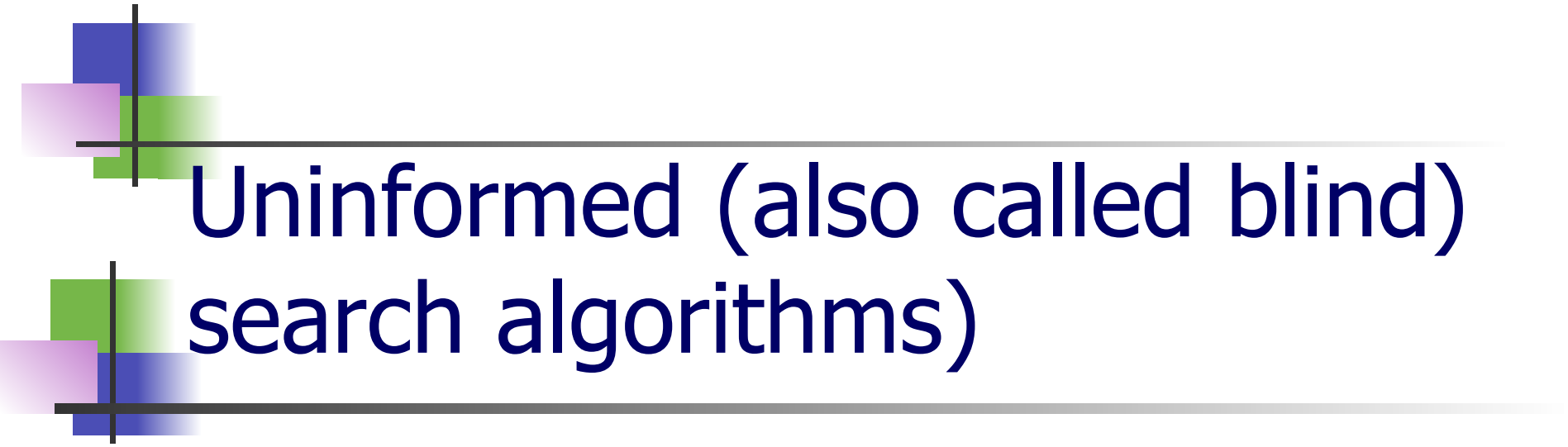
3		**Solving Problems by Searching**	12
	3.1	Definition, State space representation, Problem as a state space search, Problem formulation, Well-defined problems	
	3.2	Solving Problems by Searching, Performance evaluation of search strategies, Time Complexity, Space Complexity, Completeness, Optimality	
	3.3	Uninformed Search: Depth First Search, Breadth First Search, Depth Limited Search, Iterative Deepening Search, Uniform Cost Search, Bidirectional Search	
	3.4	Informed Search: Heuristic Function, Admissible Heuristic, Informed Search Technique, Greedy Best First Search, A* Search, Local Search: Hill Climbing Search, Simulated Annealing Search, Optimization: Genetic Algorithm	
	3.5	Game Playing, Adversarial Search Techniques, Mini-max Search, Alpha-Beta Pruning	


Uninformed (also called blind)
search algorithms)

State space searching

- **Uninformed Searching**
- Search without information
- No knowledge
- Time Consuming
- More time and space complexity
- DFS,BFS etc

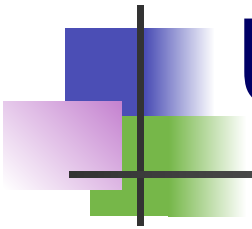
- **Informed Searching**
- Search with information
- Use knowledge to find steps to solution
- Quick solution
- Less time and space complexity
- A* , Heuristic DFS, Best First Search

Uninformed Searching

1) Search without Information
2) No knowledge
3) Time Consuming
4) More Complexity (Time, Space)
5) DFS, BFS etc.

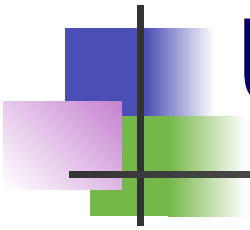
Informed Searching

1) Search with information
2) Use knowledge to find steps to solution
3) Quick solution
4) Less Complexity (Time, Space)
5) A*, Heuristic DFS, Best first Search



Uninformed search strategies

- Uninformed (blind):
  - You have no clue whether one non-goal state is better than any other. Your search is blind. You don't know if your current exploration is likely to be fruitful.
- Various blind strategies:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Iterative deepening search (generally preferred)
  - Bidirectional search (preferred if applicable)



Uninformed search strategies

- Queue for Frontier:
  - FIFO? LIFO? Priority?
- Goal-Test:
  - When inserted into *Frontier*? When removed?
- Tree Search or Graph Search:
  - Forget *Explored* nodes? Remember them?

Queue for Frontier

- FIFO (First In, First Out)
  - Results in Breadth-First Search
- LIFO (Last In, First Out)
  - Results in Depth-First Search
- Priority Queue sorted by path cost so far
  - Results in Uniform Cost Search
- Iterative Deepening Search uses Depth-First
- Bidirectional Search can use either Breadth-First or Uniform Cost Search

Search strategy evaluation

- A search strategy is defined by 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
  - 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 ∞)

Uninformrd search

1. **Breadth-first Search**
2. Depth-first Search
3. Depth-limited Search
4. Iterative deepening depth-first search
5. Uniform cost search
6. Bidirectional Search

Notes

1. Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

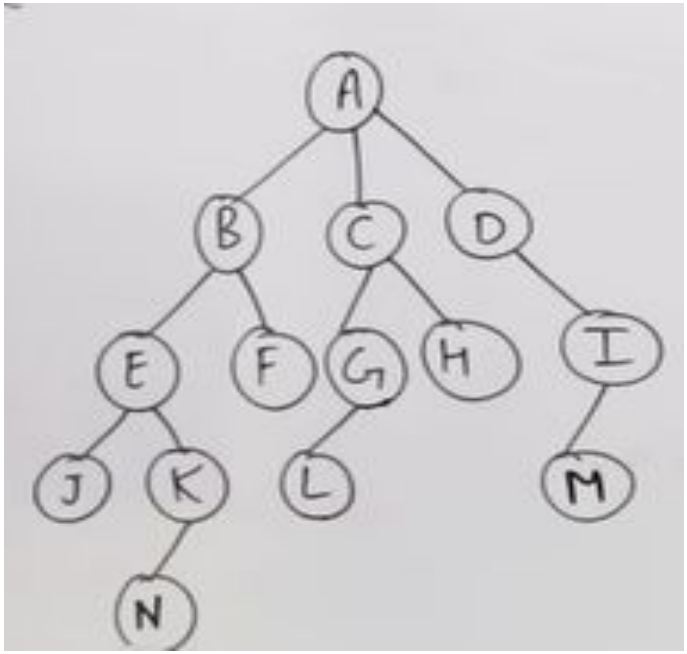
Disadvantages:

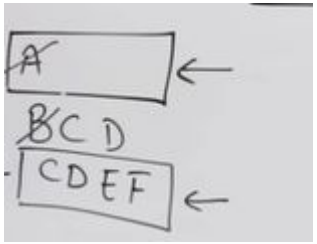
- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

Breadth First Search

- → Uninformed Search technique
- → FIFO (Queue)
- → Shallowest Node
- → Complete

- → optimal
- → Time Complexity





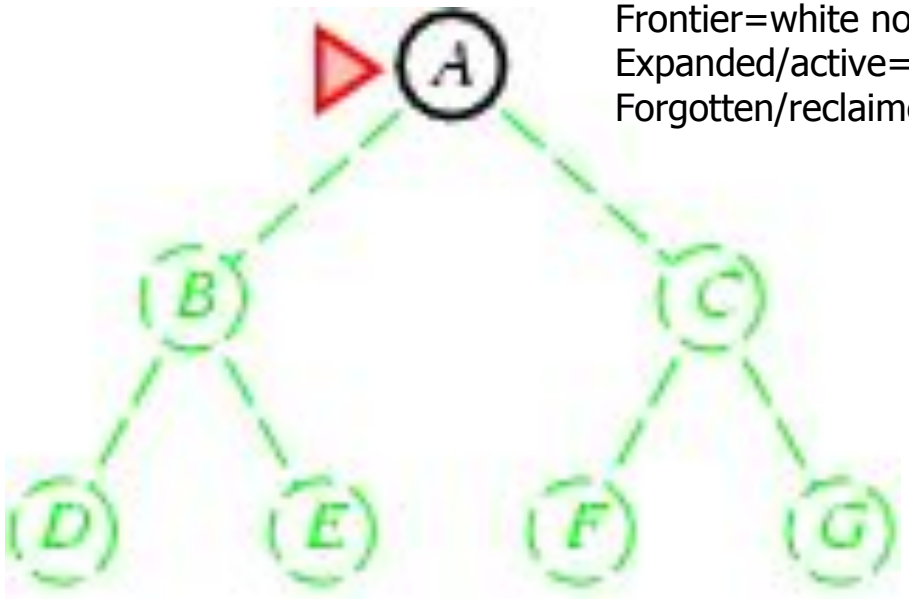
Breadth-first search

- Expand shallowest unexpanded node
- *Frontier* (or fringe): nodes in queue to be explored
- *Frontier* is a first-in-first-out (FIFO) queue, i.e., new successors go at end of the queue.
- *Goal-Test* when inserted.

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Initial state = A
Is A a goal state?

Put A at end of queue.
frontier = [A]

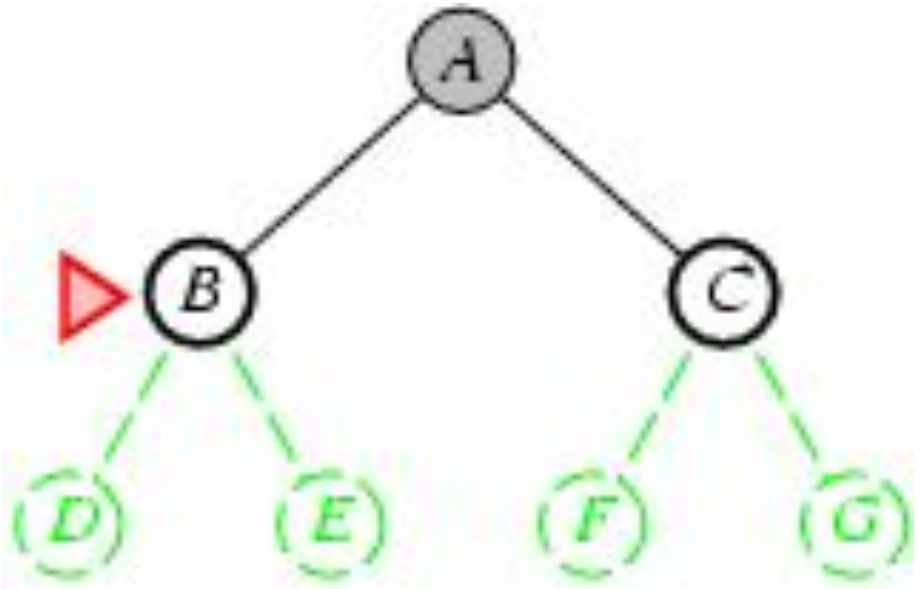


Breadth-first search

- Expand shallowest unexpanded node
- *Frontier* is a FIFO queue, i.e., new successors go at end

Expand A to B, C.
Is B or C a goal state?

Put B, C at end of queue.
frontier = [B,C]

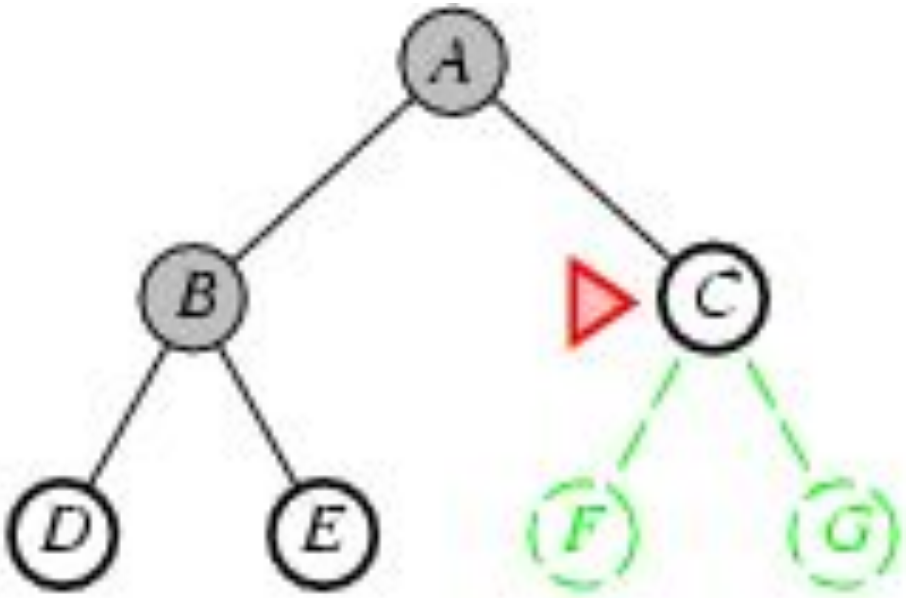


Breadth-first search

- Expand shallowest unexpanded node
- *Frontier* is a FIFO queue, i.e., new successors go at end

Expand B to D, E
Is D or E a goal state?

Put D, E at end of queue
frontier=[C,D,E]

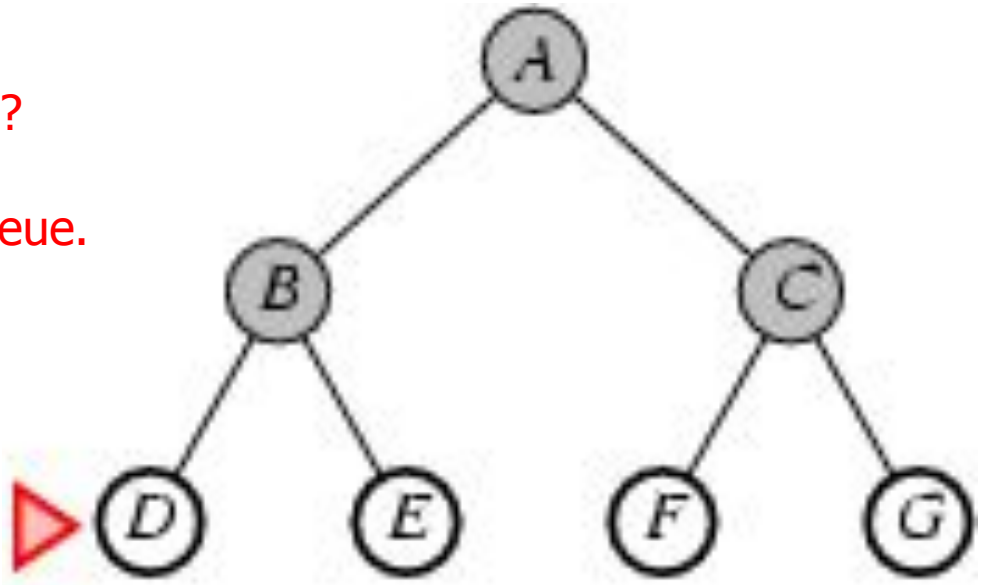


Breadth-first search

- Expand shallowest unexpanded node
- *Frontier* is a FIFO queue, i.e., new successors go at end

Expand C to F, G.
Is F or G a goal state?

Put F, G at end of queue.
frontier = [D,E,F,G]

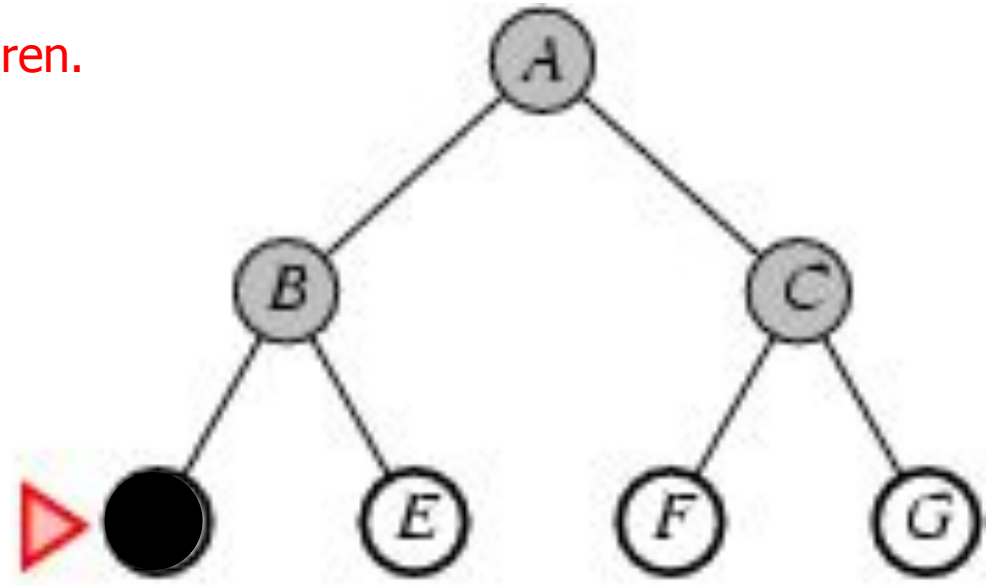


Breadth-first search

- Expand shallowest unexpanded node
- *Frontier* is a FIFO queue, i.e., new successors go at end

Expand D to no children.
Forget D.

frontier = [E,F,G]

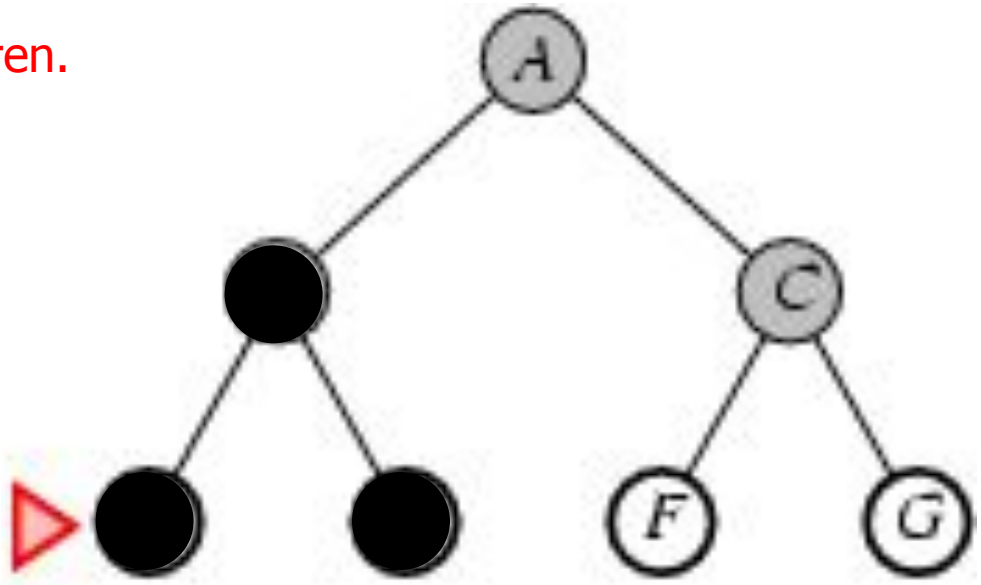


Breadth-first search

- Expand shallowest unexpanded node
- *Frontier* is a FIFO queue, i.e., new successors go at end

Expand E to no children.
Forget B,E.

frontier = [F,G]



Start state

Searching for a solution to the 8-puzzle.

Aside: in this tree, immediate duplicates are removed.

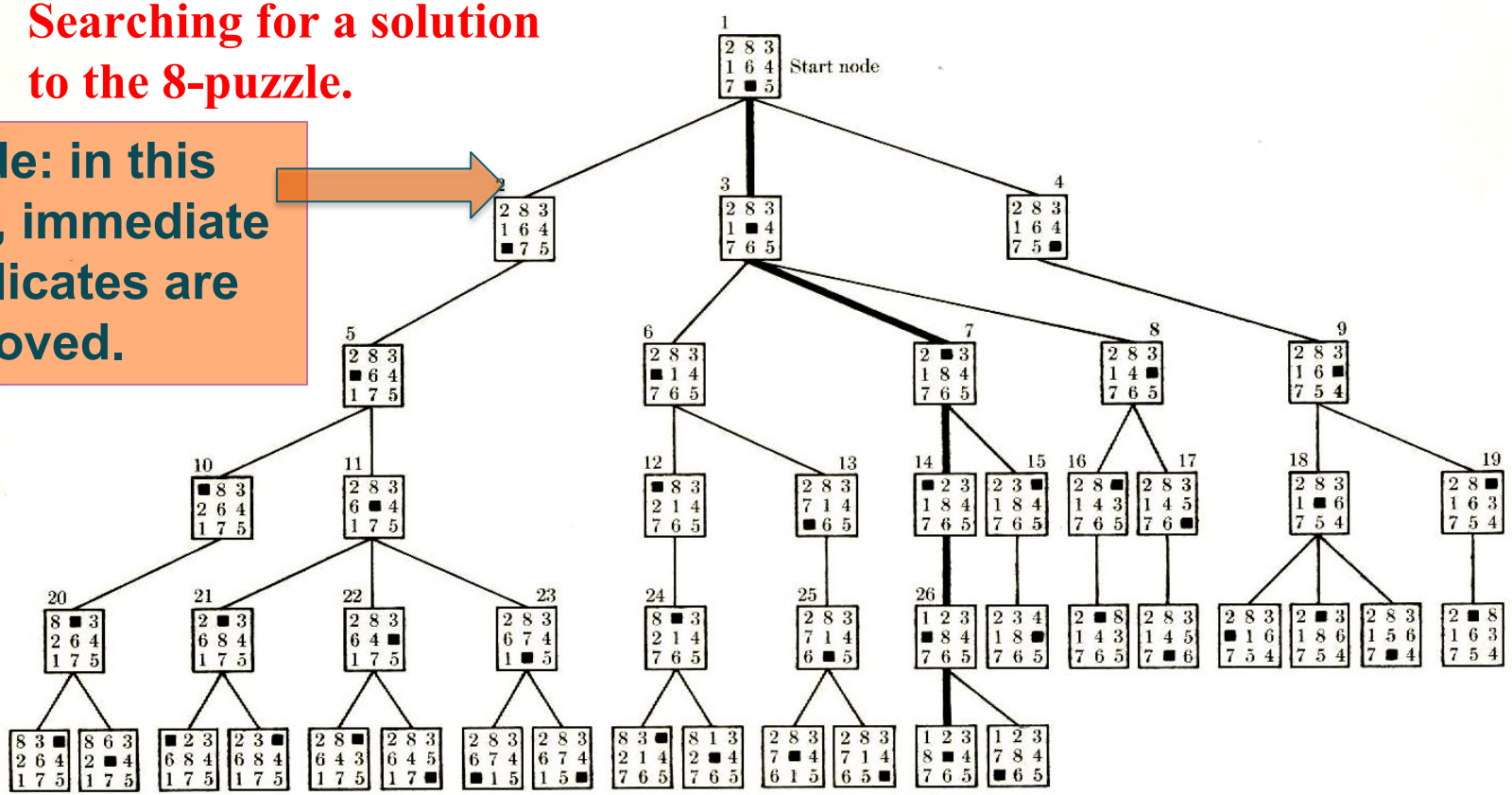
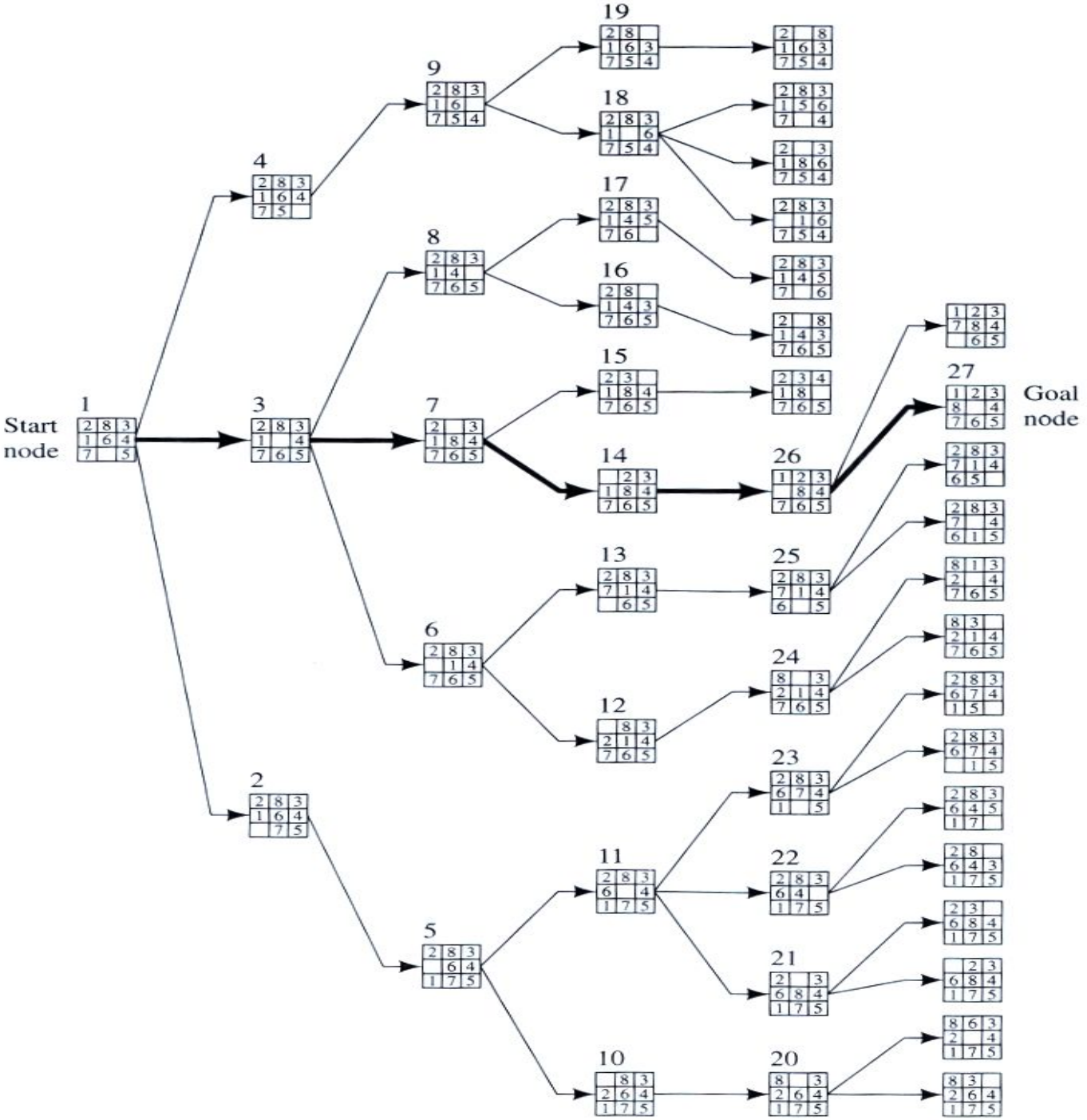


FIG. 3-2 *The tree produced by a bre*adth-first search.

A breadth-first search tree. (More detail soon.)

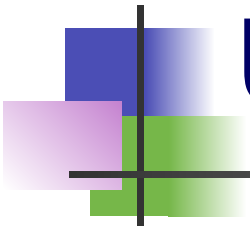
Branching factor 1, 2, or 3 (max). So, approx. 2 --- # nodes roughly doubles at each level. Number states of explored nodes grows exponentially with depth.

Example BFS


Start node
Goal node
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

Properties of breadth-first search

- Complete? Yes, it always reaches a 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? No, for general cost functions.
  Yes, if cost is a non-decreasing function only of depth.
  - With $f(d) \geq f(d-1)$, e.g., step-cost = constant:
    - All optimal goal nodes occur on the same level
    - Optimal goal nodes are always shallower than non-optimal goals
    - An optimal goal will be found before any non-optimal goal
- Space is the bigger problem (more than time)



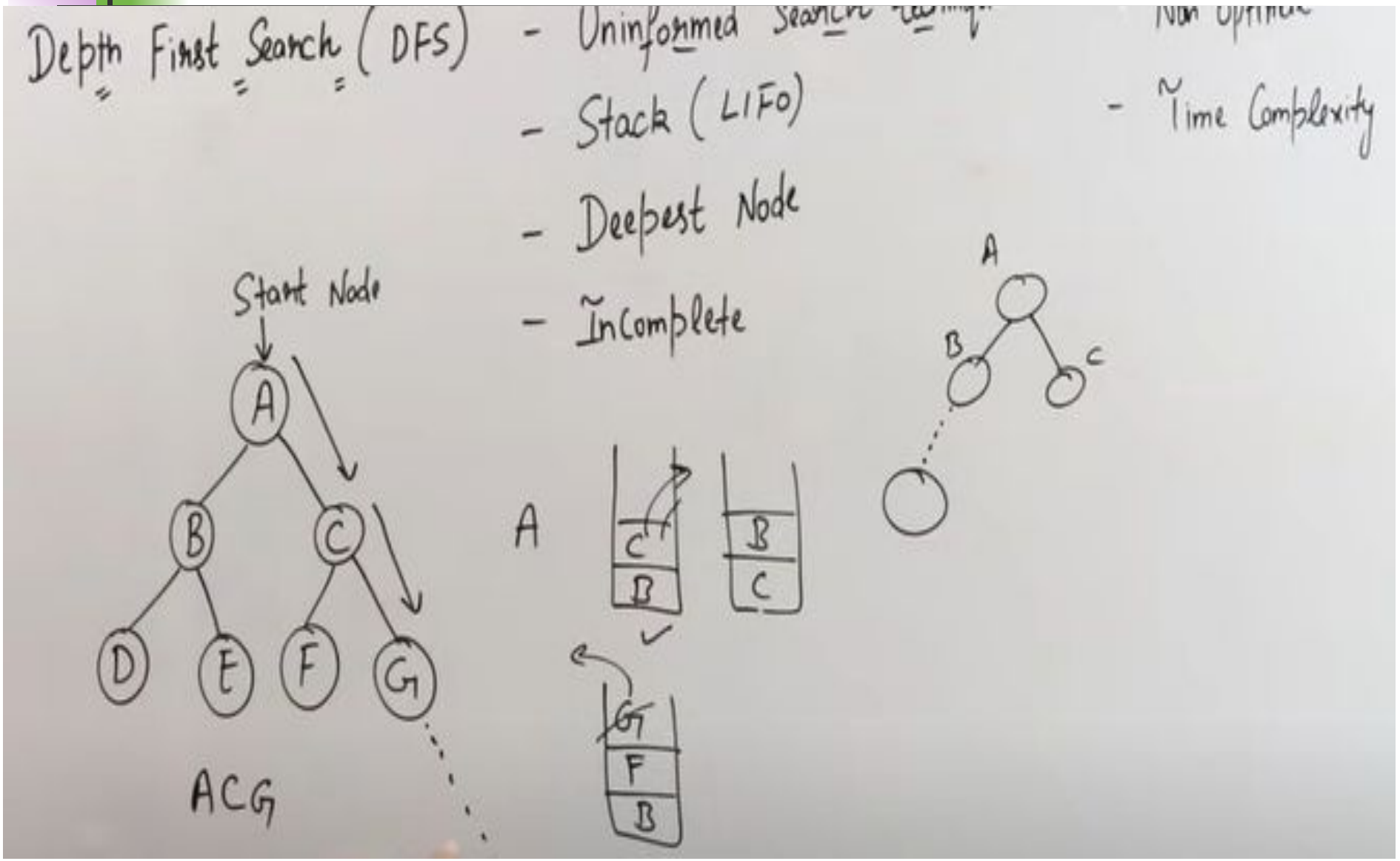
Uninformrd search

1. Breadth-first Search
2. **Depth-first Search**
3. Depth-limited Search
4. Iterative deepening depth-first search
5. Uniform cost search
6. Bidirectional Search

2. Depth-first Search

- Depth-first search isa recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.
- Note: Backtracking is an algorithm technique for finding all possible solutions using recursion.

DEPTH First Search



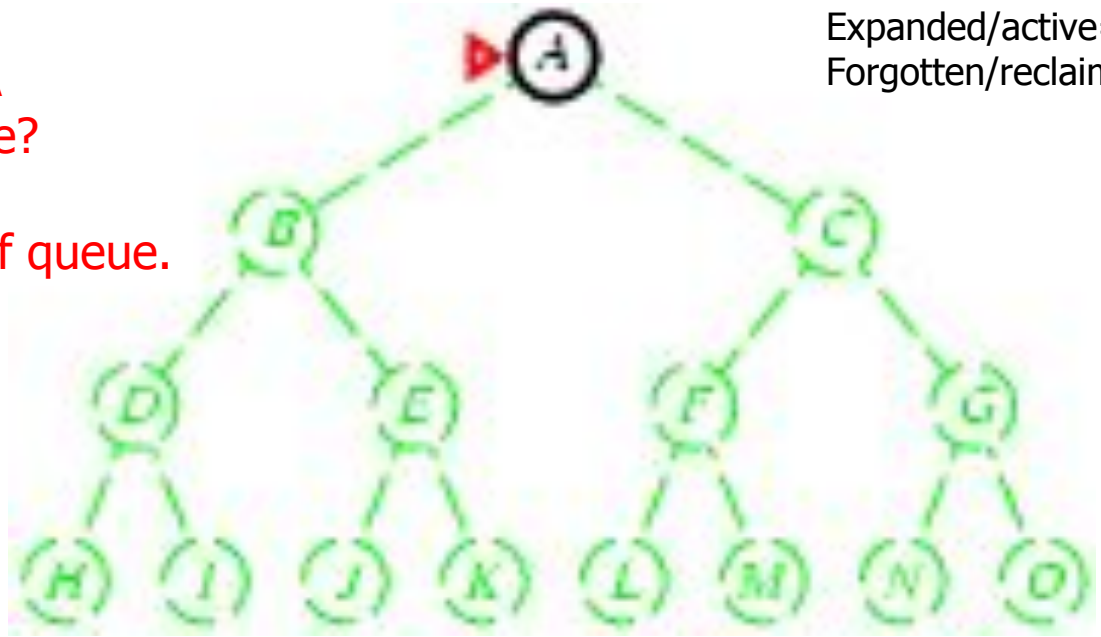
Depth-first search

- Expand *deepest* unexpanded node
- *Frontier* = Last In First Out (LIFO) queue, i.e., new successors go at the front of the queue.
- *Goal-Test* when inserted.

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Initial state = A
Is A a goal state?

Put A at front of queue.
frontier = [A]

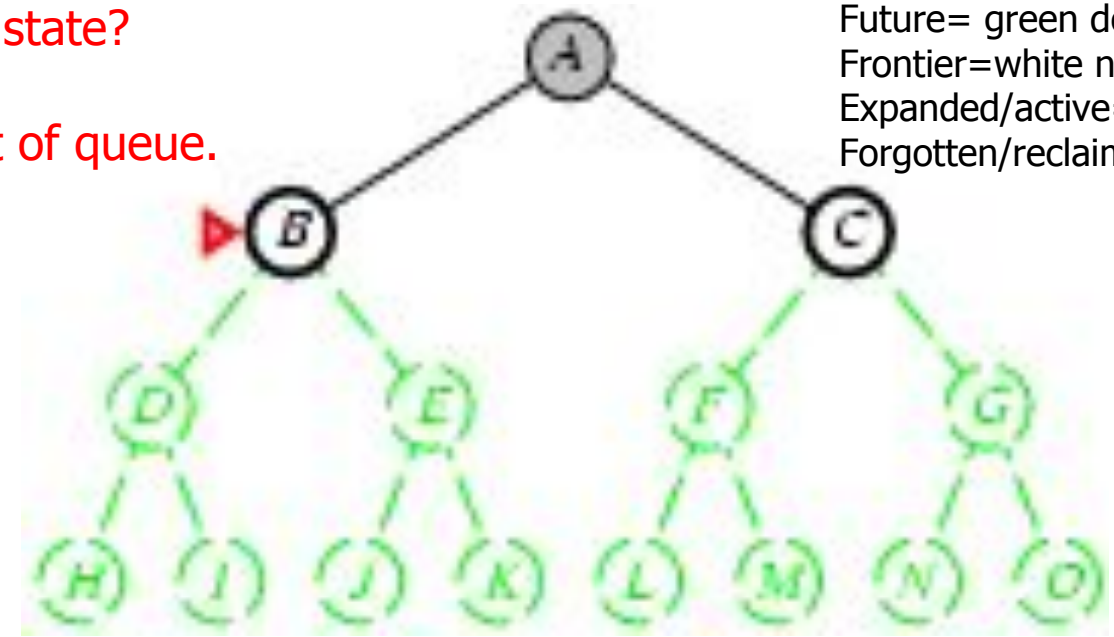


Depth-first search

- Expand deepest unexpanded node
  - *Frontier* = LIFO queue, i.e., put successors at front

Expand A to B, C.
Is B or C a goal state?

Put B, C at front of queue.
frontier = [B,C]



Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

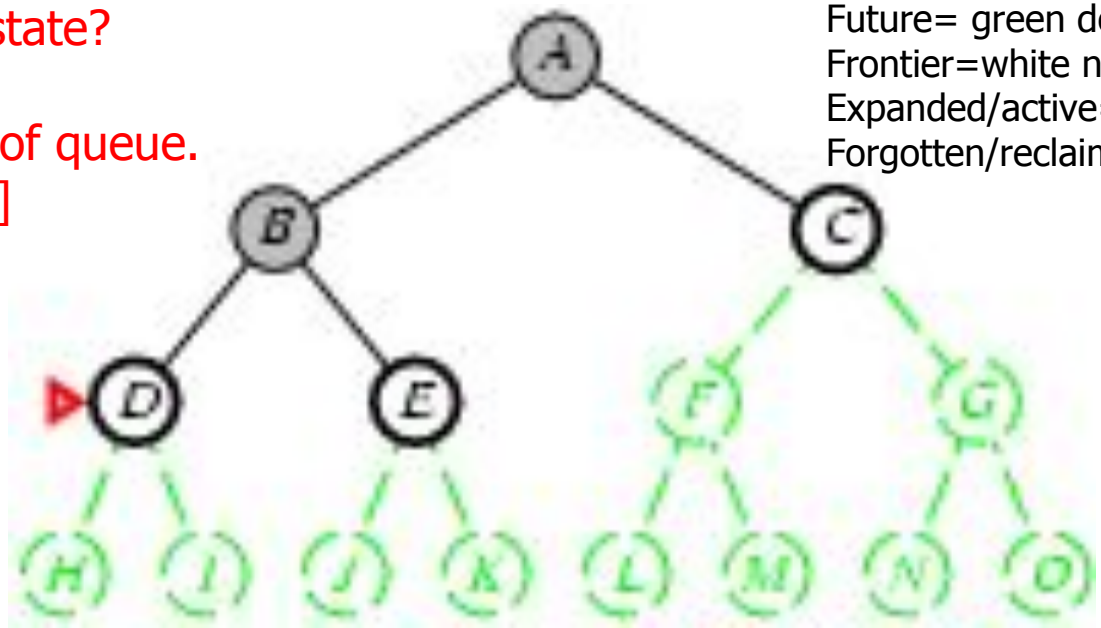
Note: Can save a space factor of *b* by generating successors one at a time.
See **backtracking search** in your book, p. 87 and Chapter 6.

Depth-first search

- Expand deepest unexpanded node
  - *Frontier* = LIFO queue, i.e., put successors at front

Expand B to D, E.
Is D or E a goal state?

Put D, E at front of queue.
frontier = [D,E,C]



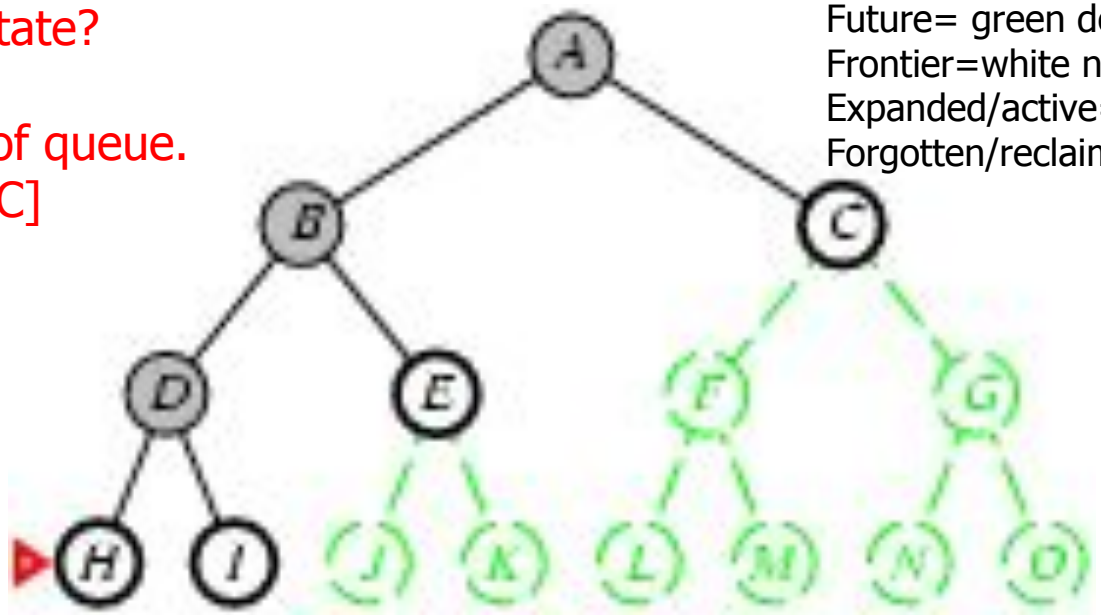
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Depth-first search

- Expand deepest unexpanded node
  - *Frontier* = LIFO queue, i.e., put successors at front

Expand D to H, I.
Is H or I a goal state?

Put H, I at front of queue.
frontier = [H,I,E,C]



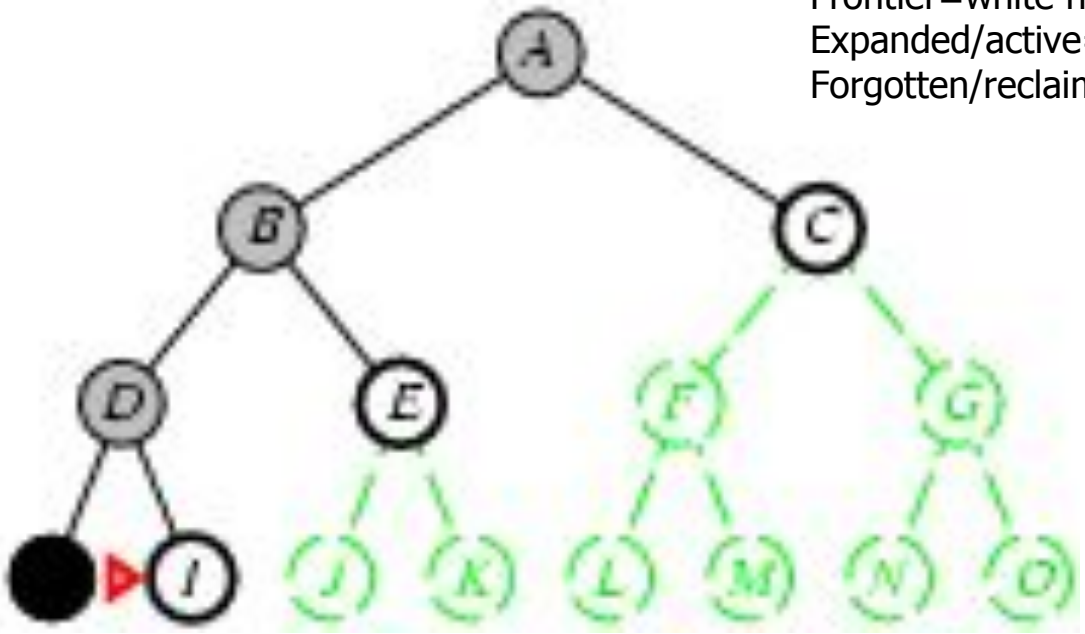
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Depth-first search

- Expand deepest unexpanded node
  - *Frontier* = LIFO queue, i.e., put successors at front

Expand H to no children.
Forget H.

frontier = [I,E,C]



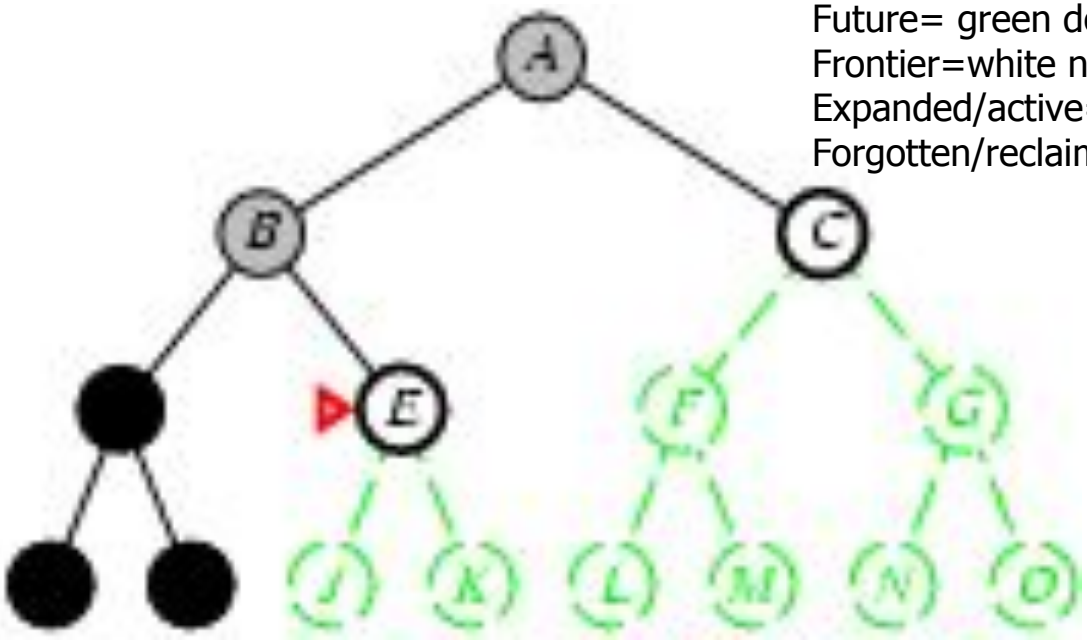
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Depth-first search

- Expand deepest unexpanded node
  - *Frontier* = LIFO queue, i.e., put successors at front

Expand I to no children.
Forget D, I.

frontier = [E,C]



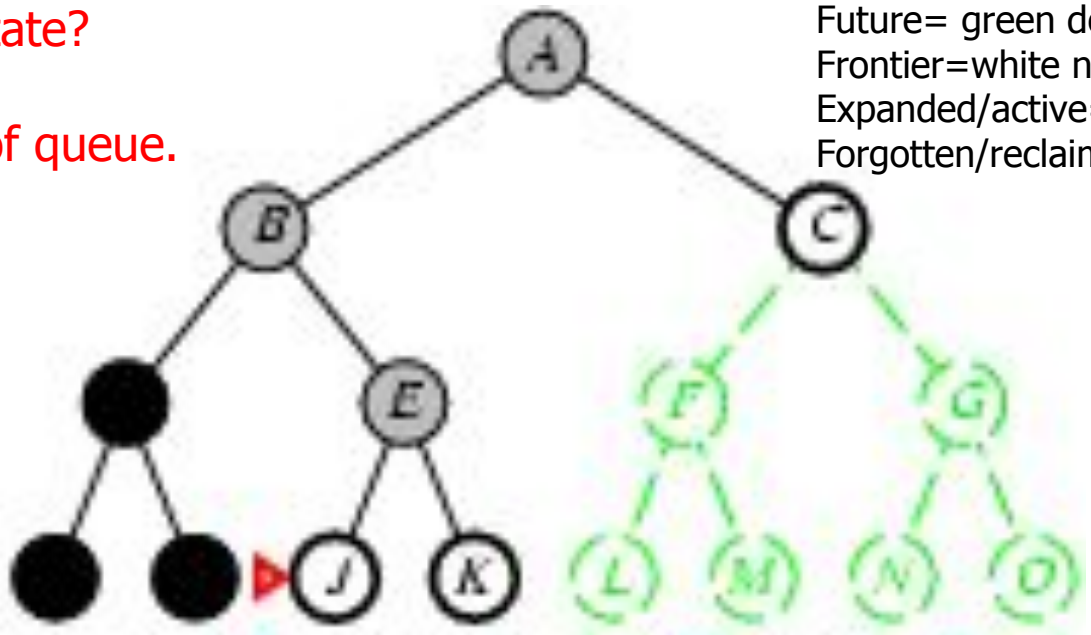
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Depth-first search

- Expand deepest unexpanded node
  - *Frontier* = LIFO queue, i.e., put successors at front

Expand E to J, K.
Is J or K a goal state?

Put J, K at front of queue.
frontier = [J,K,C]



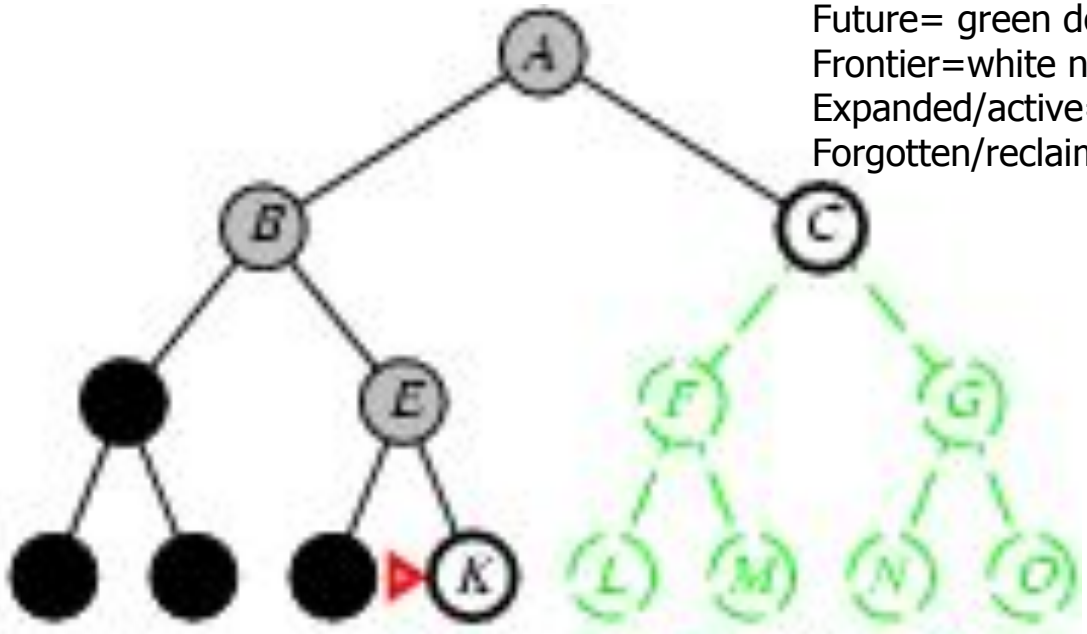
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Depth-first search

- Expand deepest unexpanded node
  - *Frontier* = LIFO queue, i.e., put successors at front

Expand I to no children.
Forget D, I.

frontier = [E,C]



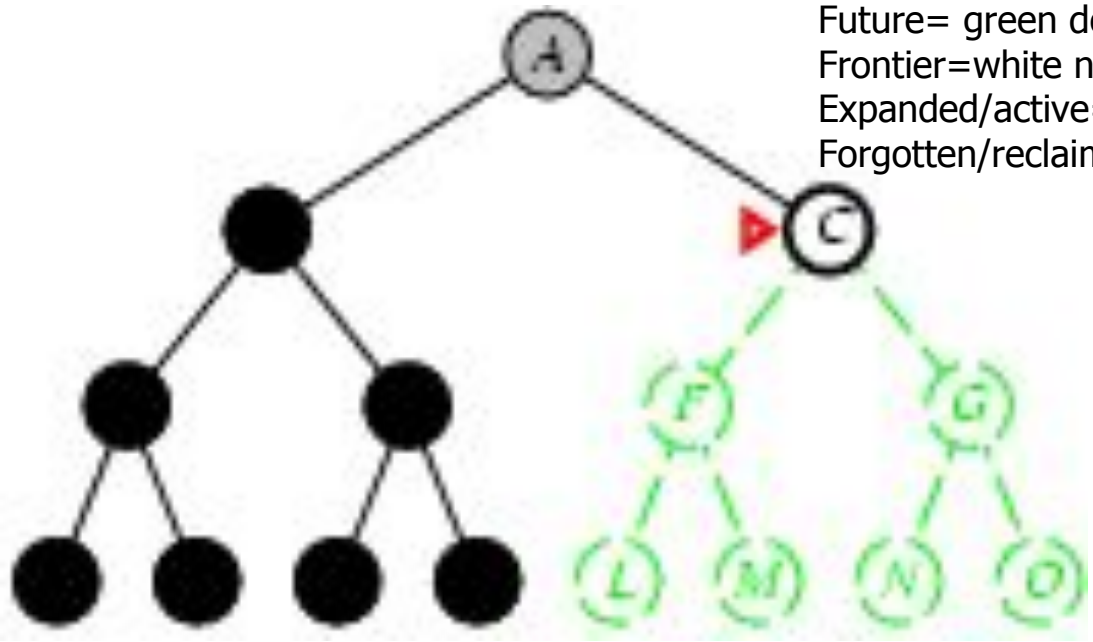
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Depth-first search

- Expand deepest unexpanded node
  - *Frontier* = LIFO queue, i.e., put successors at front

Expand K to no children.
Forget B, E, K.

frontier = [C]



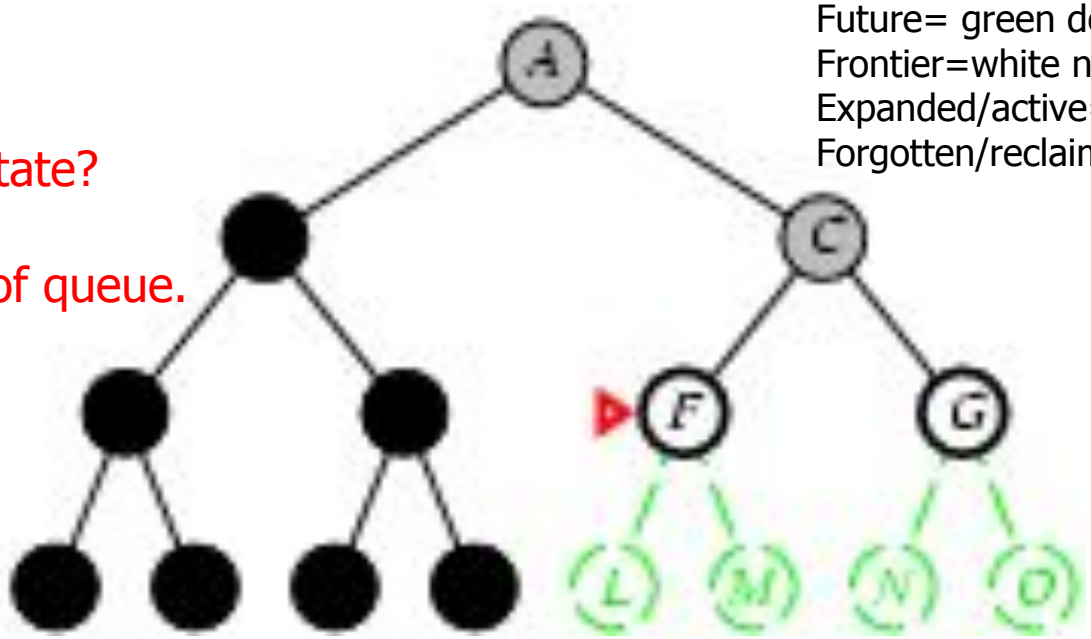
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Depth-first search

- Expand deepest unexpanded node
  - *Frontier* = LIFO queue, i.e., put successors at front

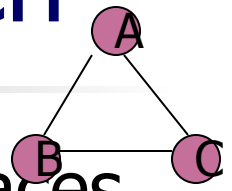
Expand C to F, G.
Is F or G a goal state?

Put F, G at front of queue.
frontier = [F,G]



Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Properties of depth-first search



- Complete? No: fails in loops/infinite-depth spaces
  - Can modify to avoid loops/repeated states along path
    - check if current nodes occurred before on path to root
  - Can use graph search (remember all nodes ever seen)
    - problem with graph search: space is exponential, not linear
  - Still fails in infinite-depth spaces (may miss goal entirely)
- Time? $O(b^m)$ with m =maximum depth of space
  - Terrible if m is much larger than d
  - If solutions are dense, may be much faster than BFS
- Space? $O(bm)$, i.e., linear space!
  - Remember a single path + expanded unexplored nodes
- Optimal? No: It may find a non-optimal goal first

Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

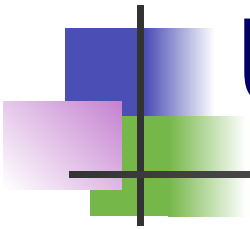
Disadvantage:

- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Analysis of BFS

- **Completeness:** DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.
- **Time Complexity:** Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:
- **$T(n)= 1+ n^2+ n^3 +.....+ n^m=O(n^m)$**
- **Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)**
- **Space Complexity:** DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.
- **Optimal:** DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

3		**Solving Problems by Searching**	12
	3.1	Definition, State space representation, Problem as a state space search, Problem formulation, Well-defined problems	
	3.2	Solving Problems by Searching, Performance evaluation of search strategies, Time Complexity, Space Complexity, Completeness, Optimality	
	3.3	Uninformed Search: Depth First Search, Breadth First Search, Depth Limited Search, Iterative Deepening Search, Uniform Cost Search, Bidirectional Search	
	3.4	Informed Search: Heuristic Function, Admissible Heuristic, Informed Search Technique, Greedy Best First Search, A* Search, Local Search: Hill Climbing Search, Simulated Annealing Search, Optimization: Genetic Algorithm	
	3.5	Game Playing, Adversarial Search Techniques, Mini-max Search, Alpha-Beta Pruning	



Uninformrd search

1. Breadth-first Search
2. Depth-first Search
3. **Depth-limited Search**
4. Iterative deepening depth-first search
5. Uniform cost search
6. Bidirectional Search

3. Depth-Limited Search Algorithm:

- A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.
- Depth-limited search can be terminated with two Conditions of failure:
- Standard failure value: It indicates that problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.

How Depth Limited Search Works

1) **Initialization:** Begin at the root node with a specified depth limit.
2) **Exploration:** Traverse the tree or graph, exploring each node's children.
3) **Depth Check:** If the current depth exceeds the set limit, stop exploring that path and backtrack.
4) **Goal Check:** If the goal node is found within the depth limit, the search is successful.
5) **Backtracking:** If the search reaches the depth limit or a leaf node without finding the goal, backtrack and explore other branches.

Applications of Depth Limited Search in AI

1) **Pathfinding in Robotics:** DLS is employed for nonholonomic motion planning of robots in the presence of obstacles. By imposing restriction on the depth it makes the robot stop after exploring a particular depth of an area and restricting the robot from too much wandering.
2) **Network Routing Algorithms:** A DLS can be implemented to compute paths between nodes in computer networks restricting the number of hops to prevent loops.
3) **Puzzle Solving in AI Systems:** DLS can be used to solve puzzles such as the 8-puzzle or Sudoku by manipulating possible moves a fixed number of times that reduces how many steps are taken in the search.
4) **Game Playing:** In AI for games, instead, DLS can be used to plan forward a few moves up to a certain level of depth to help decide how much effort to put into a given decision.

Advantages

- Depth-limited search is Memory efficient.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

Analysis of DLS

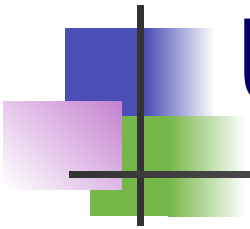
- **Completeness:** DLS search algorithm is complete if the solution is above the depth-limit.
- **Time Complexity:** Time complexity of DLS algorithm is **$O(b^{\ell})$**.
- **Space Complexity:** Space complexity of DLS algorithm is **$O(b \times \ell)$**.
- **Optimal:** Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $\ell > d$.

Problem Setup for Finding Path in Robotics using Depth Limited Search Algorithm

- **Grid Environment:** A 5×5 grid where 0 represents a free cell and 1 represents an obstacle.
- **Initial Position:** The starting position of the robot.
- **Goal Position:** The target position the robot needs to reach.
- **Movements:** The robot can move up, down, left, or right.

Function Code

Python Code

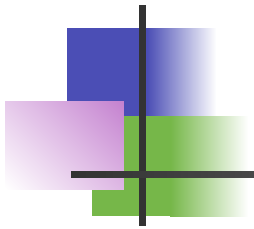


Uninformrd search

1. Breadth-first Search
2. Depth-first Search
3. Depth-limited Search
4. **Iterative deepening depth-first search**
5. Uniform cost search
6. Bidirectional Search

Iterative Deepening Search

- Search algorithm are used to solve a variety of issues, from playing games like chess and checkers to locating the shortest route on a map.
- The DFS method, one of the most popular search algorithms, searches a network or tree by travelling as far as possible along each branch before turning around. However, DFS has a critical drawback: **if the graph contains cycles, it could become trapped in an endless loop.**
- Utilizing Iterative Deepening Search (IDS) or Iterative Deepening Depth First Search is one technique to solve this issue.



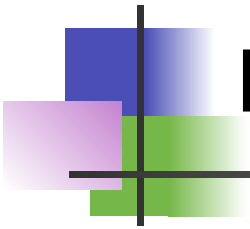
- IDS combines the benefits of DFS with Breadth First Search (BFS).
- The graph is explored using DFS, but the **depth limit steadily increased until the target is located.**
- In other words, IDS continually runs DFS, **raising the depth limit each time**, until the desired result is obtained.
- Iterative deepening is a method that makes sure the search is thorough **(i.e., it discovers a solution if one exists) and efficient (i.e., it finds the shortest path to the goal).**

How does IDS work?

- The IDS function performs iterative deepening search on the graph using a root node and a goal node as inputs until the goal is attained or the search space is used up.
- This is accomplished by regularly using the DLS function, which applies a depth restriction to DFS.
- The search ends and returns the goal node if the goal is located at any depth. The search yields None if the search space is used up (all nodes up to the depth limit have been investigated).
- The DLS function conducts DFS on the graph with the specified depth limit by taking as inputs a node, a destination node, and a depth limit.
- The search returns FOUND if the desired node is located at the current depth. The search returns NOT FOUND if the depth limit is reached but the goal node cannot be located.
- If neither criterion is true, the search iteratively moves on to the node's offspring.

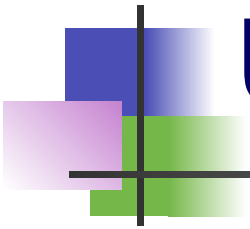
Advantages

- **It is comprehensive,** which ensures that a solution will be found if one is there in the search space. This is so that all nodes under a specific depth limit are investigated before the depth limit is raised by IDS, which does a depth-limited DFS.
- **IDS is memory-efficient:** This is because IDS decreases the algorithm's memory needs by not storing every node in the search area in memory. IDS minimises the algorithm's memory footprint by only storing the nodes up to the current depth limit.
- IDS's **ability to be utilised for both tree and graph search.** This is due to the fact that IDS is a generic search algorithm that works on any search space, including a tree or a graph.



Disadvantages

- IDS has the disadvantage of potentially **visiting certain nodes more than once,** which might **slow down the search.**
- The benefits of completeness and optimality frequently exceed this disadvantage. In addition, by employing strategies like memory or caching, the repeated trips can be minimised.

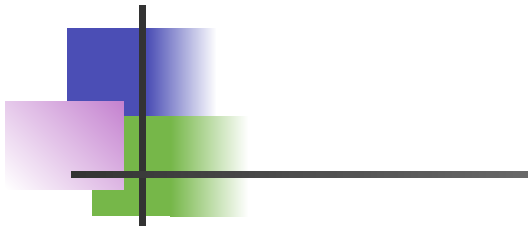


Uninformrd search

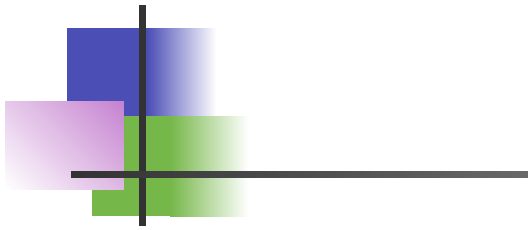
1. Breadth-first Search
2. Depth-first Search
3. Depth-limited Search
4. Iterative deepening depth-first search
5. **Uniform cost search**
6. Bidirectional Search

Uniform-Cost Search

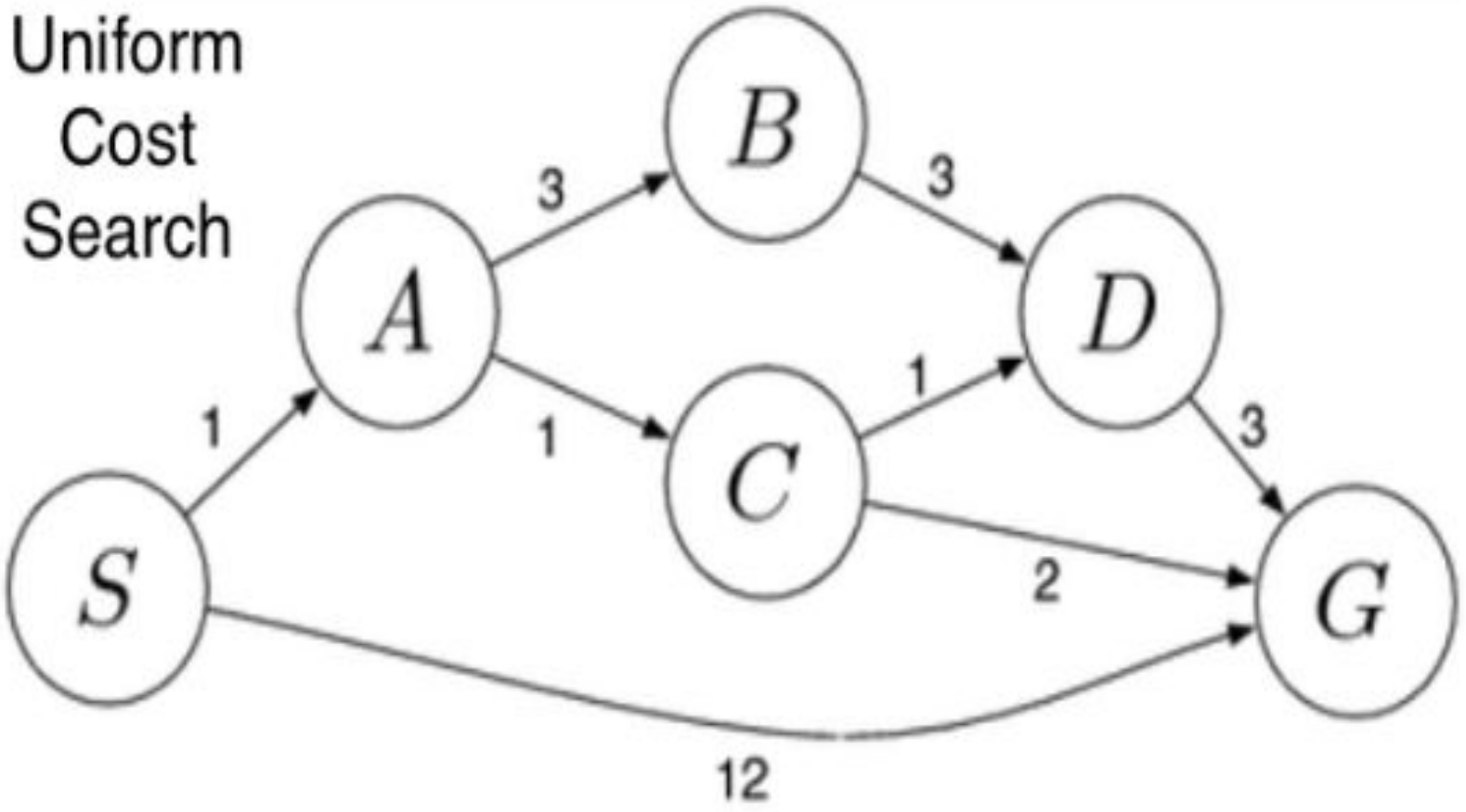
- UCS means the machine blindly follows the algorithm regardless of whether right or wrong, efficient or inefficient.
- Uniform-Cost Search is a variant of Dijikstra's algorithm. Here, instead of inserting all vertices into a priority queue, we insert only the source, then one by one insert when needed.
- In every step, we check if the item is already in the priority queue (using the visited array). If yes, we perform the decrease key, else we insert it.



- In this algorithm from the starting state, we will visit the adjacent states and will choose the least costly state then we will choose the next **least costly state** from the all un-visited and adjacent states of the visited states, in this way we will try to reach the goal state, even if we reach the goal state we will continue searching for other possible paths(if there are multiple goals).
- We will keep a priority queue that will give the least costly next state from all the adjacent states of visited states.



- UCS are **brute force** operations, and they don't have additional information about the search space; the only information they have is on how to traverse or visit the nodes in the tree.
- Thus UCS algorithms are also called **blind search** algorithms in artificial intelligence.
- The search algorithm produces the search tree **without** using any **domain knowledge**, which is a brute force in nature.
- They don't have any background information like informed search techniques algorithms on how to approach the goal or whatsoever.

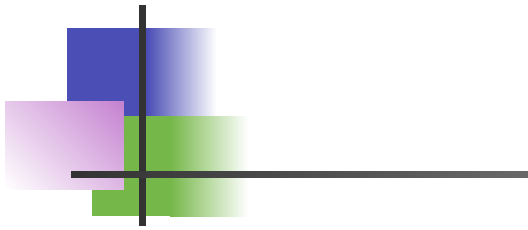

Uniform
Cost
Search
S
A
B
C
D
G
1
3
1
3
1
3
2
12

- **Advantages**

Uniform cost search is an optimal search method because at every state, the path with the least cost is chosen.

- **Disadvantages**

It does not care about the number of steps or finding the shortest path involved in the search problem, and it is only concerned about **path cost.** This algorithm may be stuck in an infinite loop.



- **Complete:** Yes (if b is finite and costs are stepped, costs are zero)
- **Time Complexity:** O(b(c/ϵ)) where, ϵ -> is the lowest cost, c -> optimal cost
- **Space complexity:** O(b(c/ϵ))
- **Optimal:** Yes (even for non-even cost)

Uniform-cost search

Breadth-first is only optimal if path cost is a non-decreasing function of depth, i.e., f(d) ≥ f(d-1); e.g., constant step cost, as in the 8-puzzle.

Can we guarantee optimality for variable positive step costs ≥ε?

(Why ≥ε? To avoid infinite paths w/ step costs 1, ½, ¼, …)

Uniform-cost Search:

Expand node with smallest path cost g(n).

- *Frontier* is a priority queue, i.e., new successors are merged into the queue sorted by g(n).
  - Can remove successors already on queue w/higher g(n).
    - Saves memory, costs time; another space-time trade-off.
- *Goal-Test* when node is popped off queue.

Uniform-cost search

Uniform-cost Search:

Expand node with smallest path cost g(n).

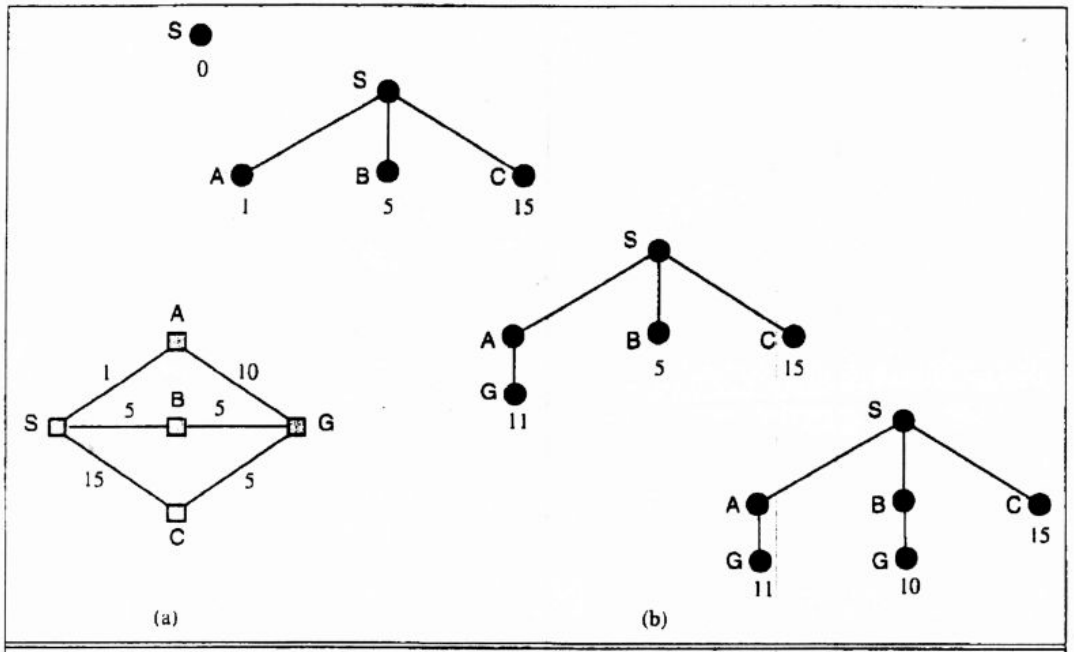


Figure 3.13 A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with $g(n)$. At the next step, the goal node with $g = 10$ will be selected.

Proof of Completeness:

Given that every step will cost more than 0, and assuming a finite branching factor, there is a finite number of expansions required before the total path cost is equal to the path cost of the goal state. Hence, we will reach it.

Proof of optimality given completeness:

Assume UCS is not optimal.
Then there must be an (optimal) goal state with path cost smaller than the found (suboptimal) goal state (invoking completeness).
However, this is impossible because UCS would have expanded that node first by definition.
Contradiction.

Uniform-cost search

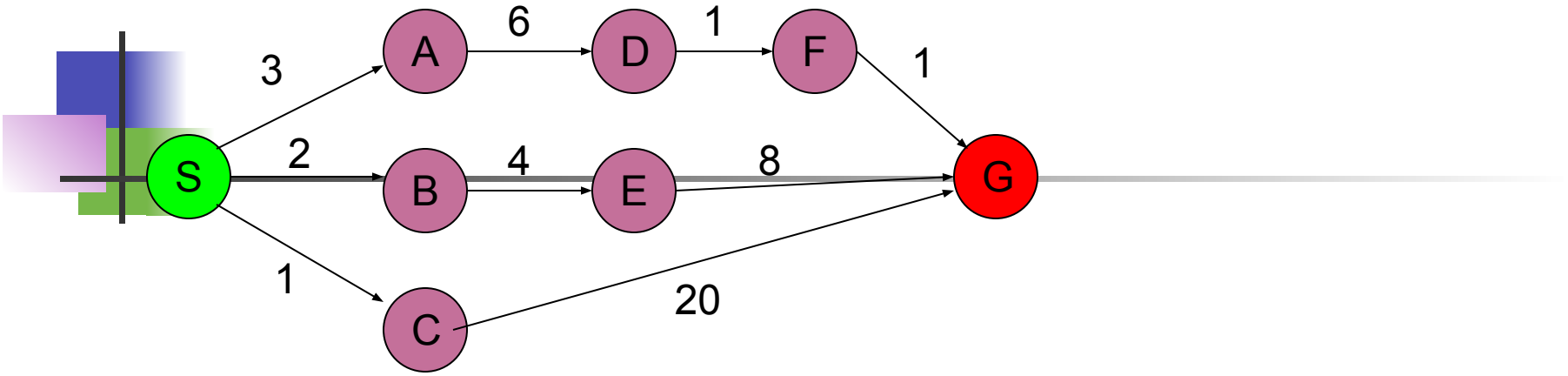
Implementation: *Frontier* = queue ordered by path cost.
Equivalent to breadth-first if all step costs all equal.

Complete? Yes, if b is finite and step cost $\geq \varepsilon > 0$.
(otherwise it can get stuck in infinite loops)

Time? # of nodes with *path cost* ≤ cost of optimal solution.
$O(b^{\lfloor 1+C^*/\varepsilon \rfloor}) \approx O(b^{d+1})$

Space? # of nodes with path cost ≤ cost of optimal solution.
$O(b^{\lfloor 1+C^*/\varepsilon \rfloor}) \approx O(b^{d+1})$

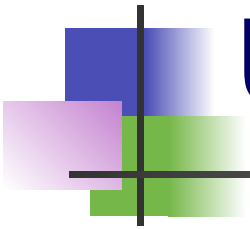
Optimal? Yes, for any step cost $\geq \varepsilon > 0$.



The graph above shows the step-costs for different paths going from the start (S) to the goal (G).

Use uniform cost search to find the optimal path to the goal.

Exercise for at home

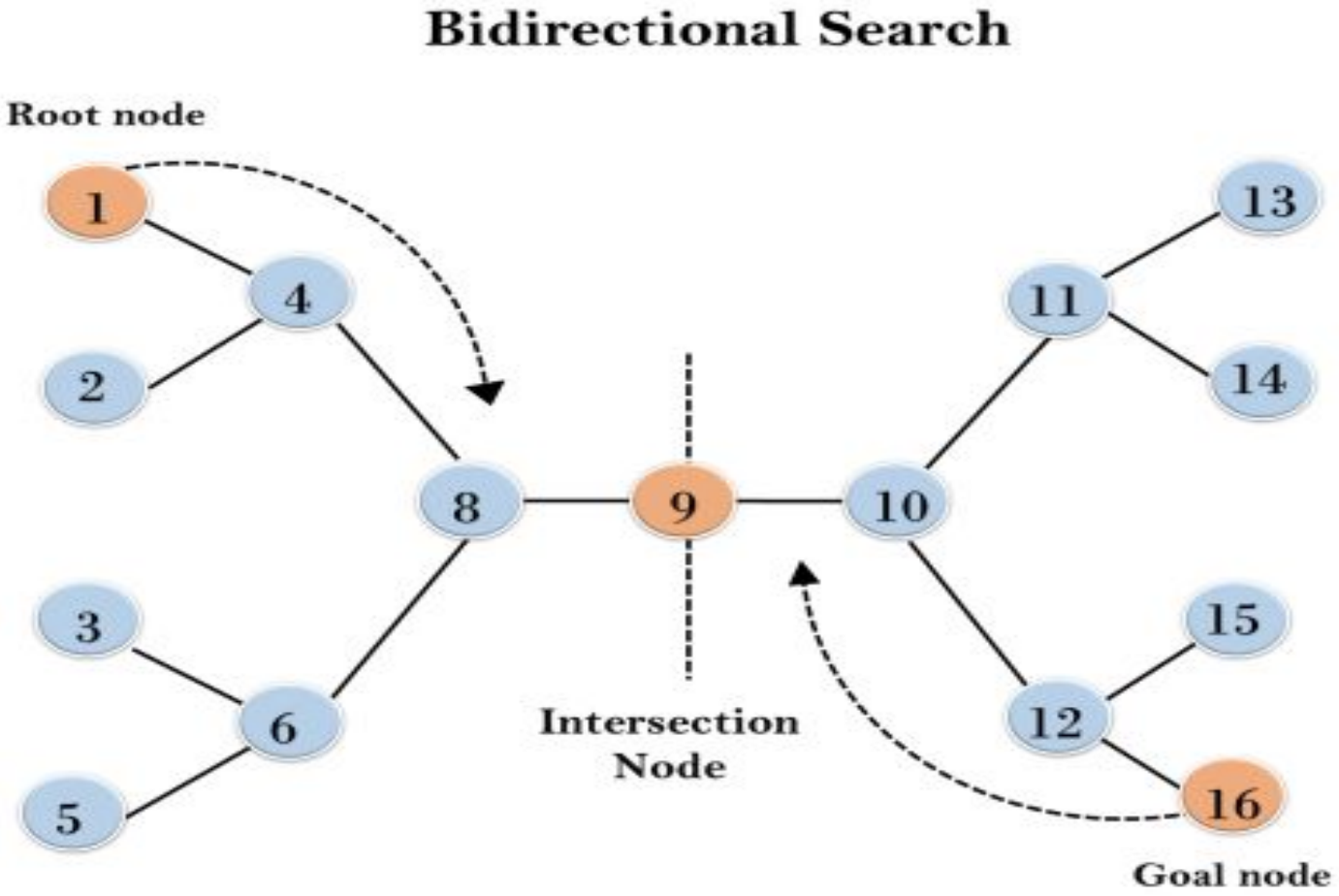


Uninformrd search

1. Breadth-first Search
2. Depth-first Search
3. Depth-limited Search
4. Iterative deepening depth-first search
5. Uniform cost search
6. **Bidirectional Search**

Bidirectional Search

- Bidirectional search algorithm runs **two simultaneous searches,** one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node.
- Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex.
- The search stops when these two graphs intersect each other.


Bidirectional Search
Root node
1
2
3
4
5
6
8
9
10
11
12
13
14
15
16
Intersection Node
Goal node

Advantages

- Bidirectional search is fast.
- Bidirectional search requires less memory
- The graph can be extremely helpful when it is very large in size and there is no way to make it smaller. In such cases, using this tool becomes particularly useful.
- The cost of expanding nodes can be high in certain cases. In such scenarios, using this approach can help reduce the number of nodes that need to be expanded.

Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.
- Finding an efficient way to check if a match exists between search trees can be tricky, which can increase the time it takes to complete the task.

Iterative deepening search

- To avoid the infinite depth problem of DFS,
  only search until depth L,
  i.e., we don't expand nodes beyond depth L.
  ☐ Depth-Limited Search

- What if solution is deeper than L? ☐ Increase L iteratively.
  ☐ Iterative Deepening Search

- This inherits the memory advantage of Depth-first search

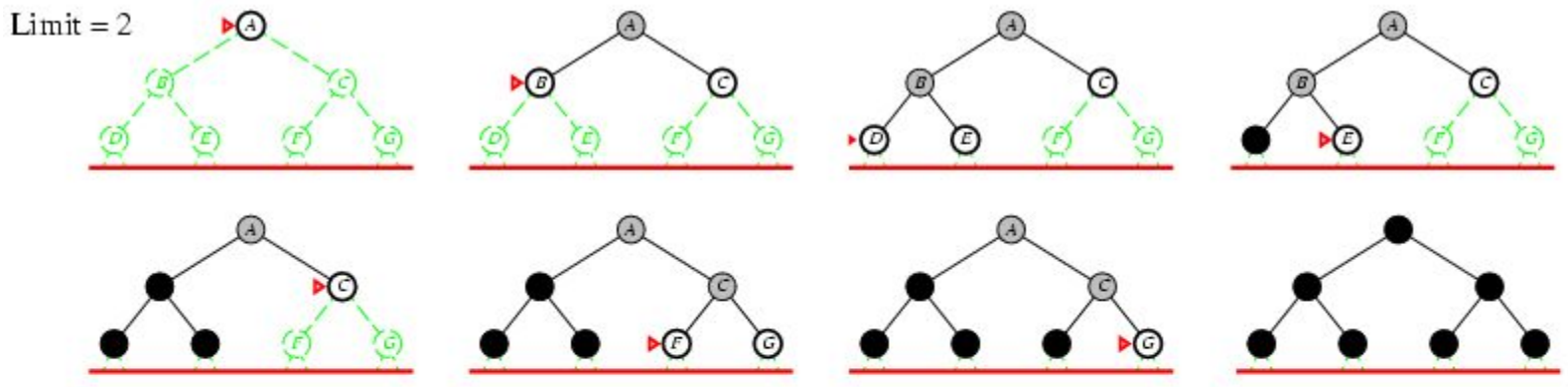
- Better in terms of space complexity than Breadth-first search.

Iterative deepening search L=0

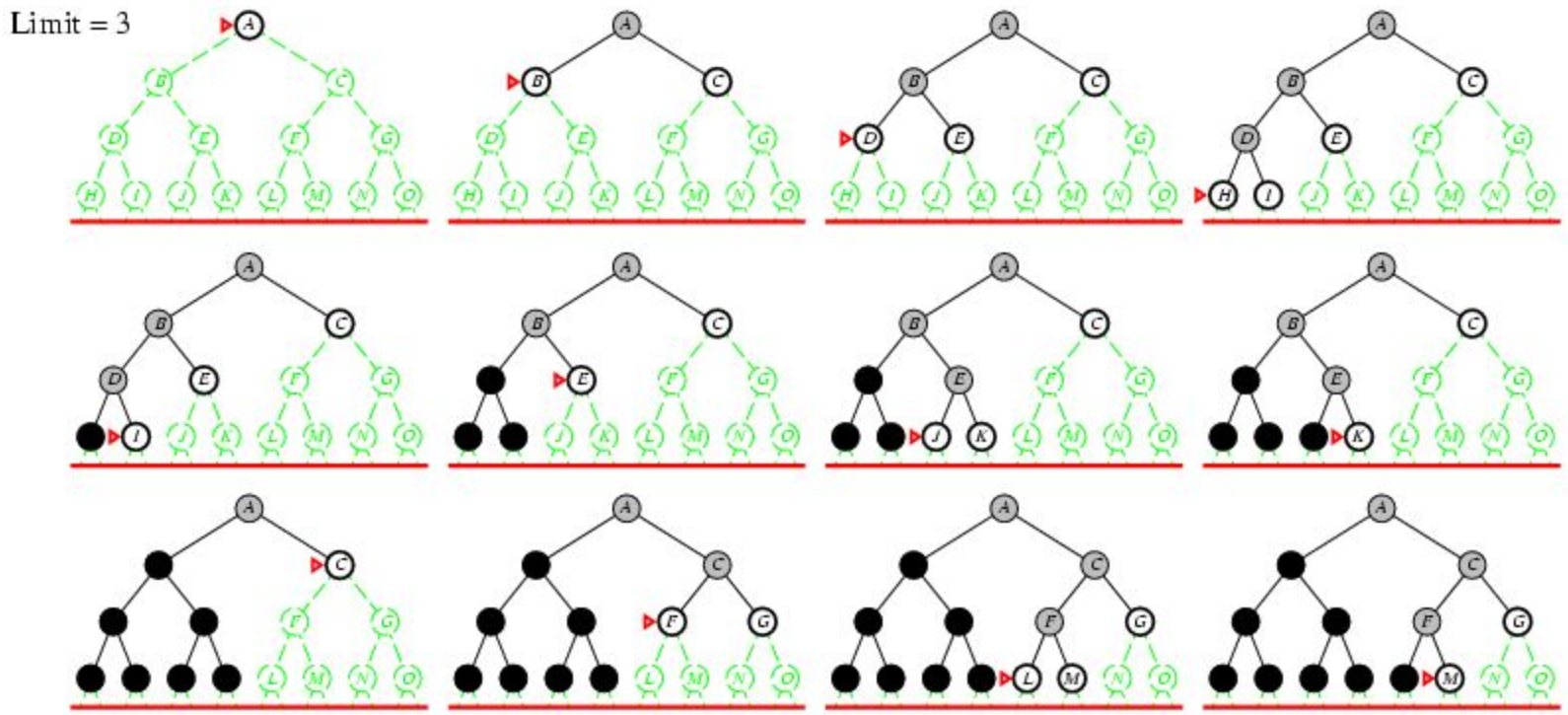
Limit = 0

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*:

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

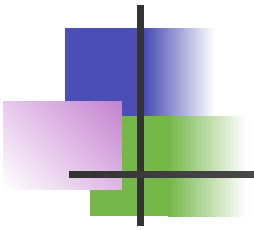
- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

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

- For *b = 10*, *d = 5*,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}450$

$O(b^d)$

Properties of iterative deepening search



- Complete? Yes
- Time? $O(b^d)$
- Space? $O(bd)$
- Optimal? No, for general cost functions.
  Yes, if cost is a non-decreasing function only of depth.

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

Complexity: time and space complexity are: $O(b^{d/2})$

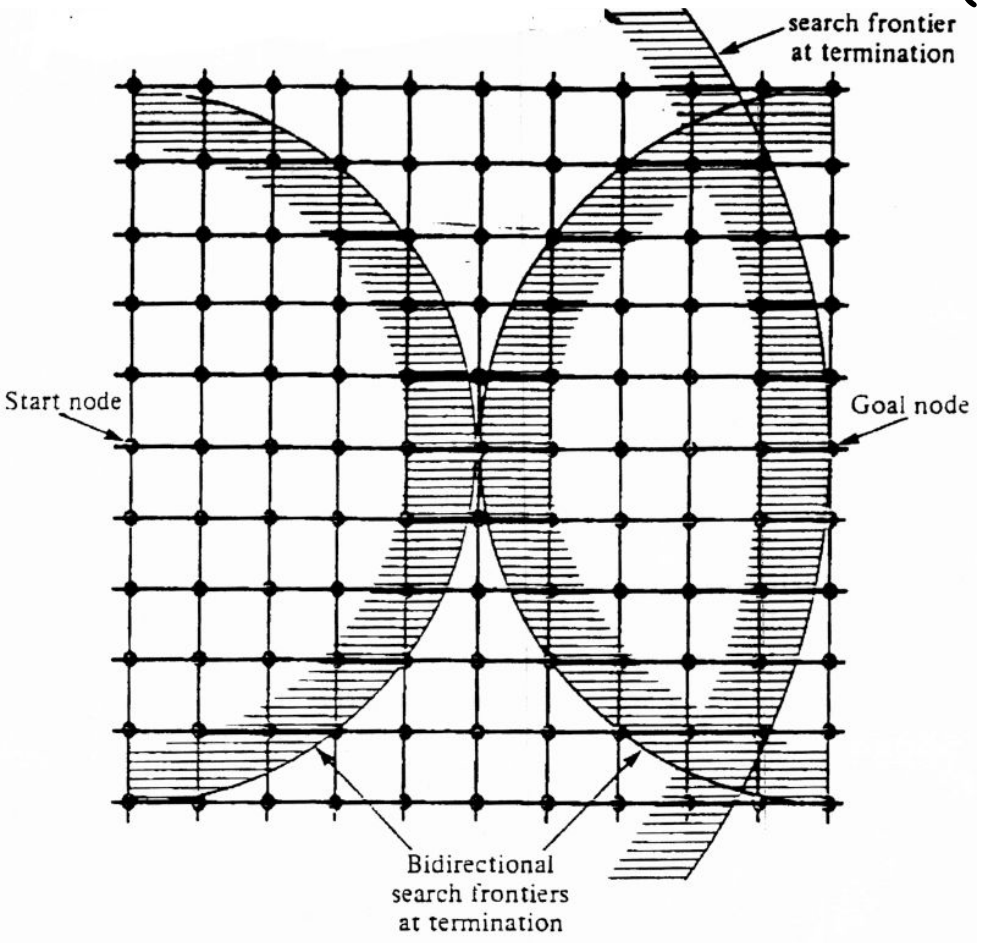


Fig. 2.10 Bidirectional and unidirectional breadth-first searches.

Summary of algorithms

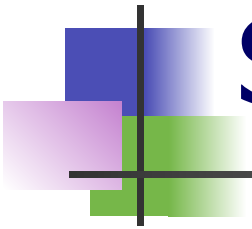
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening DLS	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon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

There are a number of footnotes, caveats, and assumptions.

Generally the preferred uninformed search strategy

[a] complete if b is finite
[b] complete if step costs $\geq \varepsilon > 0$
[c] optimal if step costs are all identical
(also if path cost non-decreasing function of depth only)
[d] if both directions use breadth-first search
(also if both directions use uniform-cost search with step costs $\geq \varepsilon > 0$)

Note that $d \leq \lfloor 1+C^*/\varepsilon \rfloor$



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