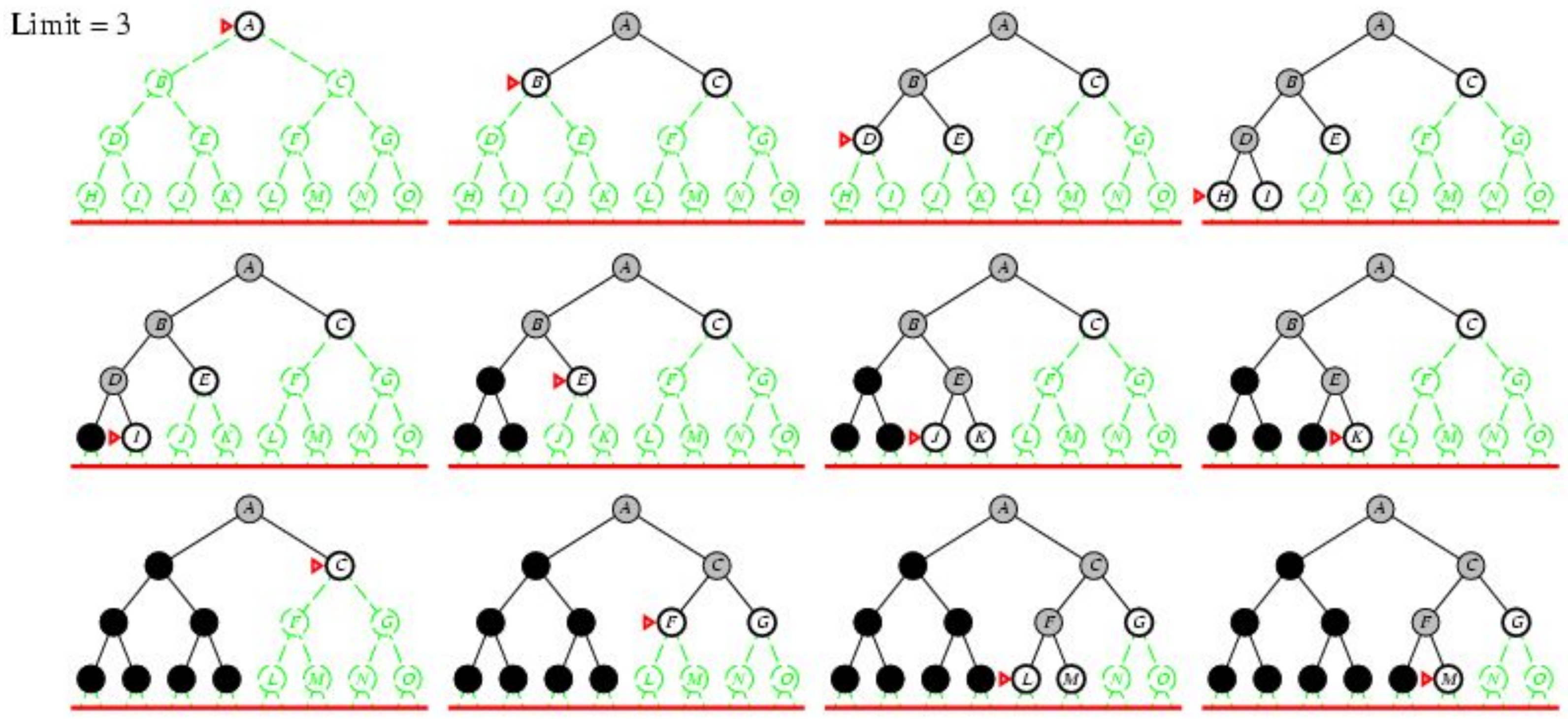
Iterative deepening search *L*=1



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*:

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

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

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

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 = 123,450

N_{BFS} = ……………………………………………………………………………… = 1,111,100

$$O(b^d) \neq O(\overset{\nwarrow}{b^{d+1}})$$

Properties of iterative deepening search

Complete? Yes

Time? $O(b^d)$

Space? $O(bd)$

Optimal? Yes, if step cost = 1 or increasing function of depth.

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

```
function ITERATIVE-DEEPENING-SEARCH($problem$) returns a solution, or failure
   for $depth$ = 0 to $\infty$ do
      $result \leftarrow$ DEPTH-LIMITED-SEARCH($problem, depth$)
      if $result \neq$ cutoff then return $result$
```

Bidirectional Search

Idea

simultaneously search forward from S and backwards from G

stop when both “meet in the middle”

need to keep track of the intersection of 2 open sets of nodes

What does searching backwards from G mean

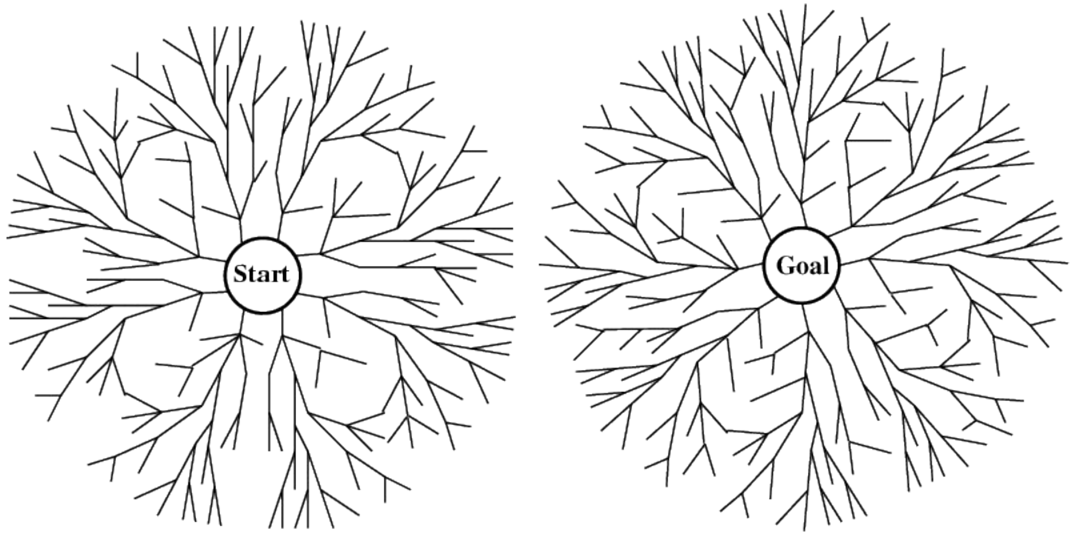
need a way to specify the predecessors of G

- this can be difficult,
- e.g., predecessors of checkmate in chess?

which to take if there are multiple goal states?

where to start if there is only a goal test, no explicit list?

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.

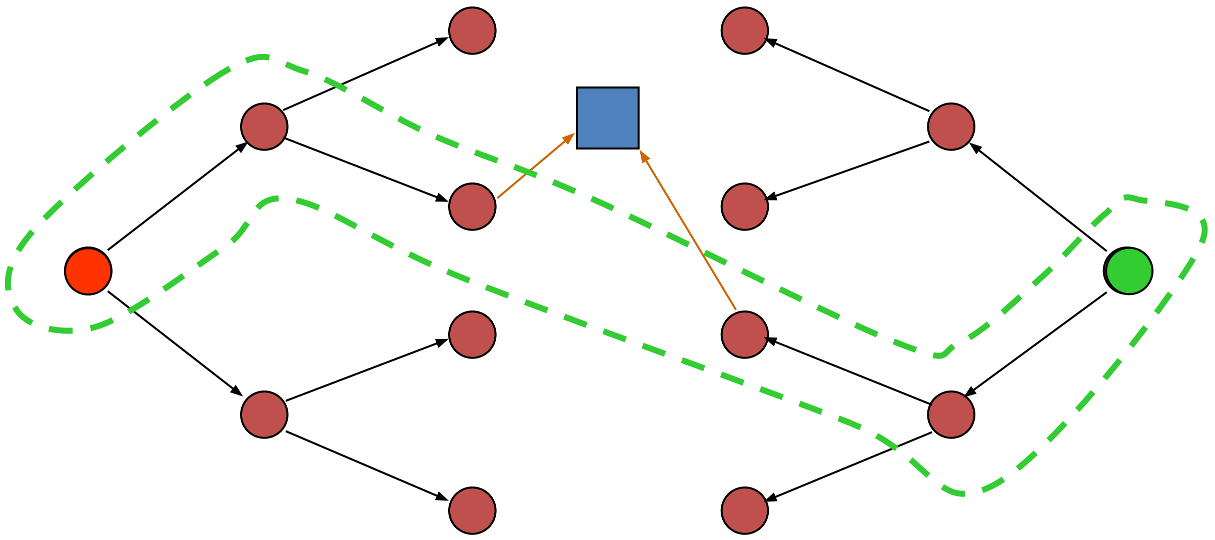
Requires the ability to generate “predecessor” states.

Can (sometimes) lead to finding a solution more quickly.

Time complexity: O($b^{d/2}$).   Space complexity: O($b^{d/2}$).

Bidirectional Search

2 fringe queues: FRINGE1 and FRINGE2



The predecessor of each node should be efficiently computable.