# Artificial Intelligence
# ENCS 434

## **Uninformed Search**



Aziz M. Qaroush - Birzeit University

# Overview

- Search as Problem-Solving
  - problem formulation
  - problem types

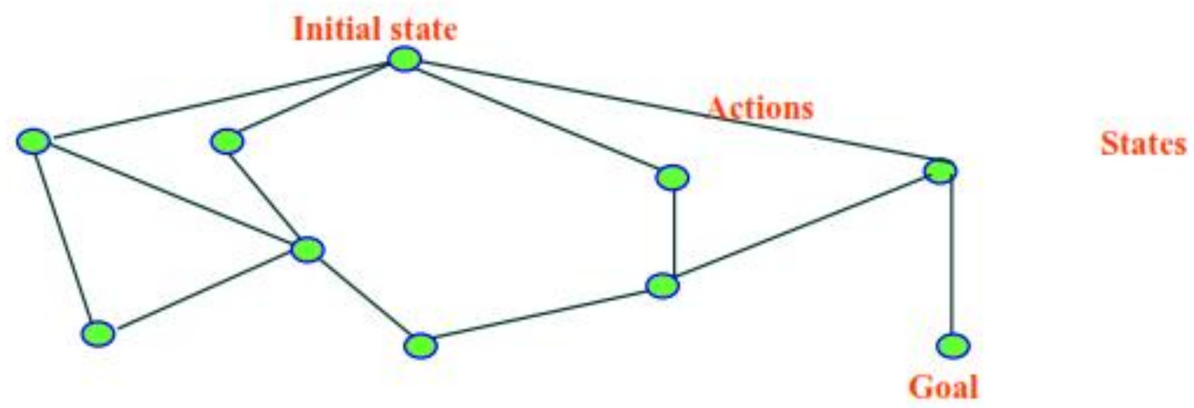
- Uninformed Search
  - breadth-first
  - depth-first
  - depth-limited search
  - iterative deepening
  - bi-directional search

# Problem-Solving Agents

- agents whose task it is to solve a particular problem
  - problem formulation
    - what are the possible states of the world relevant for solving the problem
    - what information is accessible to the agent
    - how can the agent progress from state to state
  - goal formulation
    - what is the goal state
    - what are important characteristics of the goal state
    - how does the agent know that it has reached the goal
    - are there several possible goal states
      - are they equal or are some more preferable

# Well-Defined Problems

- **Initial state:** The state the agent knows itself to be in. **(e.g., initial chessboard, current positions of objects in world, …)**
- **Action/Operator:** A set of actions that moves the problem from one state to another. (**e.g. chess move, robot action, simple change in location**).
- **Neighbourhood:** The set of all possible states reachable from a given state
- **State space:** The set of all states reachable from the initial state by any sequence of actions.
- **Path:** Any sequence of actions leading from one state to another.
- **Goal test:** A test applicable to a single state problem to determine if it is the goal state. (**e.g., winning chess position, target location**)
- **Path cost:** The function that assigns a cost to the path; **(e.g. How much it costs to take a particular path).**



# Selecting States and Actions

- states describe distinguishable stages during the problem-solving process
  - dependent on the task and domain
- actions move the agent from one state to another one by applying an operator to a state
  - dependent on states, capabilities of the agent, and properties of the environment
- choice of suitable states and operators
  - can make the difference between a problem that can or cannot be solved (in principle, or in practice)

# Example Problems

- There are two types of problems:

1. **Toy problems, which intended to illustrate various problem solving methods.**
   - State: The location of each of the eight tiles in one of the nine squares.
   - Operators: Blank moves left, right, up or down.
   - Goal test: State matches the goal configuration shown.
   - Path cost: Each step costs 1, so the path cost is just the length of the path.

| | | |
|---|---|---|
| 3 | 1 | |
| 7 | 6 | 2 |
| 5 | 8 | 4 |

**Start state**

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |

**Goal state**

- 8-puzzle has 362,800 states
- 15-puzzle has 10^12 states
- 24-puzzle has 10^25 states

- **Why search algorithms?**

**So, we need a principled way to look for a solution in these huge search spaces**


Aziz M. Qaroush - Birzeit University

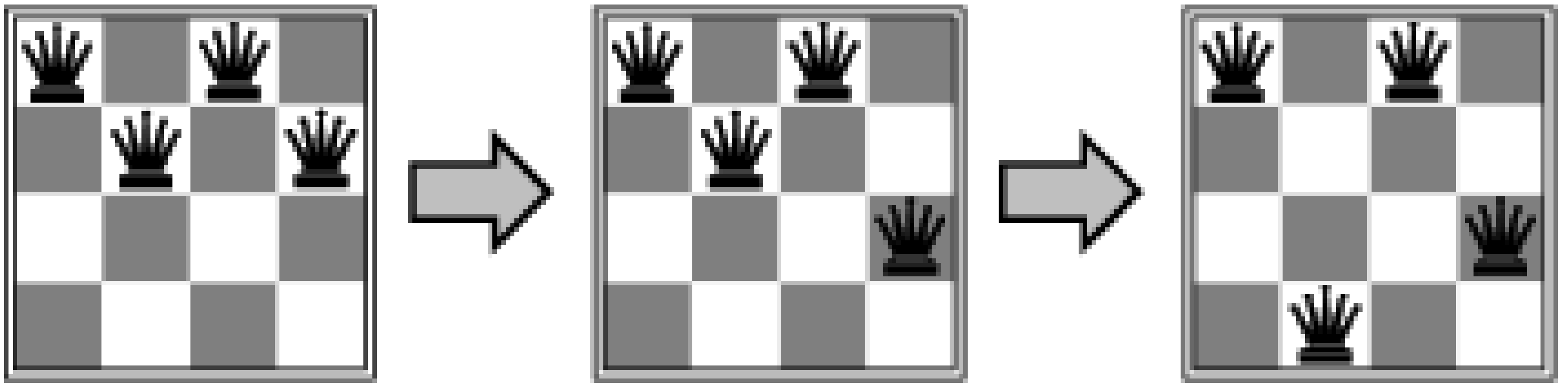
# Example Problems

2. Real world problems, which tend to be more difficult and whose solutions people actually care about

- **Route finding:** Route finding is defined in terms of specified locations and transitions along links between them. Its applications are (routing in computer networks, automated travel advisory system)
- **VLSI layout:** Positioning (transistors, resistors, capacitors, etc) and connections among a million gates in the Silicon chip is complicated and finding optimal way to place components on a printed circuit board so that:
  - Minimize surface area
  - Minimize number of signal layers
  - Minimize number of connections from one layer to another
  - Minimize length of some signal lines (e.g., clock line)
  - Distribute heat throughout board
- **Robot navigation:** Is similar to the route finding problem but in this case there is infinite set of possible actions and states.

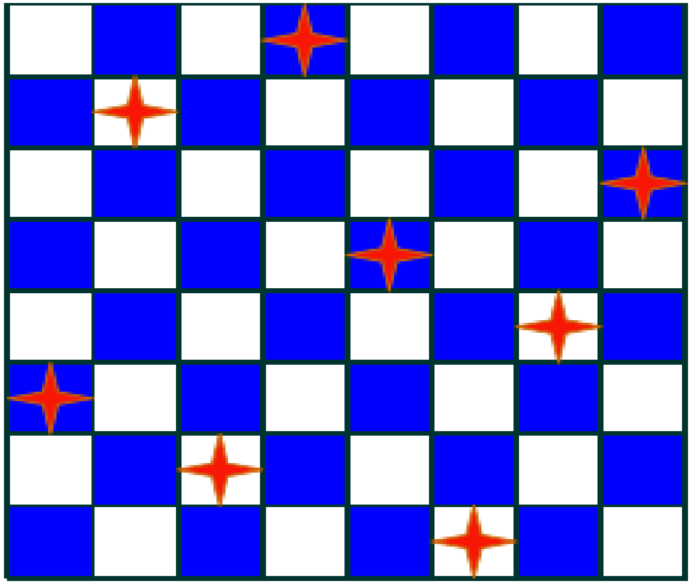
# *n*-Queens

- Put $n$ queens on an $n \times n$ board with no two queens on the same row, column, or diagonal



# 8-Queens

- incremental formulation
  - states
    - arrangement of up to 8 queens on the board
  - initial state
    - empty board
  - successor function (operators)
    - add a queen to any square
  - goal test
    - all queens on board
    - no queen attacked
  - path cost
    - irrelevant (all solutions equally valid)



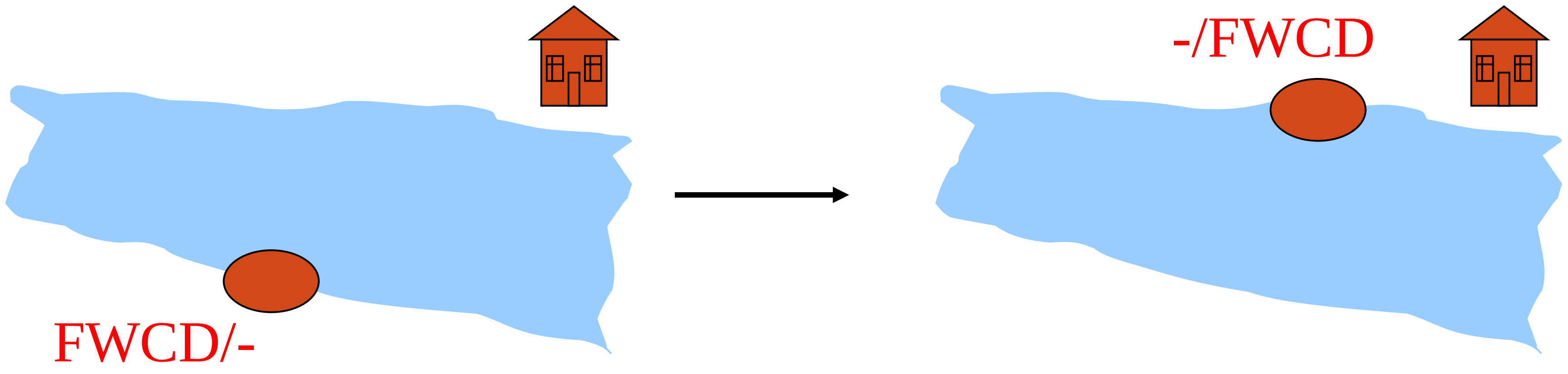
A solution

- Properties: $3*10^{14}$ possible sequences; can be reduced to 2,057

# The River Problem

- The River Problem:

**F=Farmer W=Wolf D=Duck C=Corn /=River**




How can the farmer safely transport the wolf, the duck and the corn to the opposite shore?

# The River Problem

- Problem formulation:
  - State representation: location of farmer and items in both sides of river
    [items in South shore / items in North shore] : (FWDC/-, FD/WC, C/FWD …)
  - Initial State: farmer, wolf, duck and corn in the south shore FWDC/-
  - Goal State: farmer, duck and corn in the north shore
    -/FWDC
  - Operators: the farmer takes in the boat at most one item from one side to the other side
    (F-Takes-W, F-Takes-D, F-Takes-C, F-Takes-Self [himself only])
  - Path cost: the number of crossings

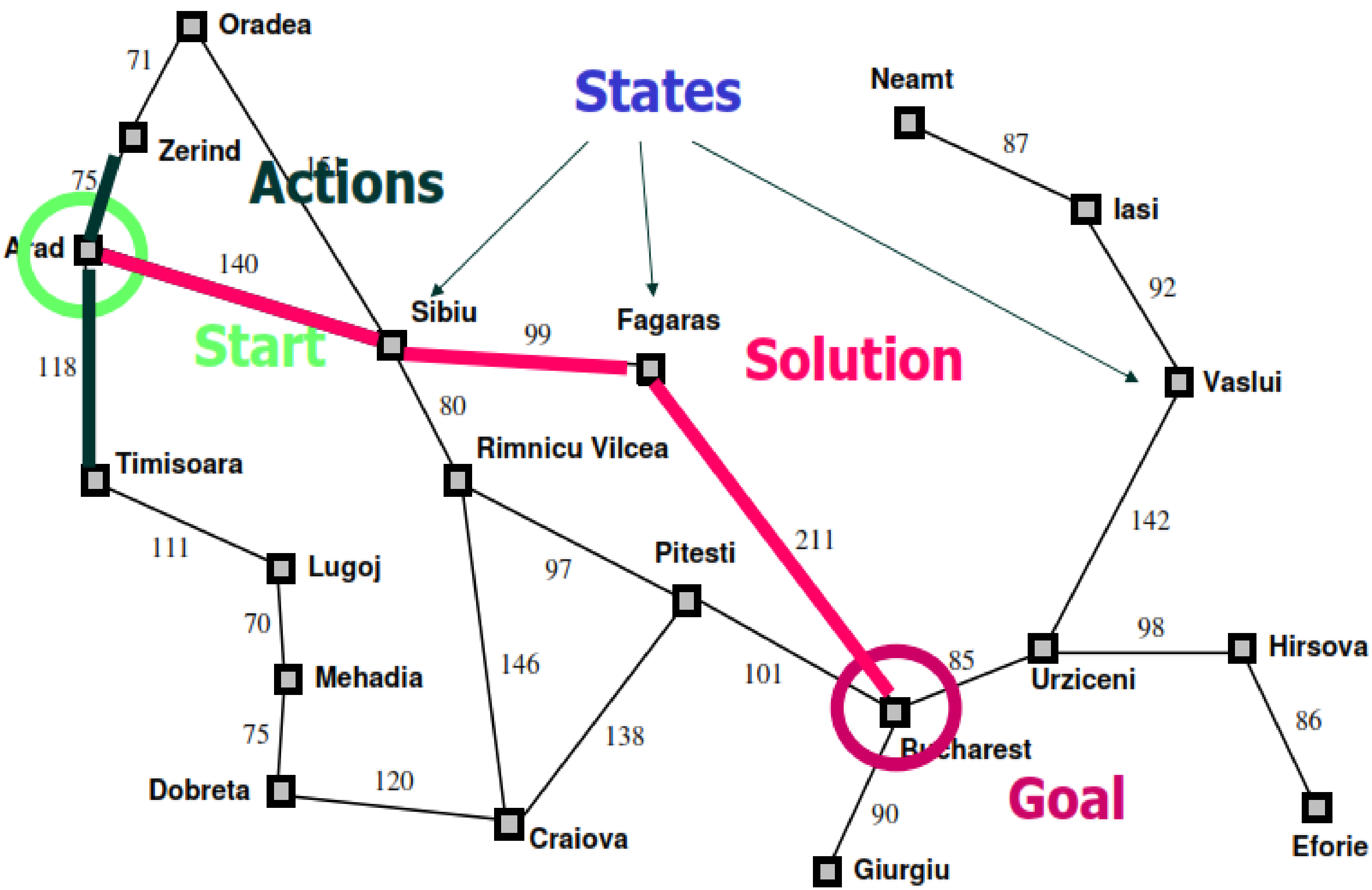
# Route Finding

- states
  - locations
- initial state
  - starting point
- successor function (operators)
  - move from one location to another
- goal test
  - arrive at a certain location
- path cost
  - may be quite complex
    - money, time, travel comfort, scenery, …

# Romania Map

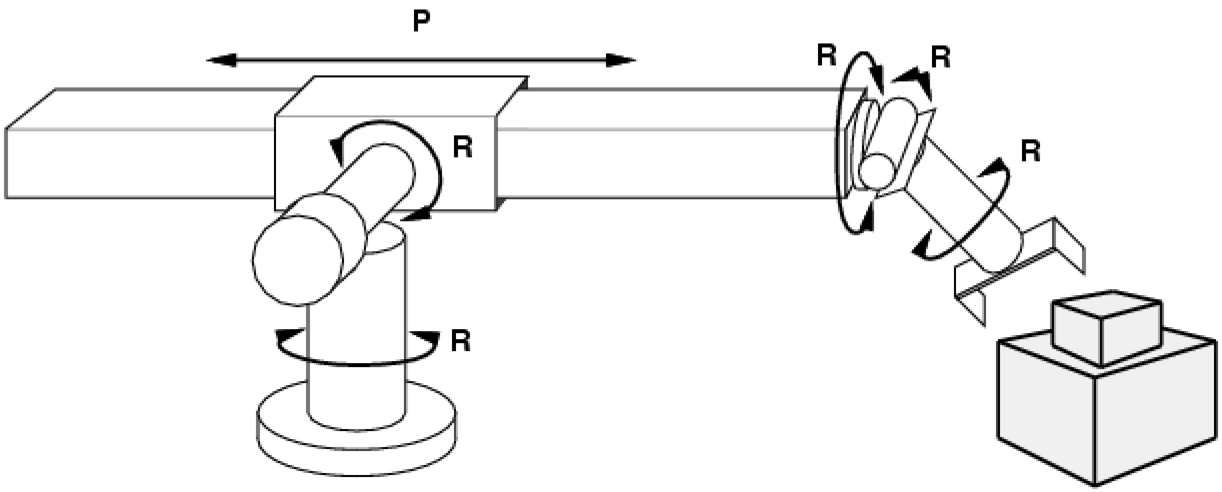
- **In Romania, on vacation, Currently in Arad.**
- **Flight leaves tomorrow from Bucharest.**
- **Formulate goal:**
  - Be in Bucharest
- **Formulate problem:**
  - States: various cities
- **Operators: drive between cities**
- **Find solution:**
  - Sequence of cities, such that total
  - driving distance is minimized,
  - e.g. Arad, Sibiu, Fagaras, Bucharest.
- **Finding shortest path**
  - Action: Move from city X to city Y
  - State: Which city you're on
  - Goal Test: Am I in Bucharest?
  - Cost: 1 for each city I visit

# Romania Map



Oradea
71
Zerind
75
Arad
140
118
Timisoara
111
Lugoj
70
Mehadia
75
Dobreta
120
Craiova
Sibiu
99
Fagaras
80
Rimnicu Vilcea
97
Pitesti
146
138
101
211
Bucharest
90
Giurgiu
85
Urziceni
98
Hirsova
86
Eforie
142
Vaslui
92
Iasi
87
Neamt
States
Actions
Start
Solution
Goal

# Robotic assembly




- states?: real-valued coordinates of robot joint angles parts of the object to be assembled
- actions?: continuous motions of robot joints
- goal test?: complete assembly
- path cost?: time to execute

# Searching for Solutions

- traversal of the search space
  - from the initial state to a goal state
  - legal sequence of actions as defined by successor function (operators)
- general procedure
  - check for goal state
  - expand the current state
    - determine the set of reachable states
    - return "failure" if the set is empty
  - select one from the set of reachable states
  - move to the selected state
- a search tree is generated
  - nodes are added as more states are visited

# Search Terminology

- search tree
  - generated as the search space is traversed
    - the search space itself is not necessarily a tree, frequently it is a graph
    - the tree specifies possible paths through the search space
  - expansion of nodes
    - as states are explored, the corresponding nodes are expanded by applying the successor function
      - this generates a new set of (child) nodes
    - the fringe (frontier) is the set of nodes not yet visited
      - newly generated nodes are added to the fringe
  - search strategy
    - determines the selection of the next node to be expanded
    - can be achieved by ordering the nodes in the fringe
      - e.g. queue (FIFO), stack (LIFO), "best" node w.r.t. some measure (cost)

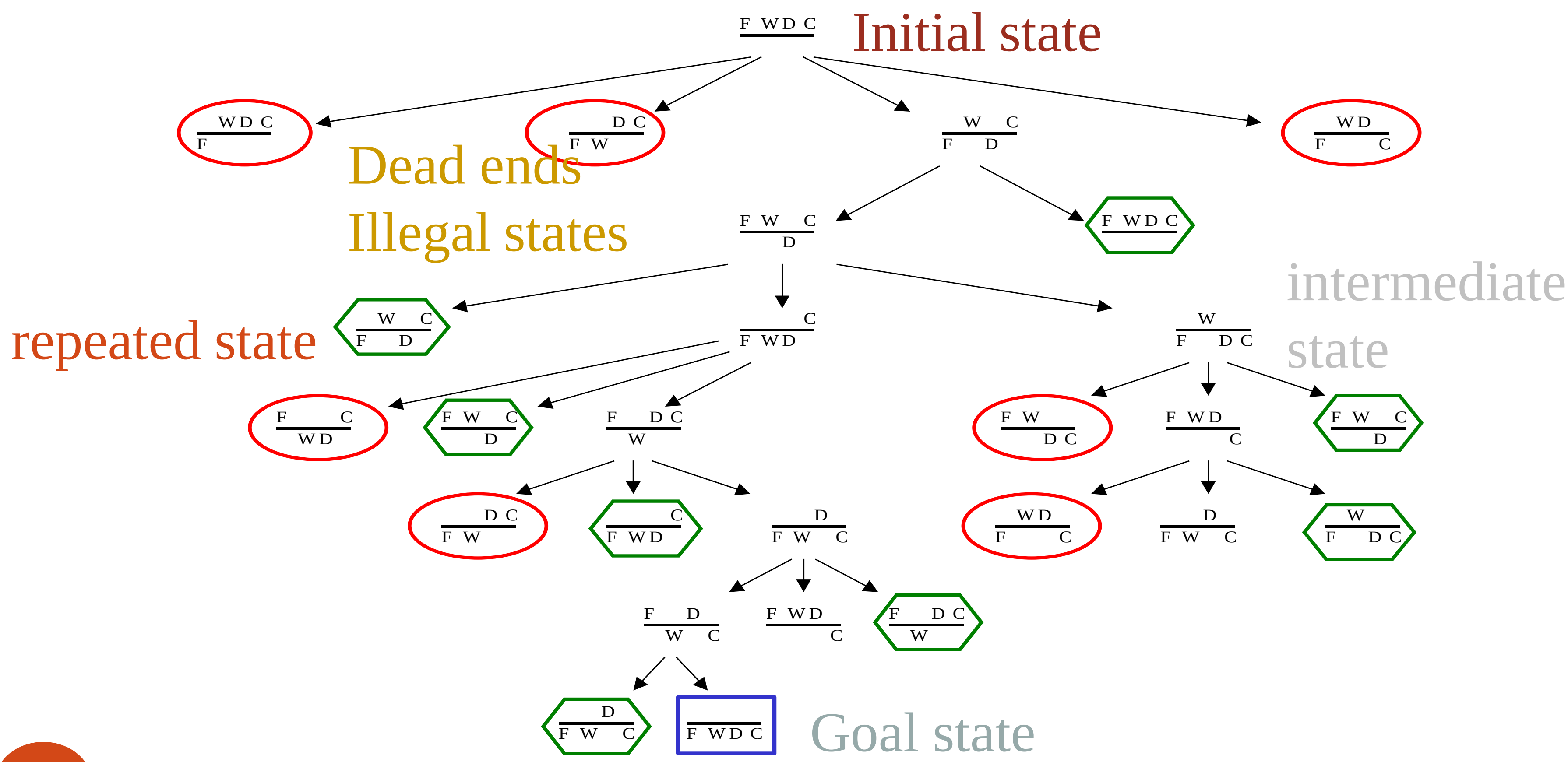
# Search Methods - State space:

- **A problem is solved by moving from the initial state to the goal state by applying valid operators in sequence.**

- **Thus the state space is the set of states reachable from a particular initial state.**

# State space – Example 1

## The River Problem





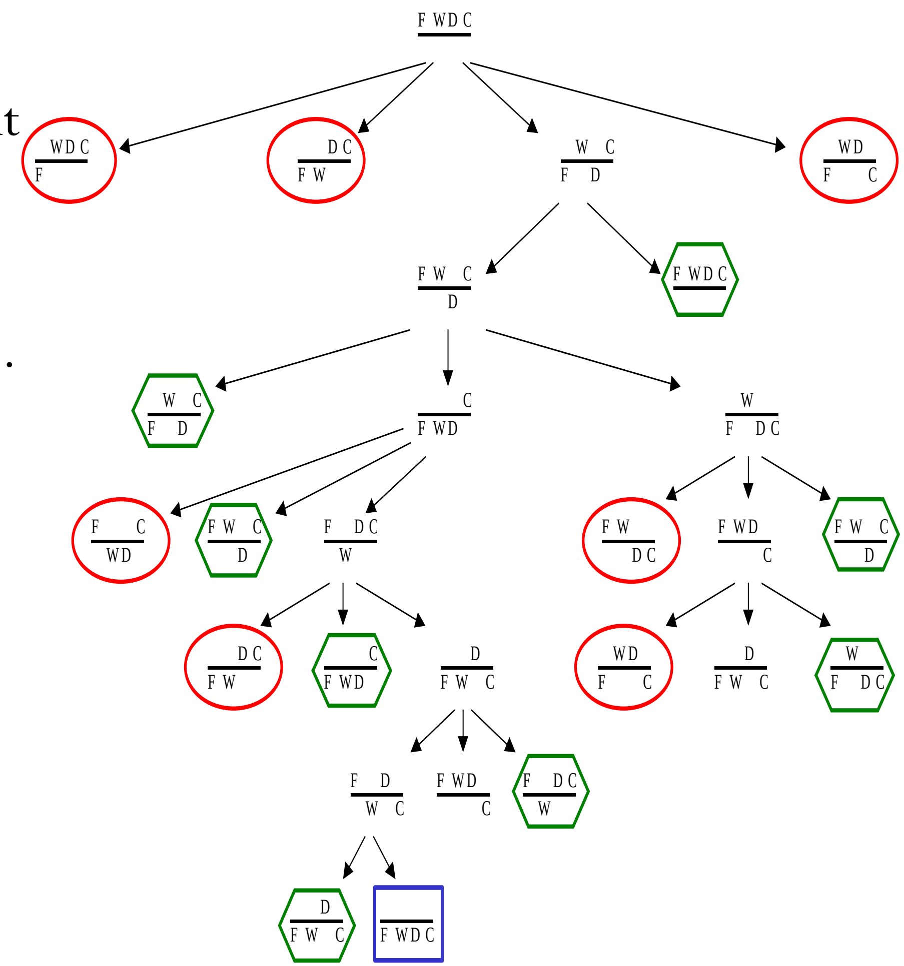

Aziz M. Qaroush - Birzeit University

# State space – Example 1

- Searching for a solution:

  We start with the initial state and keep using the operators to expand the parent nodes till we find a goal state.

- …but the search space might be large…
- …really large…
- So we need some systematic way to search.



# State space – Example 1

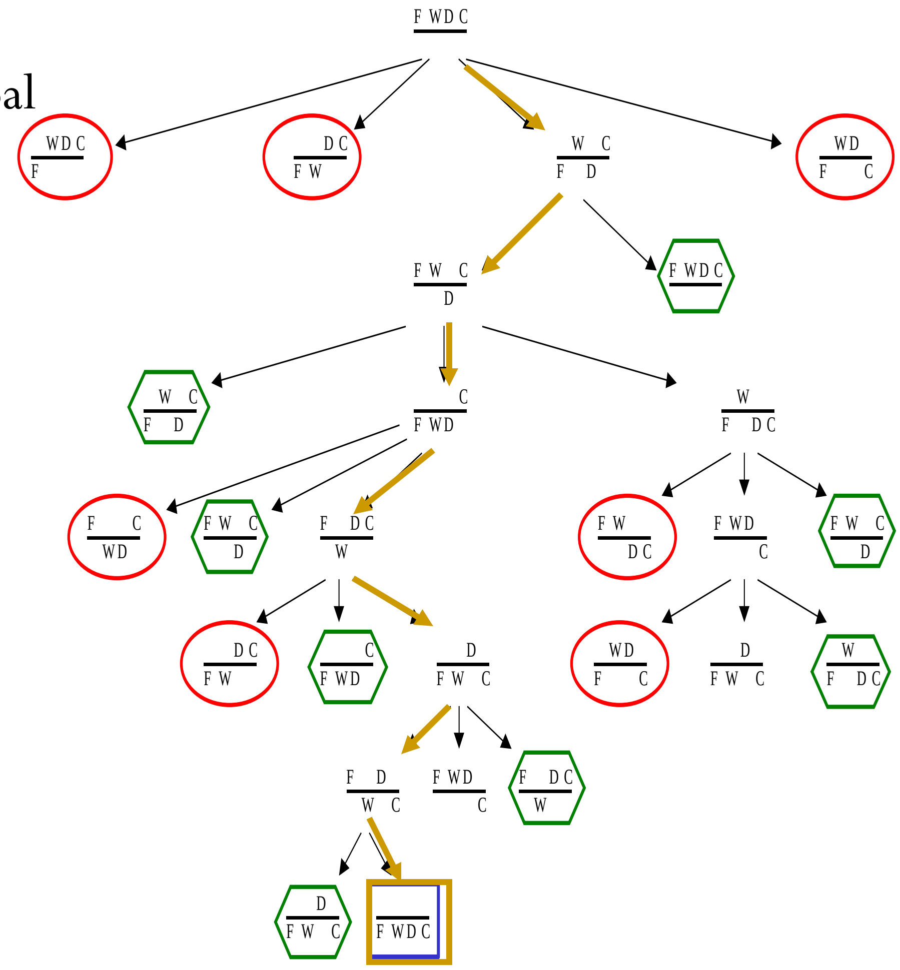
- Problem solution:

  A problem solution is simply the set of operators (actions) needed to reach the goal state from the initial state:

  F-Takes-D, F-Takes-Self, F-Takes-W,

  F-Takes-D, F-Takes-C, F-Takes-Self,

  F-Takes-D.



# State space – Example 2

- A hungry monkey is in a room.
- Bananas have been hung from the center of the ceiling of the room.
- In the corner of the room there is a chair.
- The monkey wants the bananas but he can't reach them.
- What shall he do?



Aziz M. Qaroush - Birzeit University

# State space – Example 2

If the monkey is cleaver enough, he can reach the bananas by placing the chair directly below the bananas and climbing on the top of the chair.

# State space – Example 2

The state space (w, x, y, z)

w – monkey coordinates on the floor

x = {1, if monkey is on the chair; 0, otherwise}

y – coordinates of the chair on the floor

z = {1, if the monkey knocked the bananas down; 0, otherwise}

Production rules (or operators):

1. move (u): (w, 0, y, z) —> (u, 0, y, z)

2. carry (v): (w, 0, w, z) —> (v, 0, v, z)

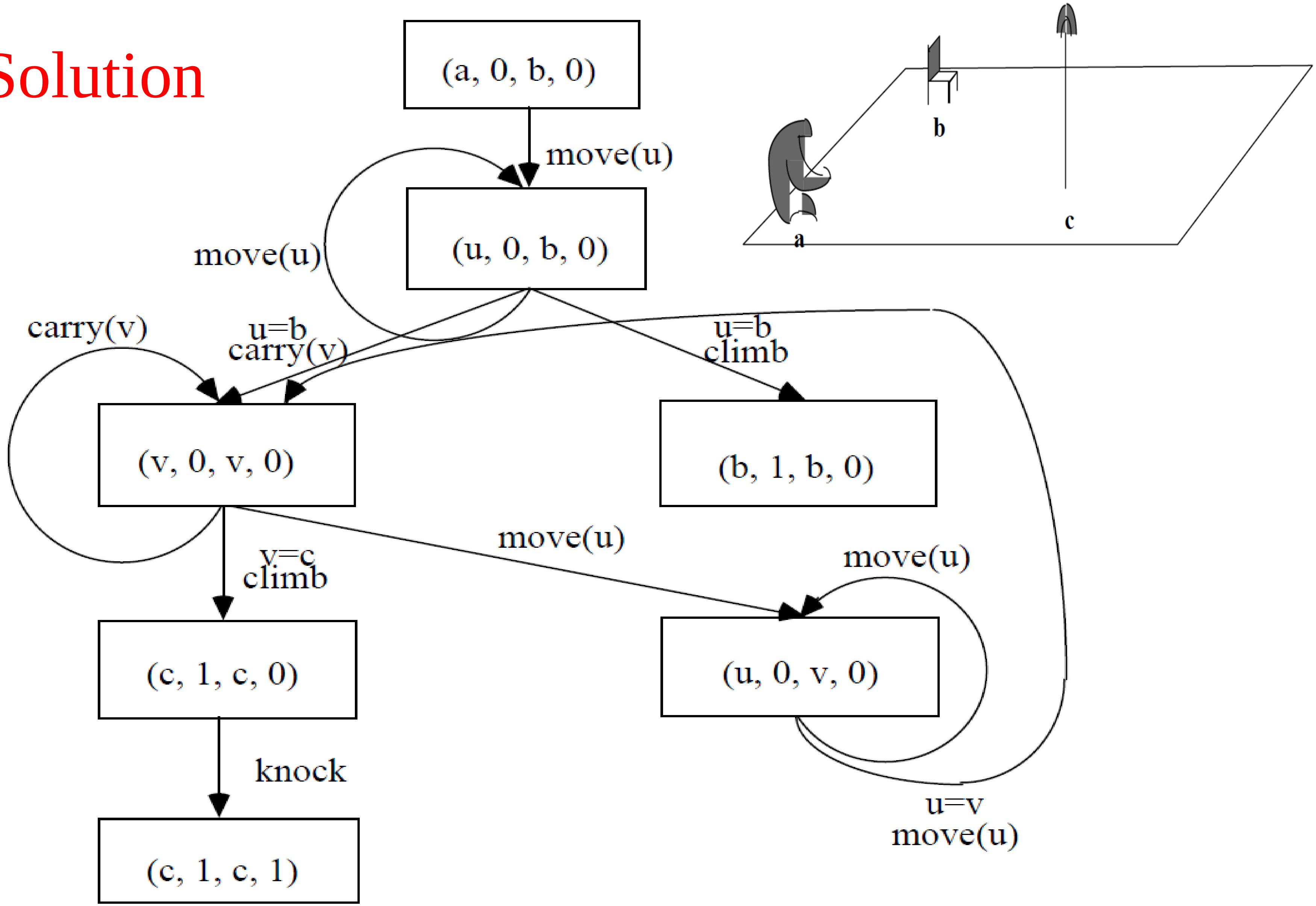
3. climb : (v, 0, v, z) —> (v, 1, v, z)

4. knock : (c, 1, c, 0) —> (c, 1, c, 1)

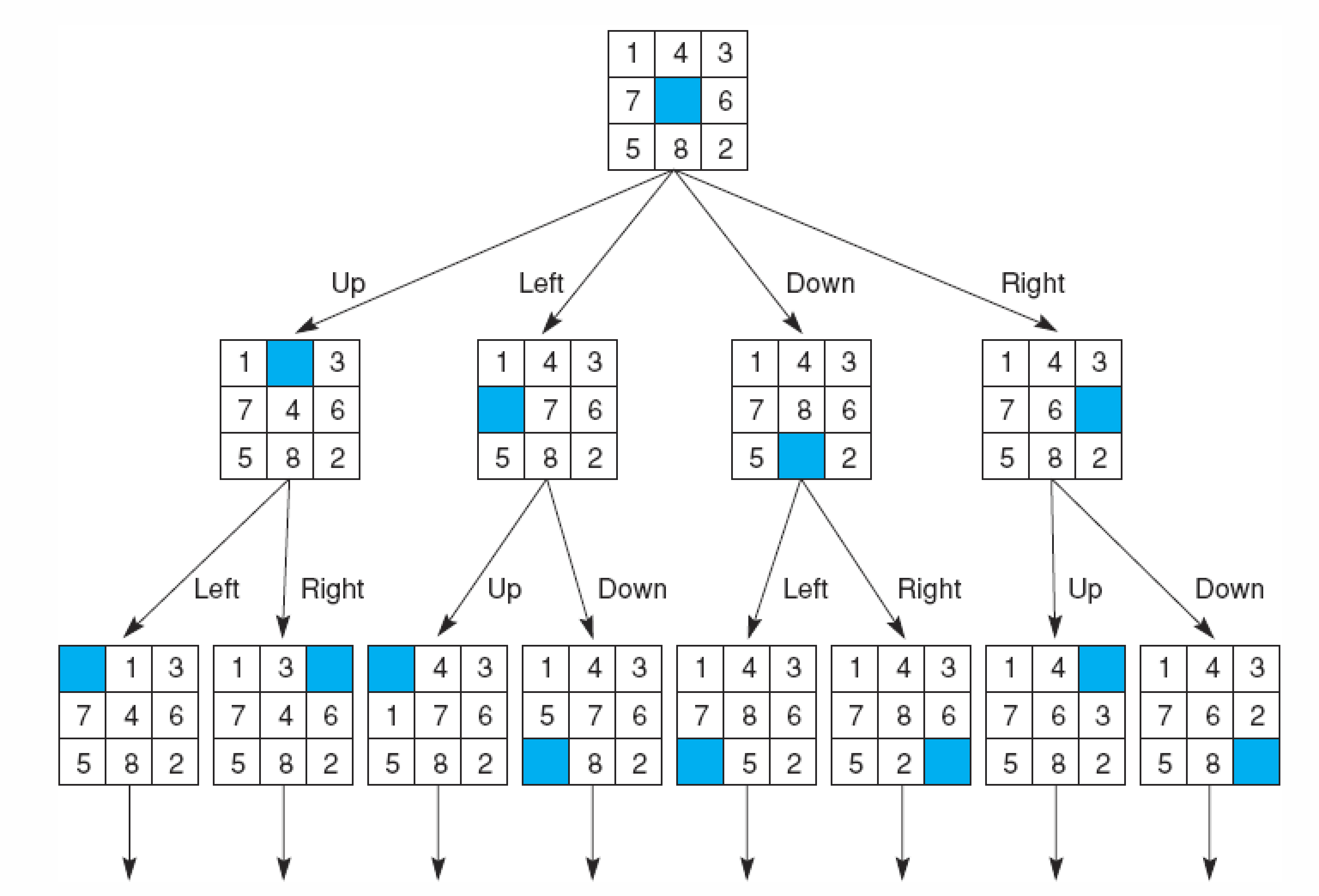
(c - coordinates of bananas in horizontal plane)

# State space – Example 2

## Solution



# State space – Example 3




The 8 puzzle search space consists of 8! states (40320)

# Generic Search Algorithms

- Basic Idea: Off-line exploration of state space by generating successors of already-explored states (also known as *expanding states).*

**Function** GENERAL-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** node contains goal state **then return** *solution*
**else** expand node and add resulting nodes to search tree.
**end**

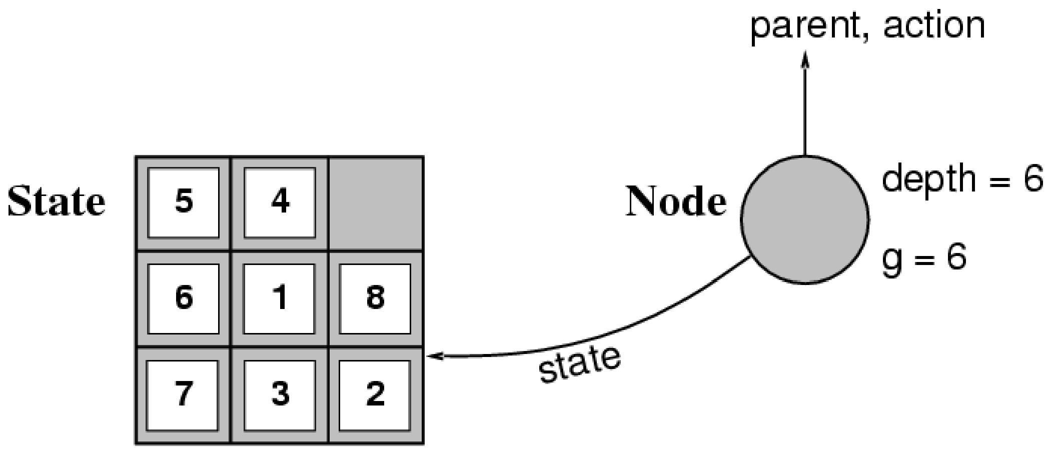
# Implementation of Generic Search Algorithm

```
function general-search(problem, QUEUEING-FUNCTION)
nodes = MAKE-QUEUE(MAKE-NODE(problem.INITIAL-STATE))
loop do
    if EMPTY(nodes) then return "failure"
     node = REMOVE-FRONT(nodes)
    if problem.GOAL-TEST(node.STATE) succeeds then return solution(node)
      nodes = QUEUEING-FUNCTION(nodes, EXPAND(node,
   problem.OPERATORS))
end
```

A nice fact about this search algorithm is that we can use a single algorithm to do many kinds of search. The only difference is in how the nodes are placed in the queue. ***The choice of queuing function is the main feature.***

# Implementation: states vs. nodes

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost g(x), depth




- The `Expand` function creates new nodes, filling in the various fields and using the `SuccessorFn` of the problem to create the corresponding states.

# Search Strategies

- Uninformed (blind) search strategies: There is no information about the number of steps or the path cost from the current state to the goal. All they can do is distinguish a goal state from a non-goal.
  - Breadth-first search
  - Uniform cost search
  - Depth-first search
  - Depth limited search
  - Iterative deepening search
  - BI-directional Search

**Ch. 3**

- Informed Search Strategies: There is an information about the path cost.
- **When strategies can determine whether one non-goal state is better than another → informed search.**
  - Best-first search
  - Greedy best-first search
  - $A^*$ search
  - Hill-climbing search
  - Simulated annealing search
  - Local beam search
  - Genetic algorithms

**Ch. 4**

# What Criteria are used to Compare different search techniques ?

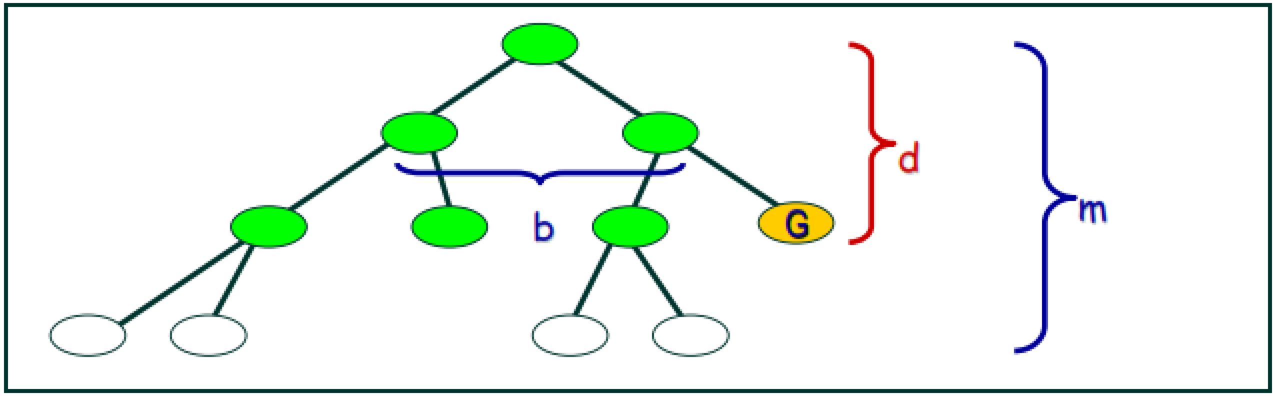
As we are going to consider different techniques to search the problem space, we need to consider what criteria we will use to compare them.

- **Completeness**: Is the technique guaranteed to find an answer (if there is one).
- **Optimality/Admissibility** : does it always find a least-cost solution?
  - an admissible algorithm will find a solution with minimum cost
- **Time Complexity**: How long does it take to find a solution.
- **Space Complexity**: How much memory does it take to find a solution.

# Time and Space Complexity ?

Time and space complexity are measured in terms of:

- The average number of new nodes we create when expanding a new node is the (effective) branching factor **b**.
- The (maximum) branching factor **b** is defined as the maximum nodes created when a new node is expanded.
- The length of a path to a goal is the depth **d**.
- The maximum length of any path in the state space **m**.



# Search Cost and Path Cost

- the *search cost* indicates how expensive it is to generate a solution
  - time complexity (e.g. number of nodes generated) is usually the main factor
  - sometimes space complexity (memory usage) is considered as well
- *path cost* indicates how expensive it is to execute the solution found in the search
  - distinct from the search cost, but often related
- *total cost* is the sum of search cost and path costs

# Breadth-First Search

- all the nodes reachable from the current node are explored first
  - achieved by the TREE-SEARCH method by appending newly generated nodes at the end of the search queue

**function** BREADTH-FIRST-SEARCH(*problem*) **returns** *solution*

    **return** TREE-SEARCH(*problem*, FIFO-QUEUE())

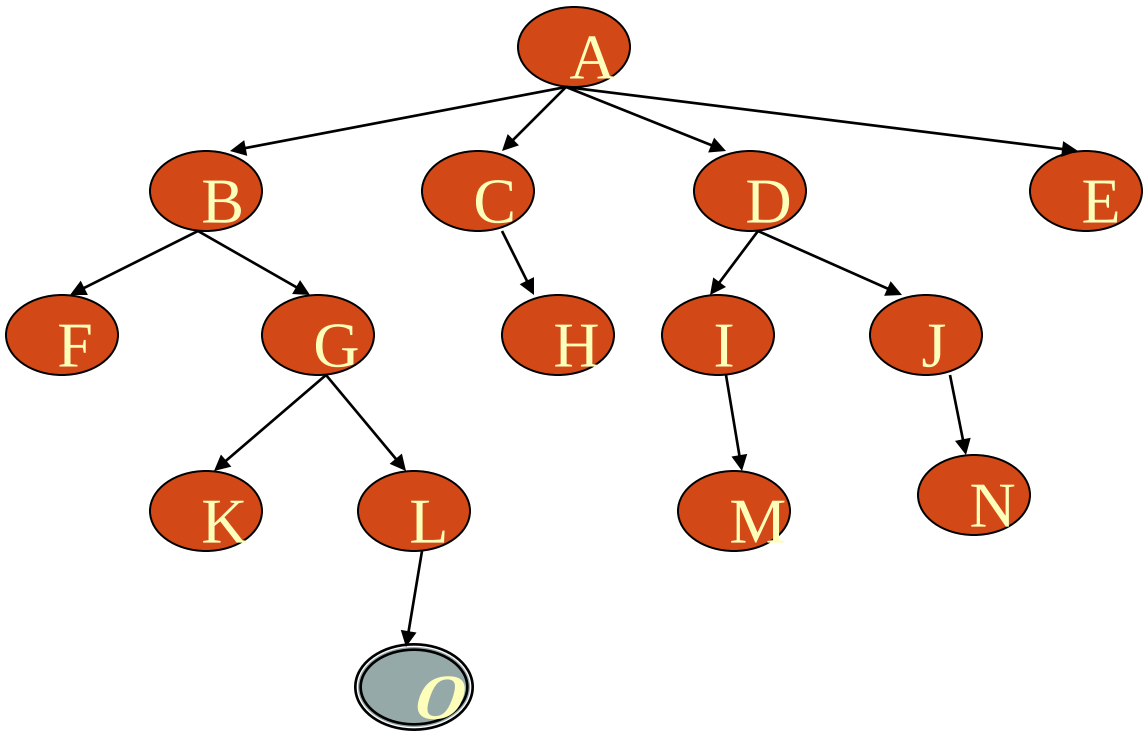
| | |
|---|---|
| Time Complexity | $b^{d+1}$ |
| Space Complexity | $b^{d+1}$ |
| Completeness | yes (for finite b) |
| Optimality | yes (for non-negative path costs) |

| | |
|---|---|
| b | branching factor |
| d | depth of the tree |

# Breadth First Search

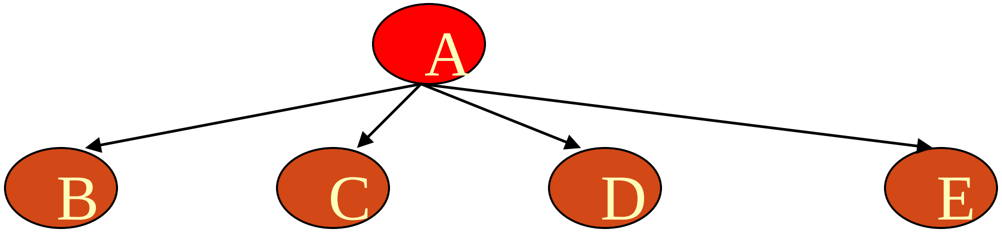
- Application1:

  Given the following state space (tree search), give the sequence of visited nodes when using BFS (assume that the node ***O*** is the goal state):



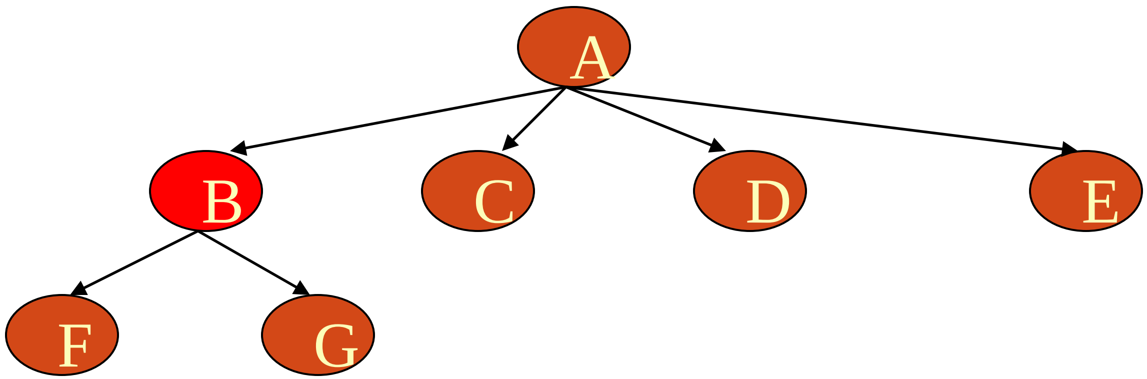
# Breadth First Search

- A,



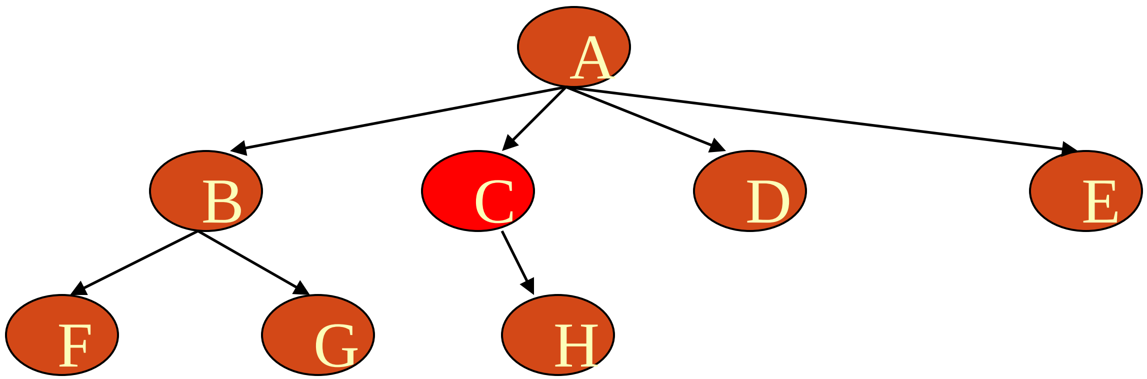
# Breadth First Search

- A,
- B,



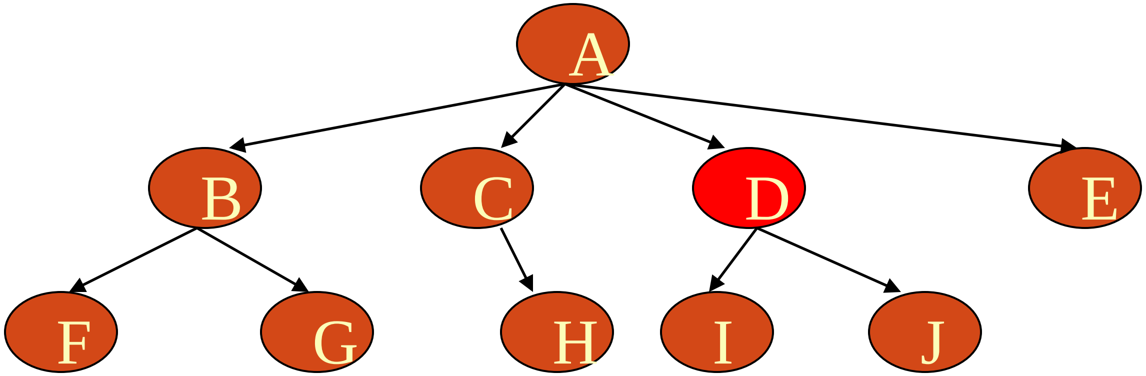
# Breadth First Search

- A,
- B,C



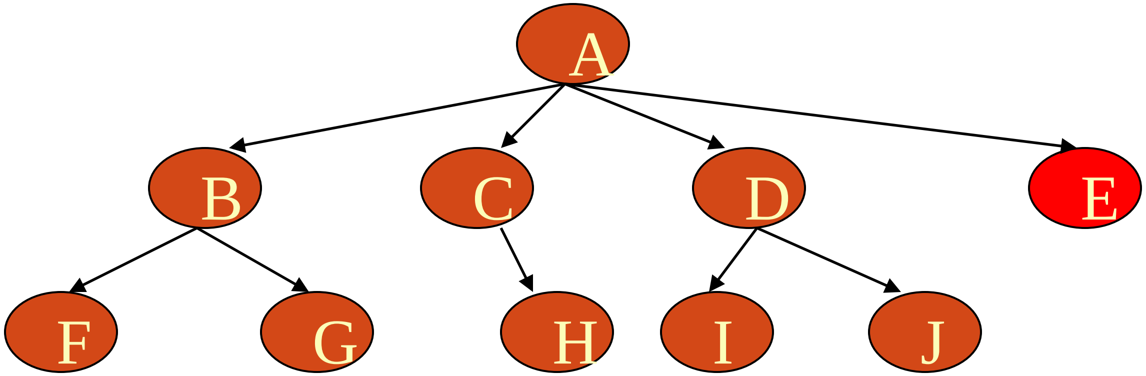
# Breadth First Search

- A,
- B,C,D



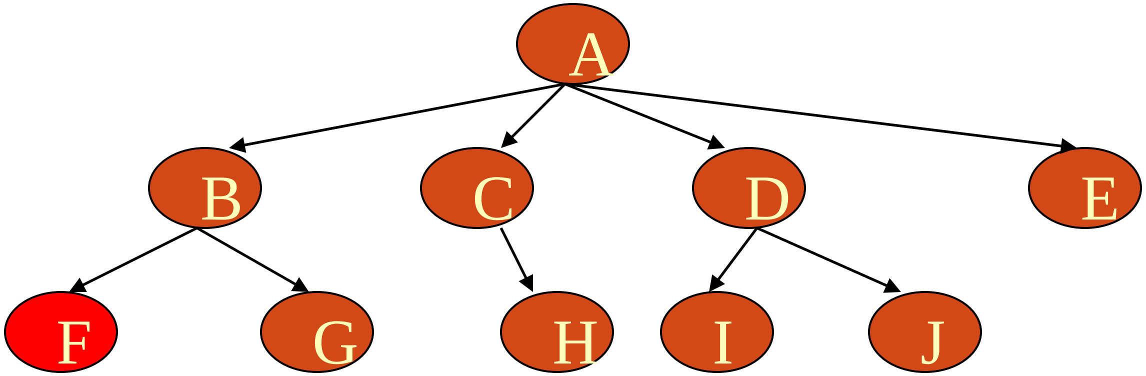
# Breadth First Search

- A,
- B,C,D,E



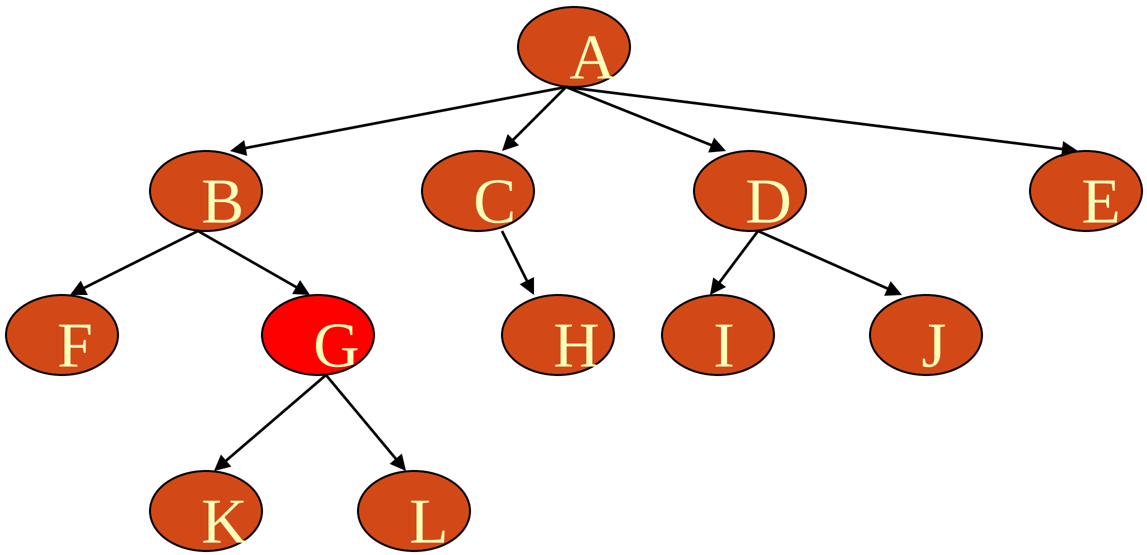
# Breadth First Search

- A,
- B,C,D,E,
- F,



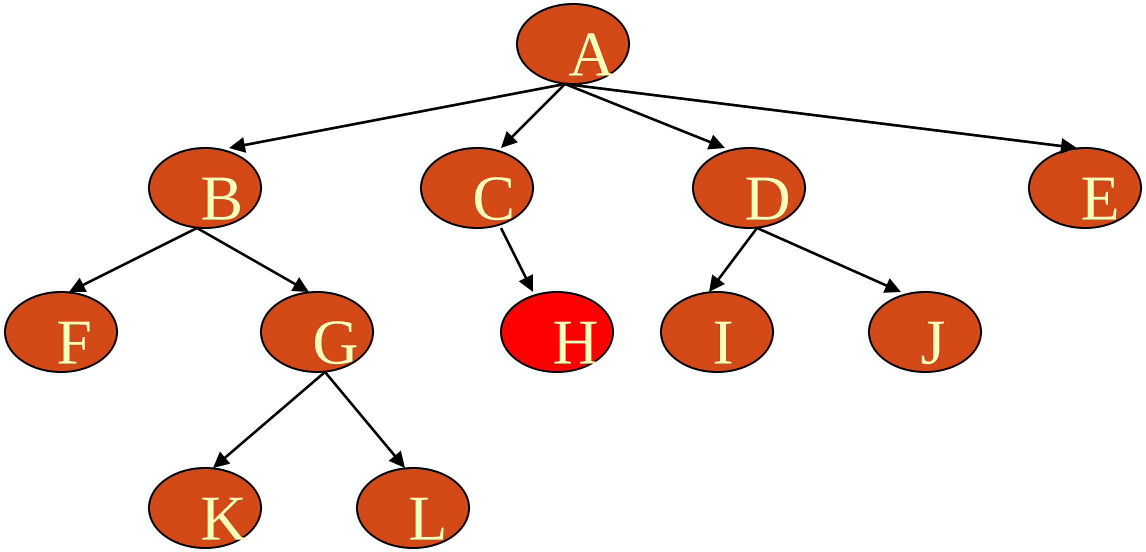
# Breadth First Search

- A,
- B,C,D,E,
- F,G



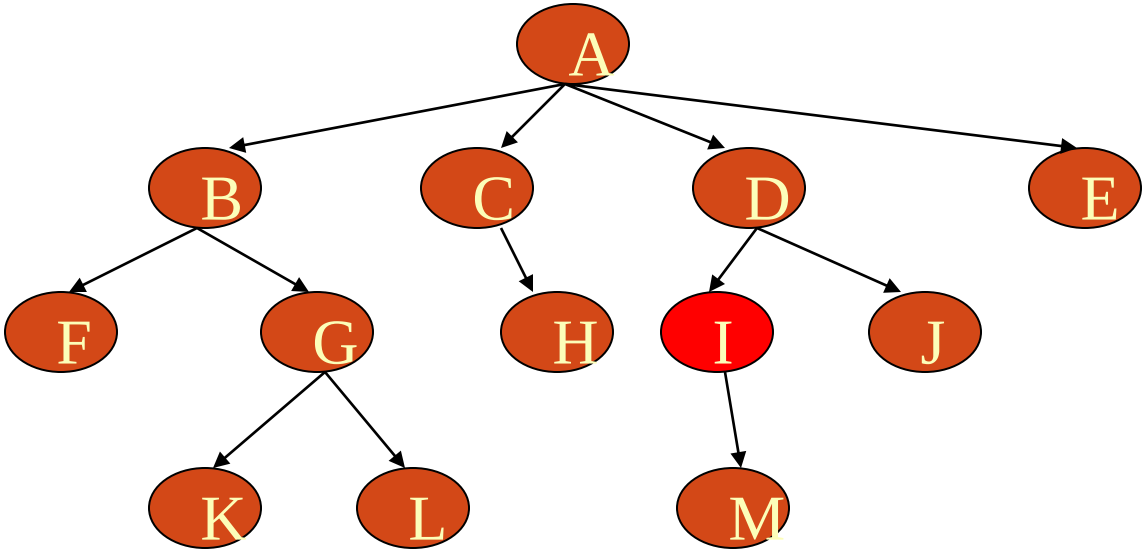
# Breadth First Search

- A,
- B,C,D,E,
- F,G,H



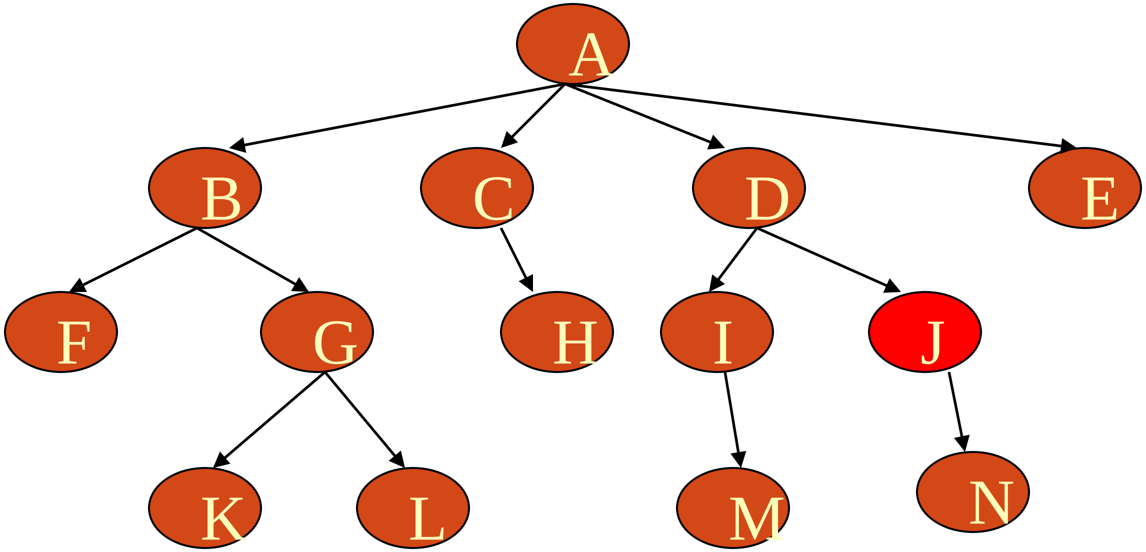
# Breadth First Search

- A,
- B,C,D,E,
- F,G,H,I



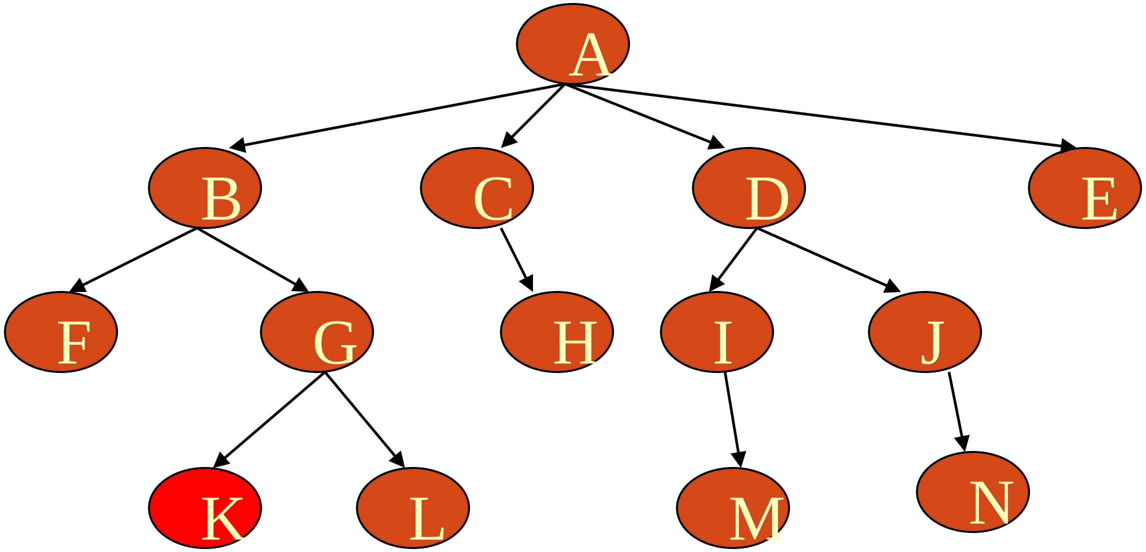
# Breadth First Search

- A,
- B,C,D,E,
- F,G,H,I,J,



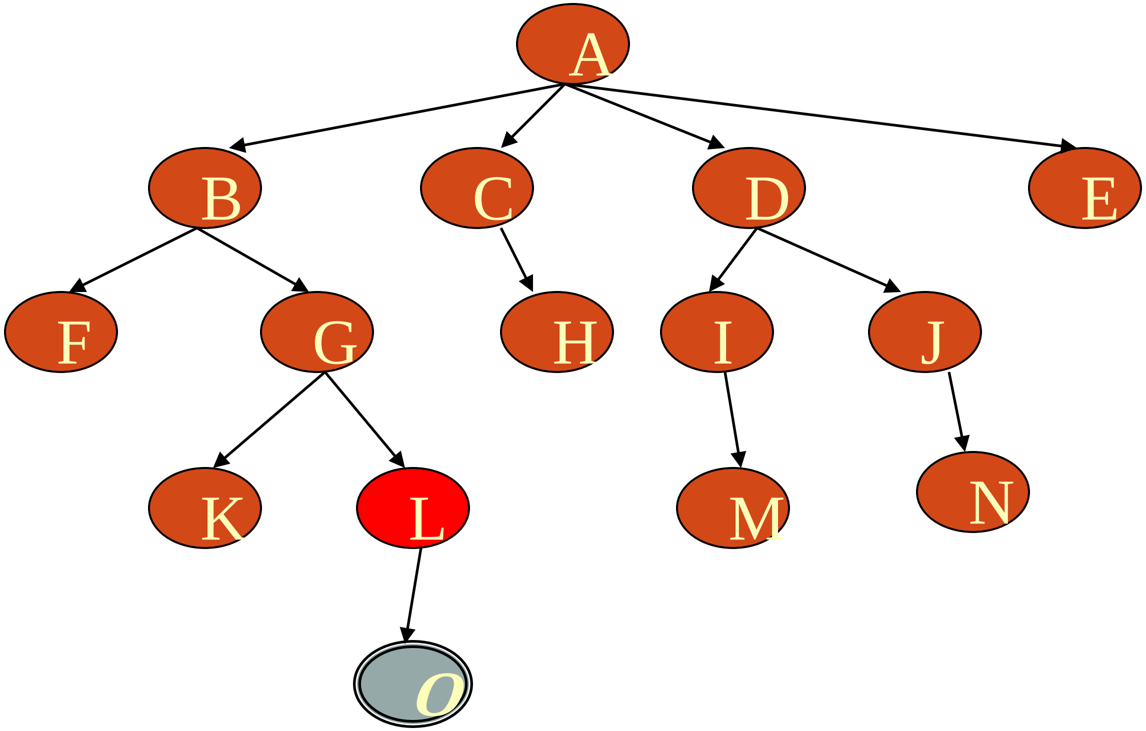
# Breadth First Search

- A,
- B,C,D,E,
- F,G,H,I,J,
- K,



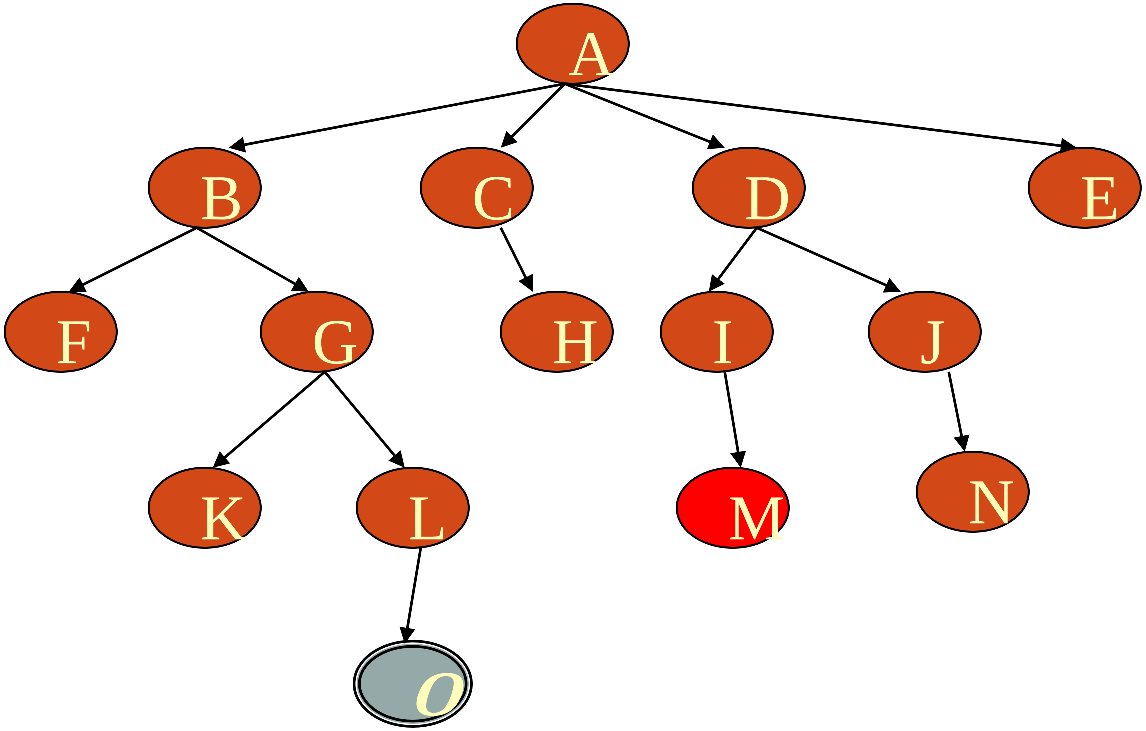
# Breadth First Search

- A,
- B,C,D,E,
- F,G,H,I,J,
- K,L



# Breadth First Search

- A,
- B,C,D,E,
- F,G,H,I,J,
- K,L, M,

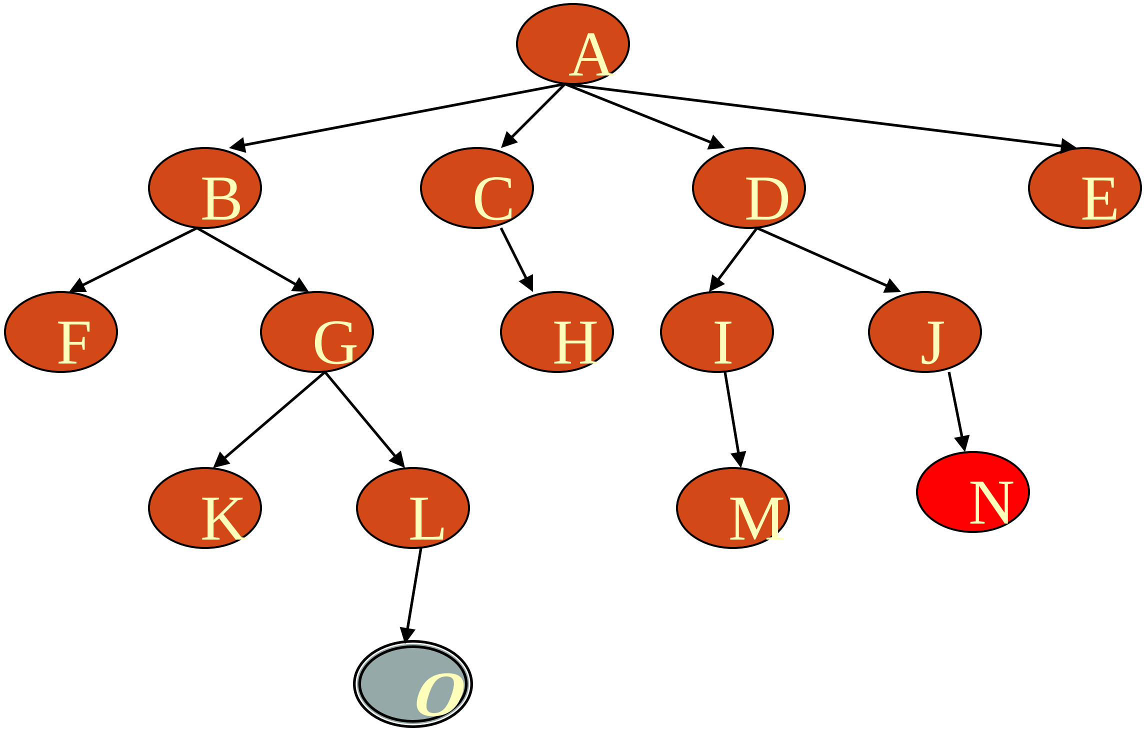





Aziz M. Qaroush - Birzeit University

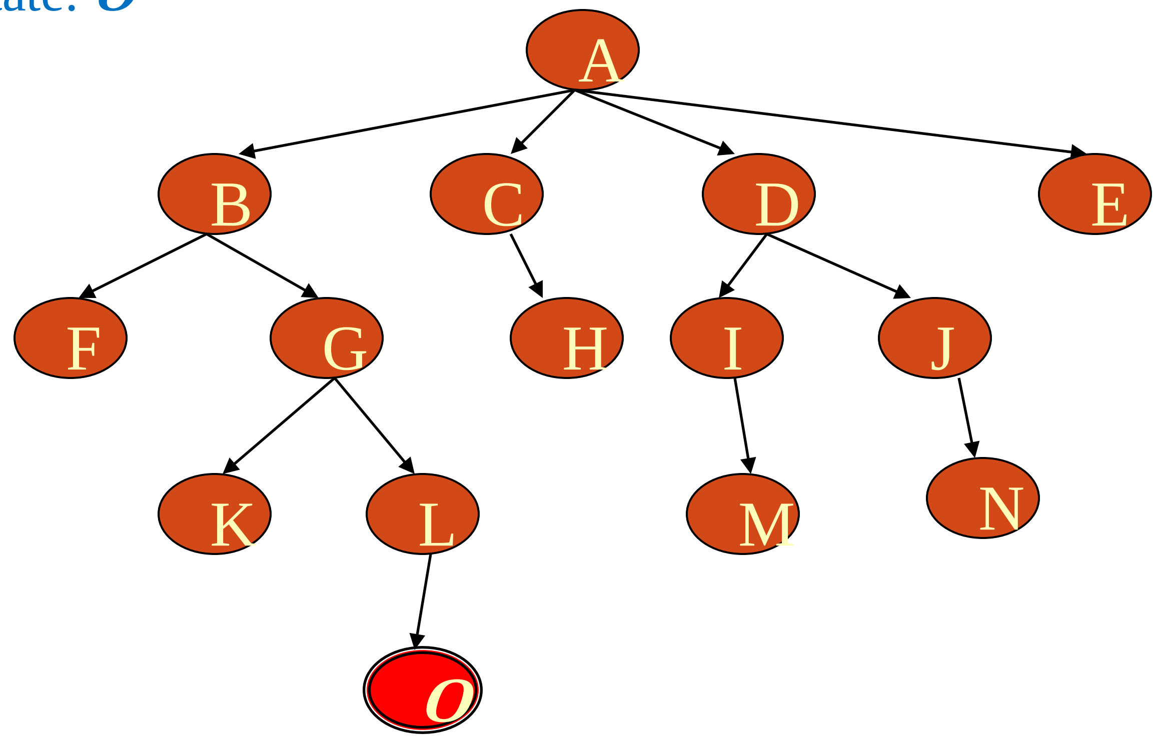
# Breadth First Search

- A,
- B,C,D,E,
- F,G,H,I,J,
- K,L, M,N,



# Breadth First Search

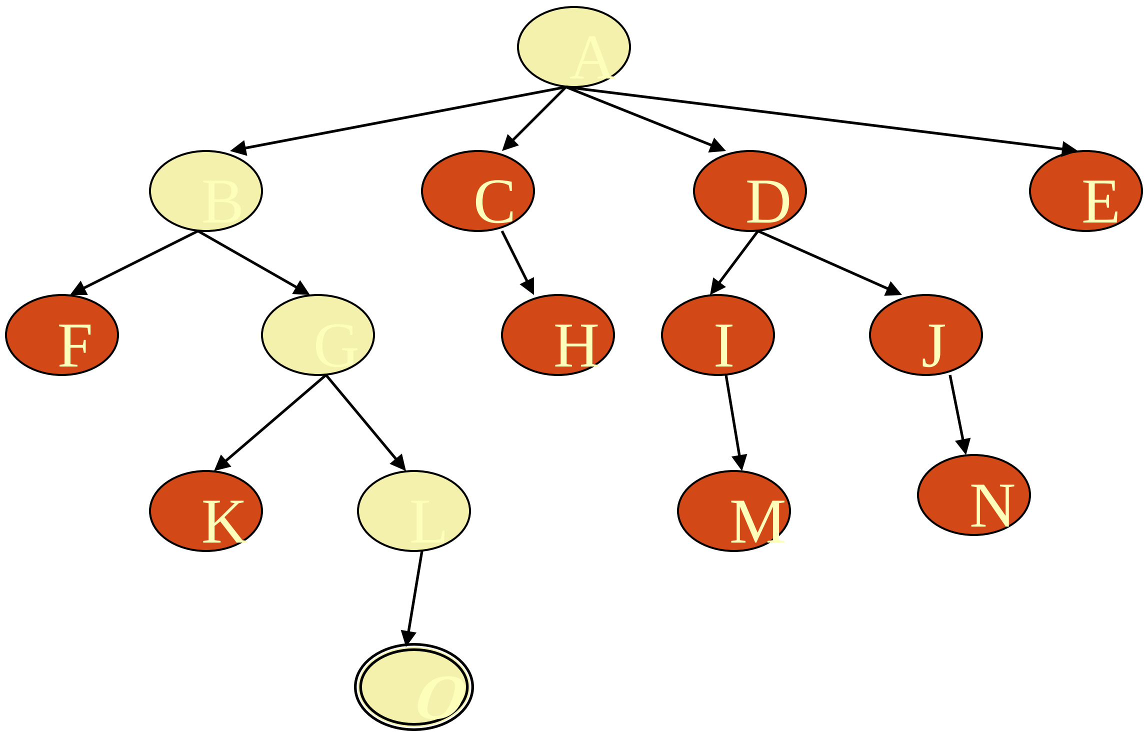
- A,
- B,C,D,E,
- F,G,H,I,J,
- K,L, M,N,
- Goal state: ***O***



# Breadth First Search

- The returned solution is the sequence of operators in the path:

  ***A, B, G, L, O***





Aziz M. Qaroush - Birzeit University

# Breadth-First Search: Evaluation

- **Completeness:**
  - Does it always find a solution if one exists?
  - YES
    - If shallowest goal node is at some finite depth d
    - Condition: If b is finite (maximum num. of succor nodes is finite)
- **Completeness:**
  - YES (if b is finite)
- **Time complexity:**
  - Assume a state space where every state has b successors.
    - Root has b successors, each node at the next level has again b successors (total $b^2$),
    - Assume solution is at depth d
    - Worst case; expand all but the last node at depth d
    - Total numb. of nodes generated:
    - $1 + b + b^2 + \ldots + b^d + b(b^d-1) = O(b^{d+1})$
- **Space complexity:** $O(b^{d+1})$
- **Optimality:**
  - Does it always find the least-cost solution?
  - In general **YES**
    - Unless actions have different cost.

# Breadth-First Search: Evaluation

- **lessons:**
  - Memory requirements are a bigger problem than execution time.
  - Exponential complexity search problems cannot be solved by uninformed search methods for any but the smallest instances.

| DEPTH | NODES | TIME | MEMORY |
|---|---|---|---|
| **2** | **1100** | **0.11 seconds** | **1 megabyte** |
| **4** | **111100** | **11 seconds** | **106 megabytes** |
| **6** | **$10^7$** | **19 minutes** | **10 gigabytes** |
| **8** | **$10^9$** | **31 hours** | **1 terabyte** |
| **10** | **$10^{11}$** | **129 days** | **101 terabytes** |
| **12** | **$10^{13}$** | **35 years** | **10 petabytes** |
| **14** | **$10^{15}$** | **3523 years** | **1 exabyte** |

Assumptions: $b = 10$; 10,000 nodes/sec; 1000 bytes/node


Aziz M. Qaroush - Birzeit University

# Uniform-Cost -First

- the nodes with the lowest cost are explored first
  - similar to BREADTH-FIRST, but with an evaluation of the cost for each reachable node
  - *g(n)* = path cost(n) = sum of individual edge costs to reach the current node

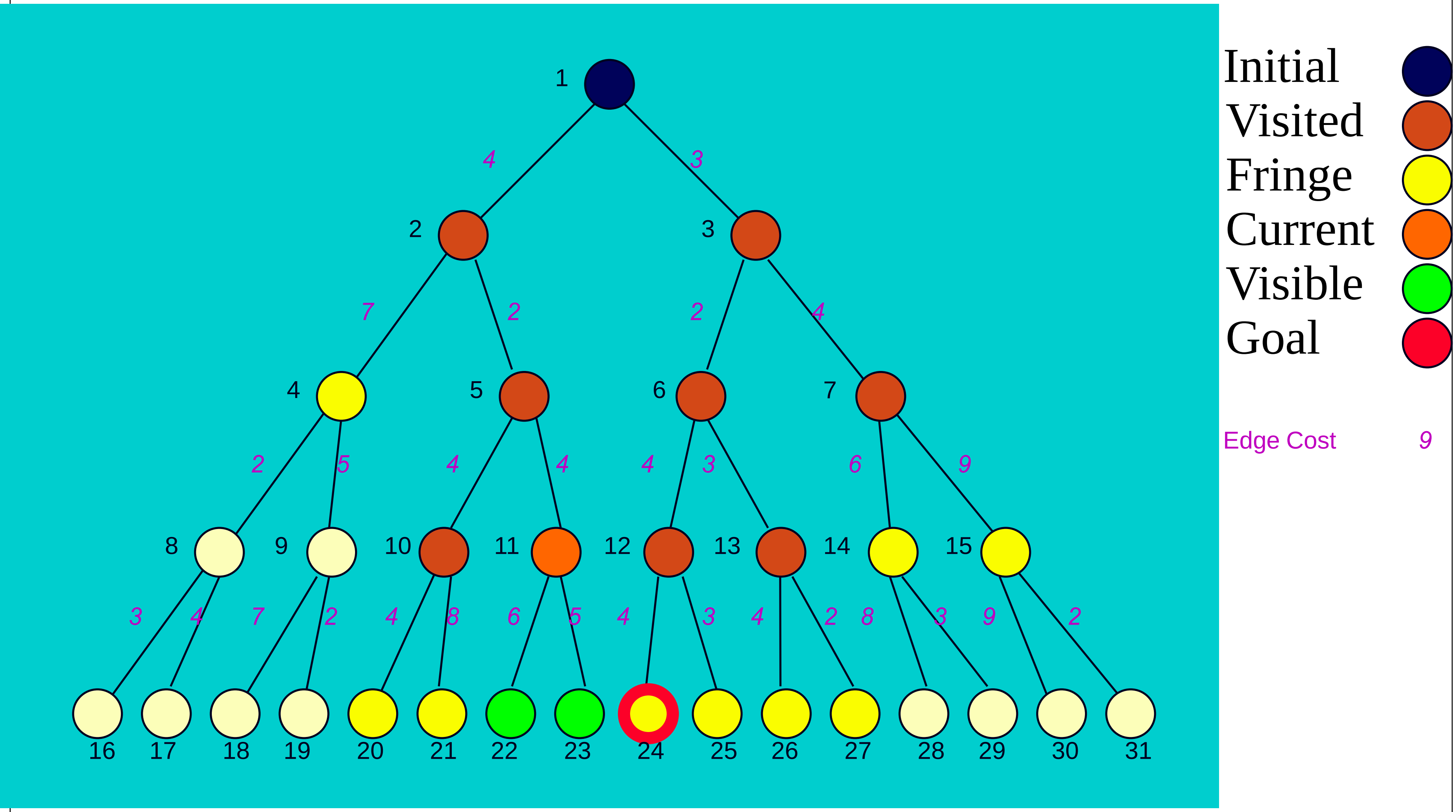
```
function UNIFORM-COST-SEARCH(problem) returns solution

        return TREE-SEARCH(problem, COST-FN, FIFO-QUEUE())
```

| | |
|---|---|
| Time Complexity | $b^{C^*/e}$ |
| Space Complexity | $b^{C^*/e}$ |
| Completeness | yes (finite b, step costs >= e) |
| Optimality | yes |

| | |
|---|---|
| b | branching factor |
| C* | cost of the optimal solution |
| e | minimum cost per action |

# Uniform-Cost Snapshot




Fringe: [27(10), 4(11), 25(12), 26(12), 14(13), 24(13), 20(14), 15(16), 21(18)]
+ [22(16), 23(15)]

# Uniform Cost Fringe Trace

1. [**1(0)**]
2. [**3(3)**, *2(4)*]
3. [**2(4)**, *6(5), 7(7)*]
4. [**6(5)**, *5(6)*, 7(7), *4(11)*]
5. [**5(6)**, 7(7), *13(8)*, *12(9),* 4(11)]
6. [**7(7)**, 13(8), 12(9), *10(10)*, *11(10)*, 4(11)]
7. [**13(8)**, 12(9), 10(10), 11(10), 4(11), *14(13), 15(16)*]
8. [**12(9)**, 10(10), 11(10), *27(10)*, 4(11), *26(12)*, 14(13), 15(16)]
9. [**10(10)**, 11(10), 27(10), 4(11), 26(12), *25(12)*, 14(13), *24(13)*, 15(16)]
10. [**11(10)**, 27(10), 4(11), 25(12), 26(12), 14(13), 24(13), *20(14)*, 15(16), *21(18)*]
11. [**27(10)**, 4(11), 25(12), 26(12), 14(13), 24(13), 20(14), *23(15)*, 15(16), *22(16)*, 21(18)]
12. [**4(11)**, 25(12), 26(12), 14(13), 24(13), 20(14), 23(15), 15(16), 23(16), 21(18)]
13. [**25(12)**, 26(12), 14(13), 24(13),8(13), 20(14), 23(15), 15(16), 23(16), 9(16), 21(18)]
14. [**26(12)**, 14(13), 24(13),8(13), 20(14), 23(15), 15(16), 23(16), 9(16), 21(18)]
15. [**14(13)**, 24(13),8(13), 20(14), 23(15), 15(16), 23(16), 9(16), 21(18)]
16. [**24(13)**,8(13), 20(14), 23(15), 15(16), 23(16), 9(16), *29(16)*,21(18), *28(21)*]

    *Goal reached!*

*Notation:* [**Bold+Yellow: Current Node**;White: Old Fringe Node; *Green+Italics: New Fringe Node*].

*Assumption:* New nodes with the same cost as existing nodes are added after the existing node.

# Uniform Cost Search: Evaluation

- If COST ≡ Depth, then Uniform Cost = Breadth-First
- Completeness: Solution is guaranteed
- Same complexity in worst case as for Breadth-First
  - $O(b^d)$, i.e., exponential in d, because large sub-trees with inexpensive steps can be explored before useful paths with costly steps.
- Optimality
  - If path cost never decreases, will stop at optimal solution
  - Does not necessarily find best solution first
  - Let g (n) = path cost at node n: need

$$g(child\ (n)) \geq g(n)$$

# Breadth-First vs. Uniform-Cost

- breadth-first always expands the shallowest node
  - only optimal if all step costs are equal
- uniform-cost considers the overall path cost
  - optimal for any (reasonable) cost function
    - non-zero, positive
  - gets bogged down in trees with many fruitless, short branches
    - low path cost, but no goal node
- both are complete for non-extreme problems
  - finite number of branches
  - strictly positive search function

# Depth-First

- continues exploring newly generated nodes
  - achieved by the TREE-SEARCH method by appending newly generated nodes at the beginning of the search queue
    - utilizes a Last-In, First-Out (LIFO) queue, or stack

**function** DEPTH-FIRST-SEARCH(*problem*) **returns** *solution*

**return** TREE-SEARCH(*problem,* LIFO-QUEUE())

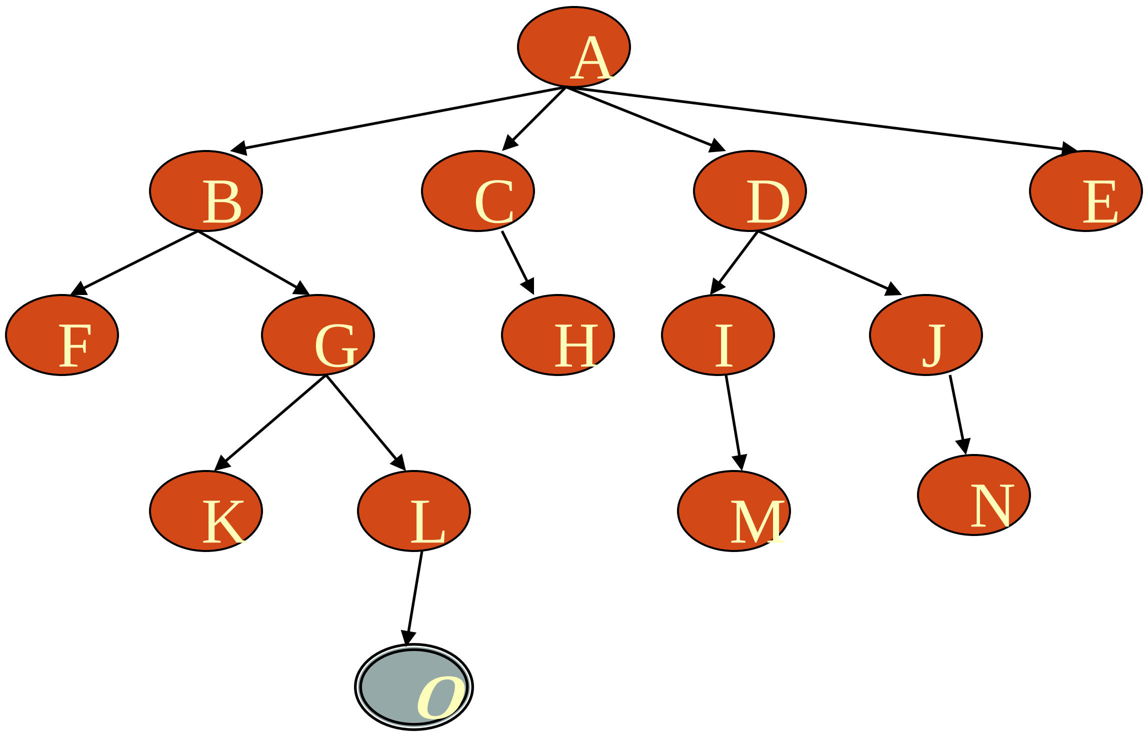
| | |
|---|---|
| Time Complexity | $b^m$ |
| Space Complexity | b*m |
| Completeness | no (for infinite branch length) |
| Optimality | no |

| | |
|---|---|
| b | branching factor |
| m | maximum path length |

# Depth First Search (DFS)

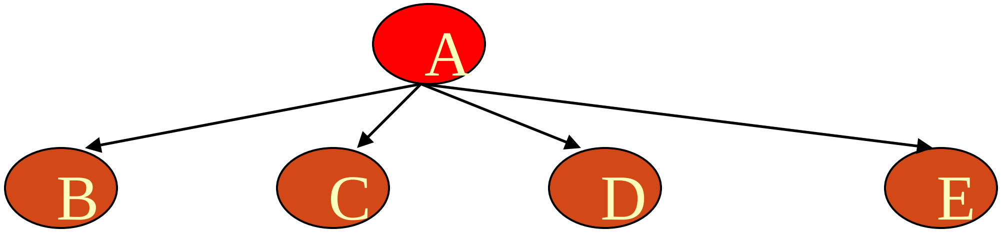
- Application2:

  Given the following state space (tree search), give the sequence of visited nodes when using DFS (assume that the node ***O*** is the goal state):



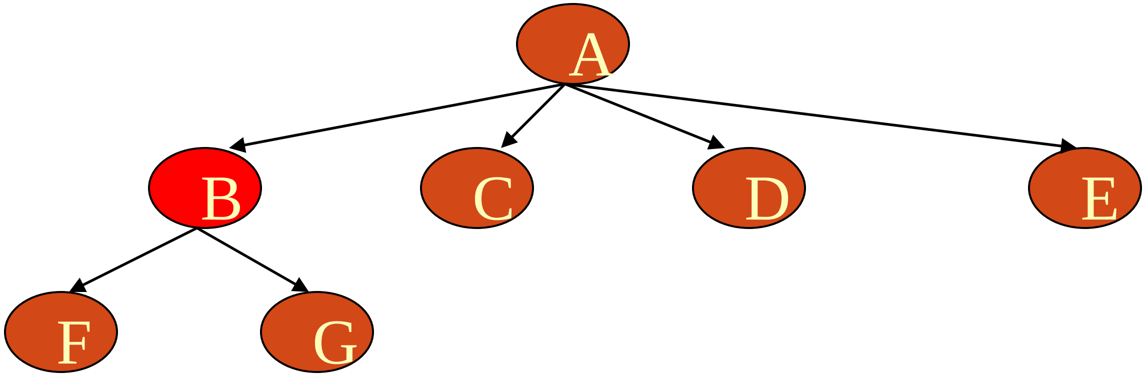
# Depth First Search

- A,



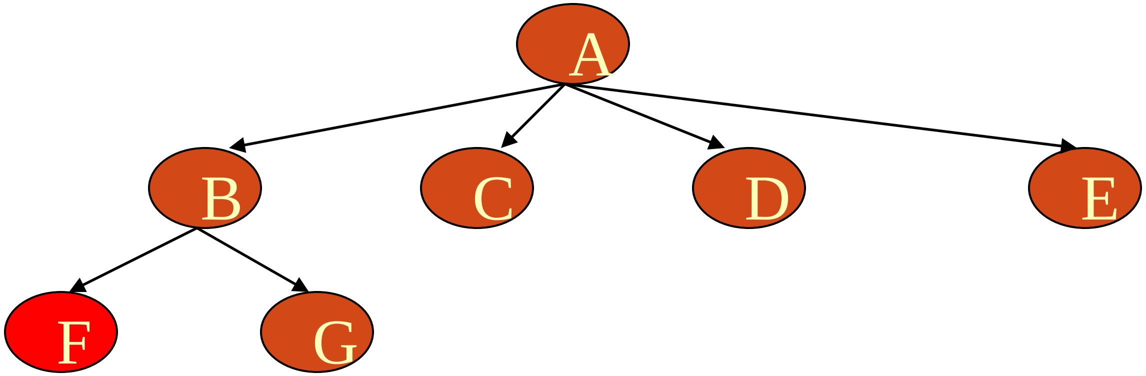
# Depth First Search

- A,B,



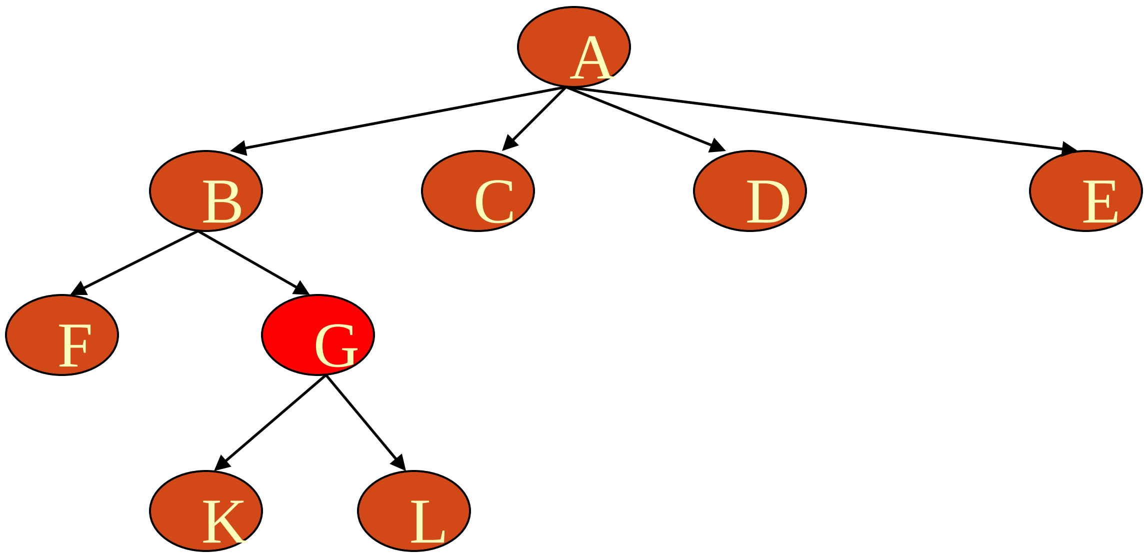
# Depth First Search

- A,B,F,



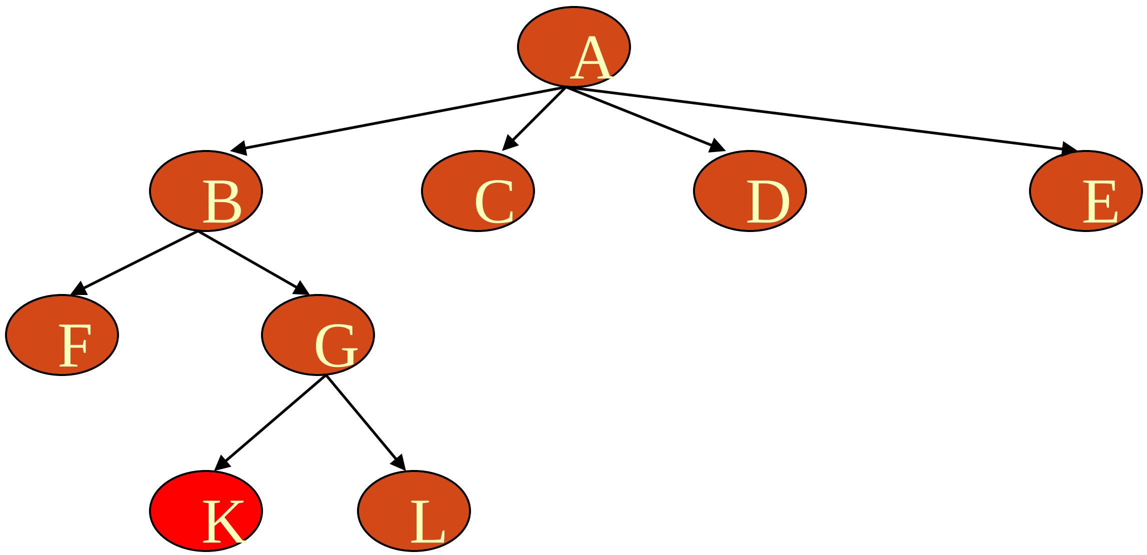
# Depth First Search

- A,B,F,
- G,



# Depth First Search

- A,B,F,
- G,K,

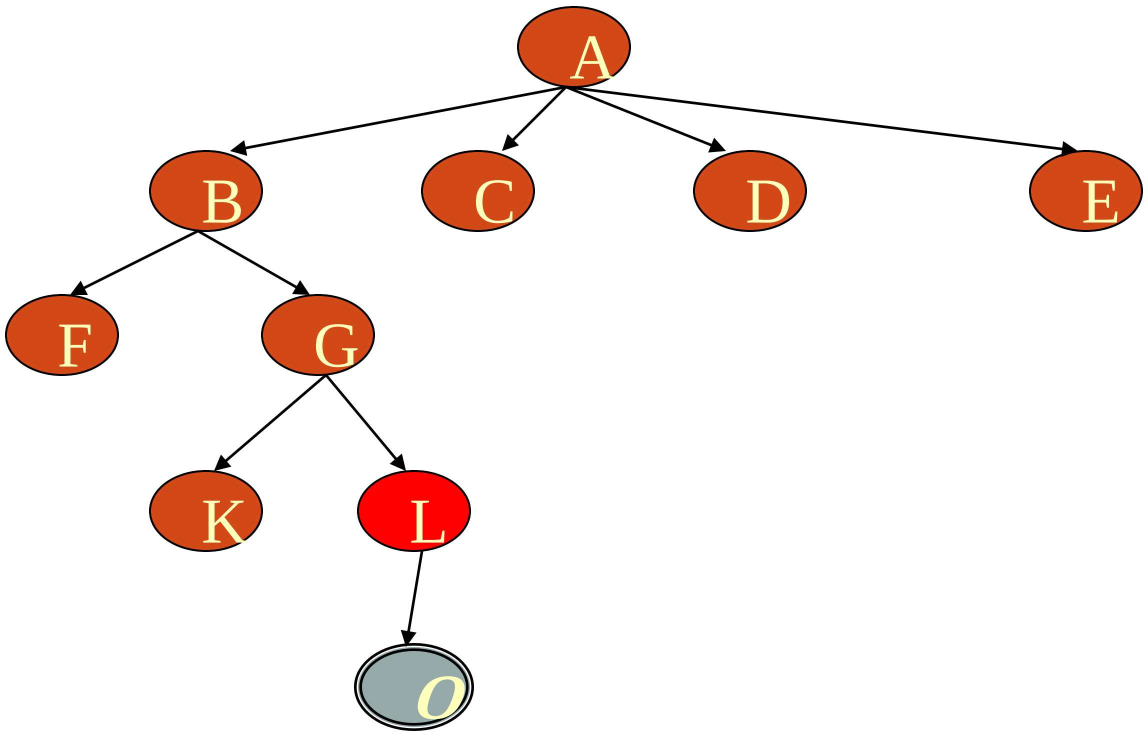





Aziz M. Qaroush - Birzeit University

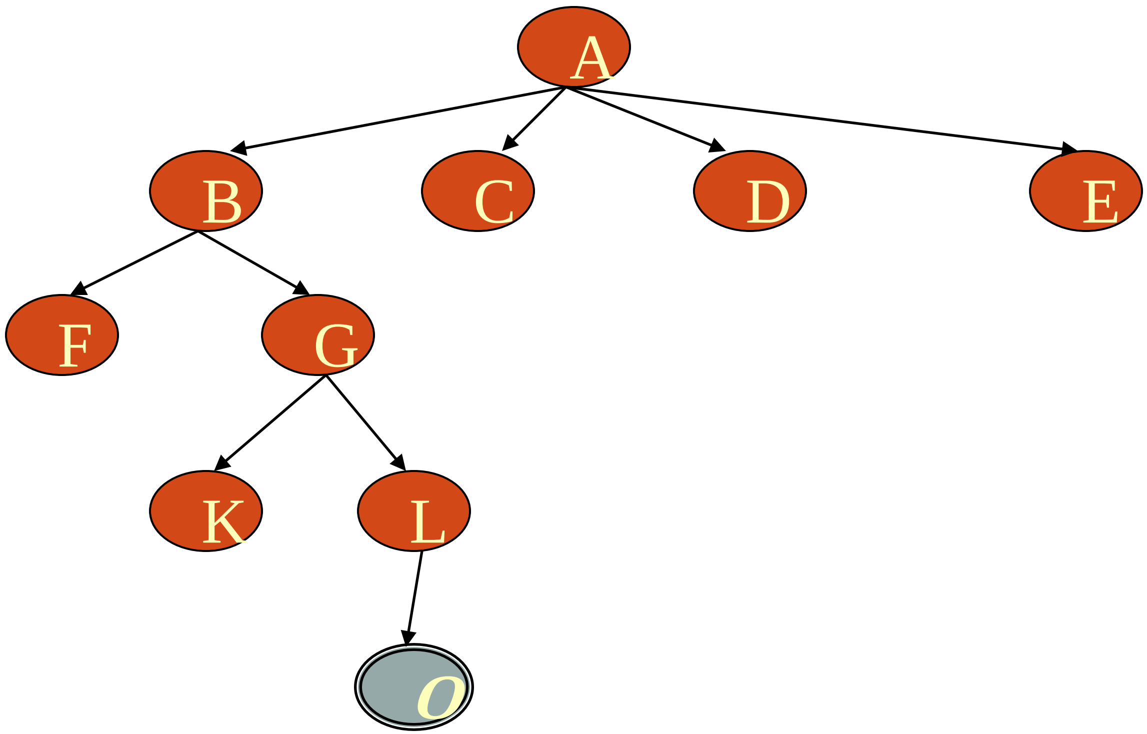
# Depth First Search

- A,B,F,
- G,K,
- L,



# Depth First Search

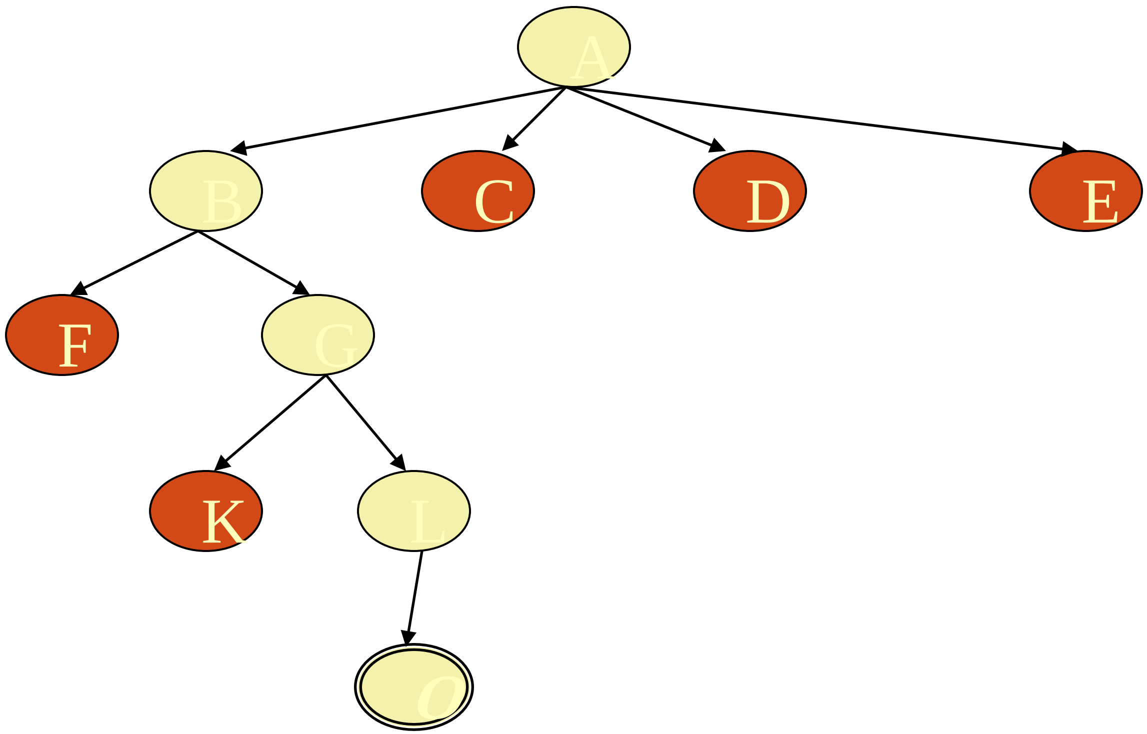
- A,B,F,
- G,K,
- L, *O: Goal State*



# Depth First Search

The returned solution is the sequence of operators in the path:

***A, B, G, L, O***

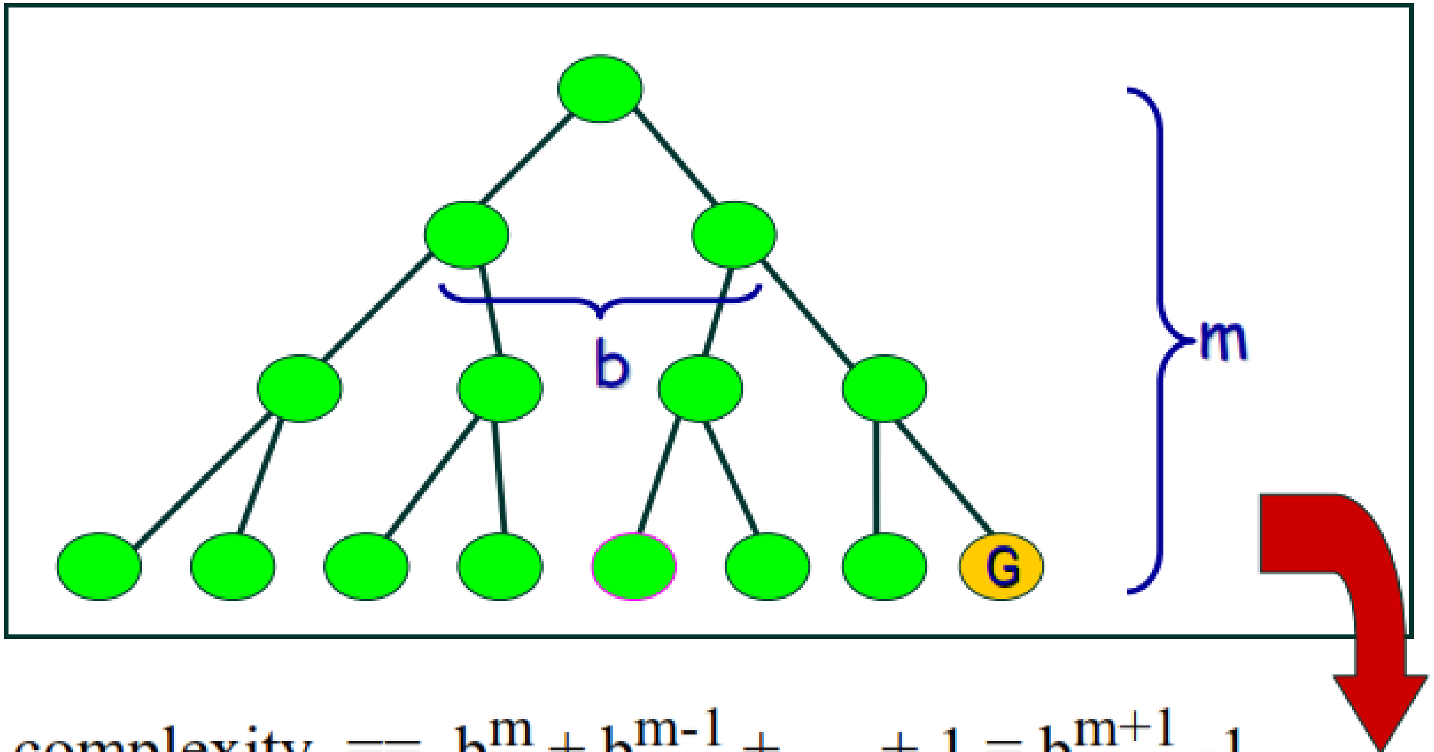


# Depth-First Search: Evaluation

- Completeness: Does it always find a solution if one exists?
  - NO: unless search space is finite and no loops are possible
- Time complexity: Still need to explore all nodes $O(b^m)$
  - Terrible if *m* is much larger than *d* (depth of optimal solution)
  - But if many solutions, then faster than BFS
- Space complexity: $O(bm+1)$
  - Must store all nodes on current path
  - Must store all unexplored sibling nodes on each hit
  - At depth m, required to store b*m nodes
  - Much better than $O(b^m)$
- Optimality: No (Might never find any solutions)

# Time Complexity of Depth-first

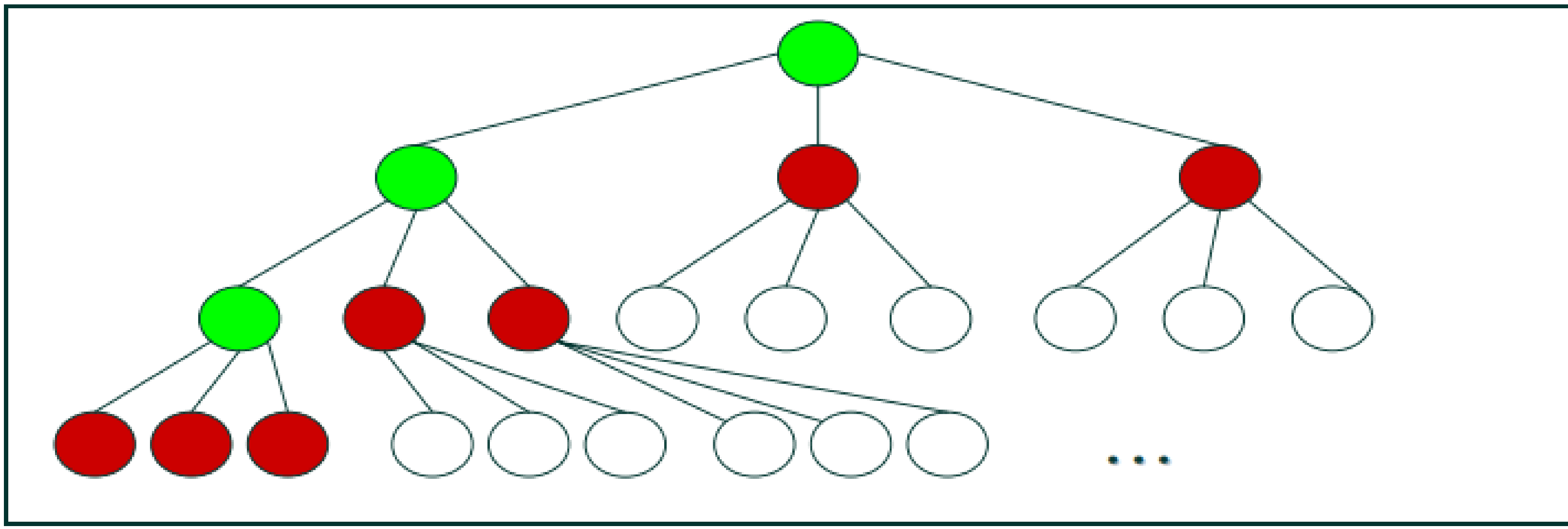
- In the worst case:
  - The goal node may be on the right-most branch,

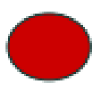



- Time complexity $== b^m + b^{m-1} + \ldots + 1 = \frac{b^{m+1} - 1}{b - 1}$
- Thus: $O(b^m)$

# Space Complexity of Depth-first

- Largest number of nodes in QUEUE is reached in bottom left-most node.
- Example: m = 3, b = 3 :




- QUEUE contains all  nodes. Thus: 7.
- In General: ((b-1) * m) + 1
- Order: O(m*b)

# Depth-First vs. Breadth-First

- depth-first goes off into one branch until it reaches a leaf node
  - not good if the goal is on another branch
  - neither complete nor optimal
  - uses much less space than breadth-first
    - much fewer visited nodes to keep track of
    - smaller fringe

- breadth-first is more careful by checking all alternatives
  - complete and optimal
    - under most circumstances
  - very memory-intensive

# Depth-Limited Search

- similar to depth-first, but with a limit
  - overcomes problems with infinite paths
  - sometimes a depth limit can be inferred or estimated from the problem description
    - in other cases, a good depth limit is only known when the problem is solved
  - based on the TREE-SEARCH method
  - must keep track of the depth

```
function DEPTH-LIMITED-SEARCH(problem, depth-limit) returns solution

        return TREE-SEARCH(problem, depth-limit, LIFO-QUEUE())
```

| | |
|---|---|
| Time Complexity | $b^l$ |
| Space Complexity | $b*l$ |
| Completeness | no (goal beyond l, or infinite branch length) |
| Optimality | no |

| | |
|---|---|
| b | branching factor |
| l | depth limit |

# Iterative Deepening

- applies LIMITED-DEPTH with increasing depth limits
  - combines advantages of BREADTH-FIRST and DEPTH-FIRST methods
  - many states are expanded multiple times
    - doesn't really matter because the number of those nodes is small
  - in practice, one of the best uninformed search methods
    - for large search spaces, unknown depth

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns solution
  for depth := 0 to unlimited do
     result := DEPTH-LIMITED-SEARCH(problem, depth-limit)
         if result != cutoff then return result
```

| | |
|---|---|
| Time Complexity | $b^d$ |
| Space Complexity | b*d |
| Completeness | yes |
| Optimality | yes (all step costs identical) |

| | |
|---|---|
| b | branching factor |
| d | tree depth |

# Iterative deepening search

Use an artificial depth cutoff, $k$.

- If search to depth $k$ succeeds: DONE.

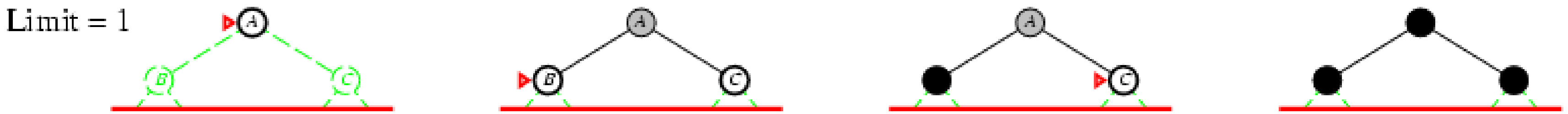
  If not: increase $k$ by 1; start over.

  (Regenerate nodes, as necessary)
- Each iteration uses

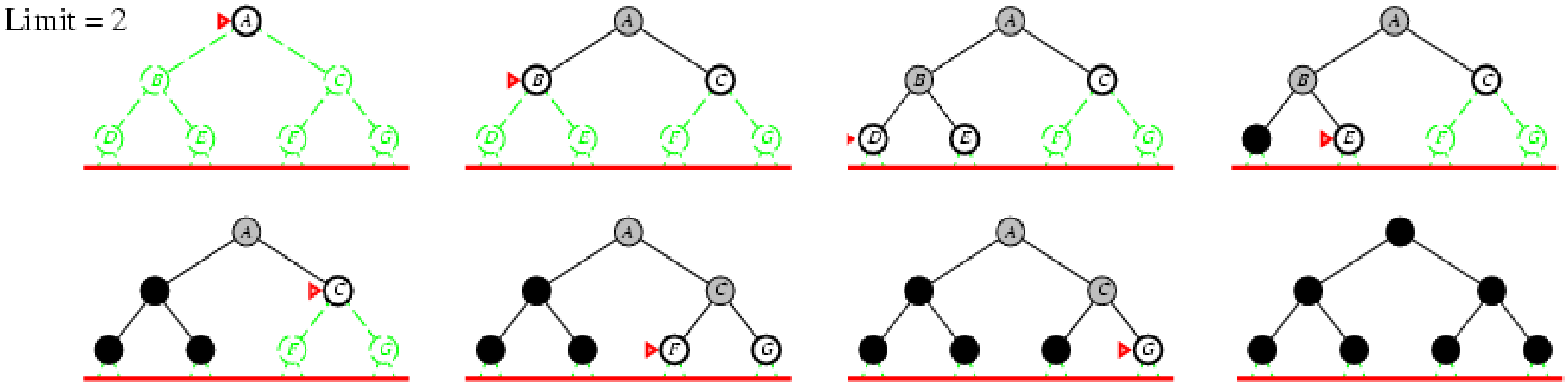
  Depth-limited Depth-First Search

Limit = 0

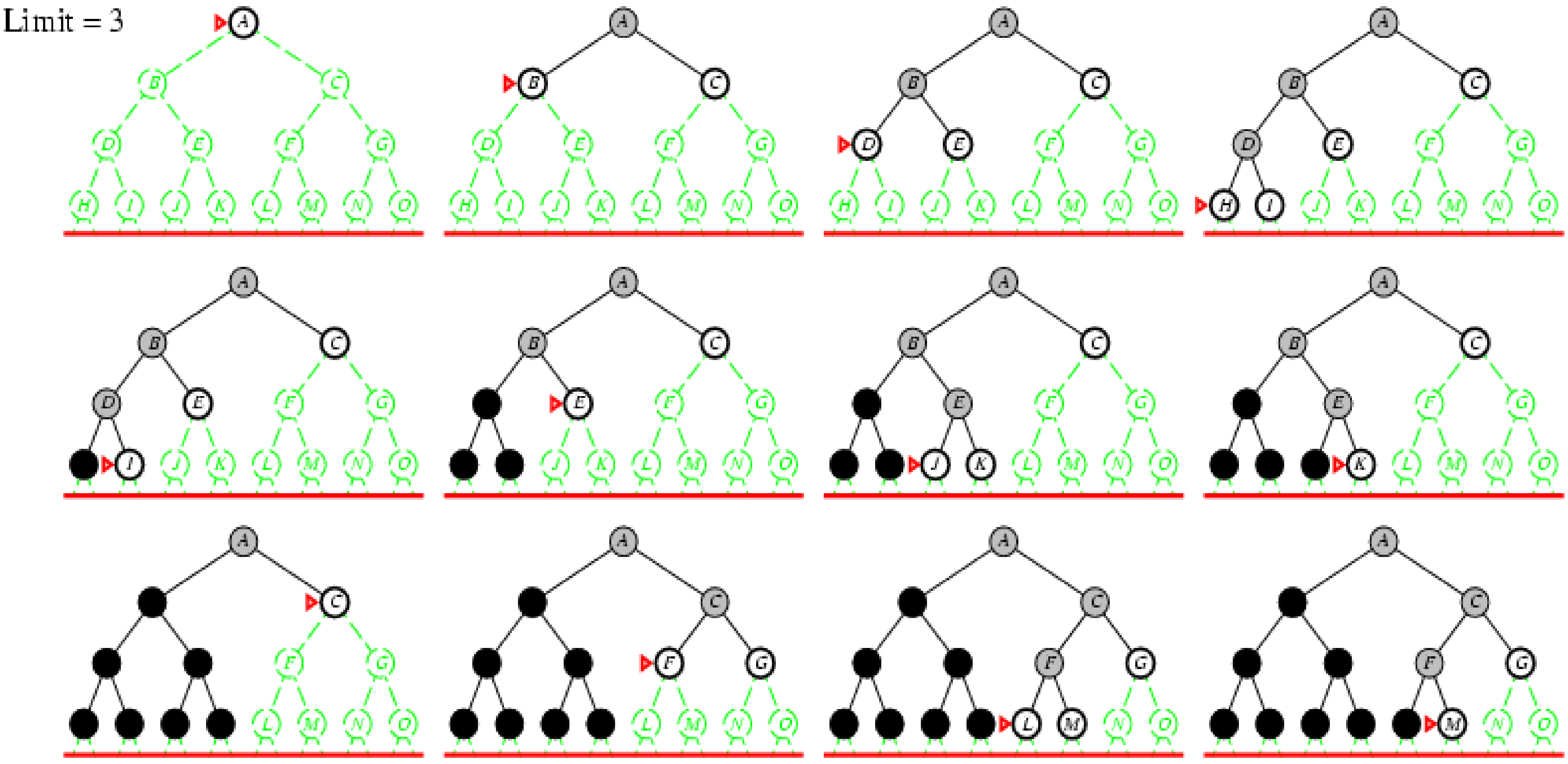
# Iterative deepening search l =1



# Iterative deepening search l =2



# Iterative deepening search l =3



# Bi-directional Search

- search simultaneously from two directions
  - forward from the initial and backward from the goal state
- may lead to substantial savings if it is applicable
- has severe limitations
  - predecessors must be generated, which is not always possible
  - search must be coordinated between the two searches
  - one search must keep all nodes in memory

| | |
|---|---|
| Time Complexity | $b^{d/2}$ |
| Space Complexity | $b^{d/2}$ |
| Completeness | yes (b finite, breadth-first for both directions) |
| Optimality | yes (all step costs identical, breadth-first for both directions) |

| | |
|---|---|
| b | branching factor |
| d | tree depth |

# Improving Search Methods

- make algorithms more efficient
  - avoiding repeated states
  - utilizing memory efficiently
- use additional knowledge about the problem
  - properties (“shape”) of the search space
    - more interesting areas are investigated first
  - pruning of irrelevant areas
    - areas that are guaranteed not to contain a solution can be discarded

# Comparing Uninformed Search Strategies

| Criterion | Breadth-first | Uninformed cost | Depth-first | Depth-limited | Iterative deepening | Bidirectional (if applicable) |
|---|---|---|---|---|---|---|
| Time | b^d | b^d | b^m | b^l | b^d | b^(d/2) |
| Space | b^d | b^d | bm | bl | bd | b^(d/2) |
| Optimal? | Yes | Yes | No | No | Yes | Yes |
| Complete? | Yes | Yes | No | Yes if l≥d | Yes | Yes |

- *b* – max branching factor of the search tree
- *d* – depth of the least-cost solution
- *m* – max depth of the state-space (may be infinity)
- l – depth cutoff