, 2022

Solving problems by Search

Goal-based Agents

Atomic-Factored-Structured

Dr. Bilal Hoteit

Table of contents

What is Search?

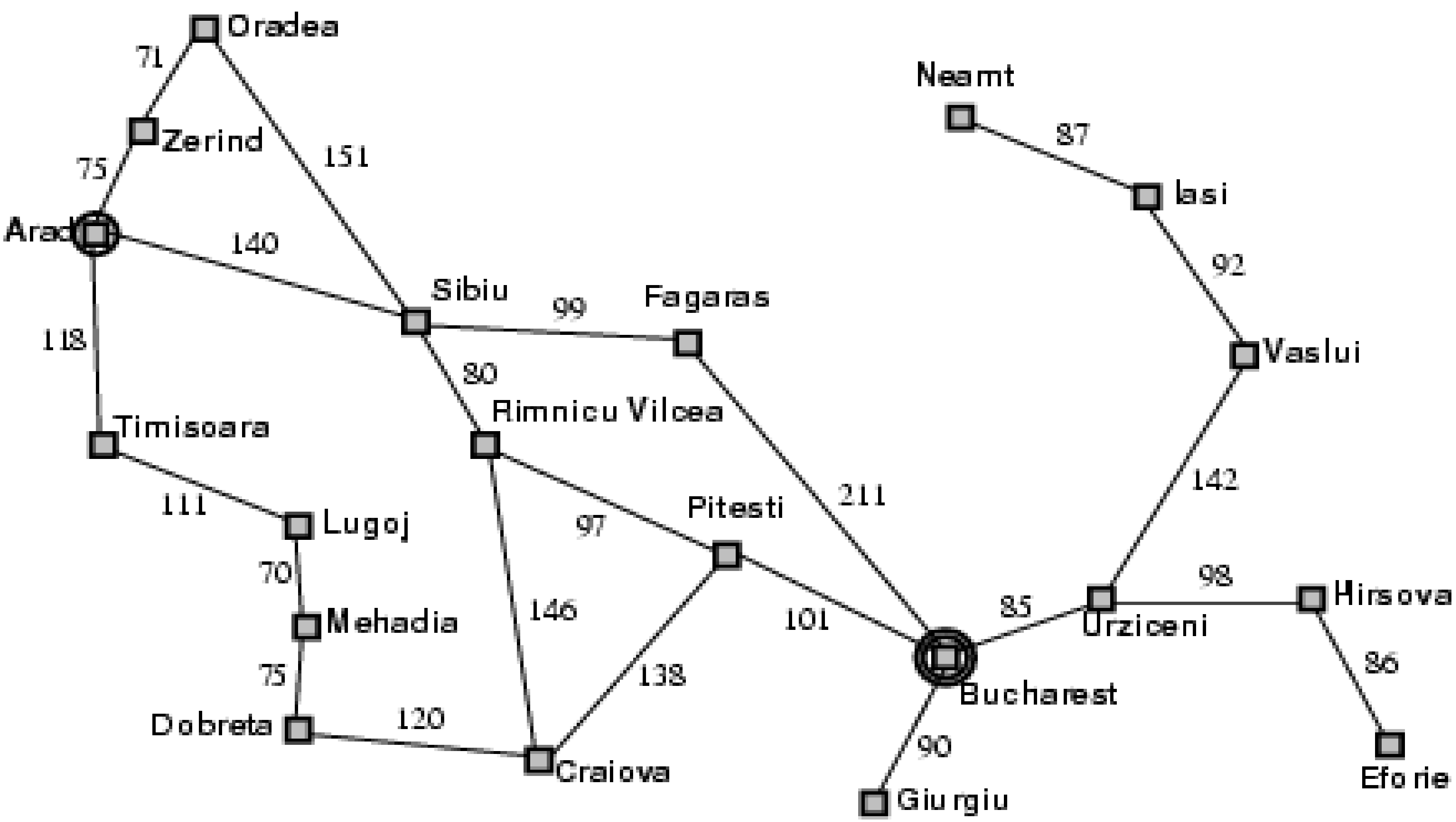
- Search is something we do all the time without realizing it.
  a. Finding this room required search.
  b. So does finding the best insurance deal.
  c. Or finding your keys.

- Computers are not, naturally good at search. BAD at Search ?
- But if they could do it, they would be very fast!
- We have to tell them how to search.
- If computers are to be efficient at search, we must very carefully tell them how to do it.

Mechanism of Search?

- All AI is search!
  a. Not totally true (obviously) but more true than you might think.
  b. Finding good/best solution to a problem amongst possible solutions.
- Ways of methodically deciding which solutions are better than others.
- Three Types of Search:
  a. Blind Search: Searching the possibilities, in some predefined order. The types of search we do in a maze.
  b. Heuristic Search: Searching the possibilities in order of their perceived value.
  c. Stochastic Search: Randomly searching in a cleaver way.

From Arad to Bucharest : Example?



Problem-solving agent.

- environment described as state space
- agent actions can change one state to some others
- current situation is one state in space
- goal situation is another state
- problem solving agent finds sequence of actions to move from current to goal state

Simplest Problem-solving agent.

- works by simulating the problem in internal representation and trying plans till a good one is discovered
- works in deterministic, static, single agent environments
- plan is made once and never changed
- works if plan is perfect – actions do what plan assumes
  no corrections to path are required
- works efficiently if space is not too large

Representation of Environment.

- abstractions of real world

a. states and state space – only relevant information in state representation. (name or key value)

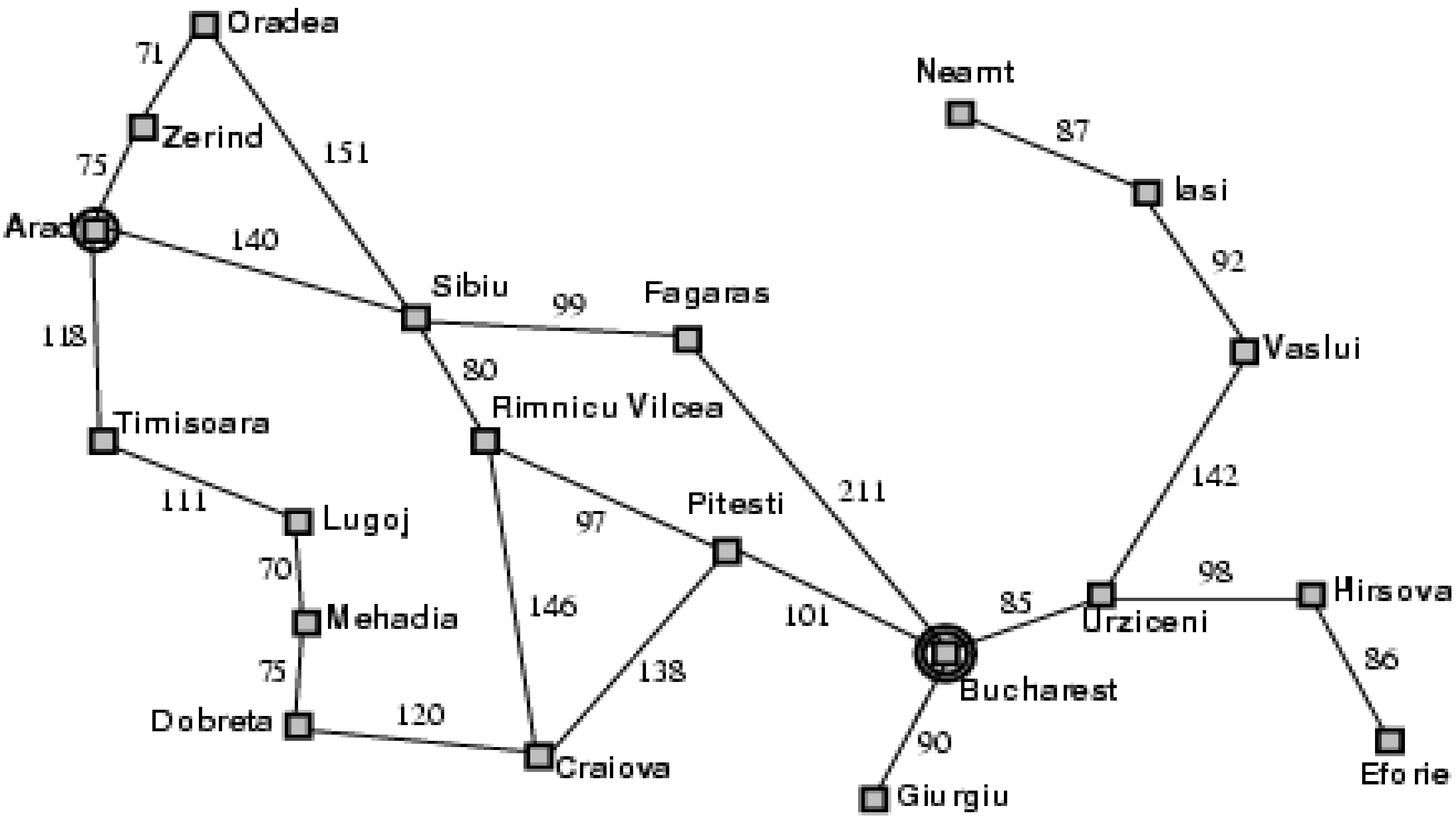
b. actions - successor function (Single move operator)

c. path cost (eg: touring problem TSP, length)

d. start state (initial node)

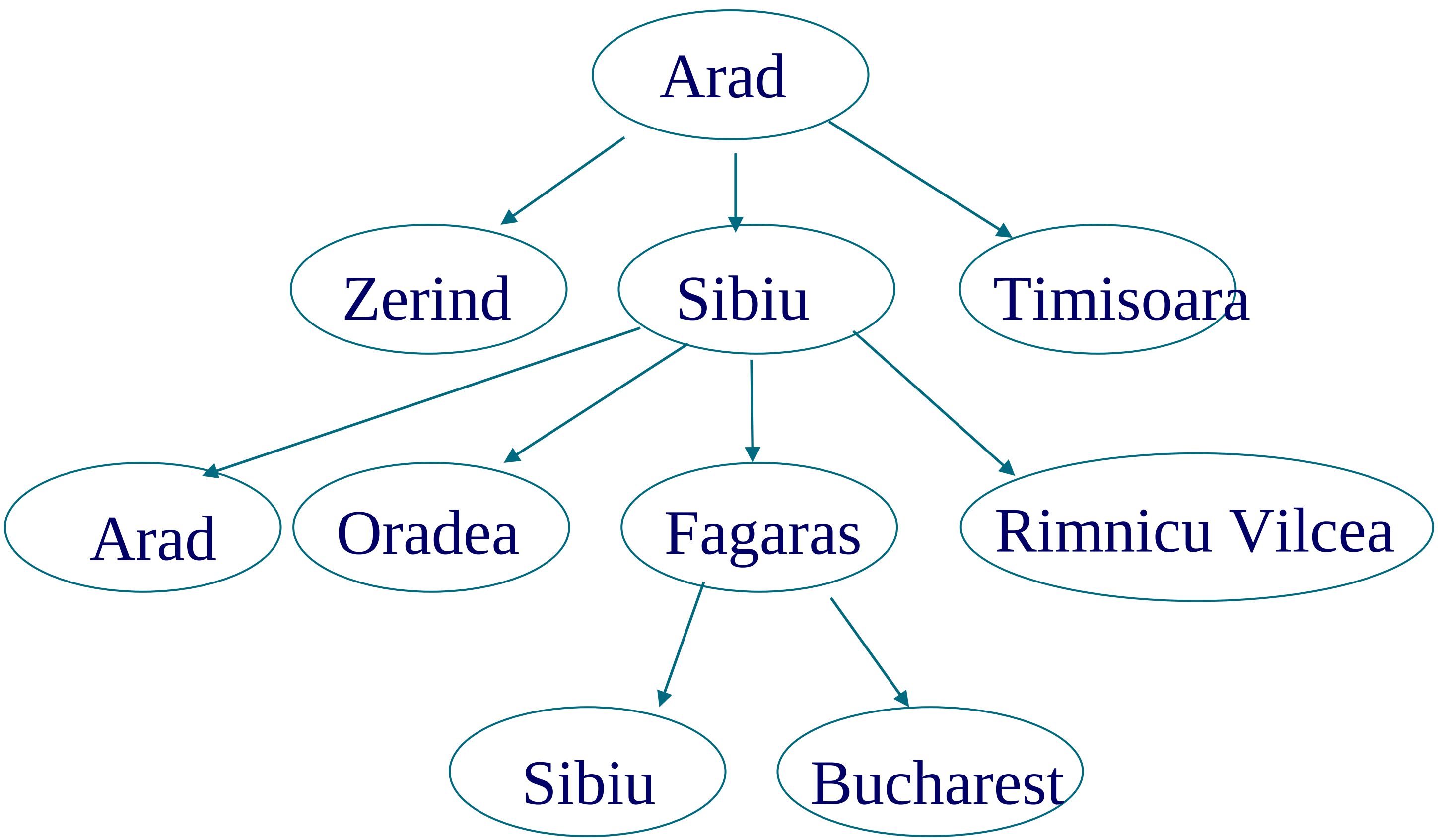
e. Goal or criterion function of state(s). (Single goal state)

From Arad to Bucharest : Example?



Representation of Environment.

- Start from initial node, pick one from all possible actions.
- Representing Search (simulation)



General concept of search Tree.

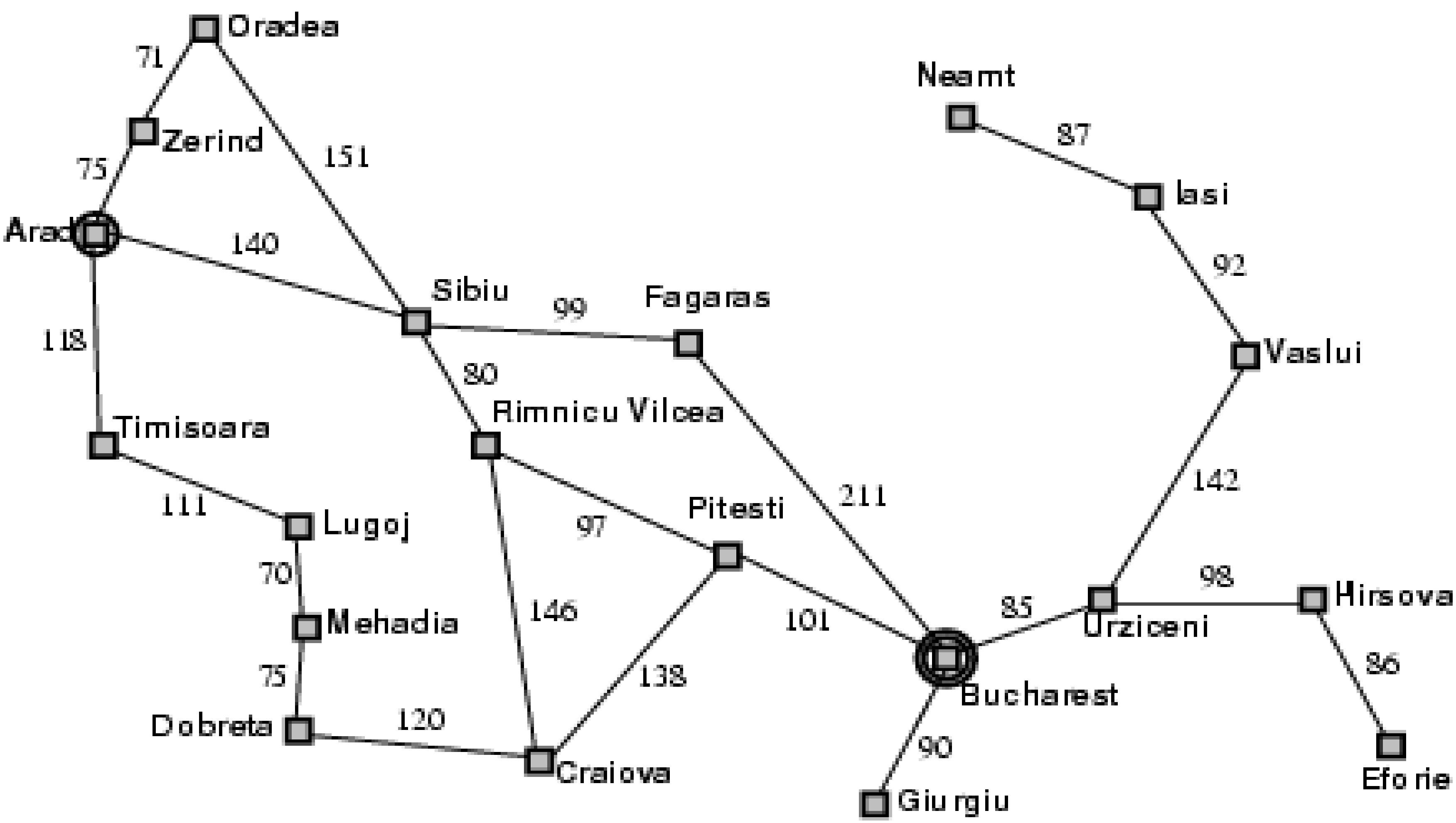
- Offline, simulated exploration of state space by generating successors of already-explored states

```
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
```

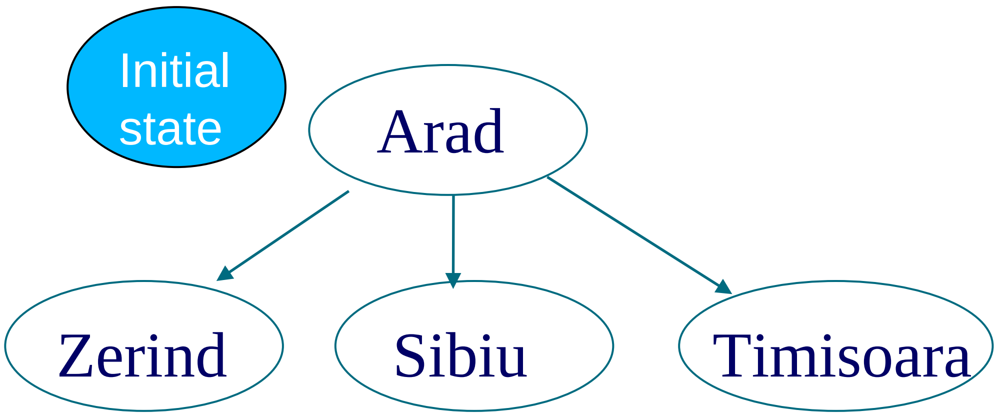
General concept of search Tree.

- startState: initial state of environment
- adjacentNode(graph): nodes of search tree: contain state, parent, action, path cost
- openList (fringe or frontier) : collection of Nodes generated, not tested yet
- action[n]: list of actions that can be taken by agent
- goalStateFound(state): evaluate a state as goal, returns Boolean
- precondition(state, action): test action applicability based on the current state, returns Boolean
- apply(node,action): apply action to get next state node(s), returns node
- makeSequence(node): returns sequence of actions, or path.

From Arad to Bucharest : Example?



Implementation of general search algorithm



- Three major approaches:

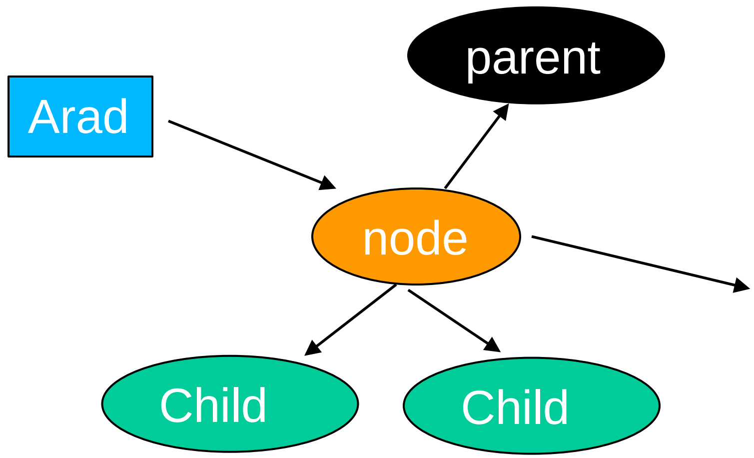
  FiFO, frontier is a queue.

  LIFO, frontier is a stack.

  Based on specific criteria.

Overview of general search algorithm

- Search tree
- Search node



Action	Right
Depth	5
Path-Cost	5
Expanded	yes

- Node expansion
  a. Evaluating the successor function on STATE(N)
  b. Generating a child of N for each state returned by the function
- Fringe of search tree
- Search strategy: At each stage it determines which node to expand

General search algorithm

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(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
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

algorithms of general search

- Several algorithms for un-informed search (Blind):

    a. breadth first
    b. Bidirectional
    c. depth first
    d. iterative deepening search
    e. Depth-limited
    f. uniform cost search

Uninformed search strategies use only the information available in the problem definition

Algorithms of general search

- A search strategy is defined by picking the order of node expansion

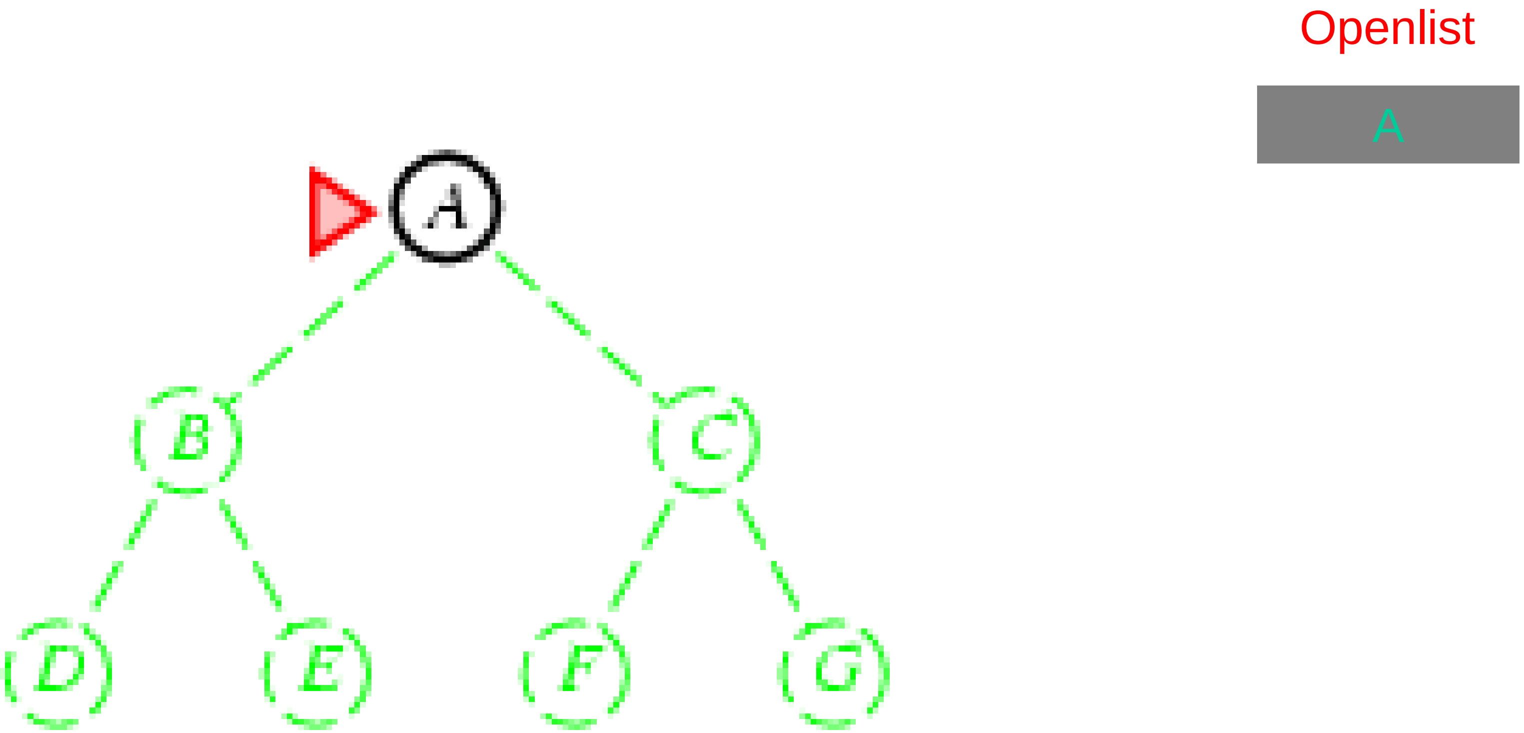
- Strategies are evaluated along the following dimensions:
  a. completeness: does it always find a solution if one exists?
  b. time complexity: number of nodes generated
  c. space complexity: maximum number of nodes in memory
  d. optimality: does it always find a least-cost solution?

- Time and space complexity are measured in terms of
  a. b: maximum branching factor of the search tree
  b. d: depth of the least-cost solution
  c. m: maximum depth of the state space (may be ∞)

- Demo: p:39 – p49. GOTO: lect0

Breadth-First Implementation

- Expand shallowest unexpanded node
- New nodes are inserted at the end of frontier (FIFO: queue)

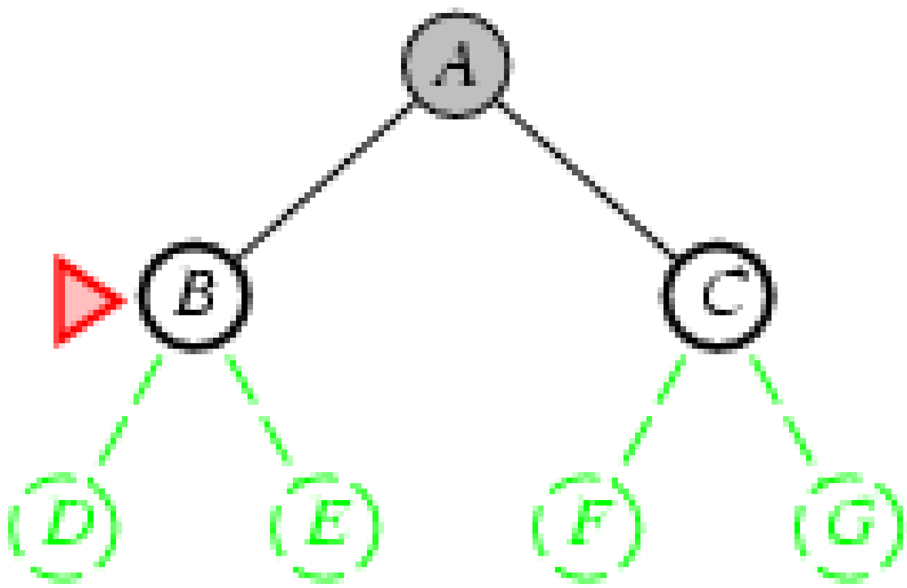


Breadth-First Implementation

- Expand shallowest unexpanded node
- New nodes are inserted at the end of frontier (FIFO: queue)

Openlist

B
C

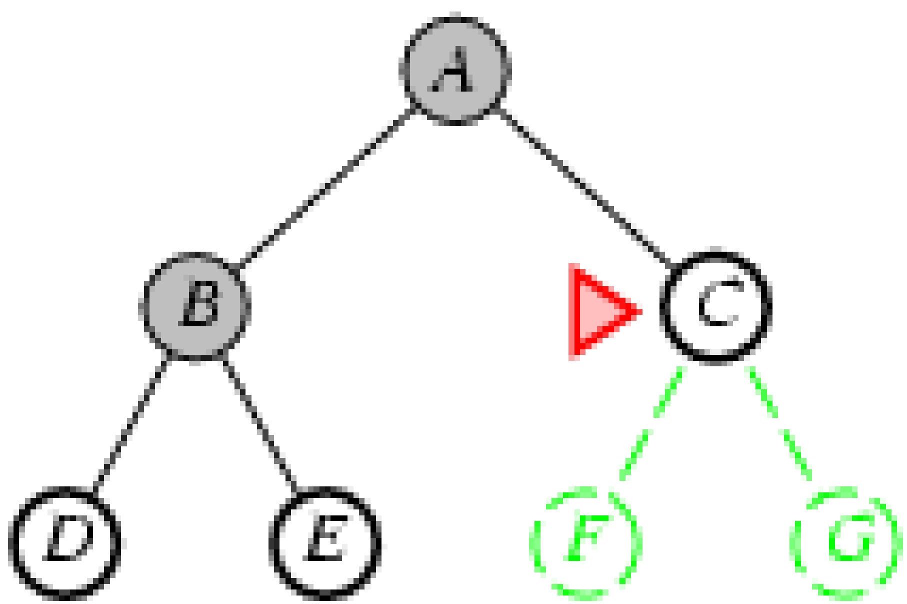


Breadth-First Implementation

- Expand shallowest unexpanded node
- New nodes are inserted at the end of frontier (FIFO: queue)

Openlist

C
D
E

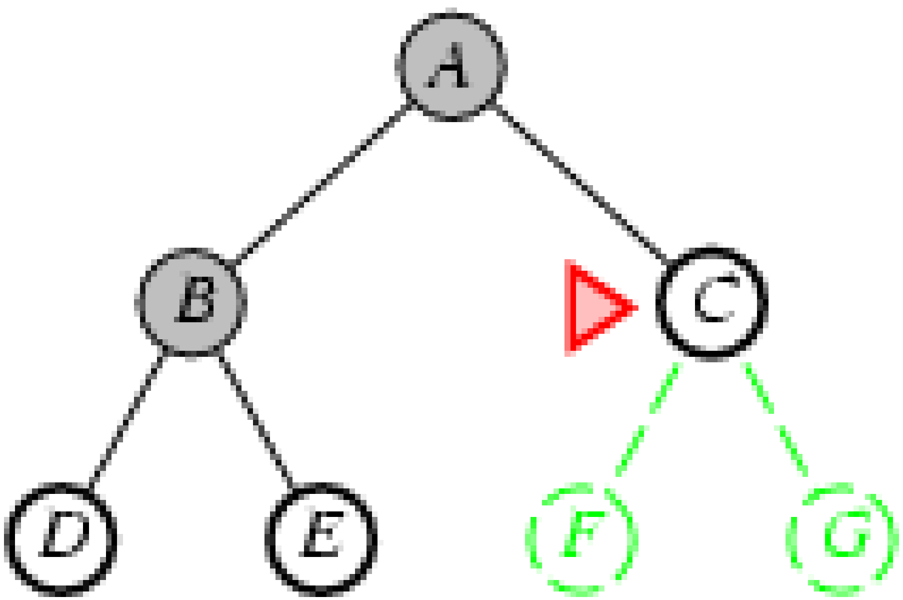


Breadth-First Implementation

- Expand shallowest unexpanded node
- New nodes are inserted at the end of frontier (FIFO: queue)

Openlist

C
D
E

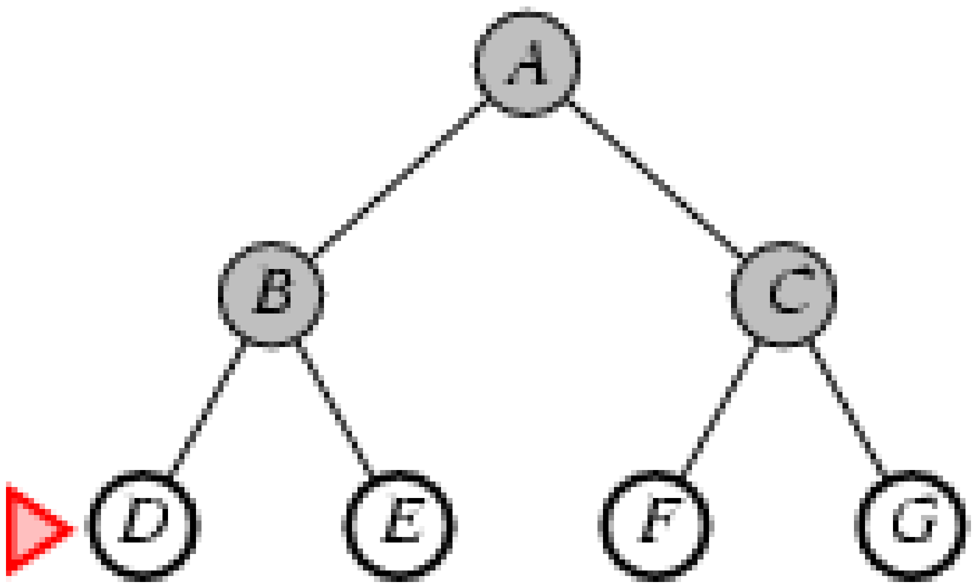


Breadth-First Implementation

- Expand shallowest unexpanded node
- New nodes are inserted at the end of frontier (FIFO: queue)

Openlist

D
E
F
G



Breadth-First Evaluation

- b: branching factor
- d: depth of shallowest goal node
- Breadth-first search is:
  a. Complete
  b. Optimal if step cost is 1
- Number of nodes generated:

$$1 + b + b^2 + \ldots + b^d = (b^{d+1}-1)/(b-1) = O(b^d)$$

- Time and space complexity is $O(b^d)$

Breadth-First Evaluation

- Time and Memory Requirements

d	# Nodes	Time	Memory
2	111	.01 msec	11 Kbytes
4	11,111	1 msec	1 Mbyte
6	$\sim 10^6$	1 sec	100 Mb
8	$\sim 10^8$	100 sec	10 Gbytes
10	$\sim 10^{10}$	2.8 hours	1 Tbyte
12	$\sim 10^{12}$	11.6 days	100 Tbytes
14	$\sim 10^{14}$	3.2 years	10,000 Tbytes

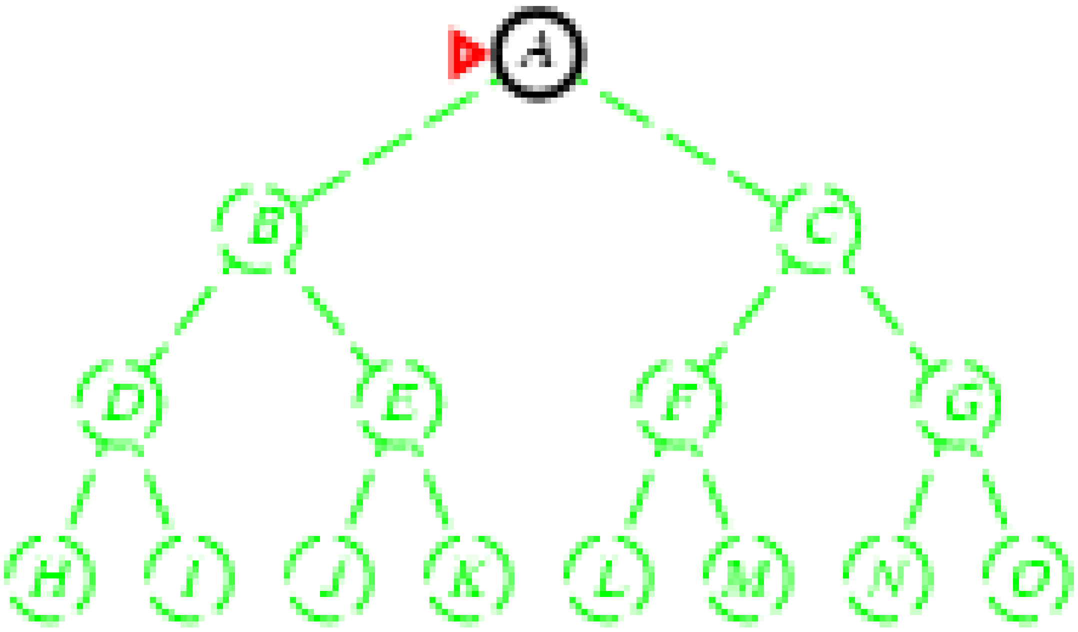
Memory requirements vs Time requirements
Assumptions: b = 10; 1,000,000 nodes/sec; 100bytes/node

Depth-First Implementation

- Expand deepest unexpanded node
- New nodes are inserted at the front of FRINGE (LIFO: stack)

Openlist

A

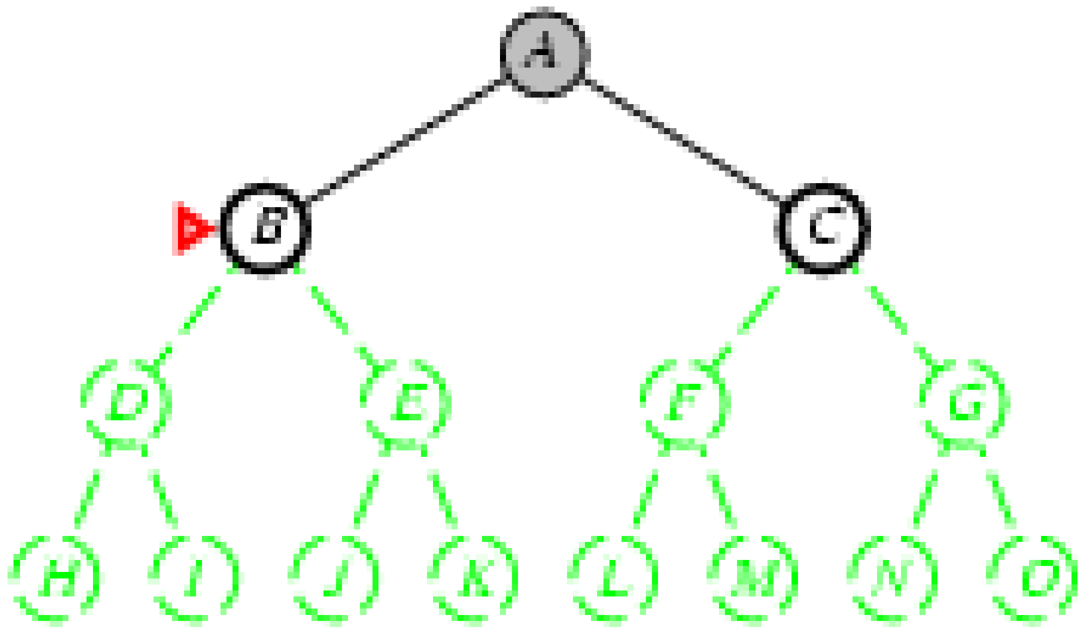


Depth-First Implementation

- Expand deepest unexpanded node
- New nodes are inserted at the front of FRINGE (LIFO: stack)

Openlist

C
B

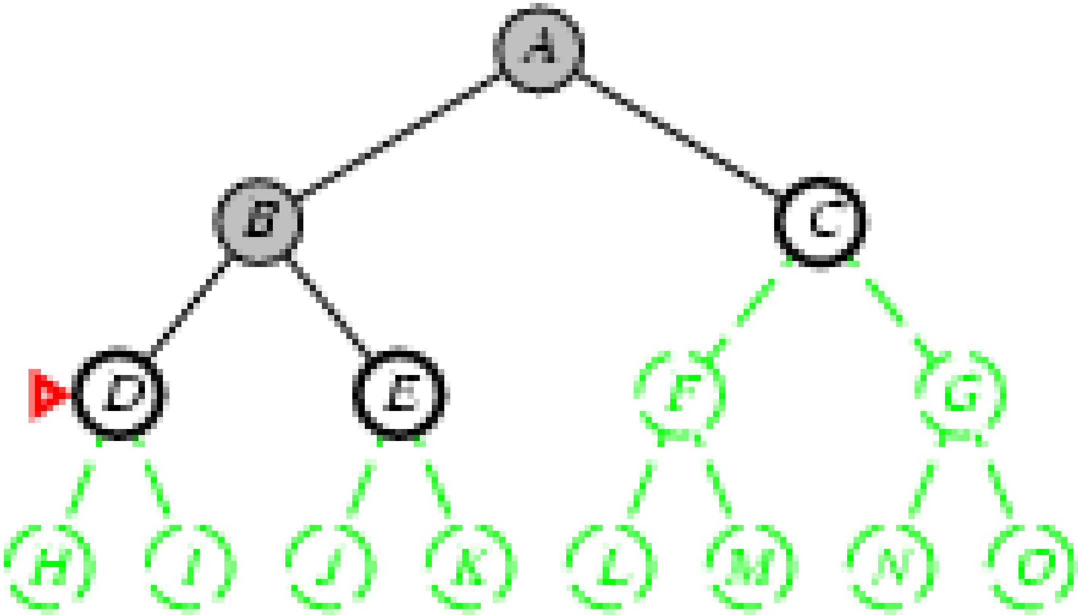


Depth-First Implementation

- Expand deepest unexpanded node
- New nodes are inserted at the front of FRINGE (LIFO: stack)

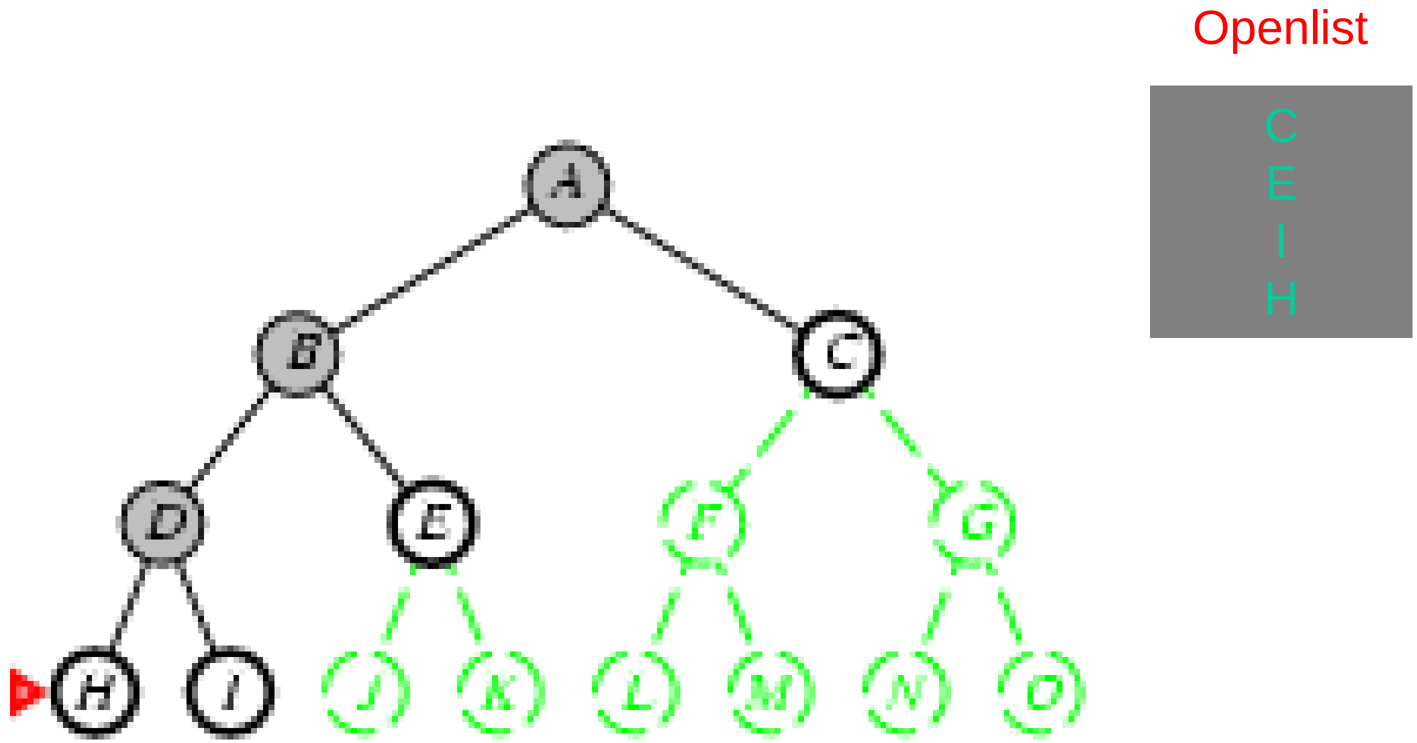
Openlist

C
E
D



Depth-First Implementation

- Expand deepest unexpanded node
- New nodes are inserted at the front of FRINGE (LIFO: stack)

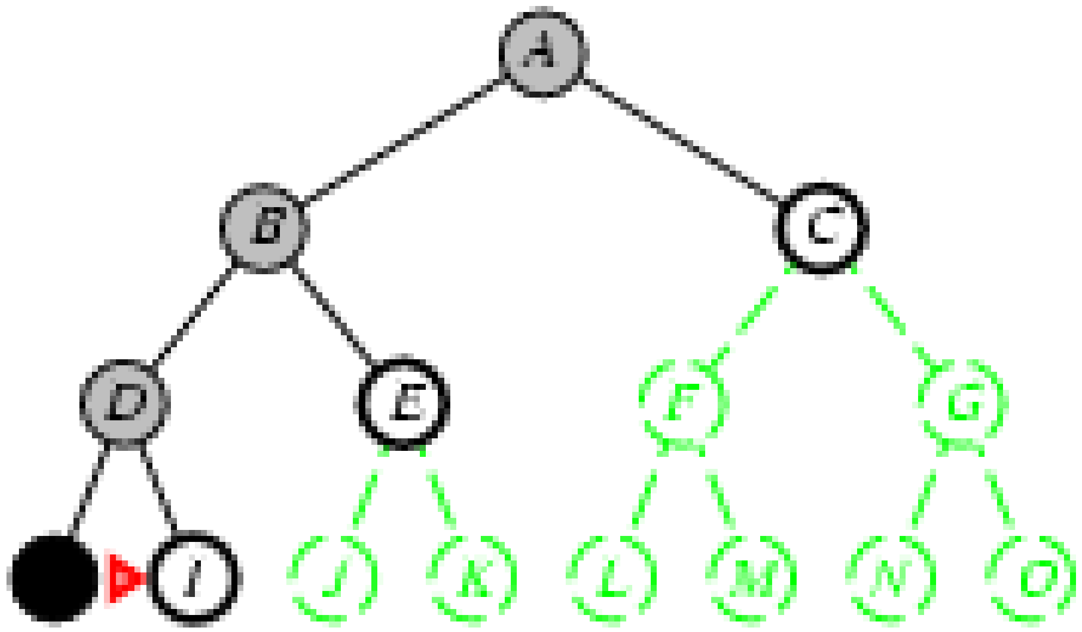


Depth-First Implementation

- Expand deepest unexpanded node
- New nodes are inserted at the front of FRINGE (LIFO: stack)

Openlist

C
E
I

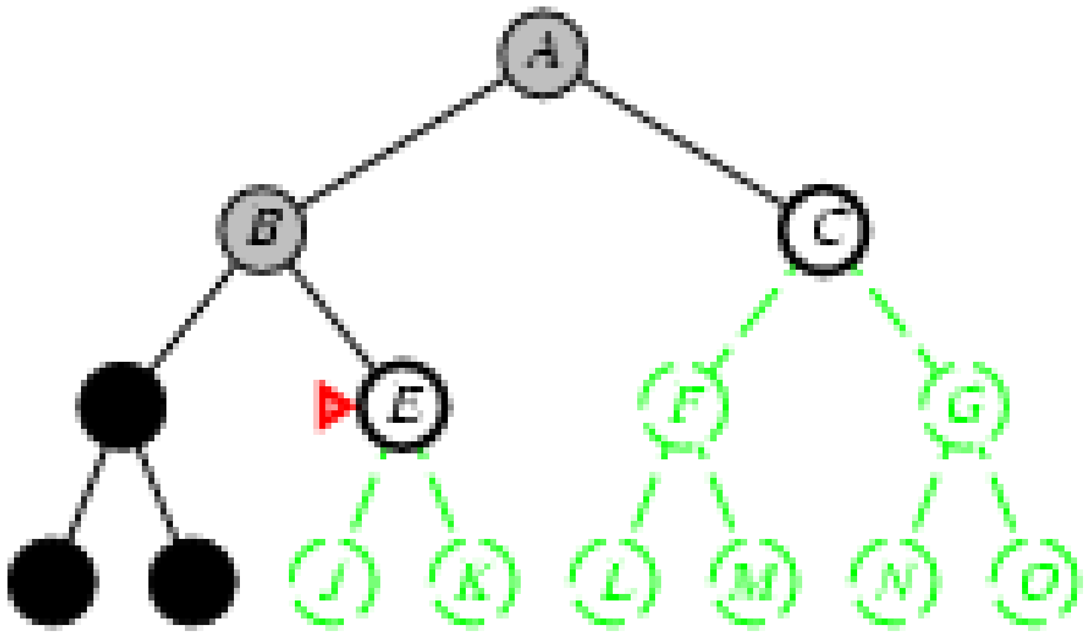


Depth-First Implementation

- Expand deepest unexpanded node
- New nodes are inserted at the front of FRINGE (LIFO: stack)

Openlist

C
E

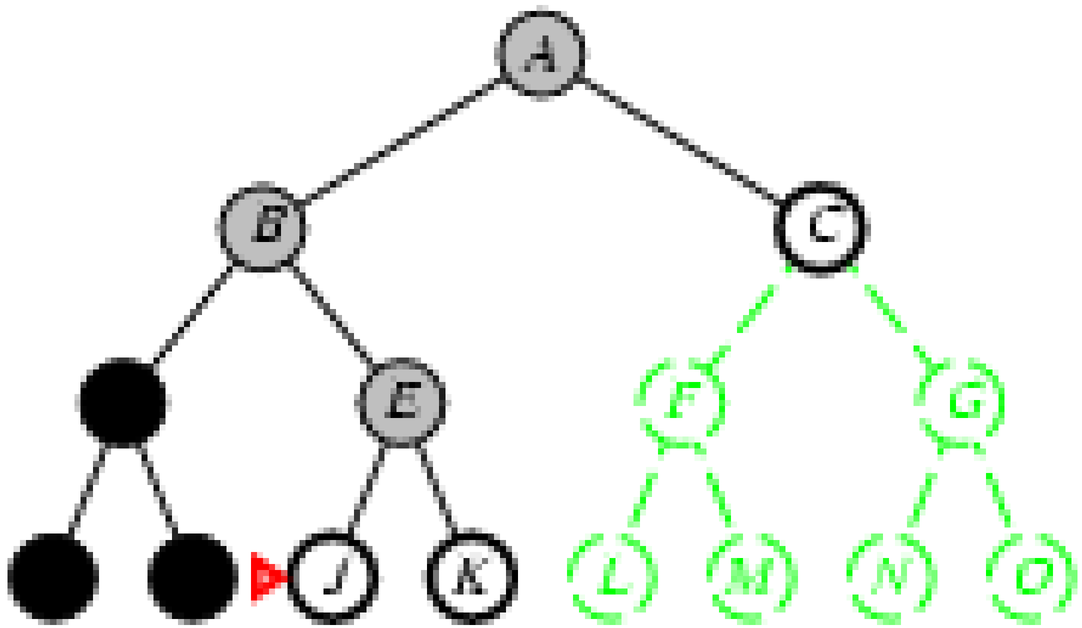


Depth-First Implementation

- Expand deepest unexpanded node
- New nodes are inserted at the front of FRINGE (LIFO: stack)

Openlist

C
K
J

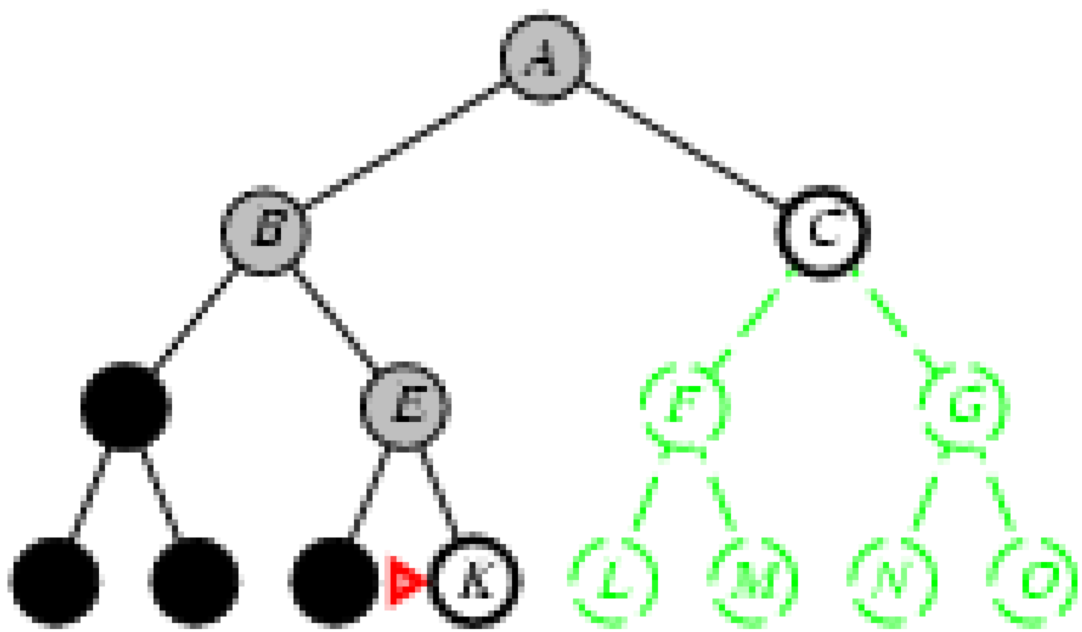


Depth-First Implementation

- Expand deepest unexpanded node
- New nodes are inserted at the front of FRINGE (LIFO: stack)

Openlist

C
K

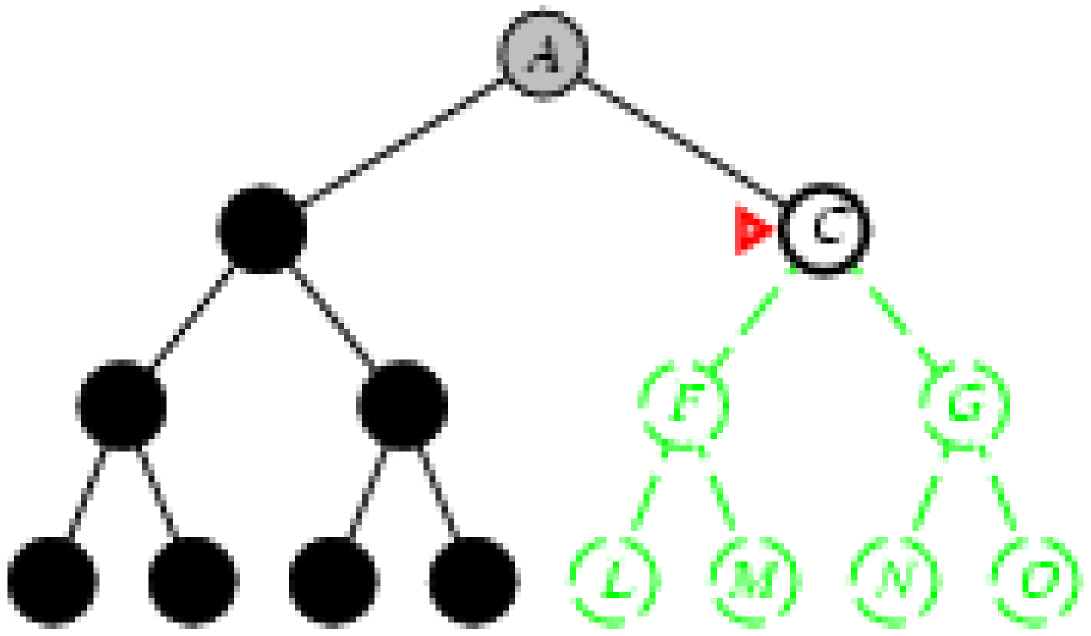


Depth-First Implementation

- Expand deepest unexpanded node
- New nodes are inserted at the front of FRINGE (LIFO: stack)

Openlist

C

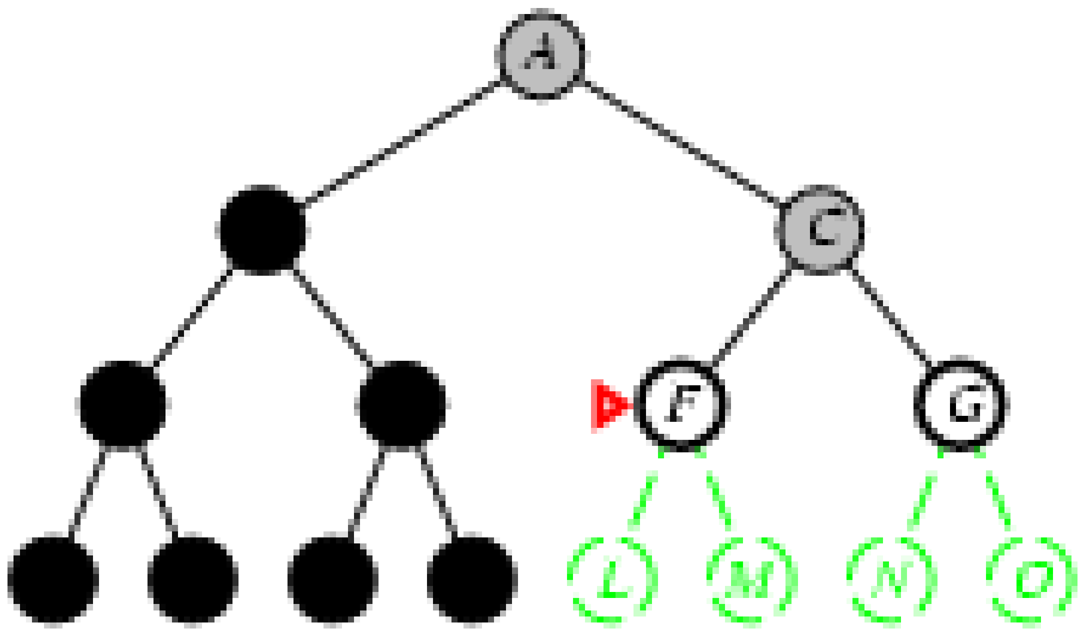


Depth-First Implementation

- Expand deepest unexpanded node
- New nodes are inserted at the front of FRINGE (LIFO: stack)

Openlist

G
F

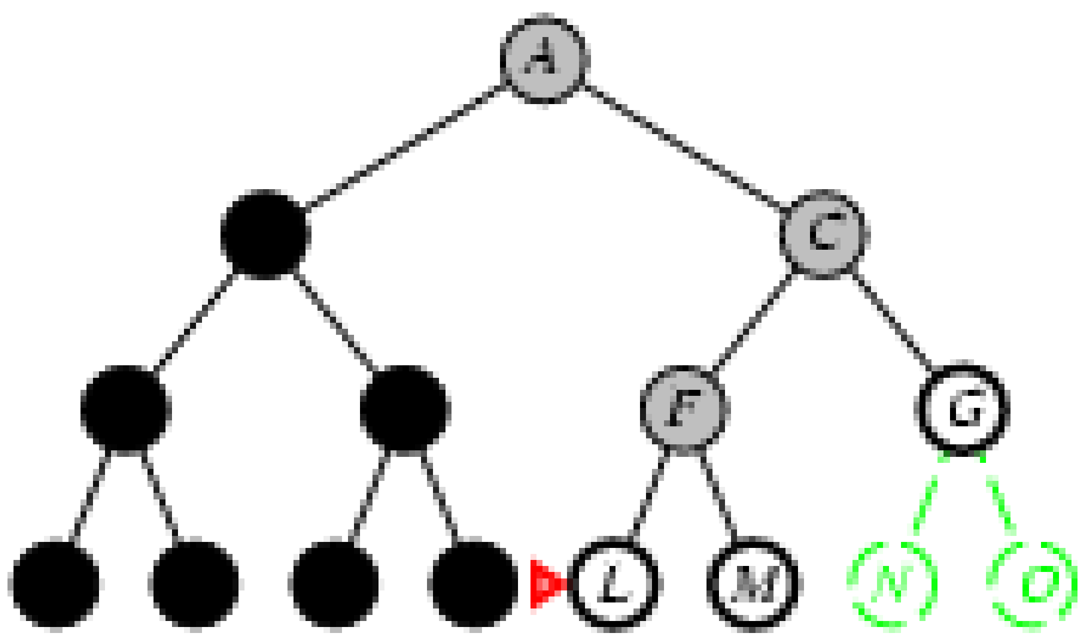


Depth-First Implementation

- Expand deepest unexpanded node
- New nodes are inserted at the front of FRINGE (LIFO: stack)

Openlist

G
M
L

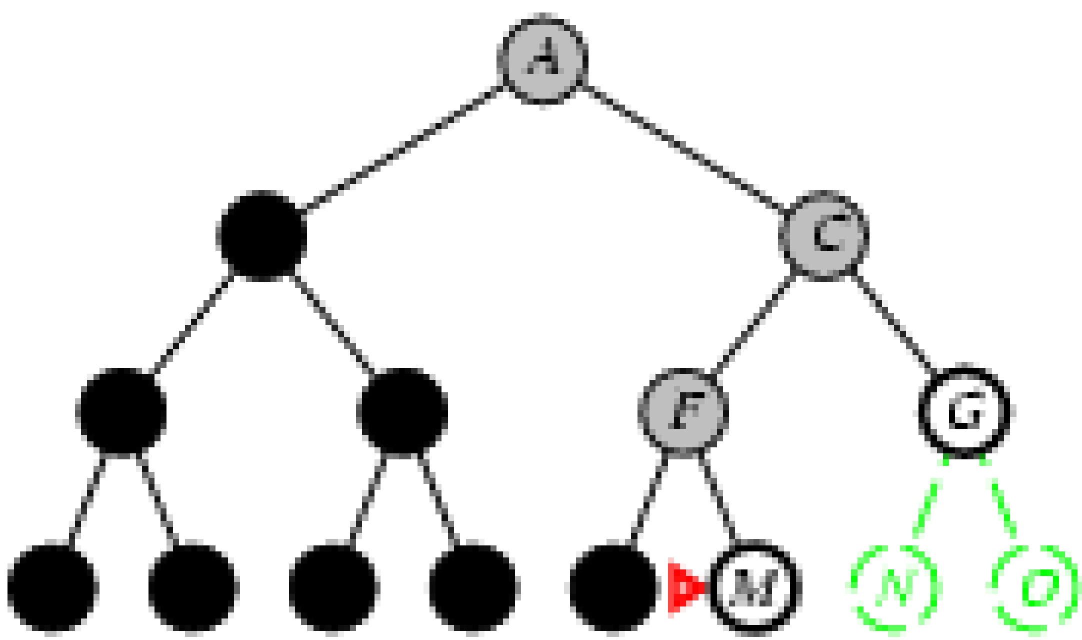


Depth-First Implementation

- Expand deepest unexpanded node
- New nodes are inserted at the front of FRINGE (LIFO: stack)

Openlist

G
M



Breadth-First Evaluation

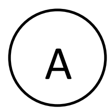
- b: branching factor
- d: depth of shallowest goal node
- m: maximal depth of a leaf node
- Depth-first search is:
  a. Complete only for finite search tree
  b. Not optimal
- Number of nodes generated:
  $1 + b + b2 + \ldots + bm = O(bm)$
- Time complexity is $O(b^m)$
- Space complexity is $O(bm)$ [or $O(m)$]
- [Reminder: Breadth-first requires $O(bd)$ time and space]

Uniform-cost Implementation

- New nodes are inserted at the front of FRINGE
- *fringe* = queue ordered by path cost, (FIFO: queue)

Openlist

A

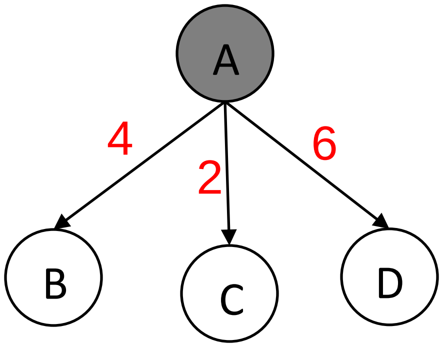


Uniform-cost Implementation

- New nodes are inserted at the front of FRINGE
- *fringe* = queue ordered by path cost, (FIFO: queue)

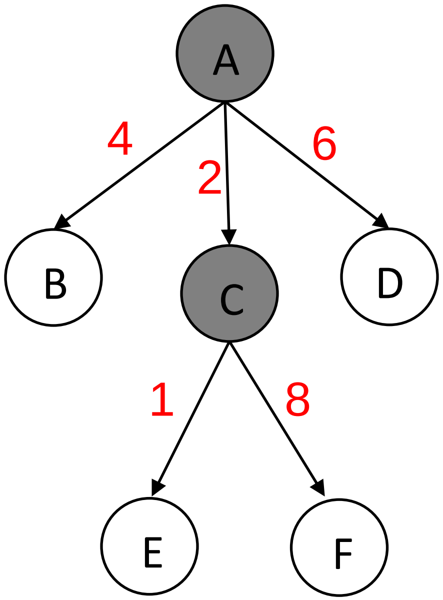
Openlist

A(0)
C(2)
B(4)
D(6)



Uniform-cost Implementation

- New nodes are inserted at the front of FRINGE
- *fringe* = queue ordered by path cost, (FIFO: queue)

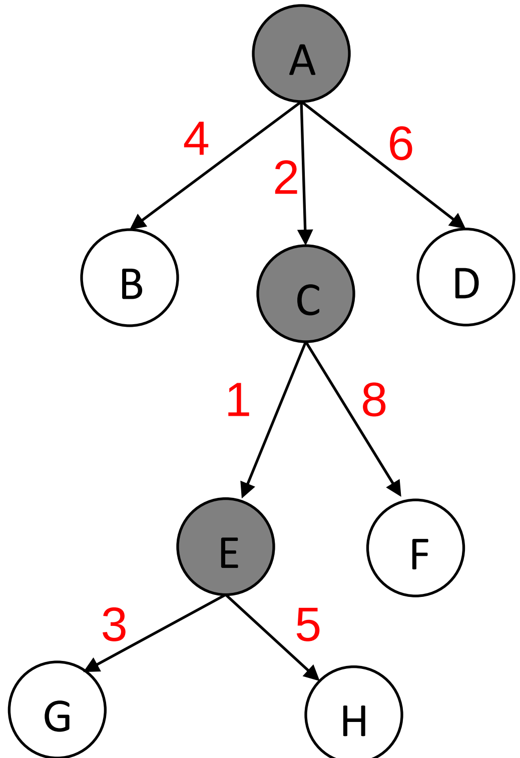


Openlist

C(2)
E(3)
B(4)
D(6)
F(10)

Uniform-cost Implementation

- New nodes are inserted at the front of FRINGE
- *fringe* = queue ordered by path cost, (FIFO: queue)



Openlist

B(4)
D(6)
G(6)
H(8)
F(10)

Uniform-cost Evaluation

- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost ≥ ε
- Time? # of nodes with g ≤ cost of optimal solution, O(bceiling(C*/ ε)) where C* is the cost of the optimal solution
- Space? # of nodes with g ≤ cost of optimal solution, O(bceiling(C*/ ε))
- Optimal? Yes – nodes expanded in increasing order of g(n)

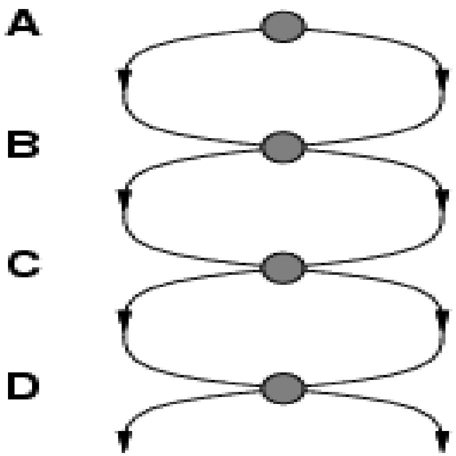
Comparison of Strategies

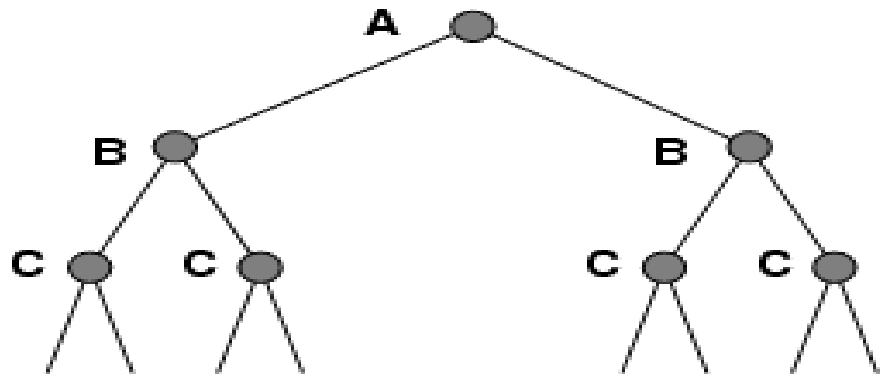
- Breadth-first is complete and optimal, but has high space complexity
- Depth-first is space efficient, but is neither complete, nor optimal
- Iterative deepening is complete and optimal, with the same space complexity as depth-first and almost the same time complexity as breadth-first

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Repeated States

- Failure to detect repeated states can turn a linear problem into an exponential one!





- Avoiding Revisited States: Requires comparing state descriptions.

  a. Initiate a new list namely known as closed or visited list
  b. Store all states associated with generated nodes in CLOSED.
  c. If the state of a new node is in CLOSED, then discard the node.

Graph search algorithm

function GRAPH-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

closed ← an empty set
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
loop do
  if *fringe* is empty **then return** failure
  node ← REMOVE-FRONT(*fringe*)
  if GOAL-TEST[*problem*](STATE[*node*]) **then return** SOLUTION(*node*)
  if STATE[*node*] is not in *closed* **then**
    add STATE[*node*] to *closed*
    fringe ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)

Simplest Graph Search Algorithm

- Create a search node from the initial state and save it in a list (called frontier)
- Create a loop until no items in frontier list
    a. Check if frontier list is empty (no solution)
    b. Select node from frontier list according to a strategy
    c. Check current node if it's the goal node (return solution - return the path)
    d. If current node not exists in visited list
        a. Expand current node
        b. Insert the current node (expanded one) in the visited list
        c. Insert the new nodes (childs) in the frontier

Strategy means which nodes from the frontier list must choose first in order to limit the search space (get quickly to the goal). There are many strategies like (LIFO, FIFO, and so on …)

Avoiding Loop back -expanding the same node several time- and that is the difference between tree search and graph search.

Simplest Graph Search Algorithm

- Demo: p:98 – p.194   GOTO: lect0
- Simulation: Python + Code
- Trade off: comparison
- Human decision. P:195

Thank you for your attention!

Questions?