Search Algorithms

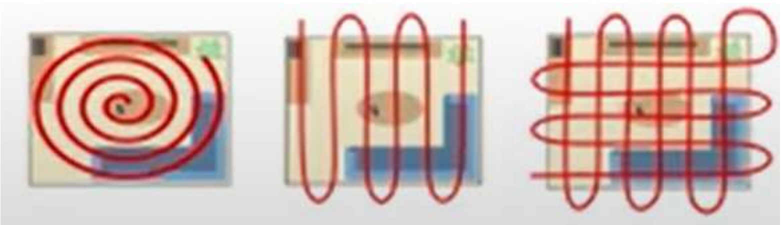
Search Algorithm

- Uninformed:
  - Breadth-first search
  - Uninform-cost search
  - Depth-first search
- Heuristic-based:
  - Greedy best-first search
  - A* Search

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  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
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Uninformed Search Strategies

- Also called ‘blind search’
- the strategies have no additional information about states beyond that provided in the problem definition.
- All they can do is **generate successors** and **distinguish a goal** state from a non-goal state.
- All search strategies are distinguished by the order in which nodes are expanded.



- Strategies that know whether one non-goal state is “more promising” than another are called informed search or heuristic search strategies

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.

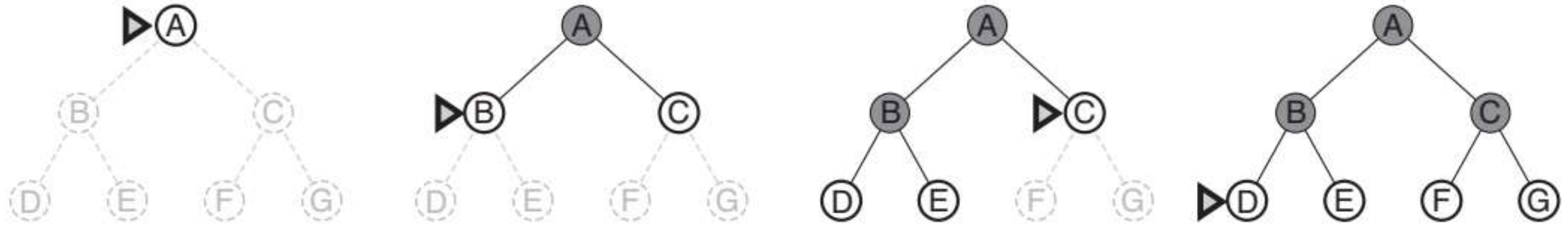


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

- This is achieved very simply by using a FIFO queue for the frontier

Cont.

```
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)
```

Cont.

- Disadvantage is Space and Time Complexity   O (b^d)
- When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node
- However, real world problem may not have equal cost.

Uniform cost search

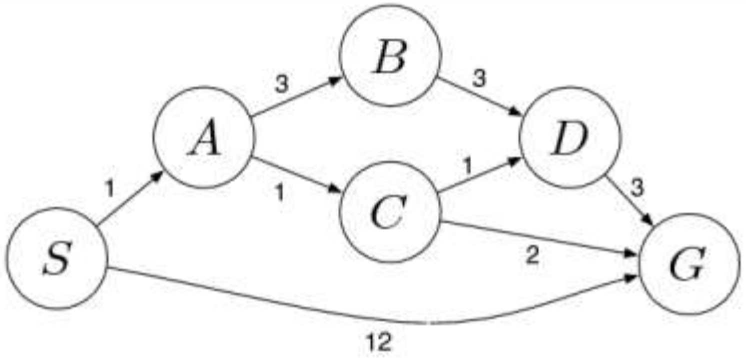
- Instead of expanding the shallowest node, uniform-cost search expands the node n with the **lowest path cost g(n)**.
- This is done by storing the frontier as a priority queue ordered by **g**
- Uniform-cost search does not care about the number of steps a path has, but only about their **total cost**

https://www.youtube.com/watch?v=XyoucHYKYSE&ab_channel=ShaulMarkovitch

UCS Algorithm

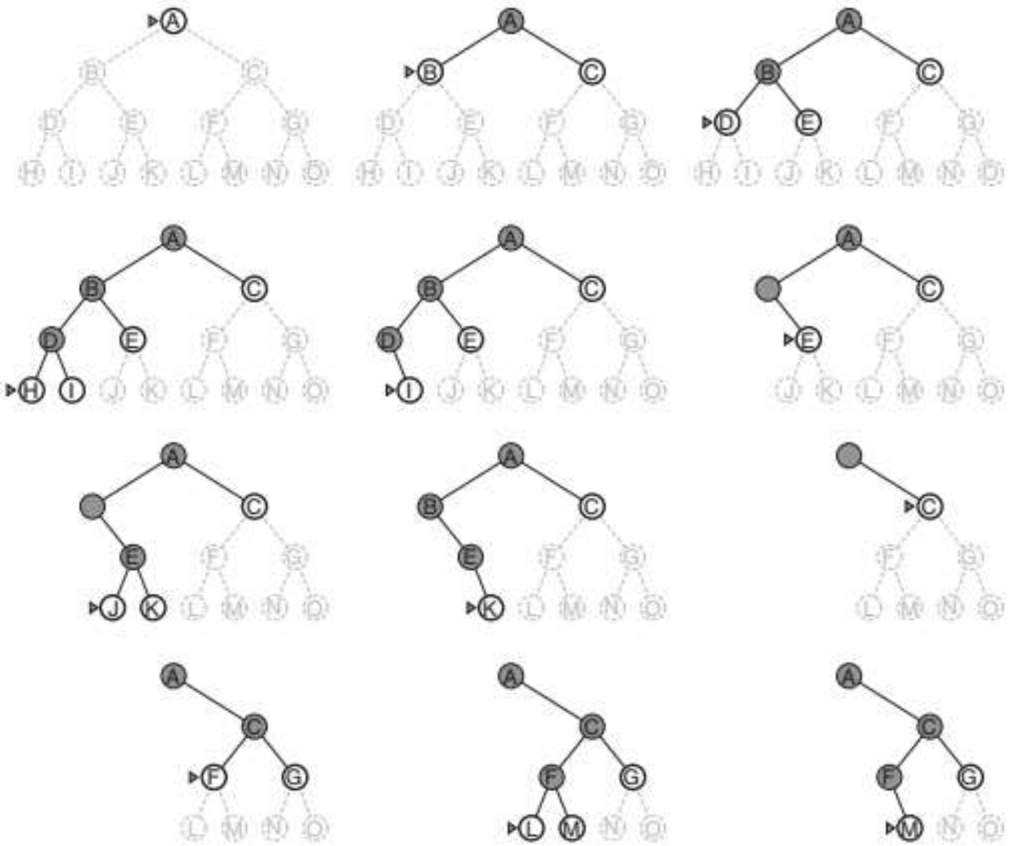
function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

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 content of priority queue
  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*



Depth First Search

- Depth-first search always **expands the deepest node** in the current frontier of the search tree.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- As those nodes are expanded, they are dropped from the frontier, so then the search **“backs up”** to the next deepest node that still has unexplored successors.
- BFS uses a FIFI queue, depth-first search uses a LIFO queue (stack)



Advantage of DFS over BFS – space complexity

- DFS has advantage over BFS when searching in a tree (not in graph)
- In case of tree search, DFS needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.
- Once a node has been expanded, it can be removed from memory as soon as its descendants have been fully explored.