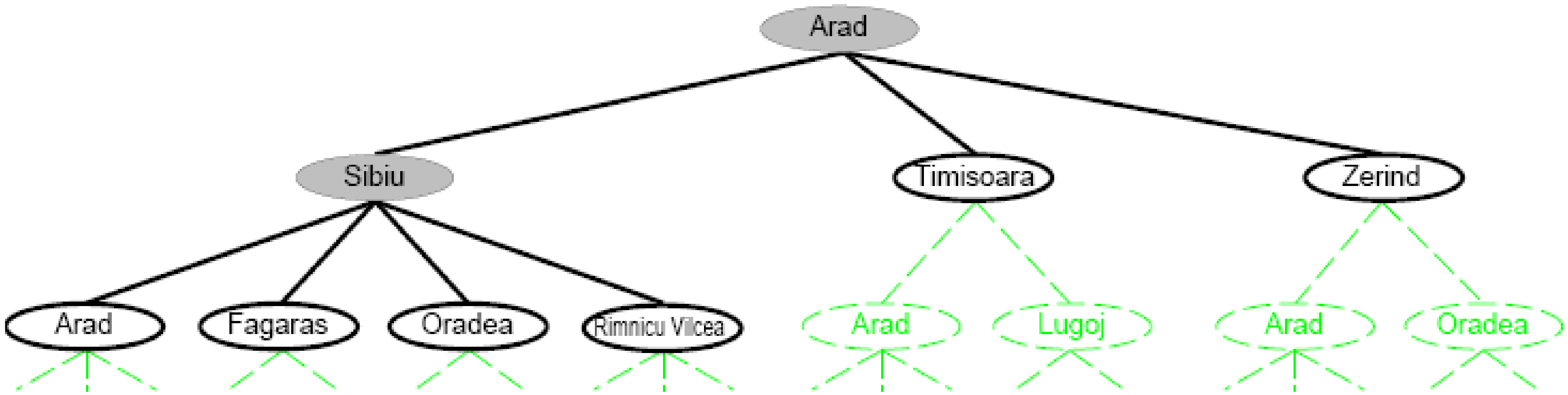
Searching For Solutions



Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded.; nodes that have been generated but not yet expanded are outlined in bold;nodes that have not yet been generated are shown in faint dashed line

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
   initialize the search tree using the initial state of problem
   loop do
      if there are no candidates for expansion then return failure
      choose a leaf node for expansion according to strategy
      if the node contains a goal state then return the corresponding solution
      else expand the node and add the resulting nodes to the search tree
```

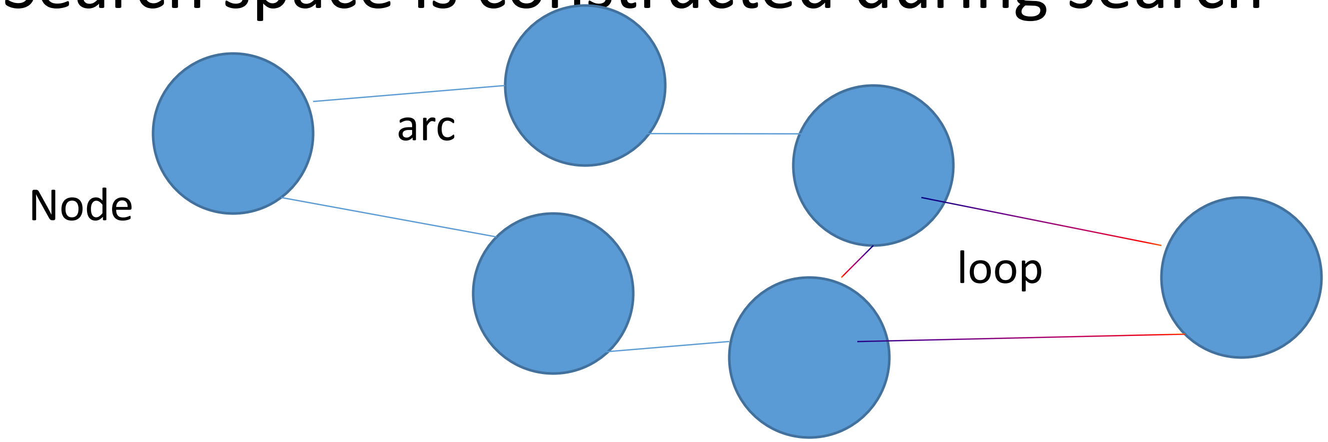
Figure 1.24 An informal description of the general tree-search algorithm

Tree Search

- Depth: The number of steps along the path from the initial state.
- Parent: the node in the search tree that generated this node;
- ACTION : the action that was applied to the parent to generate the node;
- PATH-COST :the cost, denoted by g(n),of the path from initial state to the node, as indicated by the parent pointers;
- leaf node: a node with no successors in the tree
- Fringe :Fringe is a collection of nodes that have been generated but not yet been expanded. Each element of the fringe is a leaf node.

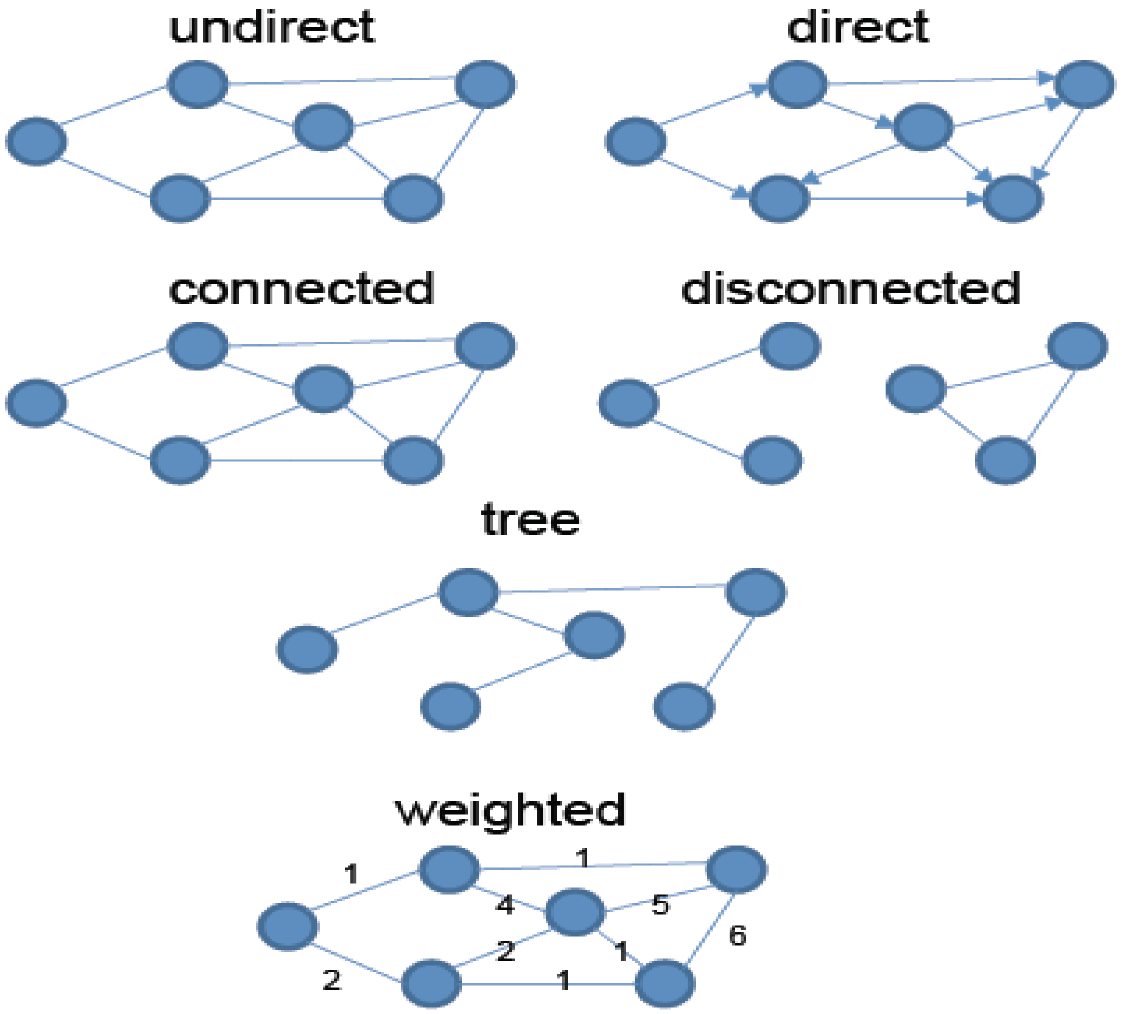
$$\text{PATH-COST}[s] \leftarrow \text{PATH-COST}[node] + \text{STEP-COST}(node, action, s)$$

- Representing the search space is the first step to enable the problem resolution
- Search space is mostly represented through graphs
- A graph is a finite set of *nodes* that are connected by *arcs*
- A *loop may exist in a graph,* where an arc lead back to the original node
- In general, such a graph is not explicitly given
- Search space is constructed during search



State Space Representation

- A graph is *undirected* if arcs do not imply a direction, *direct* otherwise
- A graph is *connected* if every pair of nodes is connected by a path
- A connected graph with no loop is called *tree*
- A *weighted graph*, is a graph for which a value is associated to each arc



```
function TREE-SEARCH(problem, fringe) returns a solution, or failure

   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
      if EMPTY?(fringe) then return failure
      node ← REMOVE-FIRST(fringe)
      if GOAL-TEST[problem] applied to STATE[node] succeeds
         then return SOLUTION(node)
      fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

```
function EXPAND(node, problem) returns a set of nodes

   successors ← the empty set
   for each ⟨action, result⟩ in SUCCESSOR-FN[problem](STATE[node]) do
      s ← a new NODE
      STATE[s] ← result
      PARENT-NODE[s] ← node
      ACTION[s] ← action
      PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
      DEPTH[s] ← DEPTH[node] + 1
      add s to successors
   return successors
```

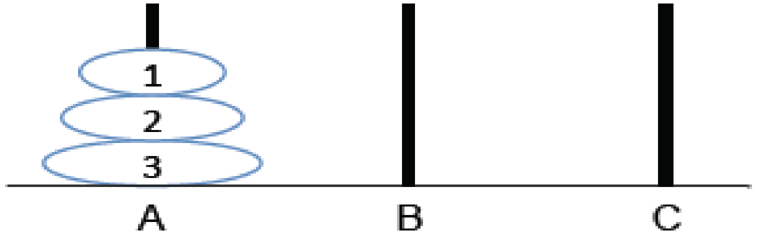
Towers of Hanoi

- 3 pegs A, B, C
- 3 discs represented as natural numbers (1, 2, 3) which correspond to the size of the discs
- The three discs can be arbitrarily distributed over the three pegs, such that the following constraint holds:

$$d_i \text{ is on top of } d_j \longrightarrow d_i < d_j$$

- Initial status: ((123)()())
- Goal status: (()()(123))

https://www.youtube.com/watch?v=aMEbboWmVCo

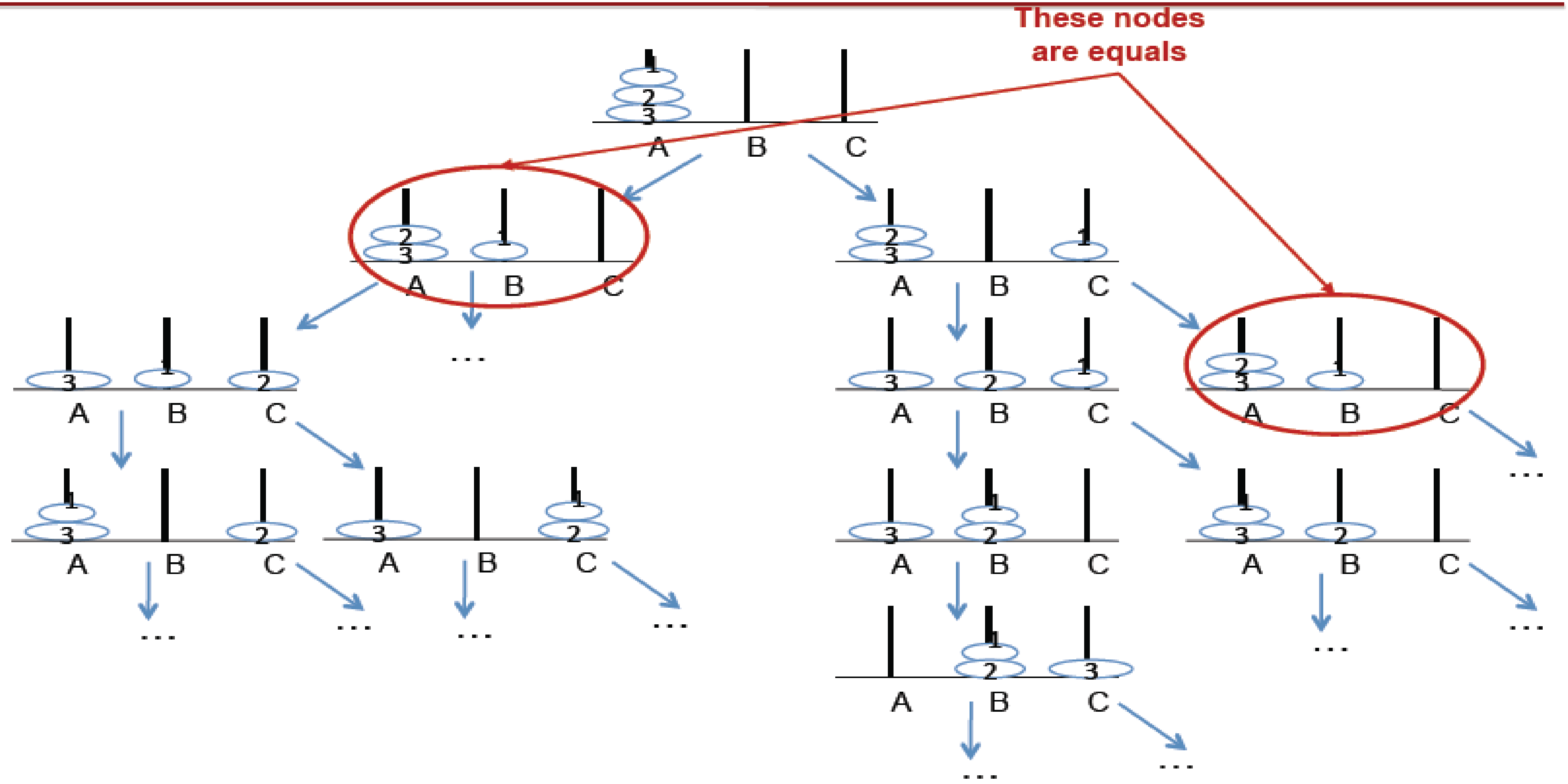


Operators:

Move *disk* to *peg*

Applying: Move 1 to C (1 ⟶ C) to the initial state ((123)()()) a new state is reached ((23)()(1))

Cycles may appear in the solution!



* A partial tree search space representation

MEASURING PROBLEM-SOLVING PERFORMANCE

- The algorithm’s performance can be measured in four ways :
- **Completeness** : Is the algorithm guaranteed to find a solution when there is one?
- **Optimality** : Does the strategy find the optimal solution
- **Time complexity** : How long does it take to find a solution?
- **Space complexity** : How much memory is needed to perform the search?

- Time and space complexity are measured in terms of

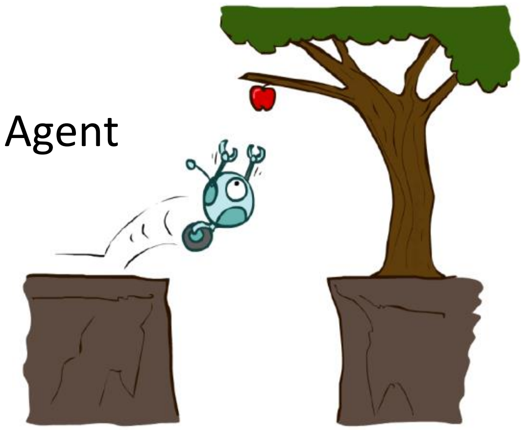
– b: maximum branching factor of the search tree

– d: depth of the shortest path solution

– m: maximum depth of the state space (may be ∞)

Different search strategies

Reflective Agent

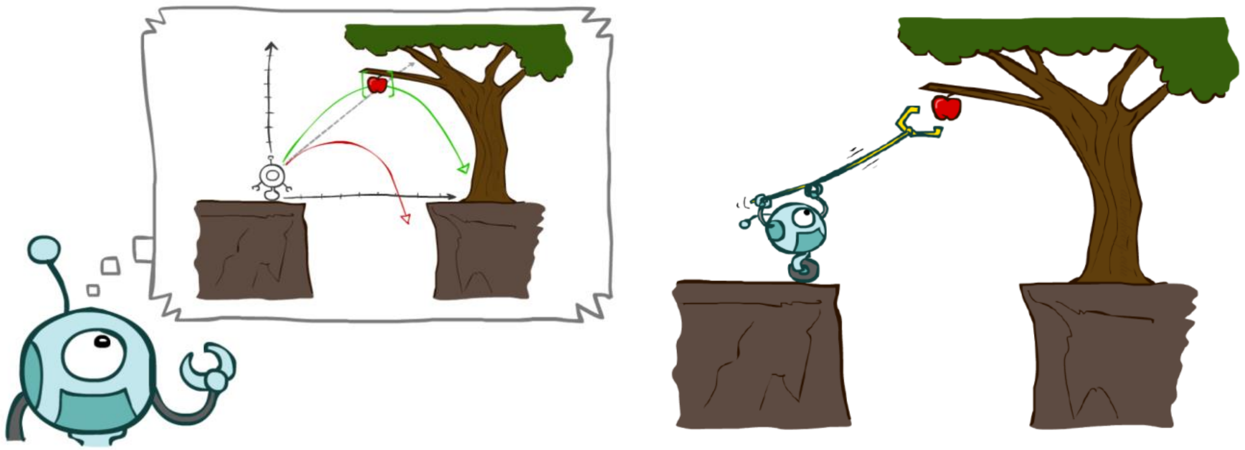


1. Blind Search strategies or Uninformed search

2. Informed Search

Planning Agent

3. Constraint Satisfaction Search



4. Adversary Search

UNINFORMED SEARCH STRATGES

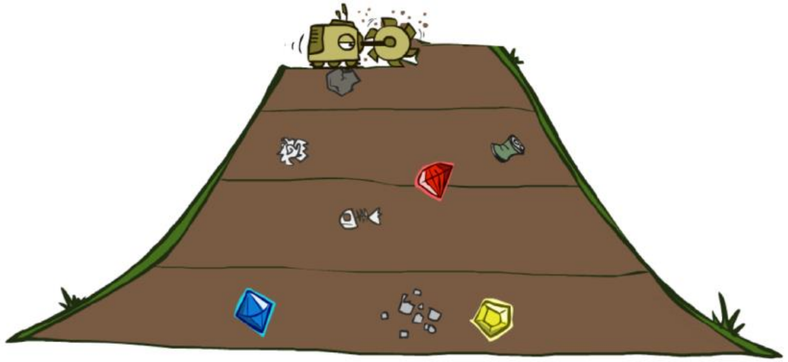
- **Strategies** that know whether one non goal state is “more promising” than another are called **Informed search or heuristic search** strategies.
- uninformed search strategies as given below.

Breadth-first search

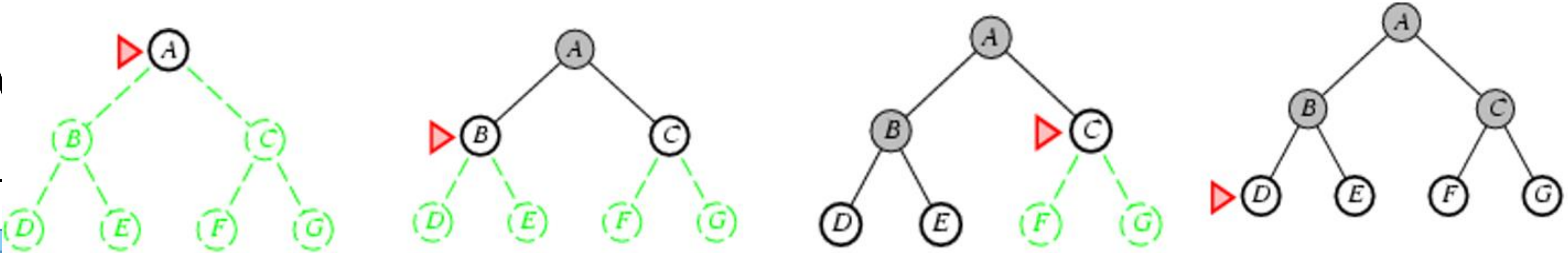
Depth-first search

BFS

- In this strategy, the root node is expanded first, then all the nodes generated by the root node are expanded next, and then *their* successors, and so on.
- **I**n general, all the nodes at depth *d* in the search tree are expanded before the nodes at depth $d + 1$.
- BFS- implemented by calling TREE-SEARCH with an <u>empty fringe that is a first-in-first-out (FIFO</u>) queue, assuring that the nodes that are visited first will be expanded first.
- Calling TREE-SEARCH(problem,FIFO-QUEUE()) ->bfs
- FIFO queue puts the newly generated successors at the end of the queue, which means that shallow nodes are expanded before deeper nodes.



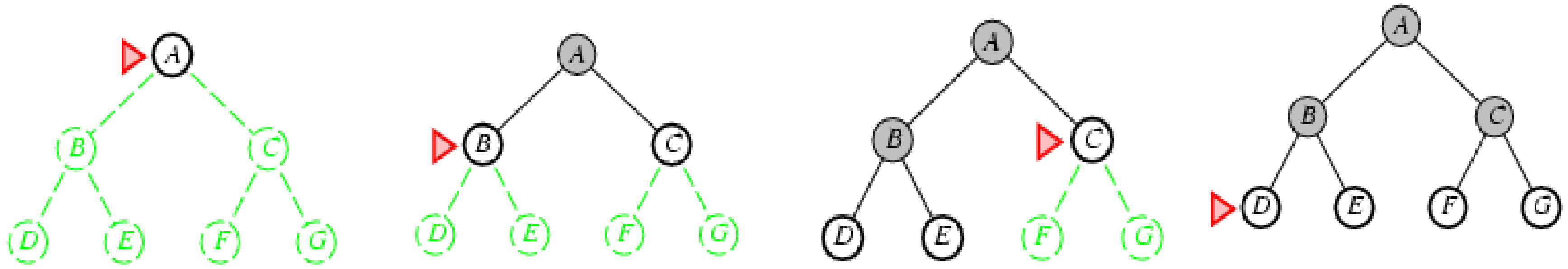
BFS A




```
function BREADTH
  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)
```

Figure 3.11 Breadth-first search on a graph.

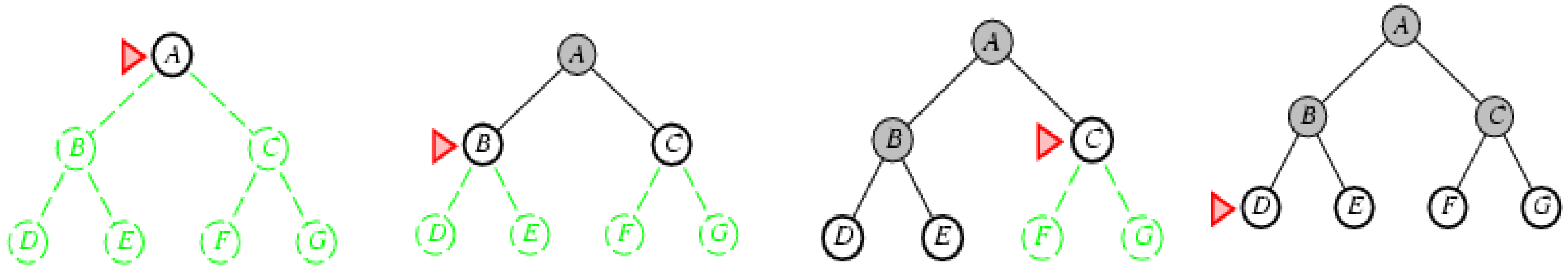
Iteration	Fringge	Closed list (Visited)	Goal test
0	A		A×
1	B,C	A	B×
2	C,D,E	A,B	C×
3	D,E,F,G	A,B,C	D×
4	E,F,G	A,B,C,D	E×
5	F,G	A,B,C,D,E	F✓



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*)

Figure 3.11 Breadth-first search on a graph.



If ‘F’ is a goal : FIFO queue

Iteration	Fringe	Closed list (Visited)	Goal test
0	A		A×
1	B,C	A	B×
2	C,D,E	A,B	C×
3	D,E,F,G	A,B,C	D×
4	E,F,G	A,B,C,D	E×
5	F,G	A,B,C,D,E	F✓

Time complexity for BFS

- Assume every state has b successors.
- The root of the search tree generates b nodes at the first level,each of which generates b more nodes,for a total of b^2 at the second level.
- Each of these generates b more nodes,yielding b^3 nodes at the third level,and so on.
- Now suppose,that the solution is at depth d. In the worst case,we would expand all but the last node at level d,generating b^{d+1} - b nodes at level d+1.
- Then the total number of nodes generated is

  $1+b + b^2 + b^3 + \ldots + b^d + (b^{d+1} + b) \quad = O(b^{d+1)}.$
- Every node that is generated must remain in memory, because it is either part of the fringe or is an ancestor of a fringe node.
- The space complexity is, therefore ,the same as the time complexity

Properties of breadth-first-search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? No, unless step costs are constant

Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs = 8640GB.

Time and Memory Requirements for BFS – $O(b^{d+1})$

Example:

- b = 10
- 10000 nodes/second
- each node requires 1000 bytes of storage

Depth	Nodes	Time	Memory
2	1100	.11 sec	1 meg
4	111,100	11 sec	106 meg
6	10^7	19 min	10 gig
8	10^9	31 hrs	1 tera
10	10^{11}	129 days	101 tera
12	10^{13}	35 yrs	10 peta
14	10^{15}	3523 yrs	1 exa

- **Disadvantage** − Since each level of nodes is saved for creating next one, it consumes a lot of memory space. Space requirement to store nodes is exponential.
- Its complexity depends on the number of nodes.
- Breadth-first search is useful when
- space is not a problem;
- few solutions may exist, and at least one has a short path length; and
- infinite paths may exist, because it explores all of the search space, even with infinite paths.
- It is a poor method when all solutions have a long path length or there is some heuristic knowledge available. It is not used very often because of its space complexity.

Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue