Depth Limited Search

- Depth-limited search avoids the pitfalls of depth-first search which is infinite path by **imposing a cutoff on the maximum depth of a path**.
- Depth-first search with depth limit l. Algorithm treats the node at the depth limit l as it has no successor nodes further.
- In this algorithm, 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.

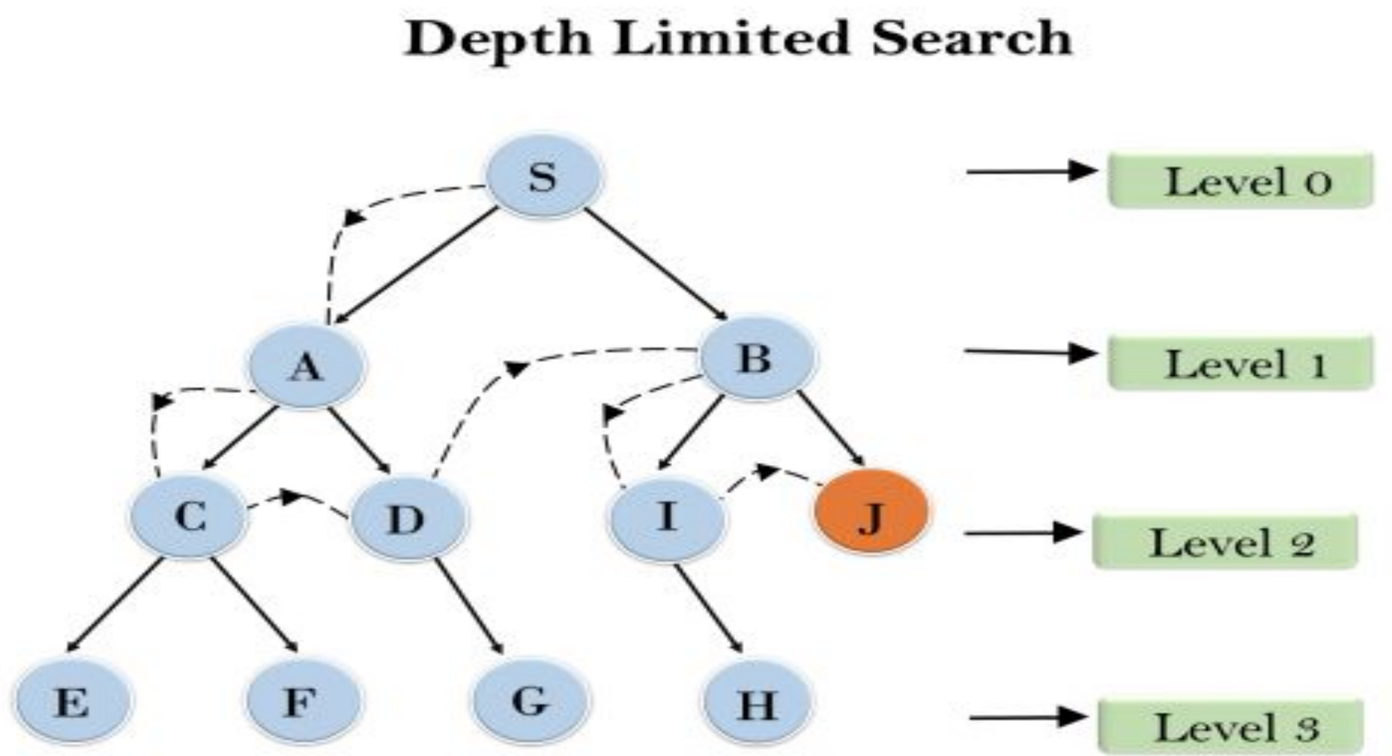
Advantages:

- 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.

DEPTH LIMITED SEARCH



PROPERTIES OF DLS

- **Complete?**
- **Yes** (unless the goal node is within the depth l)
- **Time?**
- **O(b^l)** Exponential
- **Space?**
- **O(bl)** Keeps all nodes in memory
- **Optimal?**
- **No** (depending upon search algo and heuristic property)

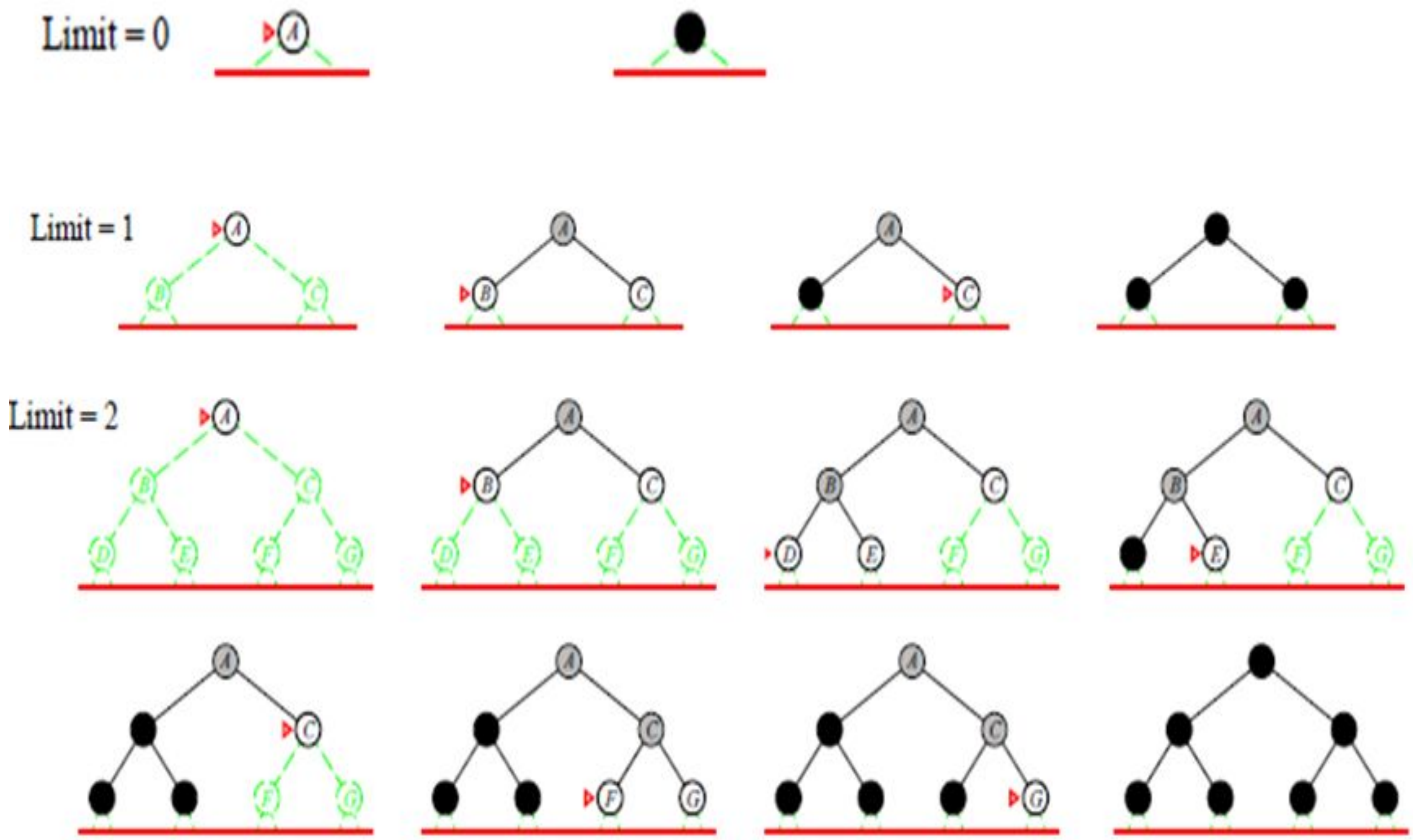
ITERATIVE DEEPENING SEARCH

- The hard part about depth-limited search is picking a good limit, which is known as diameter of the state space. for most problems, we will not know a good depth limit until we have solved the problem.
- Iterative deepening search is a strategy that sidesteps the issue of choosing the best depth limit by trying all possible depth limits: first depth 0, then depth 1, then depth 2, and so on.
- The iterative deepening algorithm is a combination of **DFS and BFS** algorithms. This search algorithm **finds out the best depth limit** and does it by iteratively increasing the depth limit until a goal is found.
- This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.
- 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.

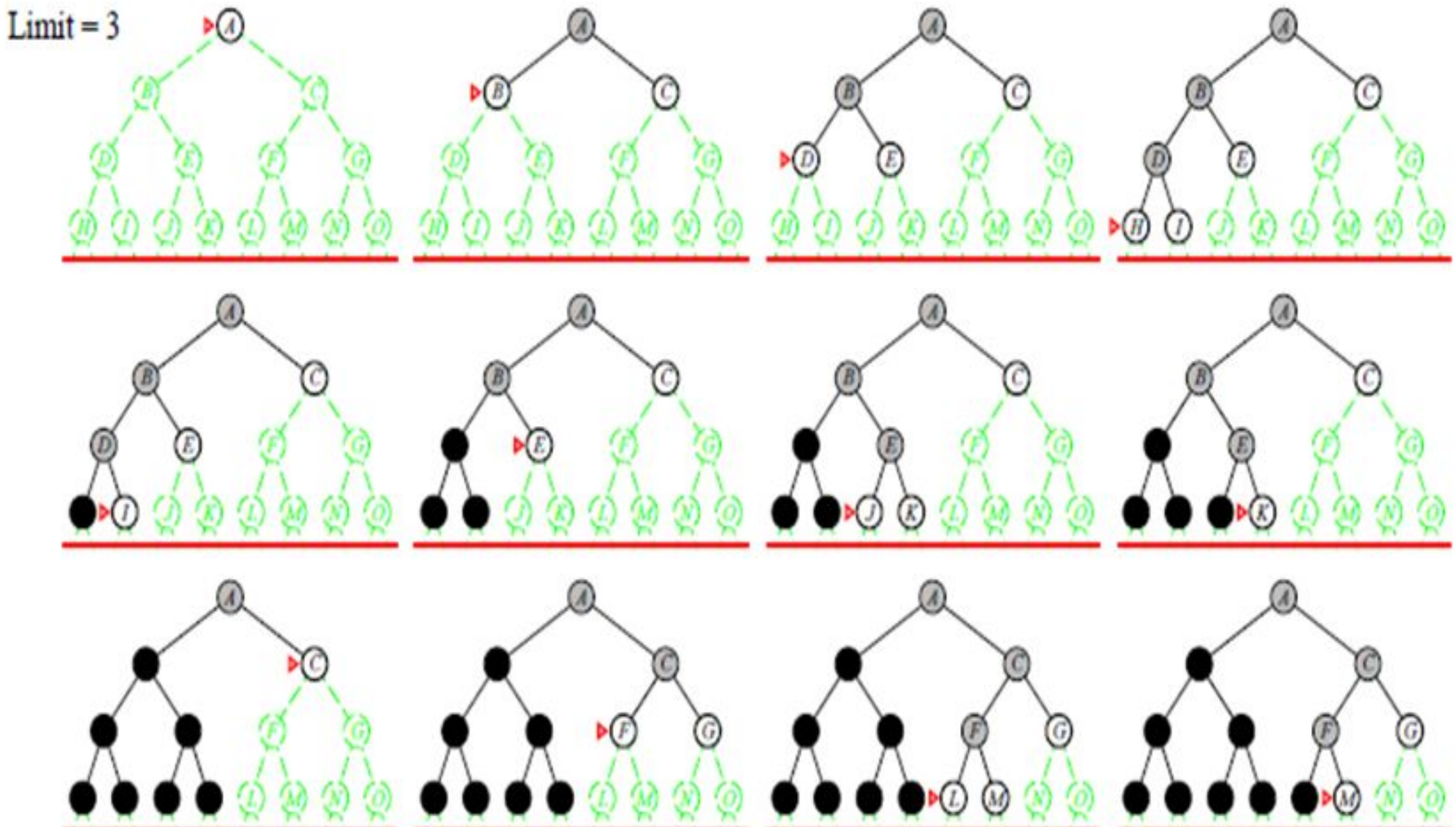
Iterative deepening search Algorithm

- Explore the nodes in DFS order.
- Set a LIMIT variable with a limit value.
- Loop each node up to the limit value and further increase the limit value accordingly.
- Terminate the search when the goal state is found.

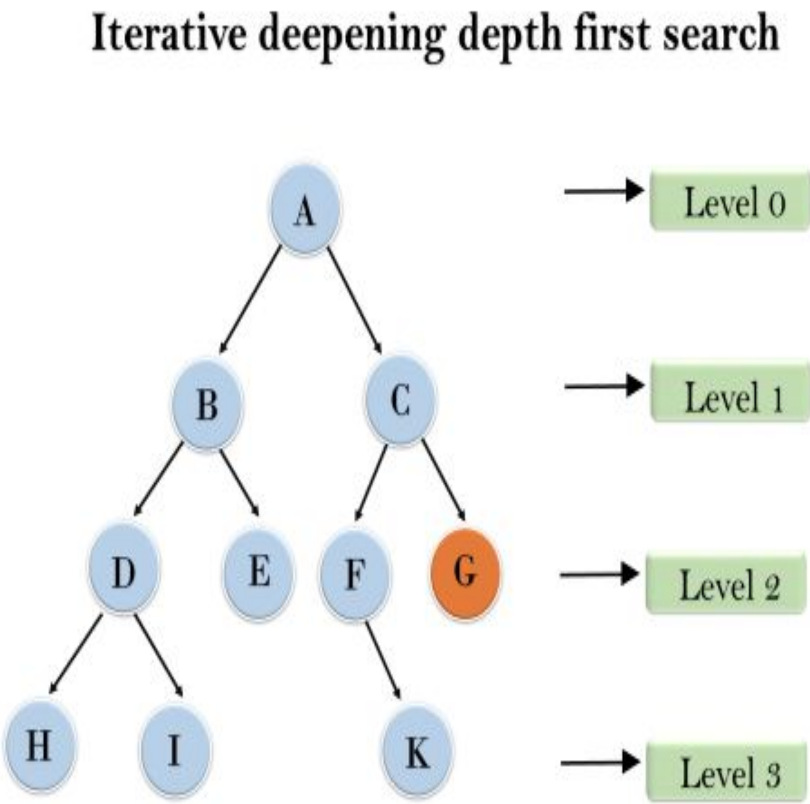
ITERATIVE DEEPENING SEARCH



ITERATIVE DEEPENING SEARCH



ITERATIVE DEEPENING SEARCH



1'st Iteration-----> A
2'nd Iteration----> A, B, C
3'rd Iteration------>A, B, D, E, C, F, G
In the third iteration, the algorithm will find the goal node.

Properties of Iterative deepening Search

- Complete?? **Yes**
- Time?? $(d + 1)\, b^0 + d b^1 + (d - 1)\, b^2 + \ldots + b^d = \mathbf{O(b^d)}$
- Space?? **O(bd)**
- Optimal?? **Yes,** if step cost = 1 it can be modified to explore a uniform-cost tree. Otherwise, not optimal but guarantees finding solution of shortest length (like BFS).
- **Disadvantages of Iterative deepening search**
- The drawback of iterative deepening search is that it seems wasteful because it generates states multiple times.
- **Note:** Generally, iterative deepening search is required when the search space is large, and the depth of the solution is unknown.

Uniform cost search

- The primary goal of the uniform-cost search is to **find a path** to the goal node which has the **lowest cumulative cost** ie sort by the **cost-so-far.**
- A uniform-cost search algorithm is implemented by the **priority queue**. It gives **maximum priority to the lowest cumulative cost** and Enqueue nodes by **path cost**.
- Uniform cost search modifies the breadth-first strategy by always expanding the lowest-cost node on the fringe. Uniform cost search is **equivalent to BFS algorithm** if the **path cost of all edges is the same.**
- **Algorithm outline:** Let **g(n) = cost of the path from the start node to the current node n**. Sort nodes by increasing value of g
  - Always select from the OPEN the node with the least g(.) value for expansion, and put all newly generated nodes into OPEN
  - Nodes in OPEN are sorted by their g(.) values (in ascending order)
  - Terminate if a node selected for expansion is a goal
- Called *“Dijkstra's Algorithm”* in the algorithm's literature and similar to *“Branch and Bound Algorithm”* in operations research literature

UNIFORM COST SEARCH

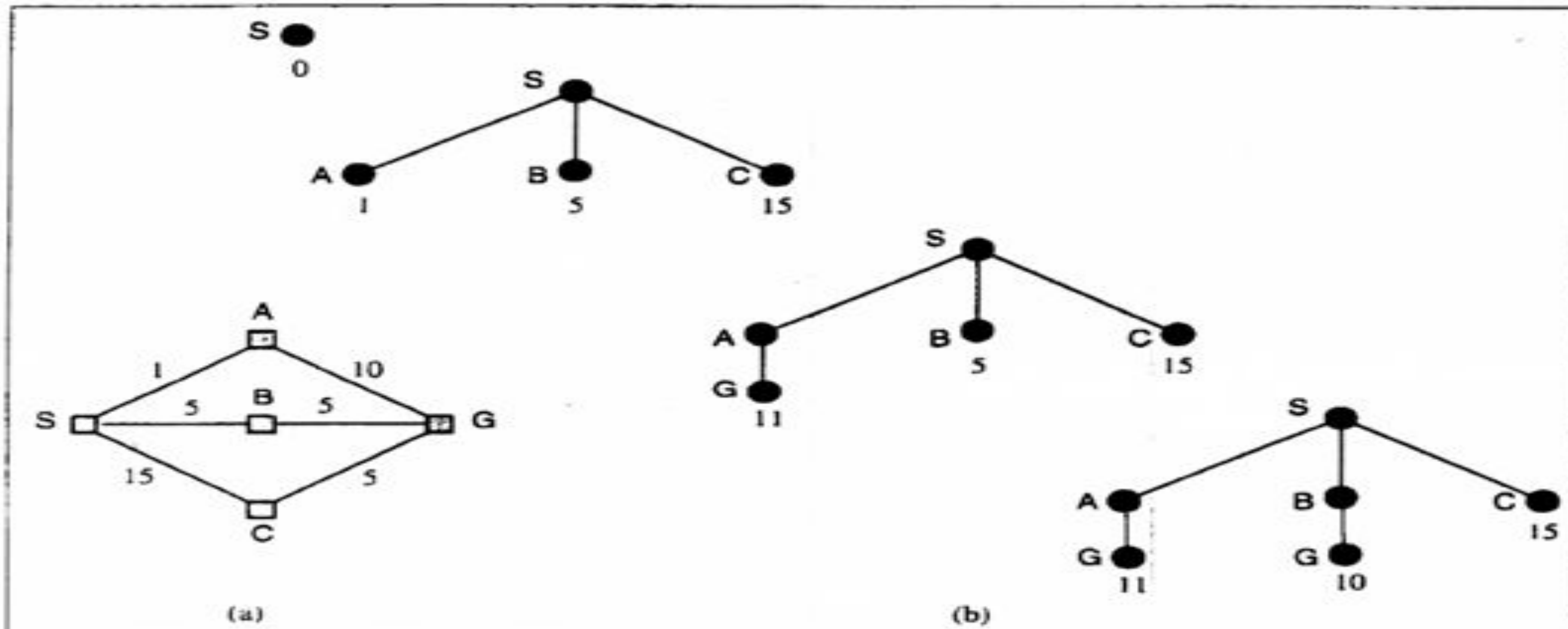
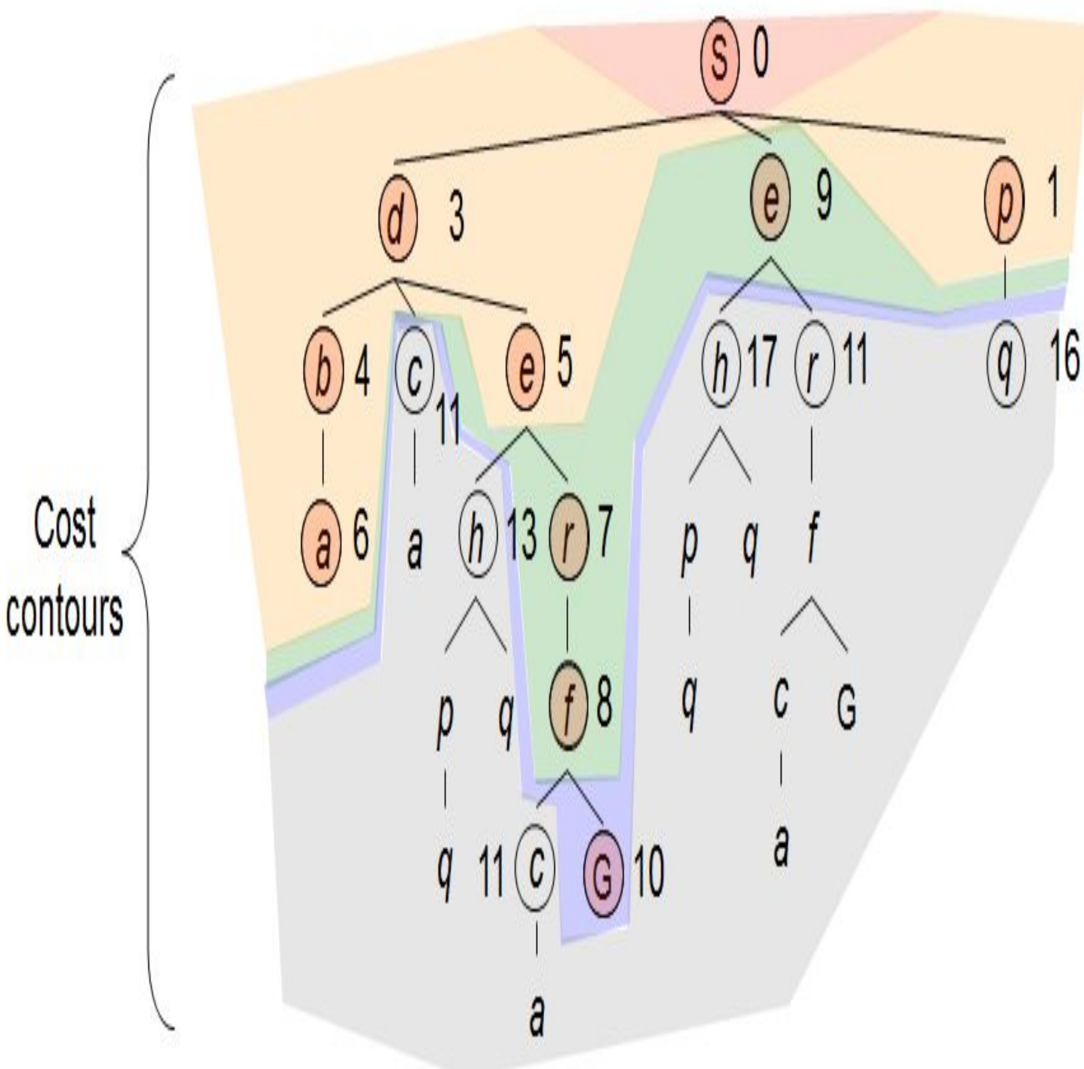


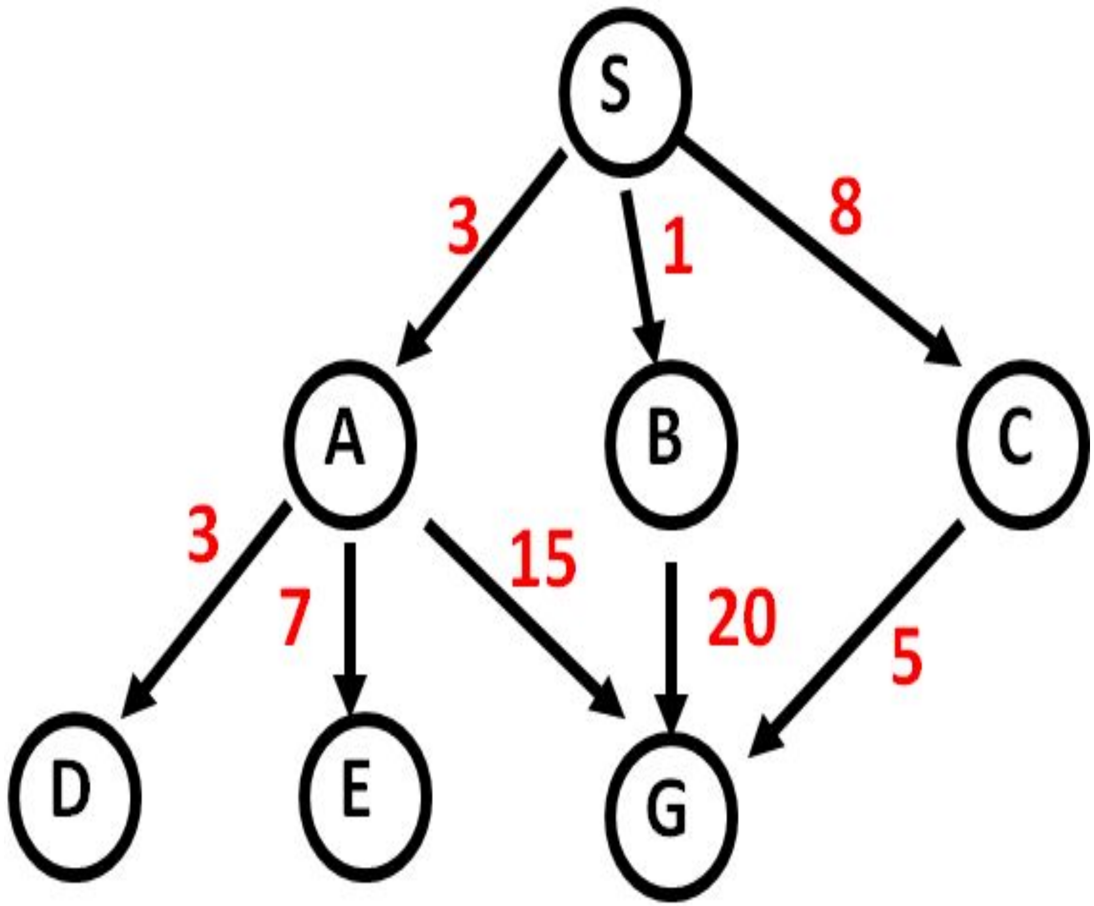
Figure 3.13 A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with $g(n)$. At the next step, the goal node with $g = 10$ will be selected.

Uniform cost search



	Expanded node	Nodes list/ open list
		$\{ S^0 \}$
1	s^0	$\{ p^1\ d^3\ e^9 \}$
2	p^1	$\{d^3\ e^9 q^{16}\}$
3	d^3	$\{ b^4\ e^5 e^9\ c^{11} q^{16} \}$
4	b^4	$\{\ e^5\ a^6\ e^9\ c^{11} q^{16}\}$
5	e^5	$\{a^6\ r^7 e^9\ c^{11}\ h^{13} q^{16}\}$
6	a^6	$\{ r^7 e^9\ c^{11}\ h^{13} q^{16} \}$
7	r^7	$\{\ f^{8} e^9\ c^{11}\ h^{13} q^{16}\}$
8	f^8	$\{ e^9\ g^{10} c^{11}\ h^{13} q^{16}\}$
9	e^9	$\{g^{10} r^{11} c^{11}\ h^{13} q^{16} h^{17}\}$

UNIFORM COST SEARCH



UNIFORM COST SEARCH

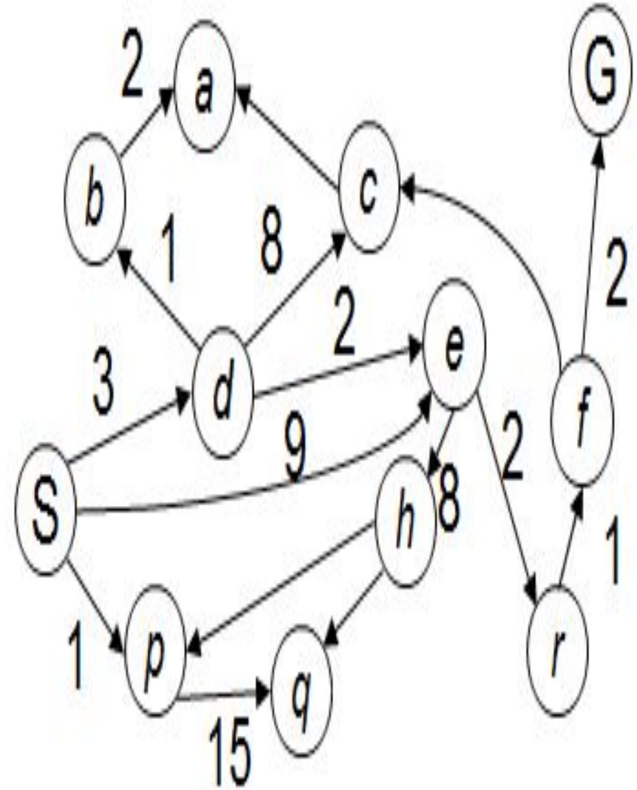
	Expanded node	Nodes list/open list
		{ S^0 }
1	S^0	{ B^1 A^3 C^8 }
2	B^1	{ A^3 C^8 G^{21} }
3	A^3	{ D^6 C^8 E^{10} G^{18} G^{21} }
4	C^8	{ C^8 E^{10} G^{18} G^{21} }
5	C^8	{ E^{10} G^{13} G^{18} G^{21} }
6	E^{10}	{ G^{13} G^{18} G^{21} }
7	G^{13}	{ G^{18} G^{21} }

- Solution path found is S C G, cost 13
- Number of nodes expanded (including goal node) = 7

UNIFORM COST SEARCH

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)



The good:

UCS is complete and optimal!

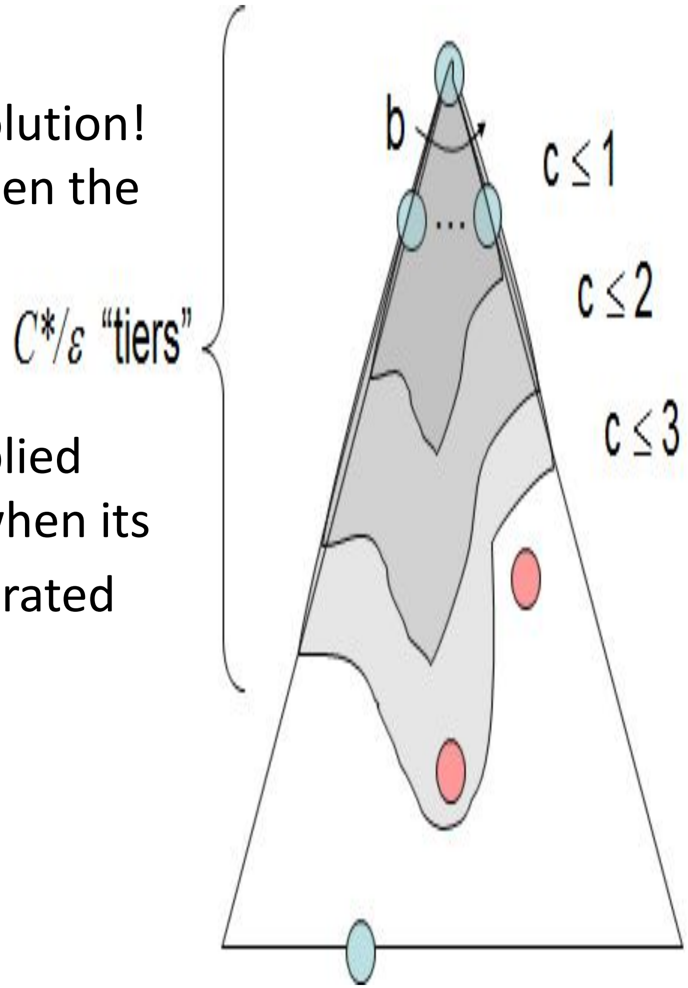
The bad:

Explores options in every “direction”. No information about goal location

UCS properties

What nodes does UCS expand?

Processes all nodes with cost less than cheapest solution!
If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)



Optimal?

Yes. Optimality depends on the goal test being applied when a node is removed from the nodes list, not when its parent node is expanded and the node is first generated

Time?

Exponential time and space complexity, $O(b^{C^*/\varepsilon€})$

How much space does the fringe take?

Has roughly the last tier, so $O(b^{C^*/\varepsilon})$

Is it complete?

Assuming best solution has a finite cost and minimum arc cost is positive, **yes**!

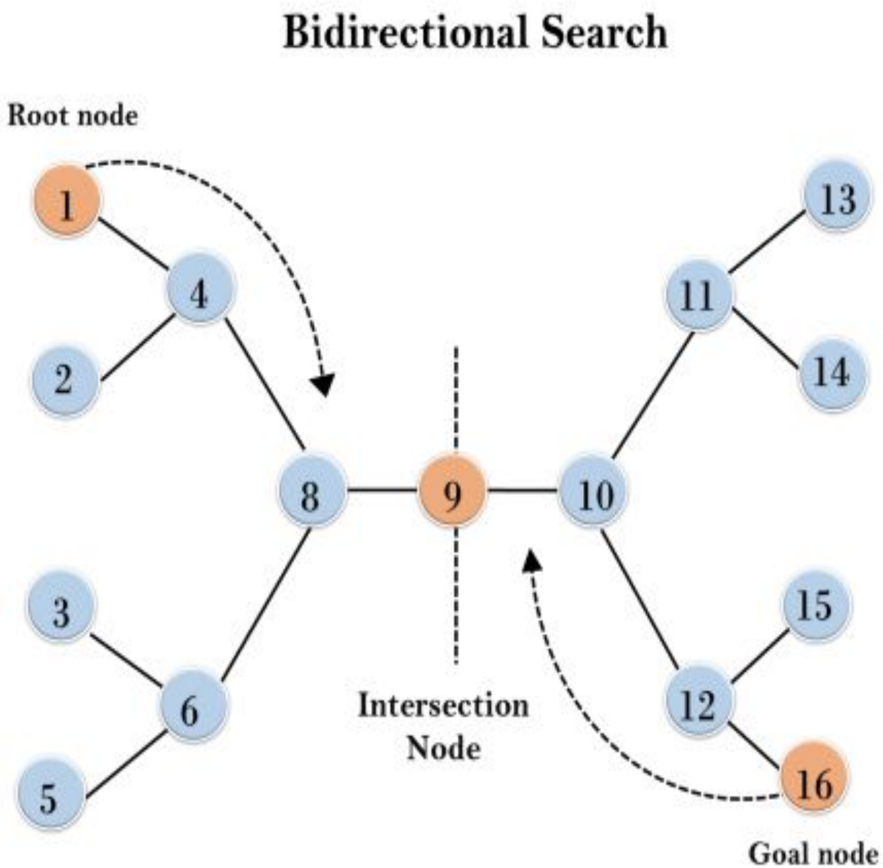
Bidirectional search

- 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.The search stops when these two graphs intersect each other.
- Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.
- 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.
- 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. need a way to specify the predecessors of G this can be difficult

When to use bidirectional approach?

- Both initial and goal states are unique and completely defined.
- The branching factor is same in both directions.

BIDIRECTIONAL SEARCH



Properties of bidirectional search

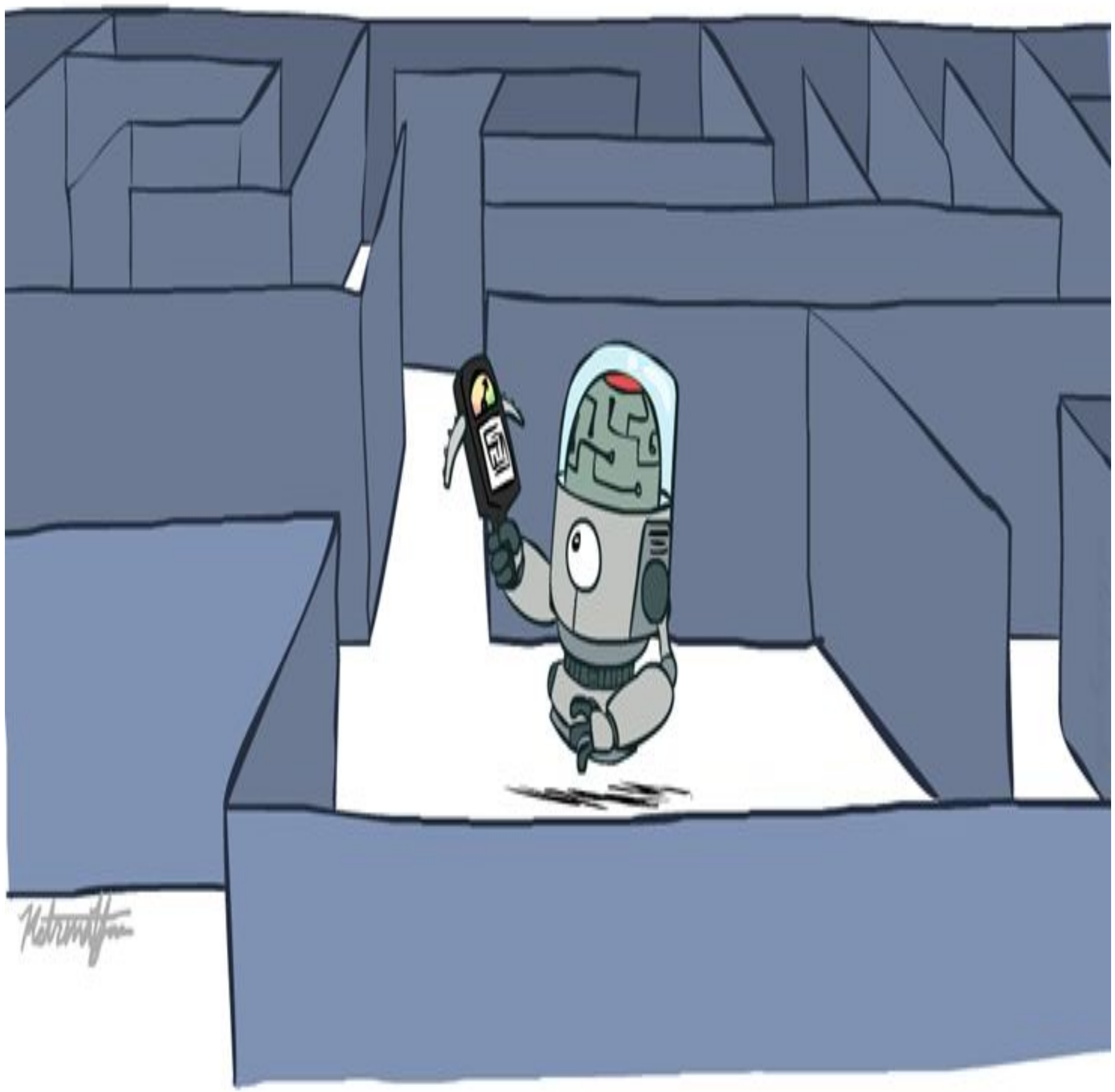
- Complete: Yes. Bidirectional search is complete.
- Optimal: Yes. It gives an optimal solution.
- Time and space complexity: Bidirectional search has $O(b^{d/2})$
- Disadvantage of Bidirectional Search
- It requires a lot of memory space.

COMPARISON

Criterion	BFS	DFS	Limited Depth	Iterative deepening	Bidirectional	Uniform Cost Search
Time	B^d	B^m	B^l	B^d	$B^{d/2}$	$B^{c*/€}$
Space	B^d	B*m	B*l	Bd	$B^{d/2}$	$B^{c*/€}$
Optimality?	Yes	No	No	Yes	Yes	Yes
Completeness	Yes	No	Yes if l≥d	Yes	Yes if e≥0	Yes if €≥0

- **m** – maximum depth of the tree, **l** –depth limit of search (AdMax
- €-smallest step cost

INFORMED SEARCH STRATEGIES

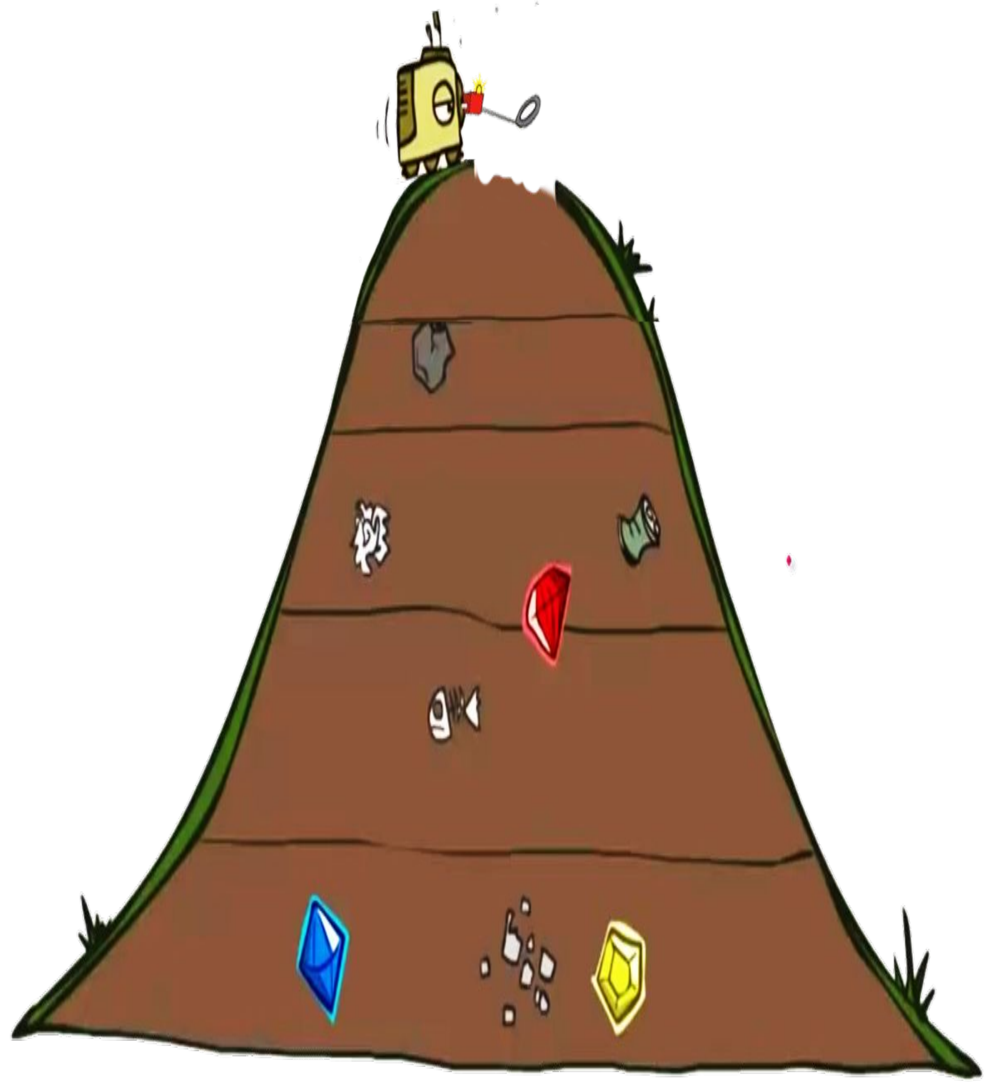


INFORMED SEARCH

- Informed search algorithms **use domain knowledge.** In an informed search, problem information is available which can guide the search. Informed search strategies can find a solution **more efficiently than an uninformed search strategy.** Informed search is also called a **Heuristic search.**
- A **heuristic** is a way which **might not always be guaranteed for best solutions** but **guaranteed to find a good solution in reasonable time**.
- Informed search can solve much complex problem which could not be solved in another way.
- It contains the problem description as well as extra information like how far is the goal node.

Example: traveling salesman problem, Greedy Search, A* Search

General Approach: Best first search



Best First Search

- A search strategy is defined by picking the **order of node expansion**
- Use an **evaluation function *f(n)*** is used to assign score for each node. **f(n)** provides an **estimate for the total cost** also known as **estimate of "desirability“**. Expand the **node n with smallest f(n)** ie most desirable unexpanded node.

Implementation:

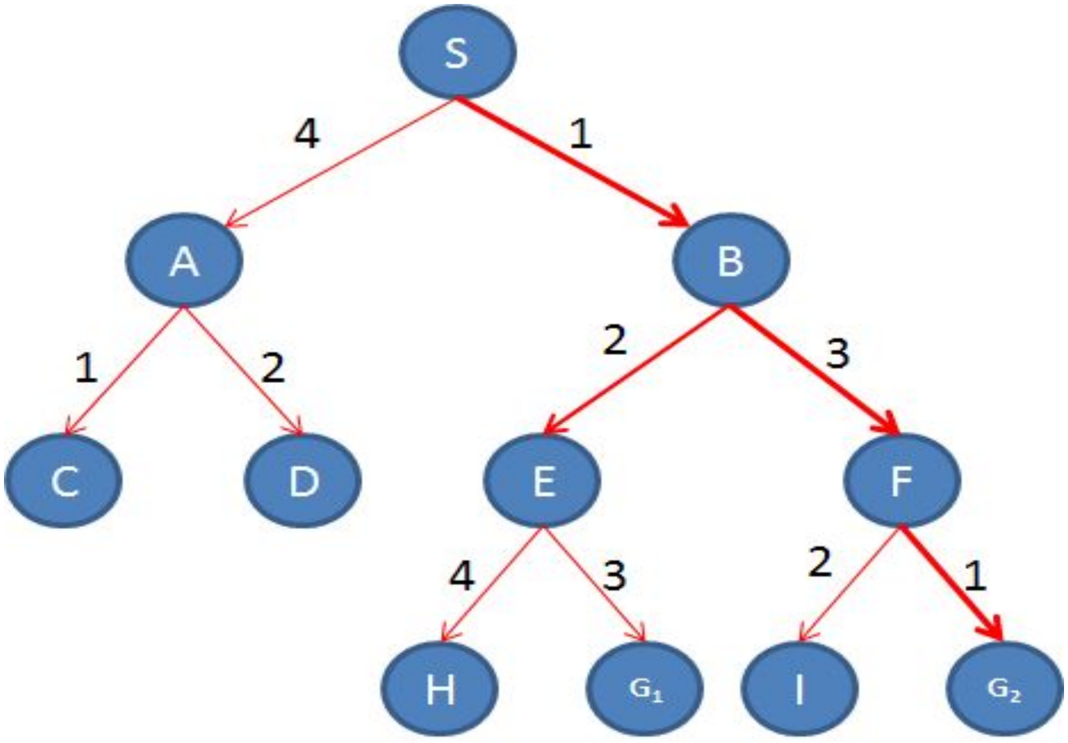
  - Order the nodes in fringe **increasing order of cost**.
  - The algorithm maintains two lists, one containing a list of candidates yet to explore (**OPEN**), and one containing a list of already visited nodes (**CLOSED**). States in OPEN are ordered according to some heuristic estimate of their “closeness” to a goal. This ordered OPEN list is referred to as ***priority queue***.
  - The algorithm always chooses the best of all unvisited nodes that have been graphed

- Choice of f determines the search strategy. The advantage of this strategy is that if the algorithm reaches a dead-end node, it will continue to try other nodes.`

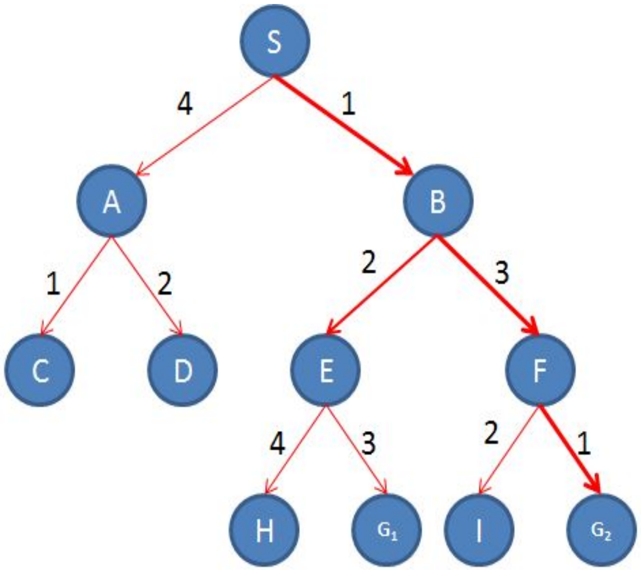
Best First Search

```
Let fringe be a priority queue containing the initial state
LOOP
    if fringe is empty return failure
    Node<- remove-first(fringe)
     if Node is a goal
          then return path from initial state to goal
           node
     else generate all the successors of the Node, and
      put the newly generated nodes into fringe
       according to their f values
END LOOP
```

BEST FIRST SEARCH EXAMPLE



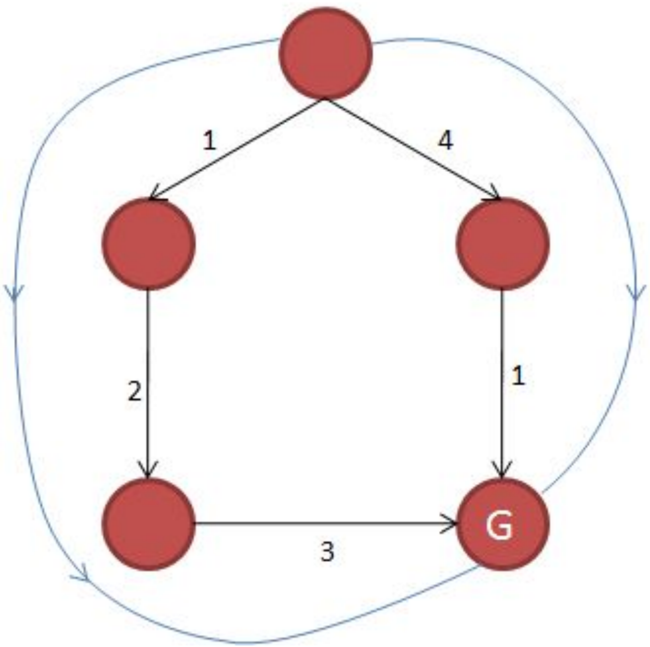
BEst FIRST SEARCH SOLUTION

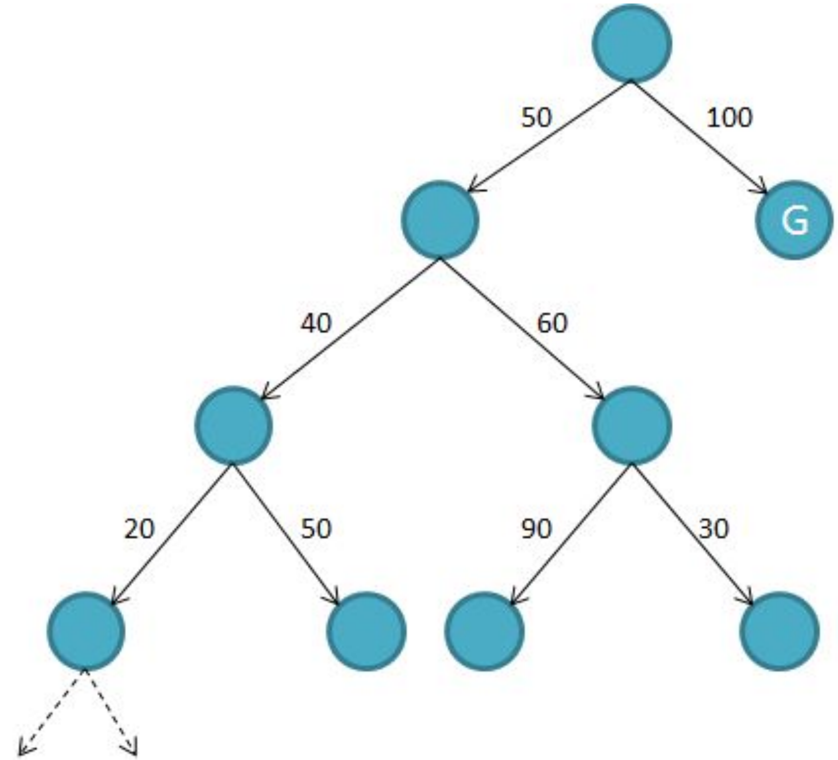


open=[S_0]; closed=[ ]
open=[B_1 , A_4]; closed=[S_0]
open=[E_3 , A_4, F_4]; closed=[S_0 , B_1]
open=[A_4 , F_4 , $G1_6$, , H_7]; closed=[S_0 , B_1 , E_3]
open=[F_4 , C_5 , $G1_6$, D_6 , H_7]; closed=[S_0 , B_1 , E_3 , A_4]
open=[C_5 , $G2_5$, $G1_6$, I_6 , D_6 , H_7]; closed=[S_0 , B_1 , E_3 , A_4 , F_4]
open=[$G2_5$, $G1_6$, I_6 , D_6 , H_7]; closed=[S_0 , B_1 , E_3 , A_4 , F_4 , C_5]

Cost = 1+3+1=5

Does best first algorithm always guarantee to find shortest path?





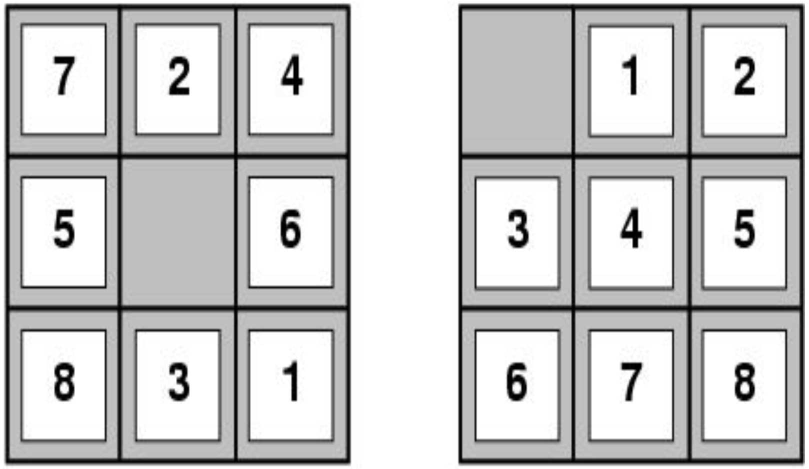
HEURISTIC FUNCTIONS

- Most of Best First Strategies use eval. func. f(n) as heuristic function h(n)
- A **heuristic function, h(n)**, is the estimated cost of the cheapest path from the state at node n, to a goal state. A node is selected for expansion in informed search algorithm based on an evaluation function that estimates cost to goal.
- A heuristic is:
  - A function that *estimates* how close a state is to a goal
  - Designed for a particular search problem
  - The value of the heuristic function is always positive. If h(n)=0, n is goal node
- **Examples:** Manhattan distance, Euclidean distance for pathing
- Heuristic is a function which is used in Informed Search finds the most promising path.
- Heuristic functions are very much dependent on the domain used. h(n) might be the estimated number of moves needed to complete a puzzle, or the estimated straight-line distance to some town in a route finder.
- Choosing an appropriate function greatly affects the effectiveness of the state-space search, since it tells us which parts of the state-space to search next.
- A heuristic evaluation function which accurately represents the actual cost of getting to a goal state, tells us very clearly which nodes in the state-space to expand next, and leads us quickly to the goal state.

EXAMPLe heuristics

E.g., for the 8-puzzle:

- ***$h_1(n)$* = number of misplaced tiles**
- ***$h_2(n)$* = total Manhattan distance(i.e., no. of squares from desired location of each tile)(cityblock, D4 distance)**
- $h_1(S)$ = ?
- $h_2(S)$ = ?

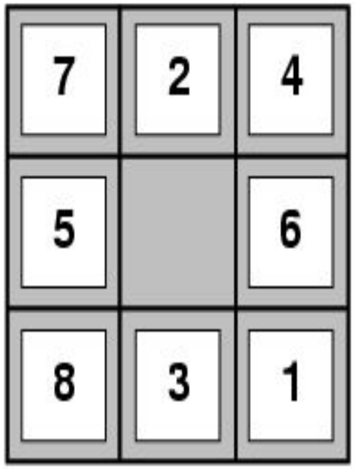


Start State

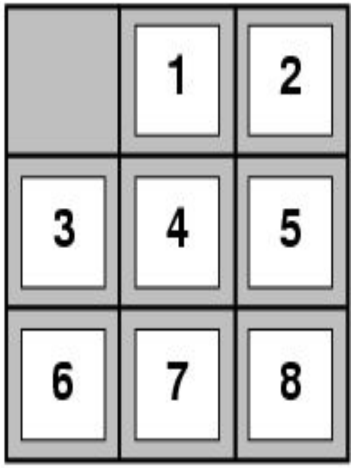
Goal State

EXAMPle heuristics

- E.g., for the 8-puzzle:
- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance(i.e., no. of squares from desired location of each tile)
- $h_1(S) = 8$
- $h_2(S) = 3+1+2+2+2+3+3+2 = 18$



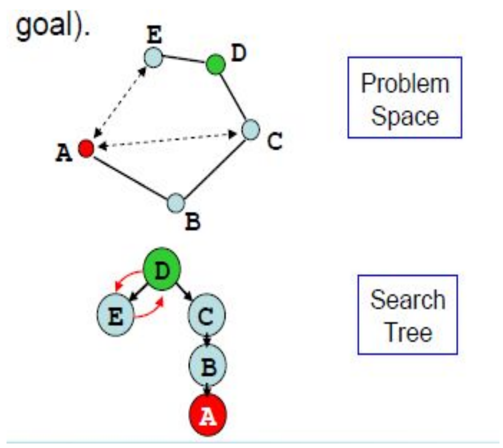
Start State



Goal State

EXAMPLE HEURISTICS

- **For Graph Search problem**
- **Straight-line distance :** The distance between two locations on a map can be known without knowing how they are linked by roads (i.e. the absolute path to the goal).



Properties of best first seaRch

- It may **get stuck** in an infinite branch that doesn't contain the goal .
- It does **not guarantee** to find the shortest path solution .

Memory requirement :

- **In best case :** as depth first search.
- **In average case :** between depth and breadth.
- **In worst case :** as breadth first search