# COL333/671: Introduction to AI

Semester I, 2021

## Solving Problems by Searching
## Uninformed Search

**Rohan Paul**

# This Class

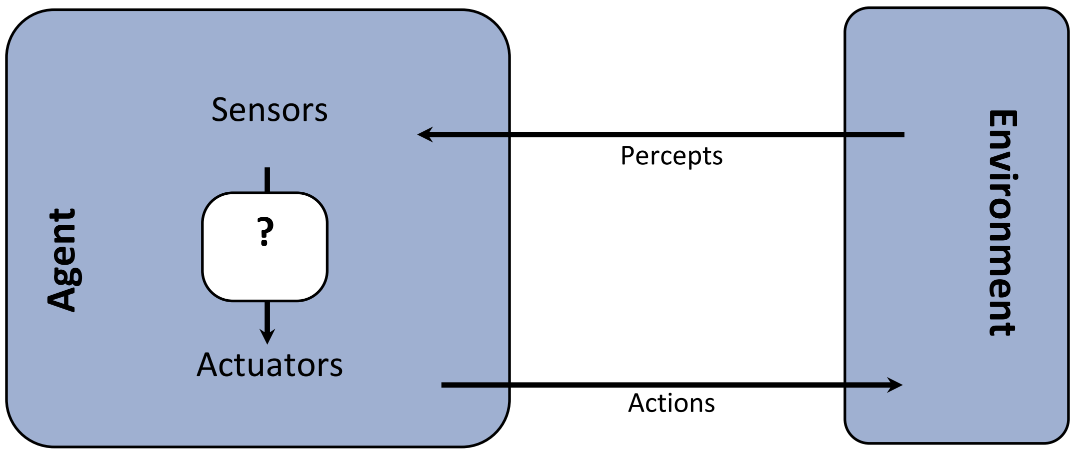
- Reflex Agents
- Problem Solving as search
  - Uninformed Search
- Reference Material
  - AIMA Ch. 3

# Acknowledgement

These slides are intended for teaching purposes only. Some material has been used/adapted from web sources and from slides by Doina Precup, Dorsa Sadigh, Percy Liang, Mausam, Dan Klein, Nicholas Roy and others.

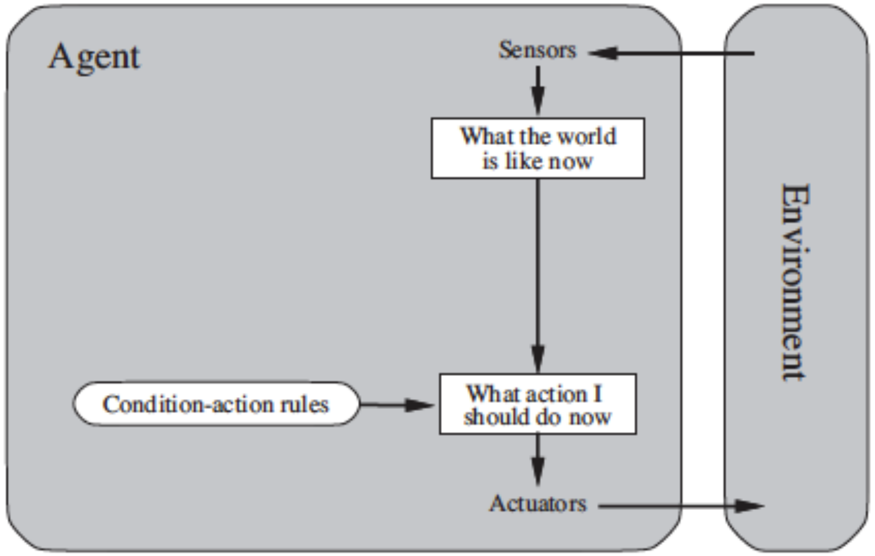
# Last time: Agent View of AI

- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.
- Examples
  - Alexa
  - Robotic system
  - Refinery controller
  - Question answering system
  - Crossword puzzle solver
  - ……



# Simple Reflex Agents

- A Reflex Agent
  - Selects action based on the current percept.
  - Directly map states to actions.
- Operate using *condition-action* rules.
  - **If** (condition) **then** (action)
- Example:
  - An autonomous car that is avoiding obstacles.
  - **If** (*car-in-front-is-braking*) **then** (*initiate-braking*)
- Problem: no notion of goals
  - The autonomous car cannot take actions that will lead to an intended destination.



# From Reflex to Problem Solving Agents

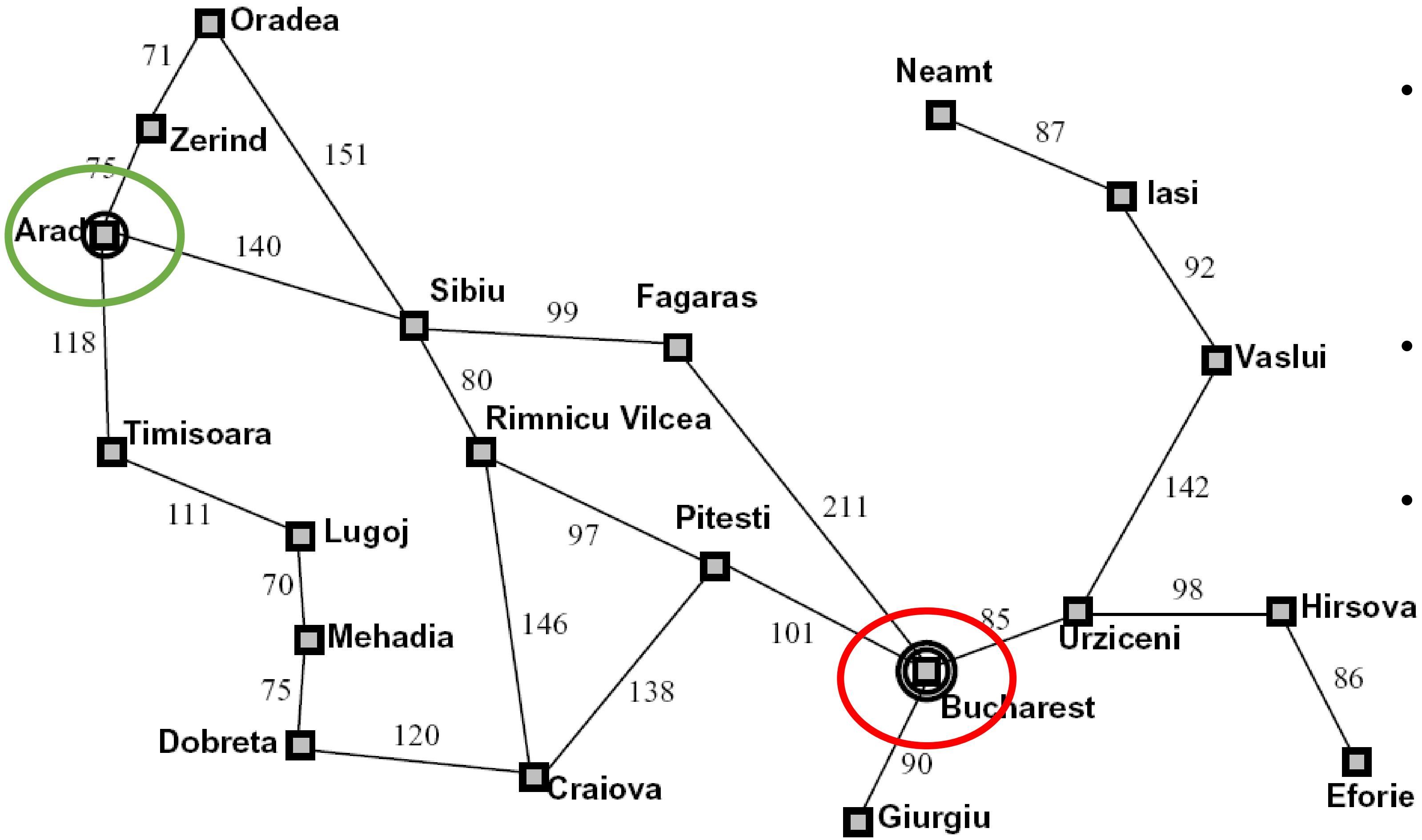
- Reflex agents
  - Directly map states to actions.
  - No notion of goals. Do not consider action consequences.

$x \longrightarrow \boxed{f} \longrightarrow$ single action $y$

- Problem Solving Agents
  - Adopt a goal
  - Consider future actions and the desirability of their outcomes
  - Solution: a sequence of actions that the agent can execute leading to the goal.
  - Today's focus.

$x \longrightarrow \boxed{f} \longrightarrow$ **action sequence** $(a_1, a_2, a_3, a_4, \ldots)$

# Example – Route Finding




- Problem:
  - Find a solution i.e., a sequence of actions (road traversals) that can take the agent to the destination in minimum time.
- Search
  - Process of looking for a sequence of actions that reaches the goal
- Note: as we will see search problems are more general than path finding.

# Search Problem Formulation

**Many problems in AI can be modeled as search problems.**

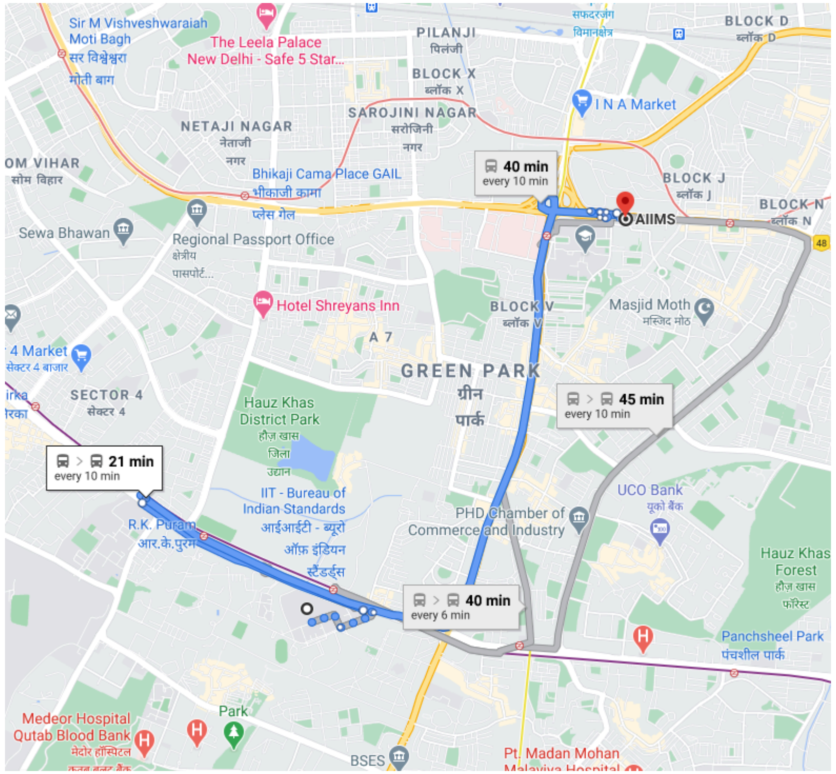
- *State space* $S$: all possible configurations of the domain of interest
- *An initial (start) state* $s_0 \in S$
- *Goal states* $G \subset S$: the set of end states
  - Often defined by a *goal test* rather than enumerating a set of states
- *Operators* $A$: the actions available
  - Often defined in terms of a *mapping from a state to its successor*

**Transition model or successor function**

# Formulating a Search Problem

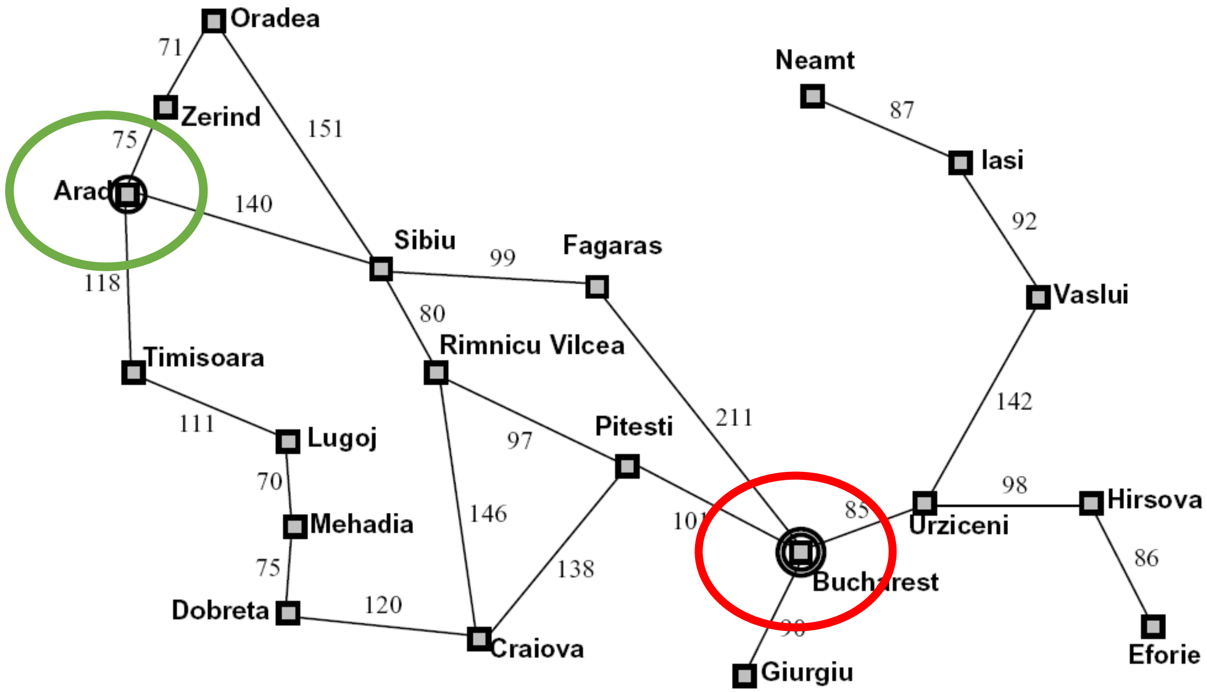
- *Path*: a sequence of states and operators
- *Path cost*: a number associated with any path
  - Measures the quality of the path
  - Usually the smaller, the better

- Find a solution which is a sequence of actions that transforms the start state to a goal state.
- Search is the process of looking for a sequence of actions that reaches the goal.




Route finding in a map.

# Example – Route Finding

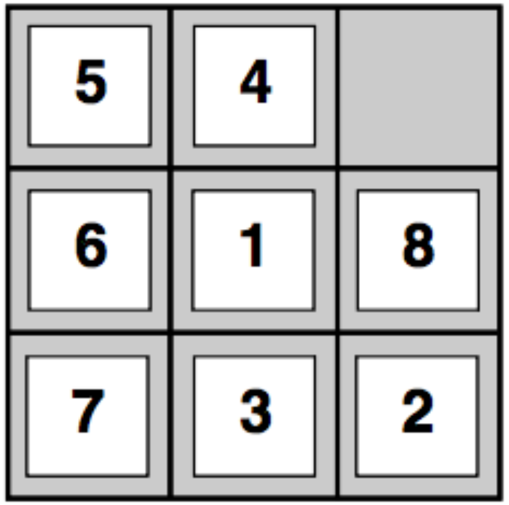



- State space:
  - All the cities on the map.
- Actions:
  - Traversing a road: Going to an adjacent city.
- Cost:
  - Distance along the road
- Start state:
  - Arad
- Goal test:
  - Is state == Bucharest?

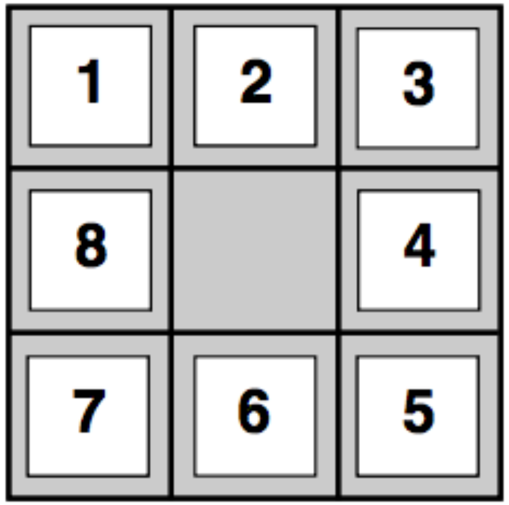
# Modeling Assumptions

- Environment is observable
  - The agent always knows it current state.
- Discrete states and actions
  - Finite number of cities.
  - At any given state there are a finite number of actions.
- Known and Deterministic action outcomes
  - The agent knows which sates are reached by each action.
  - Action has exactly one outcome when applied to a state.

# Example – The Eight Puzzle




**Start State**

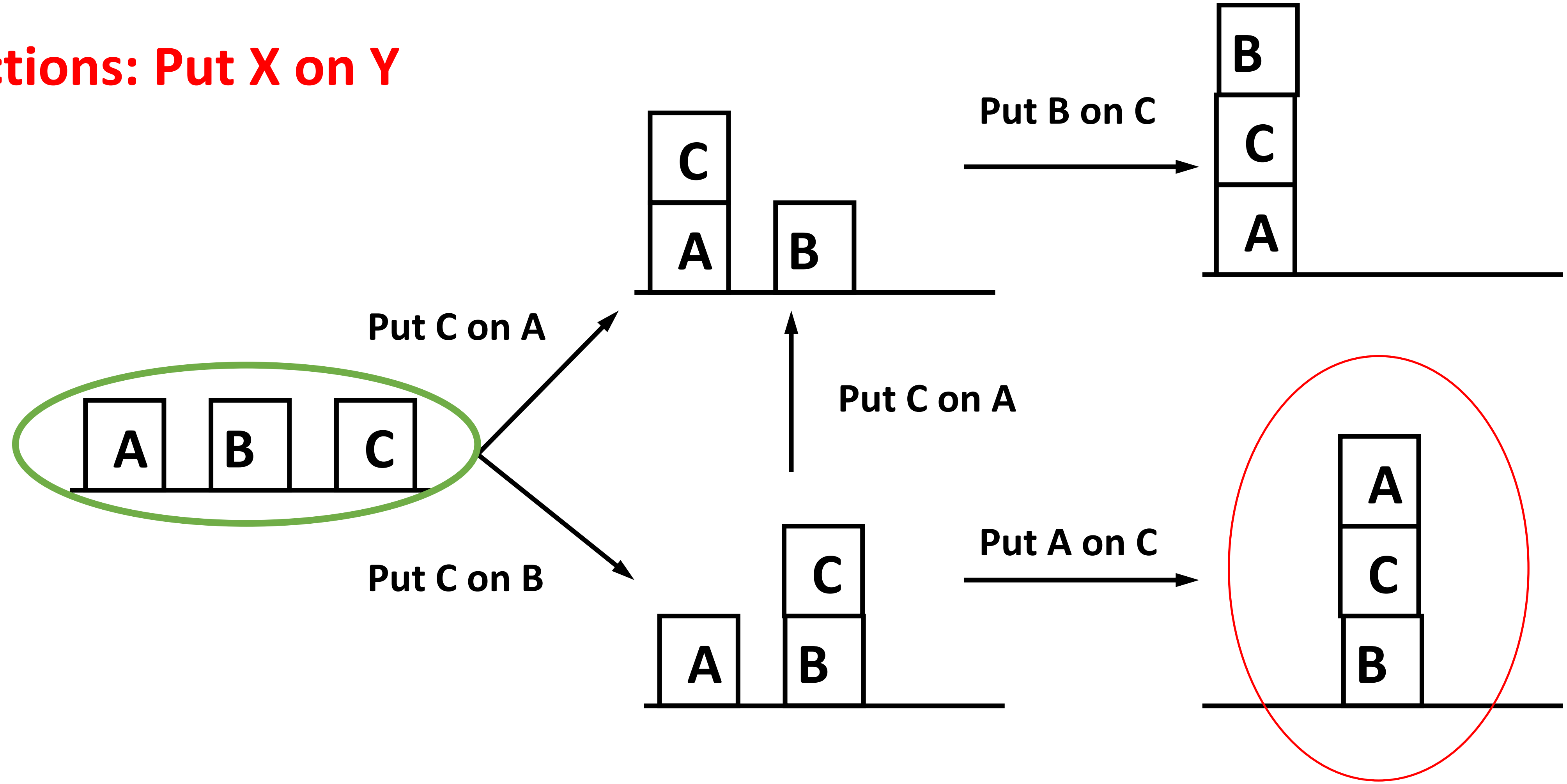



**Goal State**

- States: configurations of the puzzle
- Goals: target configuration
- Operators: swap the blank with an adjacent tile
- Path cost: number of moves

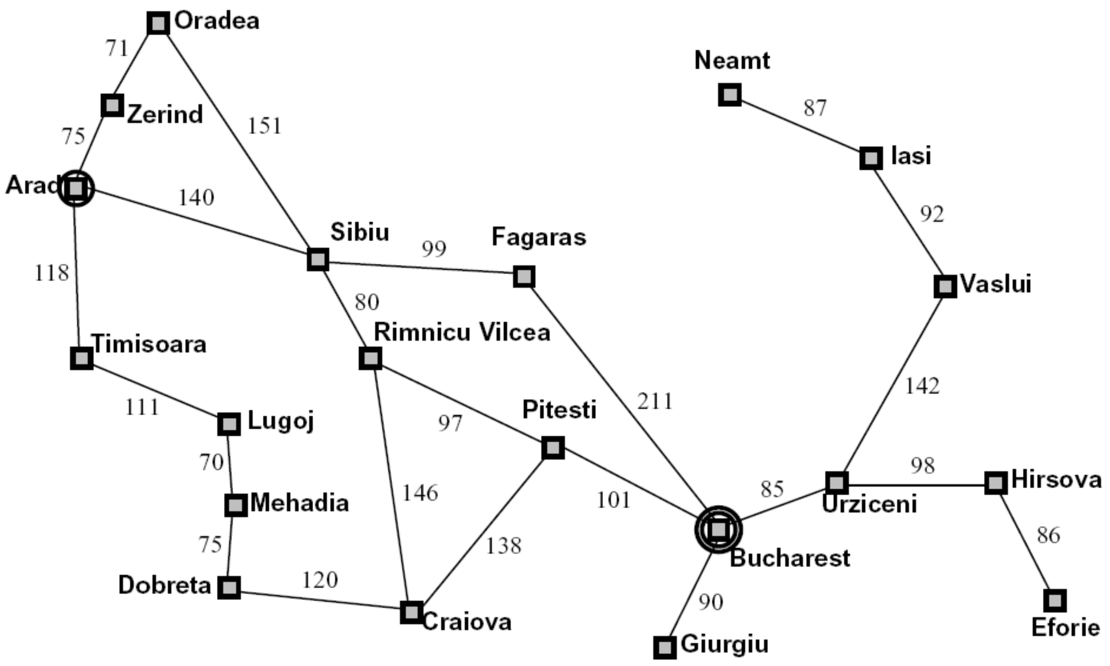
# Example – Block Manipulation

**Actions: Put X on Y**



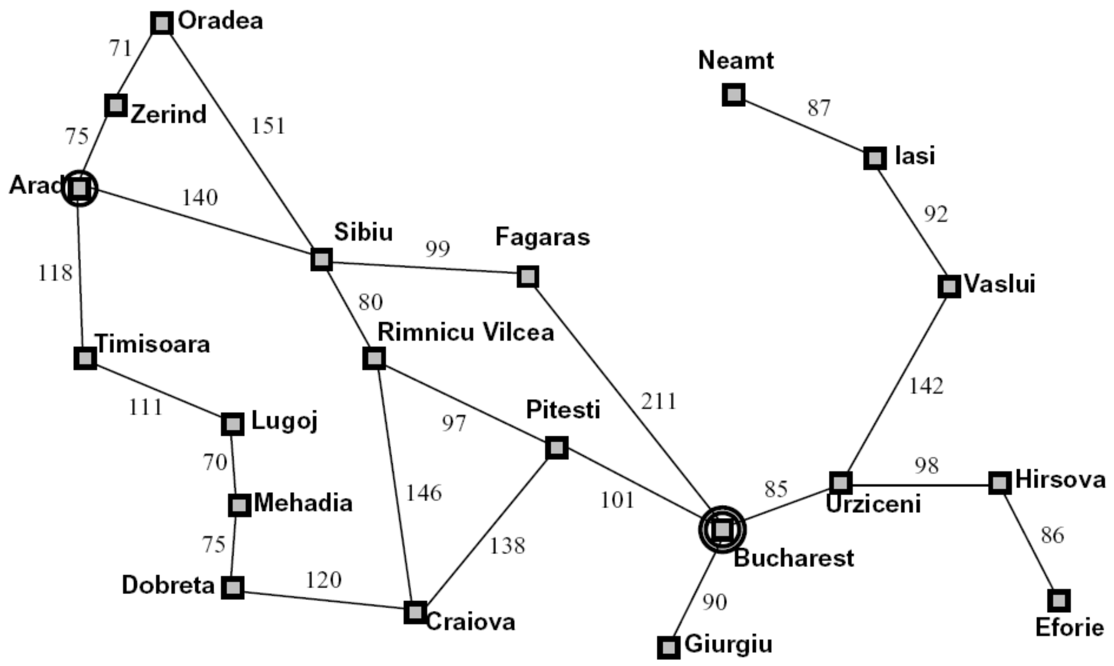
# State Space Graphs

- A representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)
  - **Each state occurs only once**
- The full graph is usually too large.
- The graph is built and explored implicitly by applying actions on states.

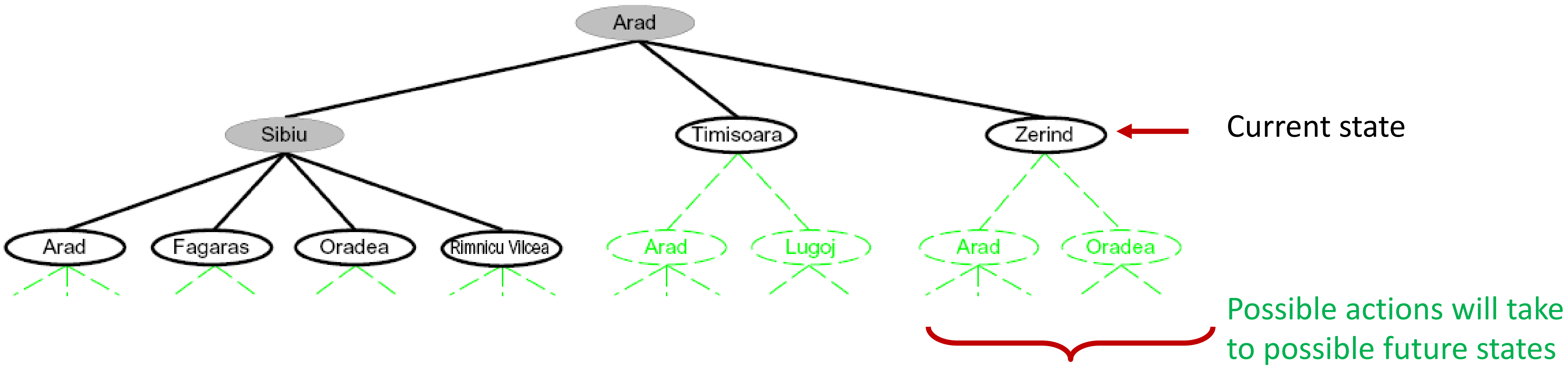


# Searching for a solution

- Once the problem is formulated, need to solve it.
- Solution – action sequences. Search algorithms work by considering various possible action sequences.



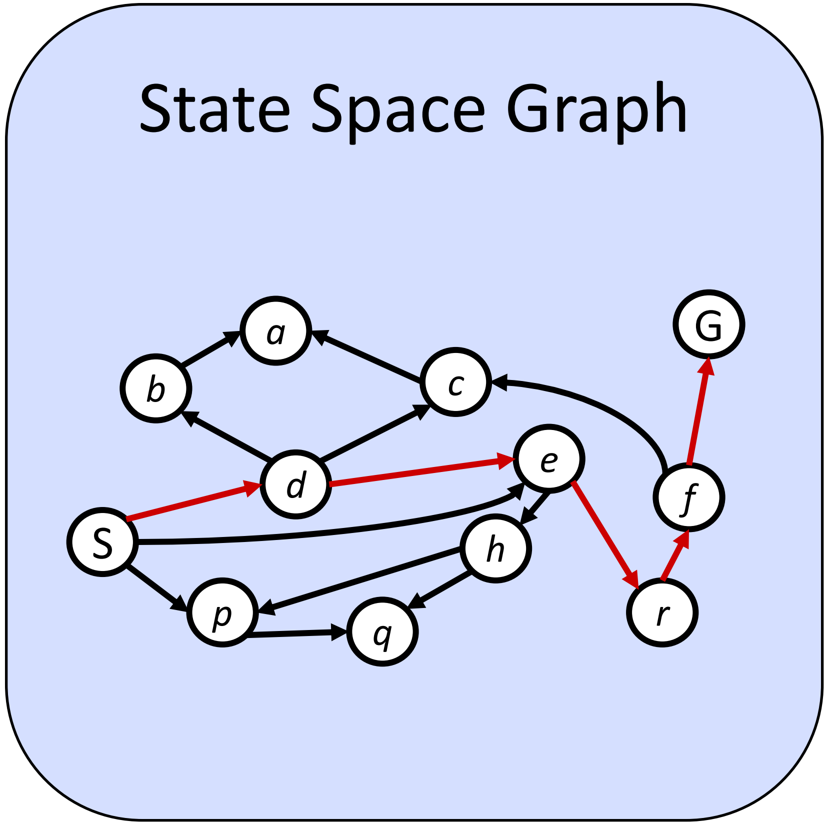
# Search Trees

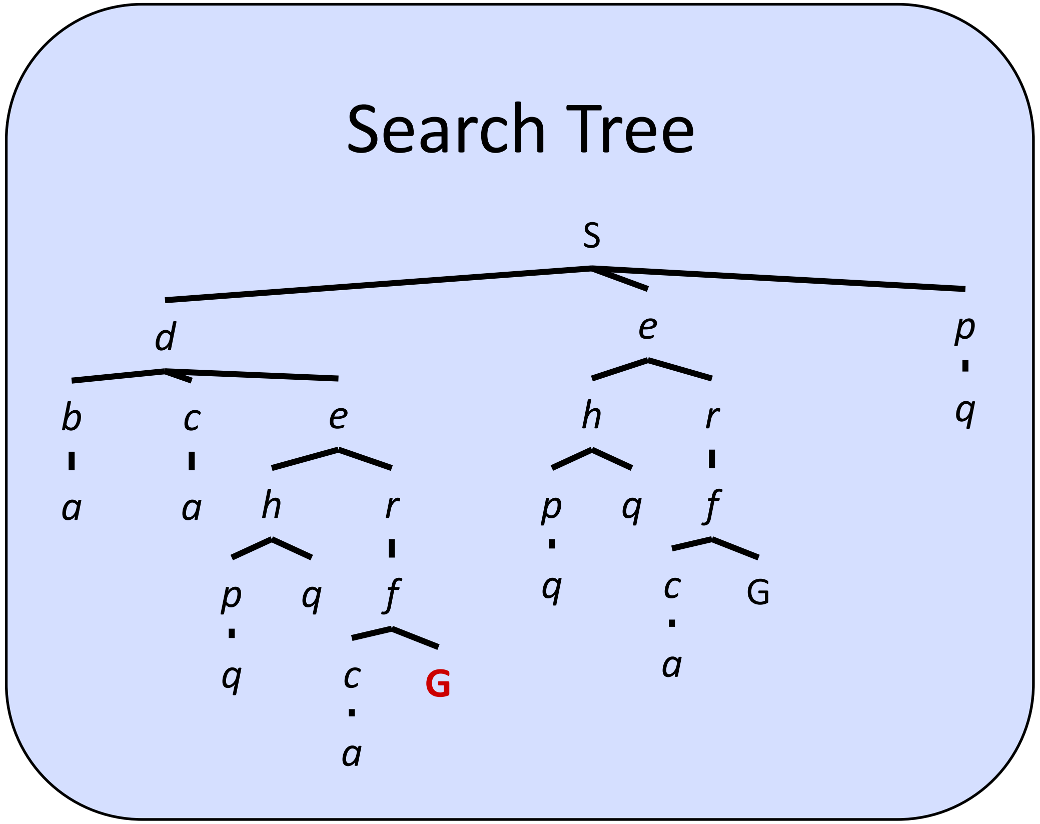



- A search tree: A "what if" tree of plans and their outcomes
  - The start state is the root node
  - Check if the node contains the goal.
  - Other wise, "expand" the node
    - Apply legal actions  on the current state to generate new set of states.
  - Frontier
    - All the nodes available for expansion.
  - In a Search Tree, nodes show states, but correspond to PLANS that achieve those states

# State Space Graph vs. Search Tree

- Each NODE in in the search tree is an entire PATH in the state space graph.
- Construct both on demand and construct as little as possible.



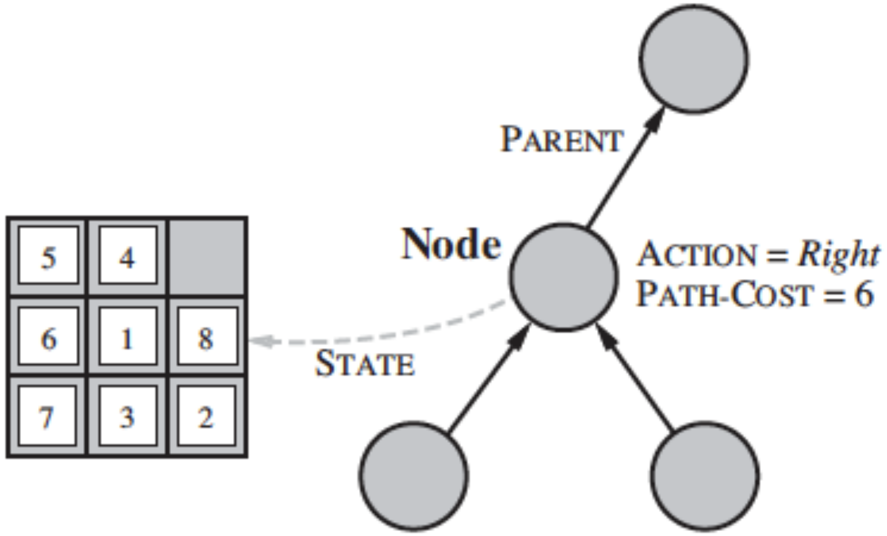



# Tree Search

```
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
    end
```

# Infrastructure for Search Algorithms

- *Defining a search node*:
  - Each node contains a state
  - Node also contains additional information, e.g.:
    * The parent state and the operator used to generate it
    * Cost of the path so far
    * Depth of the node
- *Expanding a node*:
  - Applying all legal operators to the state contained in the node
  - Generating nodes for all the corresponding successor states.



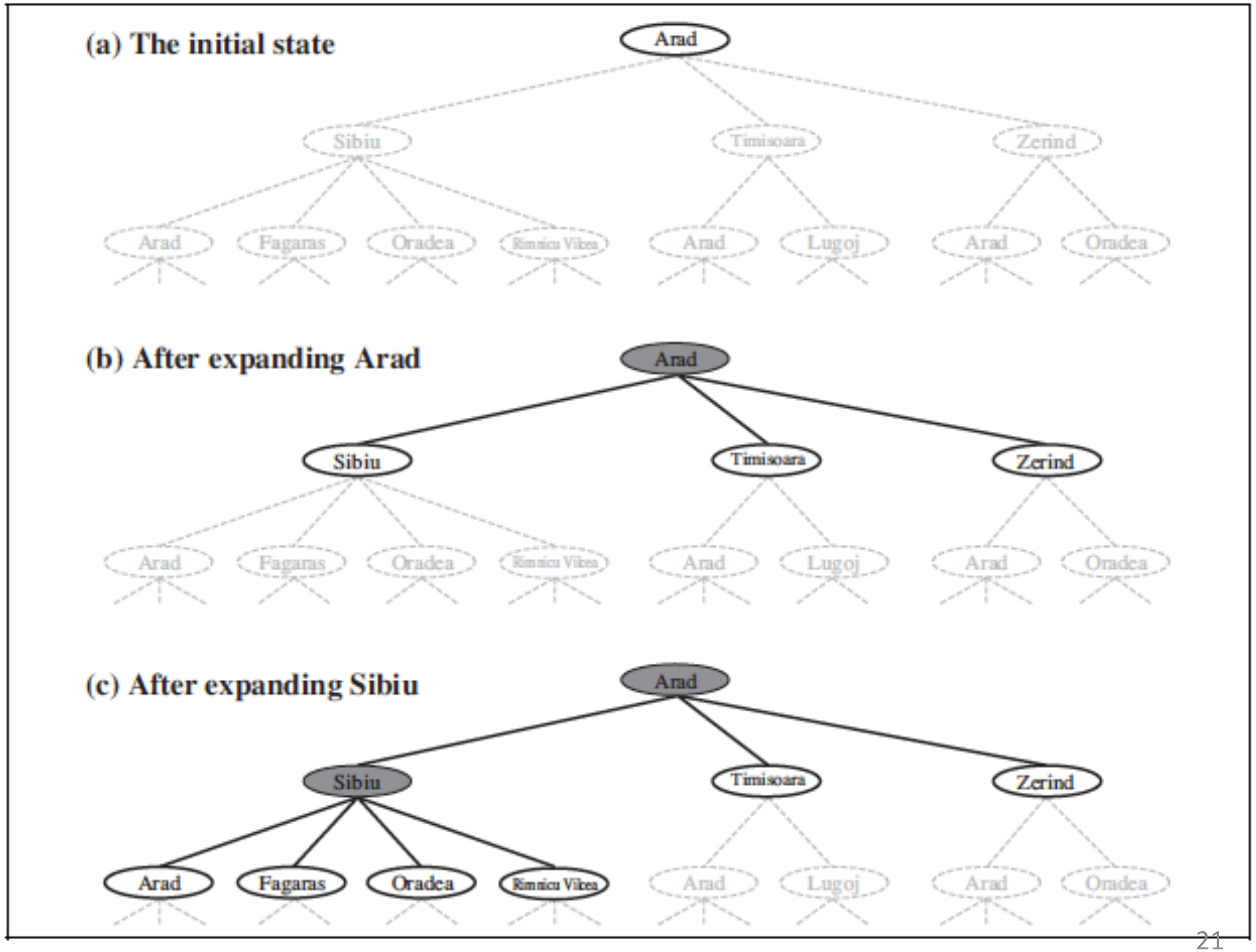

```
function CHILD-NODE(problem, parent, action) returns a node
   return a node with
      STATE = problem.RESULT(parent.STATE, action),
      PARENT = parent, ACTION = action,
      PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

# Search Tree

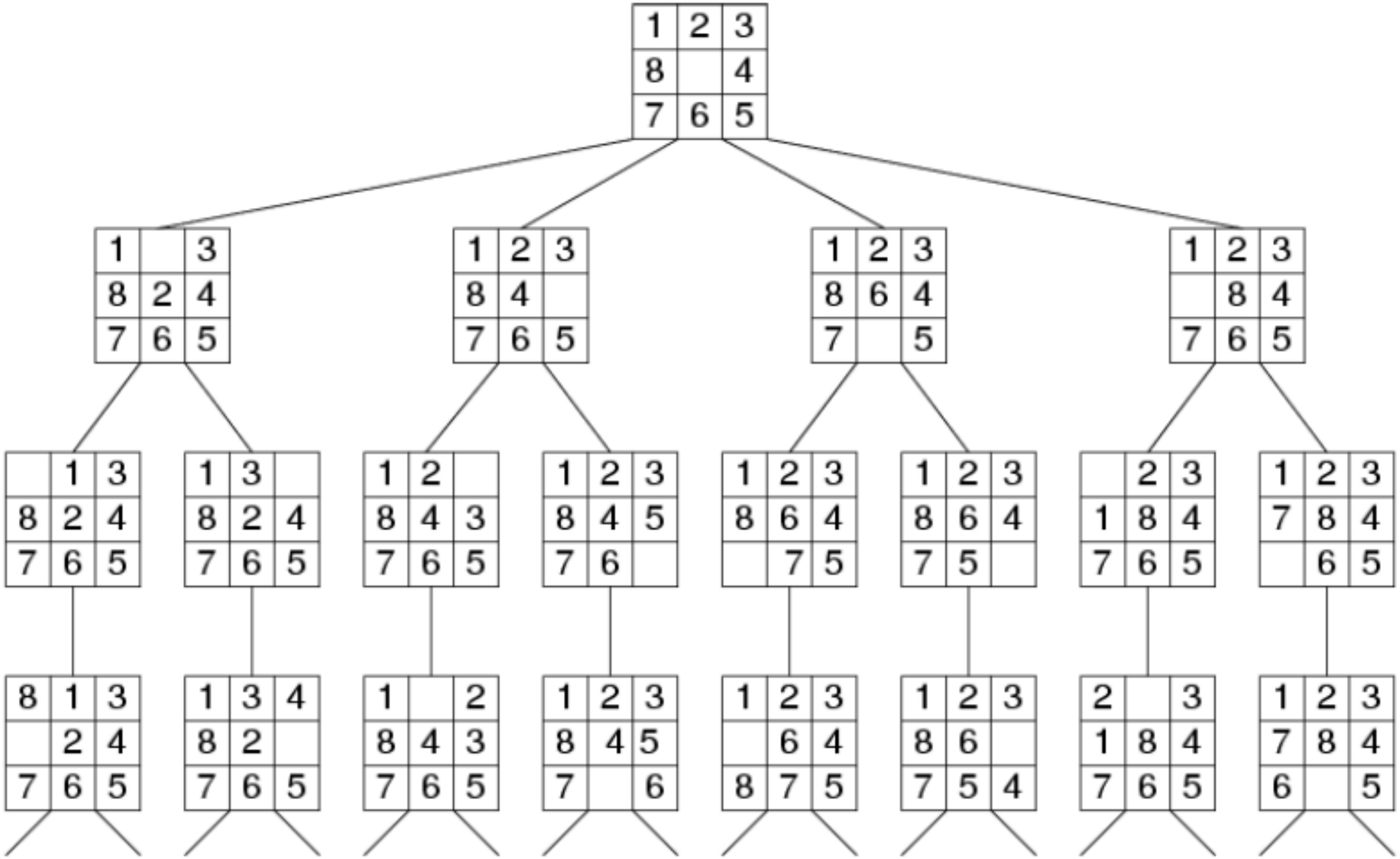
- The process of expansion while constructing the search tree.
  - Note that Arad appears again.

- Loops lead to redundancy
  - Why? Path costs are additive.

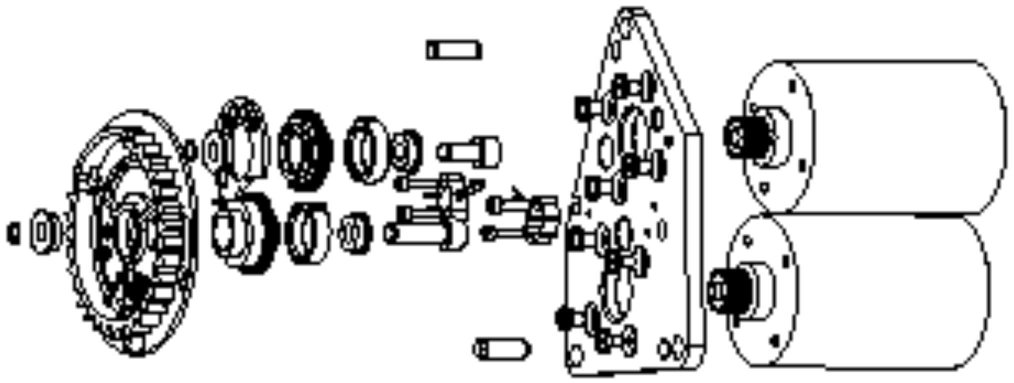
- Can we remember which nodes were expanded?

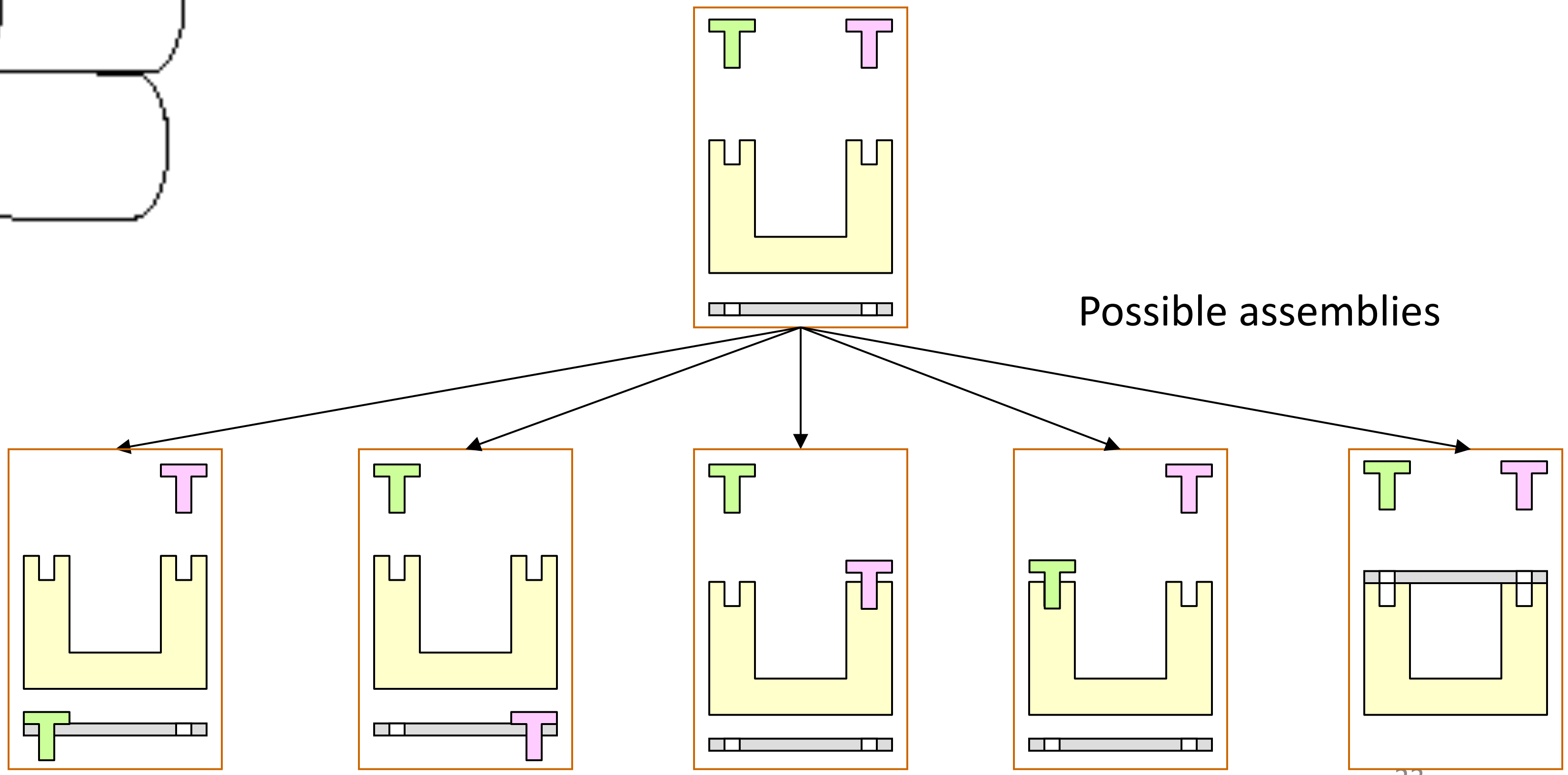


# The Eight Puzzle – State Space (Fragment)



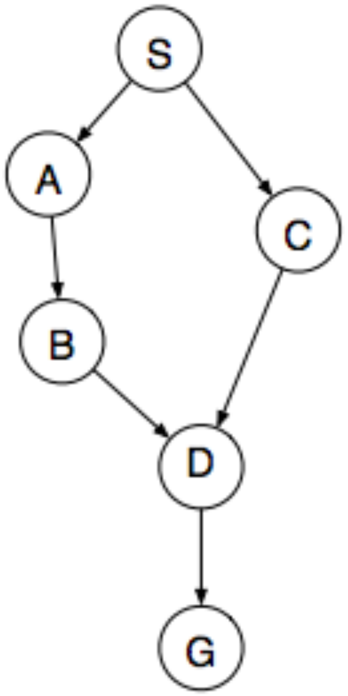
# Examples – Assembly Planning





# Revisiting States

- What if we revisit a state that was already expanded? Redundancy.
- Maintain an explored set (or closed list) to store every expanded node
  - Worst-case time and space requirements are $O(|S|)$ where $|S|$ is the number of states.



# Graph Search

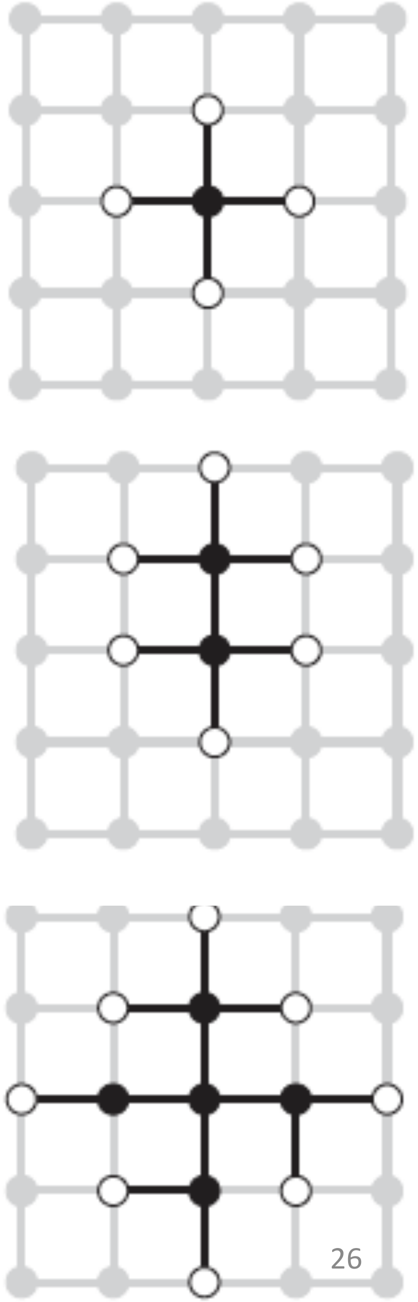
**function** GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  *initialize the explored set to be empty*
  **loop do**
    **if** the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    **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*

Newly generated nodes that match the frontier nodes or expanded nodes are discarded.

# Notion of a Frontier

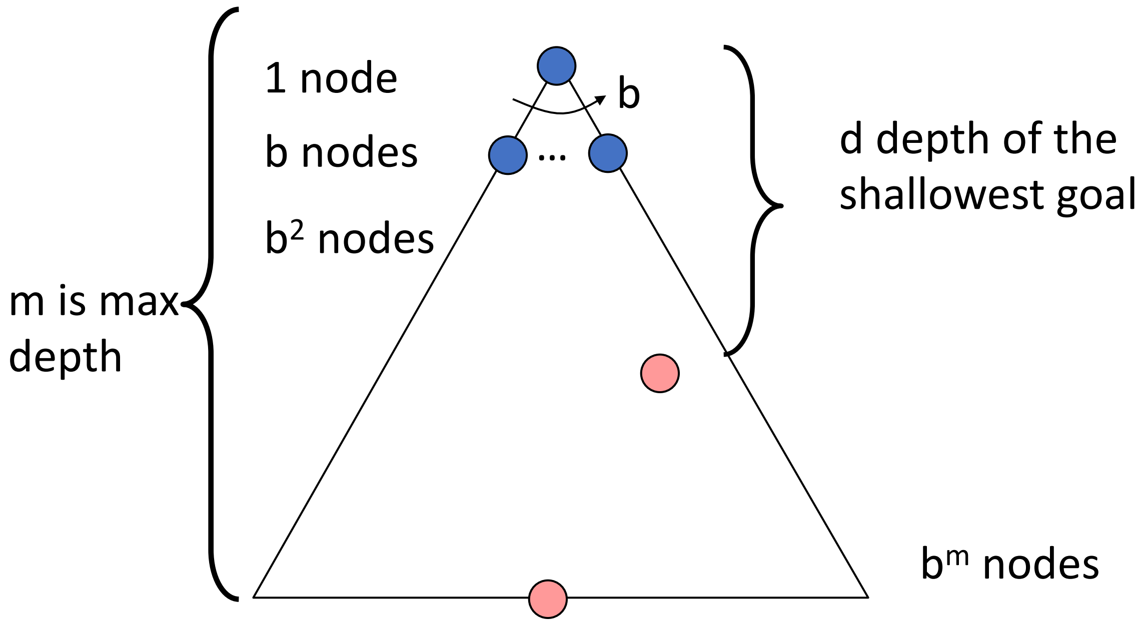
- How to manage generated nodes?
  - Need a data structure for managing nodes as they are generated.
  - Queue (characterized by the order win which they store the inserted nodes).
- Frontier
  - Separates the explored and unexplored nodes.
  - Also called open list
- Search Strategy
  - Search algorithms vary in their “strategy” to decide which nodes to explore?
  - We will see examples soon.

Progression of search



# Measuring problem-solving performance

- Cartoon of search tree:
  - b is the branching factor
  - m is the maximum depth
  - solutions at various depths
  - d is the depth of the shallowest goal node
- Number of nodes in entire tree?
  - $1 + b + b^2 + \dots b^m =$ **$O(b^m)$**
  - Each node can generate the b new nodes



# Properties of Search Algorithms

- Completeness
  - Is the search algorithm guaranteed to find a solution when there is one?
  - Should not happen that there is a solution but the algorithm does not find it (e.g., infinite loop in a part of the state space)
- Optimality
  - Is the plan returned by the  search algorithm the optimal ?
- Time Complexity
  - The number of nodes generated during search.
- Space Complexity
  - The maximum number of nodes stored in memory during search.

# Search Algorithms

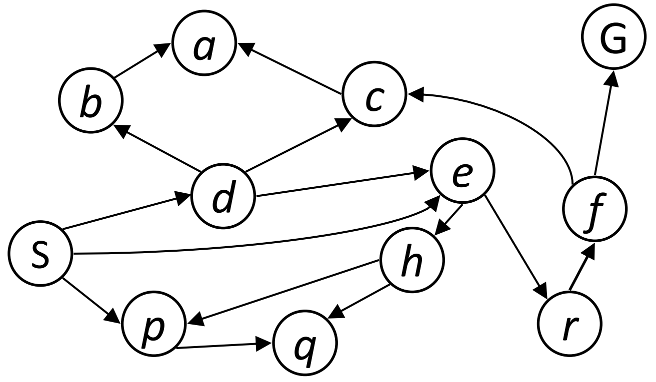
- The strategy for exploration of nodes leads to a variety of search algorithms
- **Uninformed Search**
  - Only use information about the state in the problem definition.
  - Generate successors and distinguish goal states from no-goal states.
- Informed Search
  - Use problem-specific knowledge beyond the problem definition
  - Heuristics for more efficient search

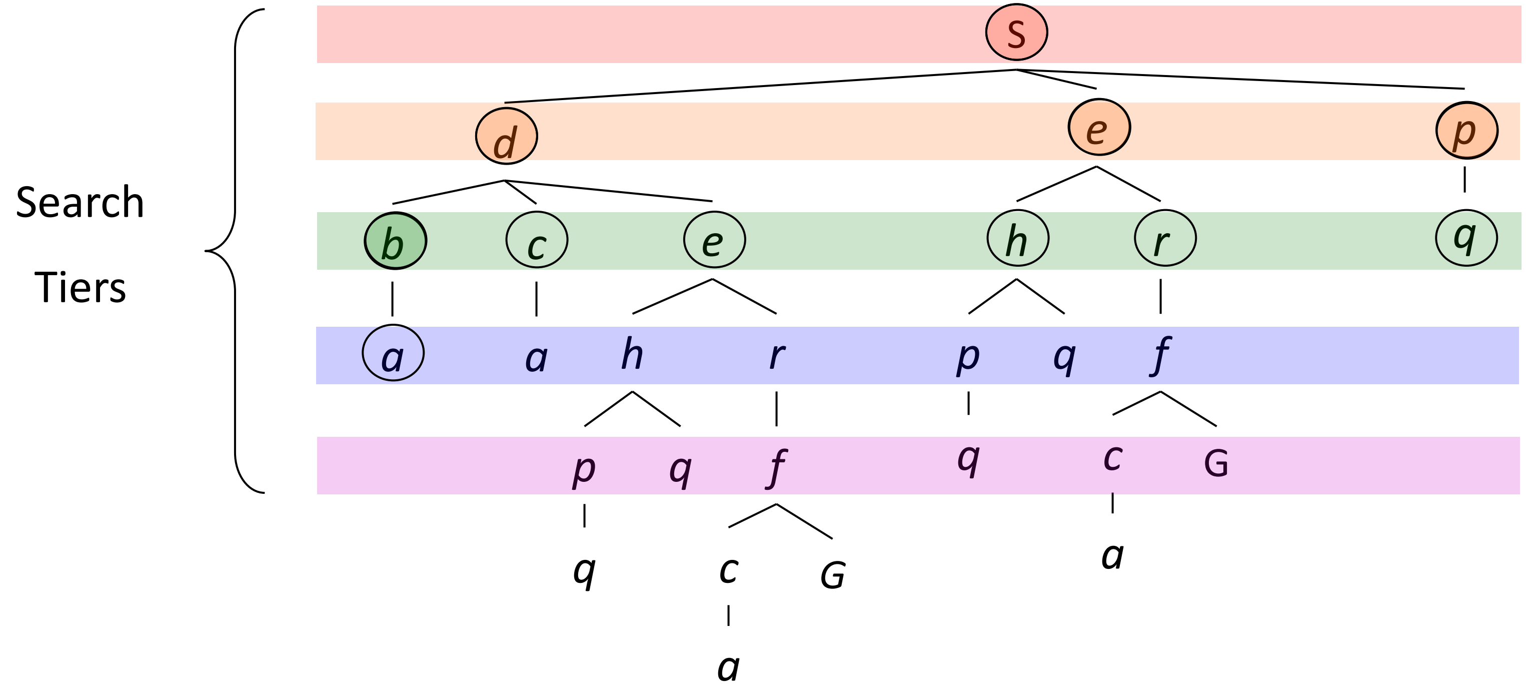
# Breadth-First Search (BFS)

**Strategy: expand a shallowest unexplored node first.**

**All the successors of a node are expanded, then their successors and so on.**

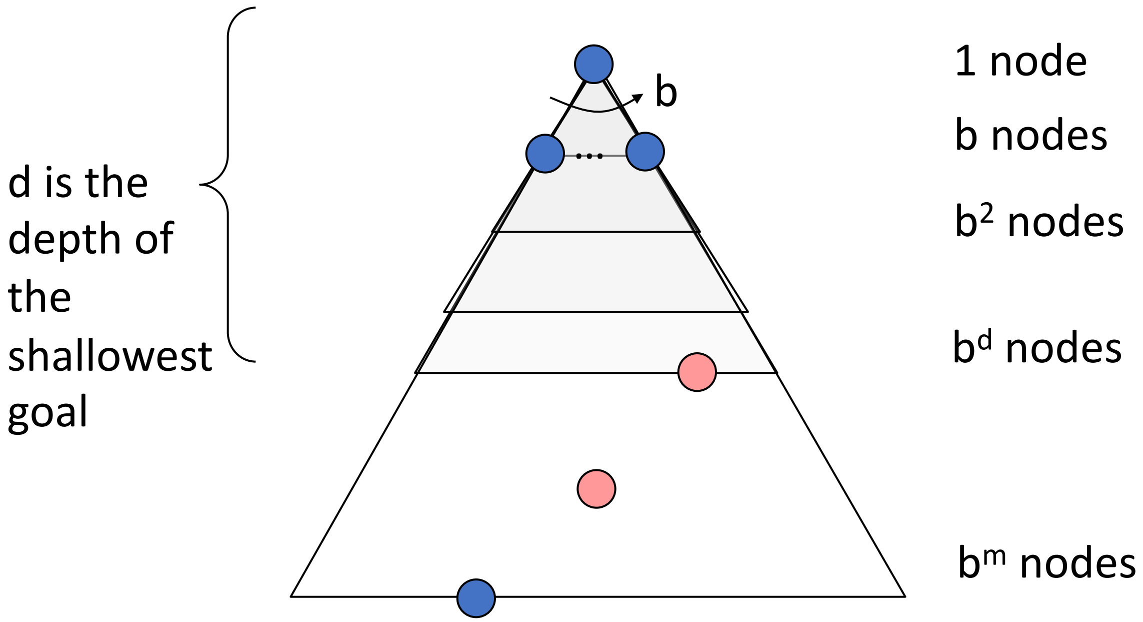
**Implementation: Frontier is a FIFO queue**



# Breadth First Search (BFS) Properties

- **Expansion Strategy**
  - Expands the ***shallowest*** unexplored node in the frontier of the search tree.
- **Time Complexity**
  - Search takes time $\mathbf{O(b^d)}$
- **Space Complexity**
  - Frontier nodes $\mathbf{O(b^d)}$
  - Explored nodes $\mathbf{O(b^{d-1})}$
  - **Memory requirement is a problem.**



# Breadth First Search (BFS) Properties

| Depth | Nodes | Time | Memory |
|---|---|---|---|
| 0 | 1 | 1 millisecond | 100 bytes |
| 2 | 111 | .1 seconds | 11 kilobytes |
| 4 | 11,111 | 11 seconds | 1 megabyte |
| 6 | $10^6$ | 18 minutes | 111 megabytes |
| 8 | $10^8$ | 31 hours | 11 gigabytes |
| 10 | $10^{10}$ | 128 days | 1 terabyte |
| 12 | $10^{12}$ | 35 years | 111 terabytes |
| 14 | $10^{14}$ | 3500 years | 11,111 terabytes |

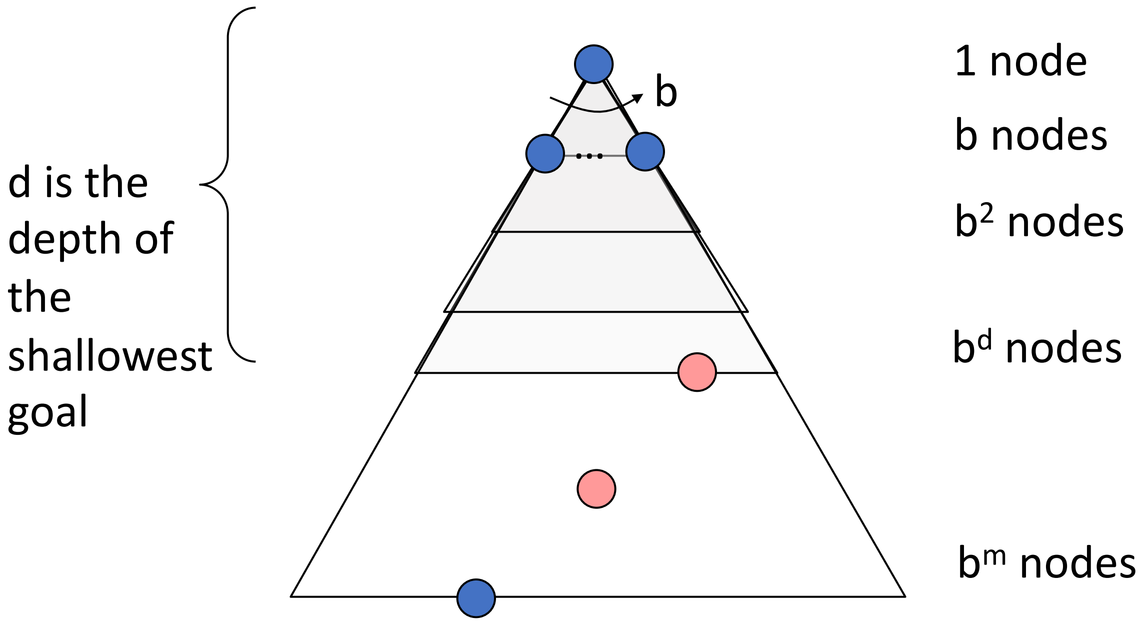
Time and memory requirements for BFS. Branching factor b = 10. 1 million nodes per second and 100 bytes per node.

Memory requirement is a big problem for BFS.

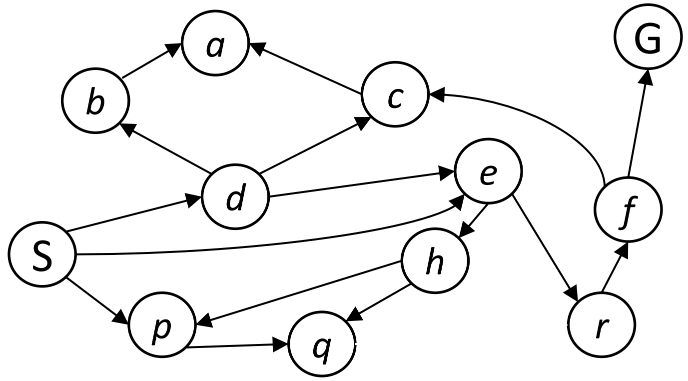
# Breadth First Search (BFS) Properties

- **Is it complete?**
  - **Yes.**
  - The shallowest goal is at a finite depth, d
  - If the branching factor, b, is finite then BFS will find it.

- **Is it optimal?**
  - **Yes.** If the path cost is a non-decreasing function of depth.
    - For example, if all edge costs are equal.



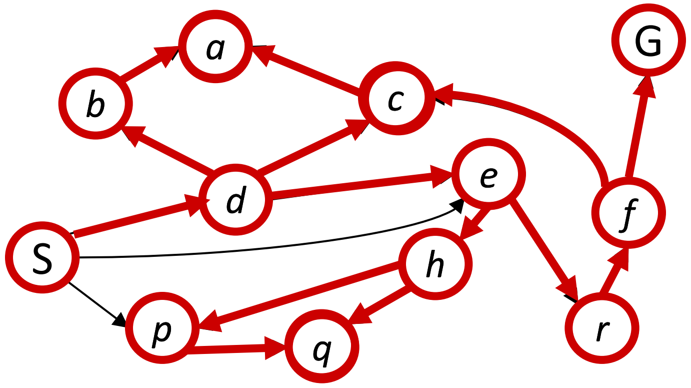
# Depth-First Search (DFS)

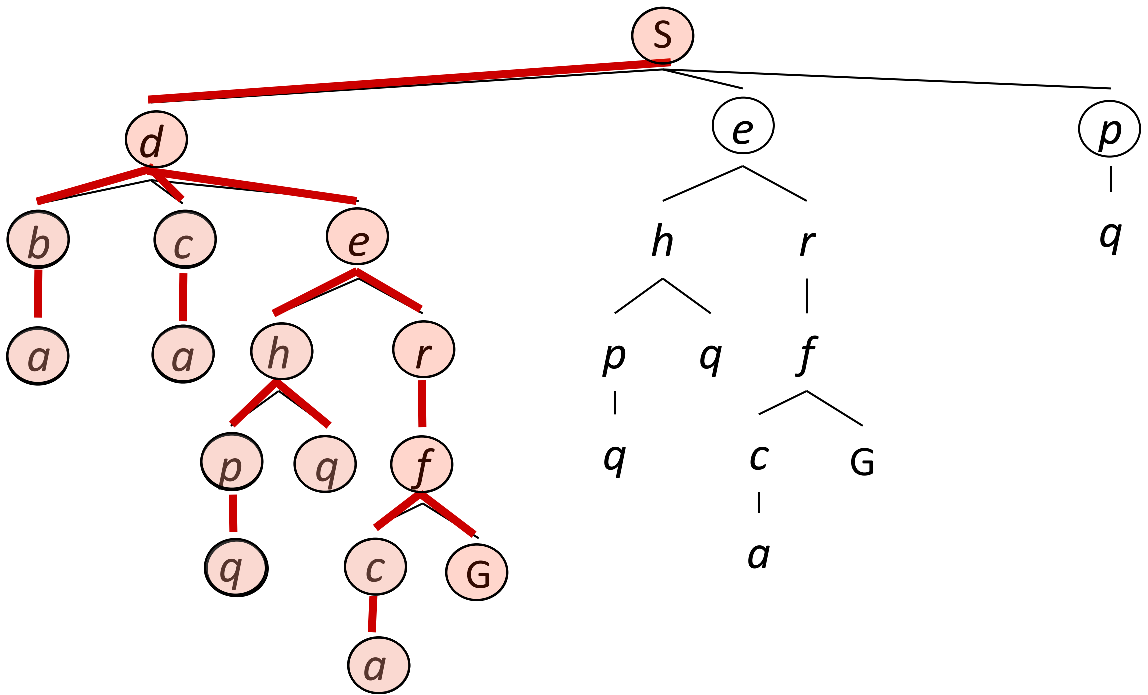


# Depth-First Search (DFS)

**Strategy: expand a deepest node first**

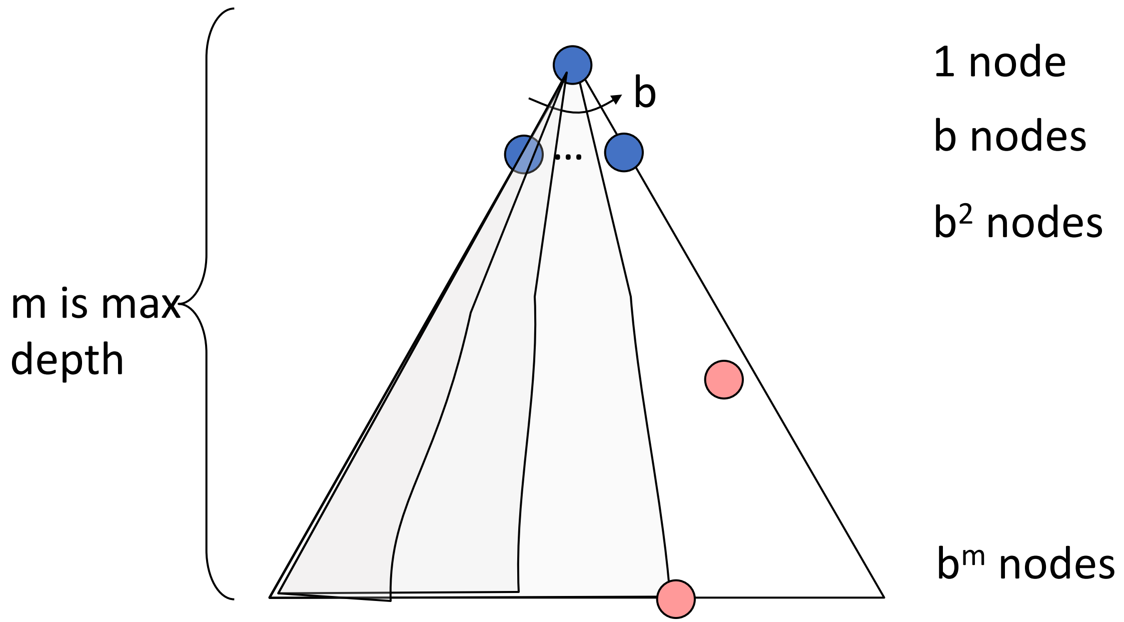
**Implementation: Frontier is a LIFO stack**



# Depth First Search (DFS) Properties

- **Expansion Strategy**
  - Expands the ***deepest*** unexplored node in the frontier of the search tree
- **Time Complexity**
  - Worst case: processes the whole tree.
  - If m is finite, takes time **$O(b^m)$**
- **Space Complexity**
  - Frontier stores:
    - Single path from the root to the leaf node.
    - Sibling nodes on the path that are unexplored.
  - **Memory requirement is low $O(bm)$**
- **Is it complete?**
  - **Yes**, if m is finite. Eventually, finds the path.
- **Is it optimal?**
  - **No**, it finds the "leftmost" solution, regardless of depth or cost




Note: Variant of graph search where the goal test is applied at node generation time. Space saving.

# DFS Variant

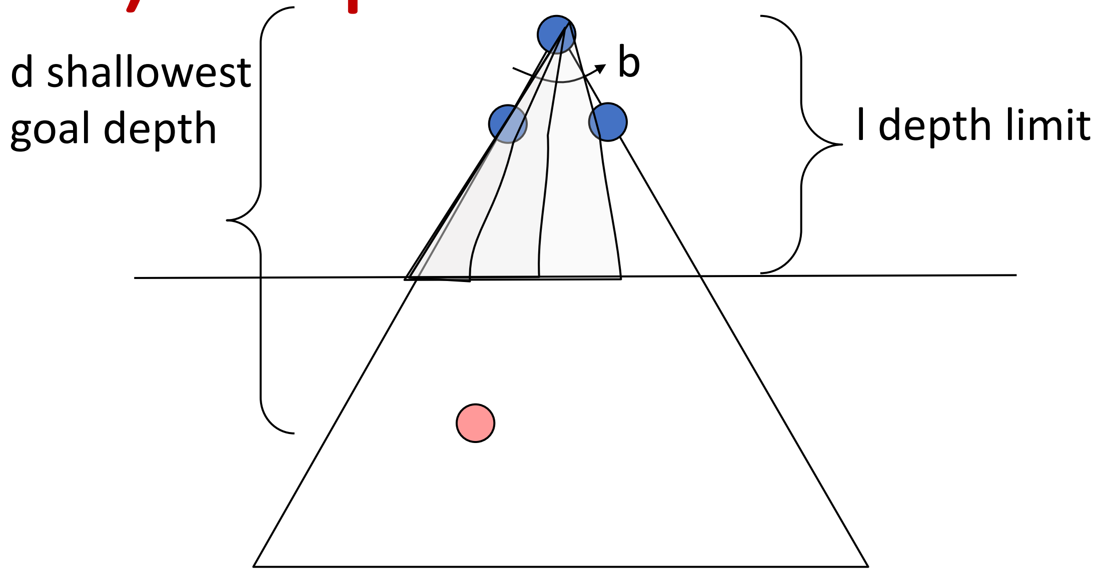
- Reducing DFS memory requirement still further
- Backtracking search
  - Only one successor is generated at a time rather than all successors
  - Each partially expanded node remembers which successor to generate next.
  - Memory saving by modifying the current state description directly rather than copying.

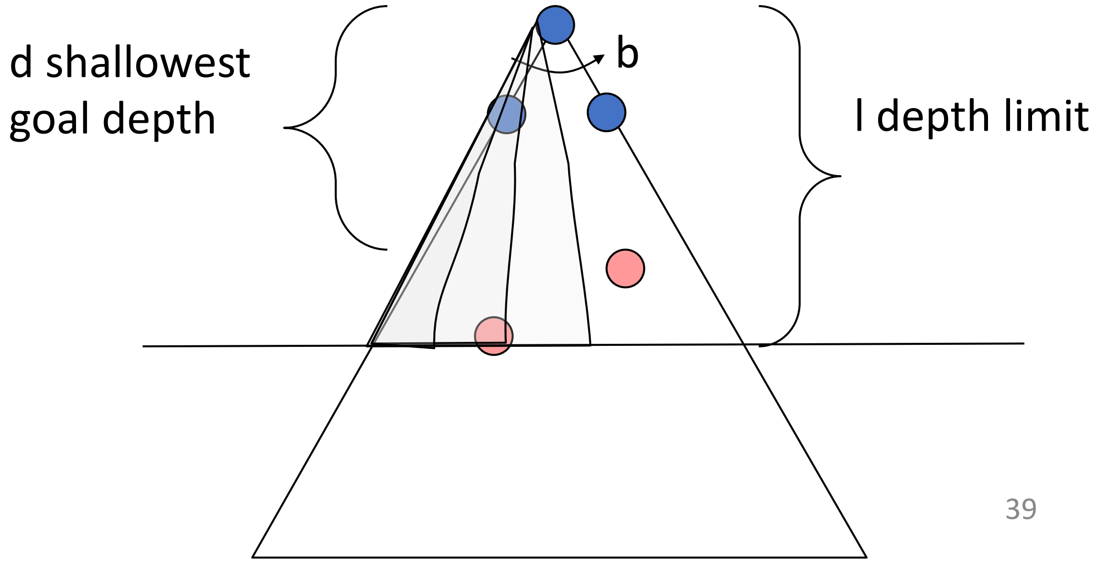
# Depth-Limited Search

- Problem
  - Depth First Search fails when the maximum goal depth is not known ahead of time
- Solution
  - **Depth Limited Search**
    - Restrict the depth of search (supply a depth limit, l)
    - Search depth-first, but terminate a path either if a goal state is found or if the maximum allowed depth is reached.
    - Equivalently, nodes at l have no successors.

# Depth Limited Search (DLS) Properties

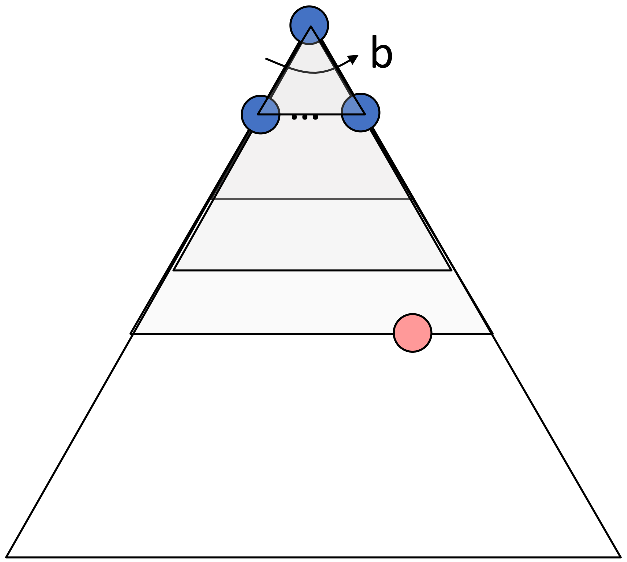
- **Termination**
  - Always terminates.
- **Time Complexity**
  - Worst case: processes the whole tree till l.
  - Time **$O(b^l)$**
- **Space Complexity**
  - Frontier is managed like Depth First Search.
  - Memory **O(bl).**
- **Is it complete?**
  - **Not complete** when goal depth is greater than the limit (d>l)
- **Is it optimal?**
  - **Not optimal** when the limit is greater than the goal depth (l > d)



# Iterative Deepening Search

- Combine DFS's space advantage with BFS's shallow-solution advantages
    - Run a DLS with depth limit 1.  If no solution…
    - Run a DLS with depth limit 2.  If no solution…
    - Run a DLS with depth limit 3.  …..



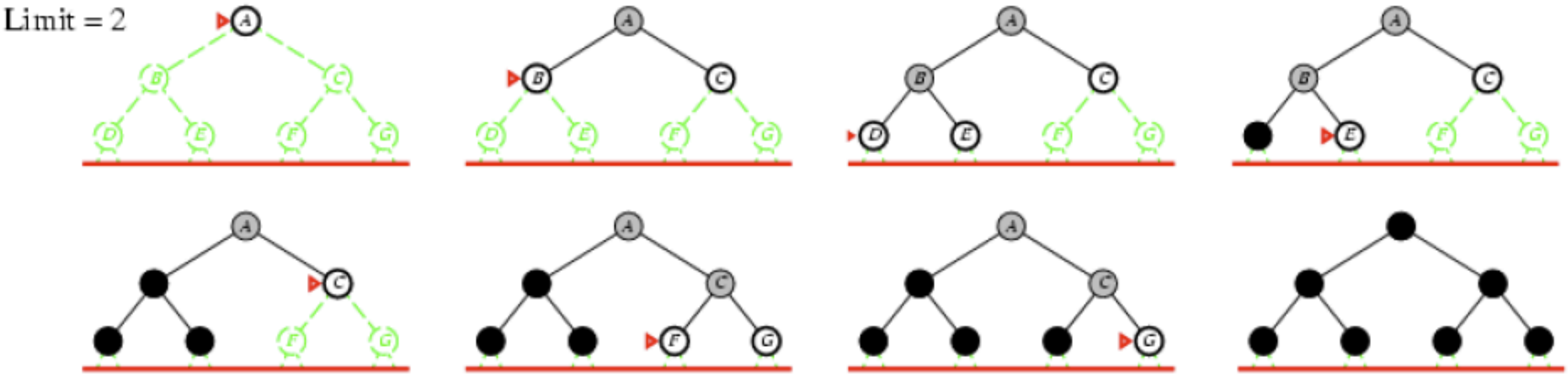

**function** ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  **for** *depth* = 0 **to** $\infty$ **do**
    *result* ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    **if** *result* ≠ cutoff **then return** *result*

# Iterative Deepening: Example




Adapted from Prof. Mausam's slide

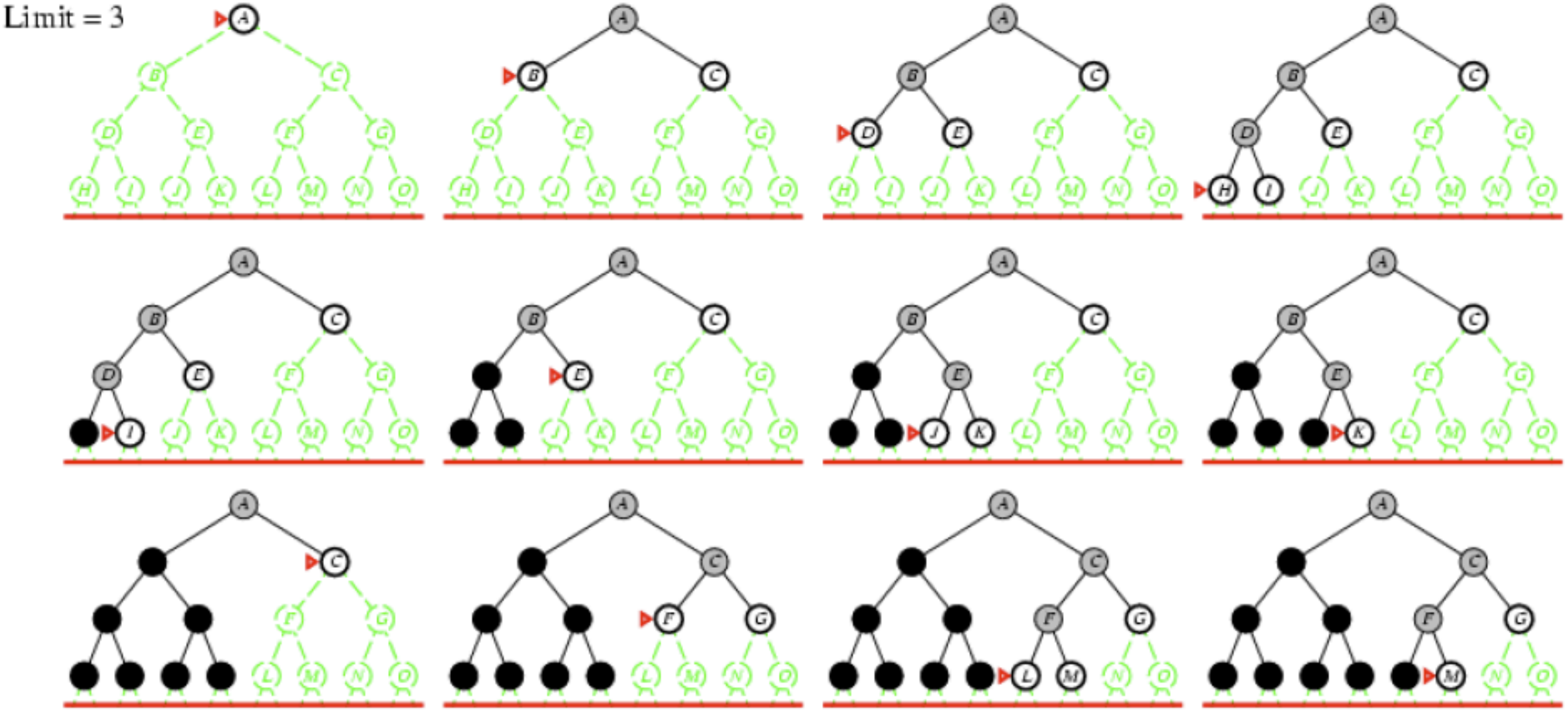
# Iterative Deepening: Example






Adapted from Prof. Mausam's slide

# Iterative Deepening: Example






Adapted from Prof. Mausam’s slide

# Iterative Deepening: Properties

- Is it wasteful to generate nodes again and again?
  - Not really!
  - The lowest level contributes the maximum. Overhead is not significant in practice.
- Asymptotic time complexity is same as BFS: **$O(b^d)$**

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:
  - $N_{DLS} = b^0 + b^1 + b^2 + ... + b^{d-2} + b^{d-1} + b^d$
- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:
  - $N_{IDS} = (d+1)b^0 + d\ b^{\wedge 1} + (d-1)b^{\wedge 2} + ... + 3b^{d-2} + 2b^{d-1} + 1b^d$

No. of times generated.

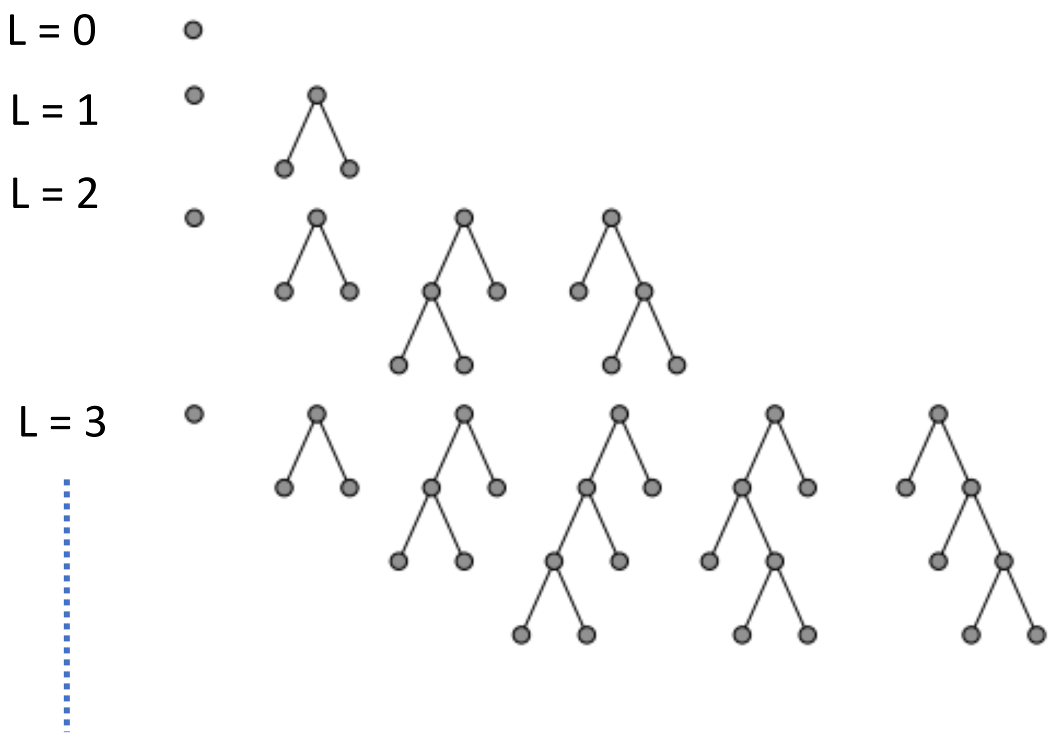
- Asymptotic ratio: (b+1)/(b-1)
- For *b = 10, d = 5,*
-
  - $N_{DLS}$ = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111
  -
  - $N_{IDS}$ = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456
  -
- Overhead = (123,456 - 111,111)/111,111 = 11%



Adapted from Prof. Mausam's slide

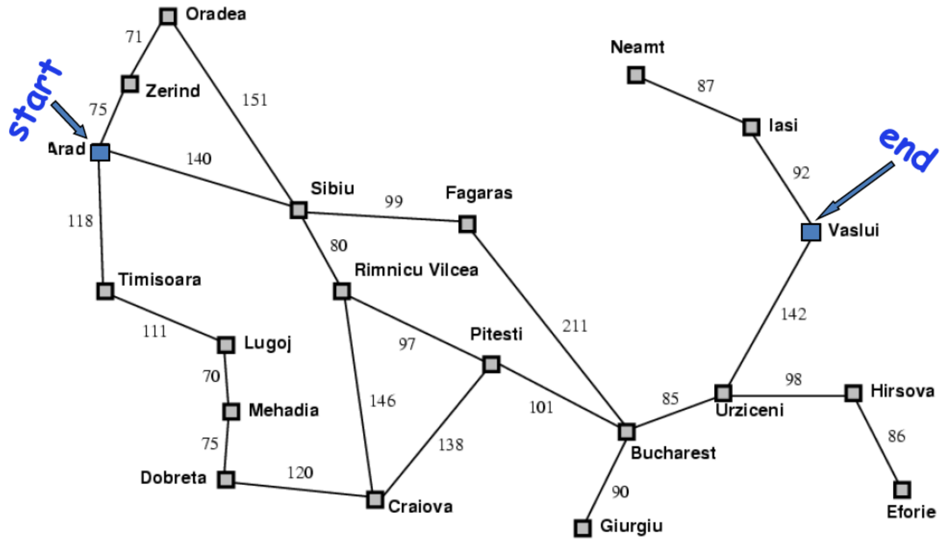
# Iterative Deepening Properties

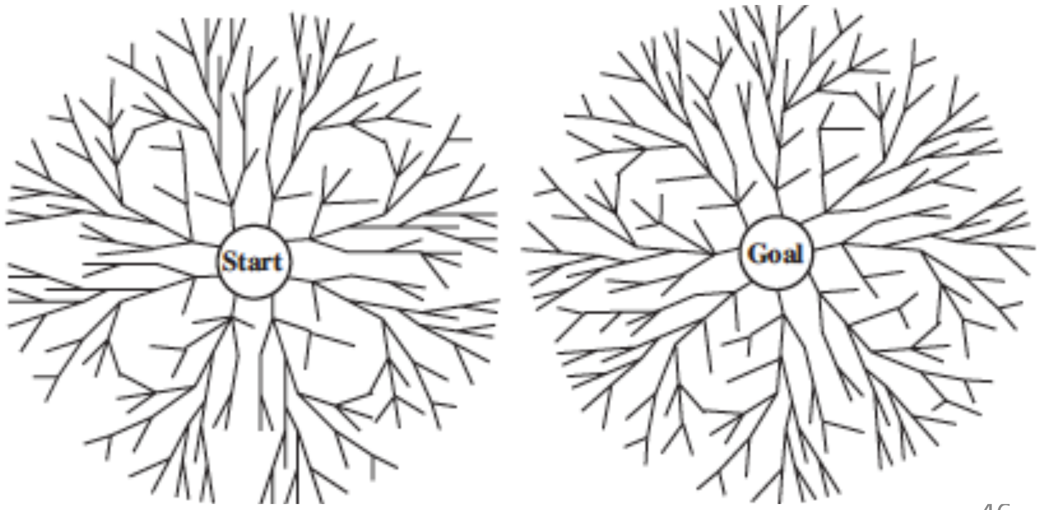
- **Time Complexity**
  - Time **O($b^d$)**
- **Space Complexity**
  - Memory **O(bd)**
  - Linear memory requirement like **DFS**
- **Is it complete?**
  - **Yes.** Complete like **BFS**
- **Is it optimal?**
  - **Yes.** Optimal like BFS (if costs are non-decreasing function of path length)
- **Relevance**
  - Preferred method for large state spaces where maximum depth of a solution is unknown



# Bi-directional Search

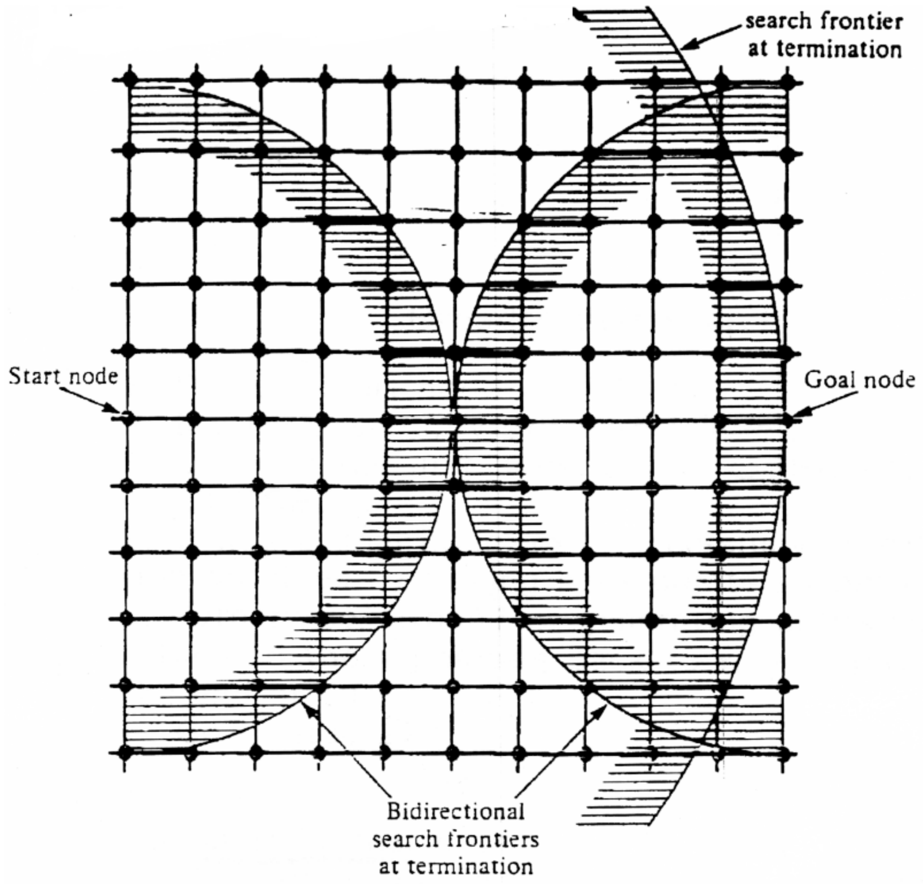
- Run one search forward from the initial state.
- Run another search backward from the goal.
- Stop when the two searches meet in the middle.



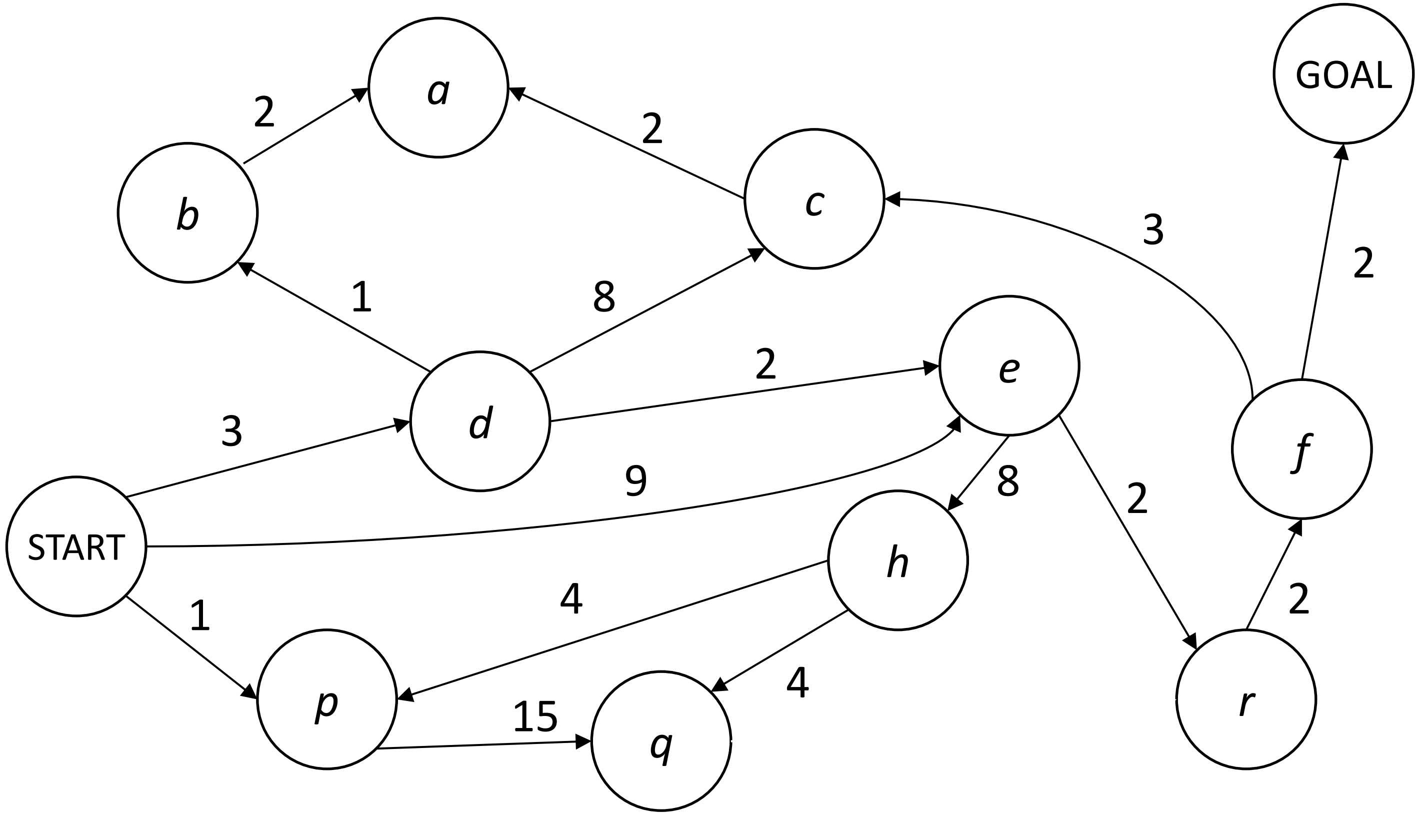
# Bi-directional Search

- Space and time complexity
  - **$O(b^{d/2})$**
  - $b^{d/2} + b^{d/2}$ is smaller than $b^d$
  - $10^{8}+10^{8} = 2.10^{8} << 10^{16}$
- Needs an efficiently computable *Predecessor()* function
  - Difficult: e.g., predecessors of checkmate in chess?
- What if there are several goal states?
  - Create a new dummy goal state whose predecessors are the actual goal states.



# Cost-insensitive vs Cost-sensitive Search

- Till now, the cost on the edges was not taken into account.
  - Worked with solution depths.
- Shortest paths (plans) were found in terms of the number of actions.
  - Did not find the least-cost path.
  - BFS, DFS ....

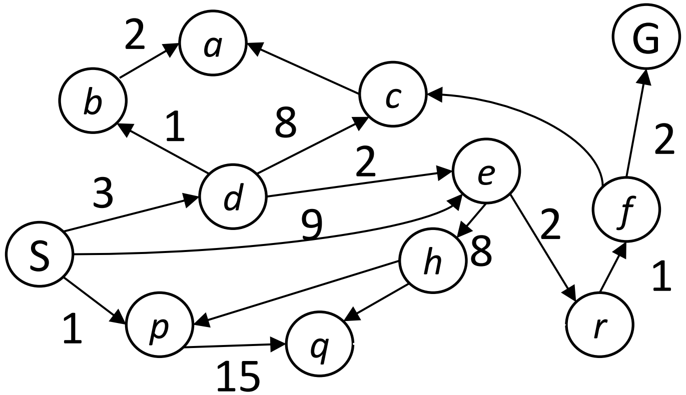


# Uniform Cost Search (UCS)

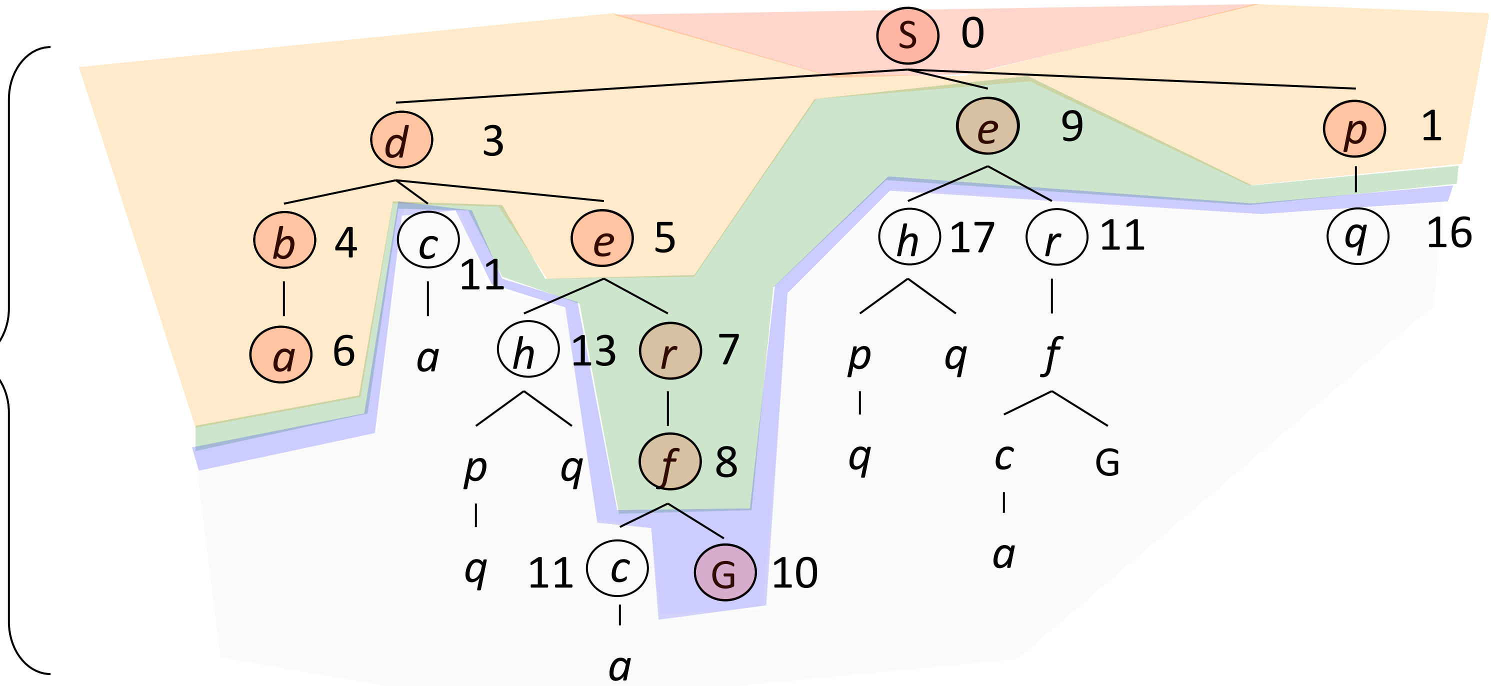
Intuition, the low-cost plans should be pursued first. Prioritise them.

Strategy: expand a cheapest node first:

**Frontier is a priority queue (priority: cumulative cost)**




Cost contours



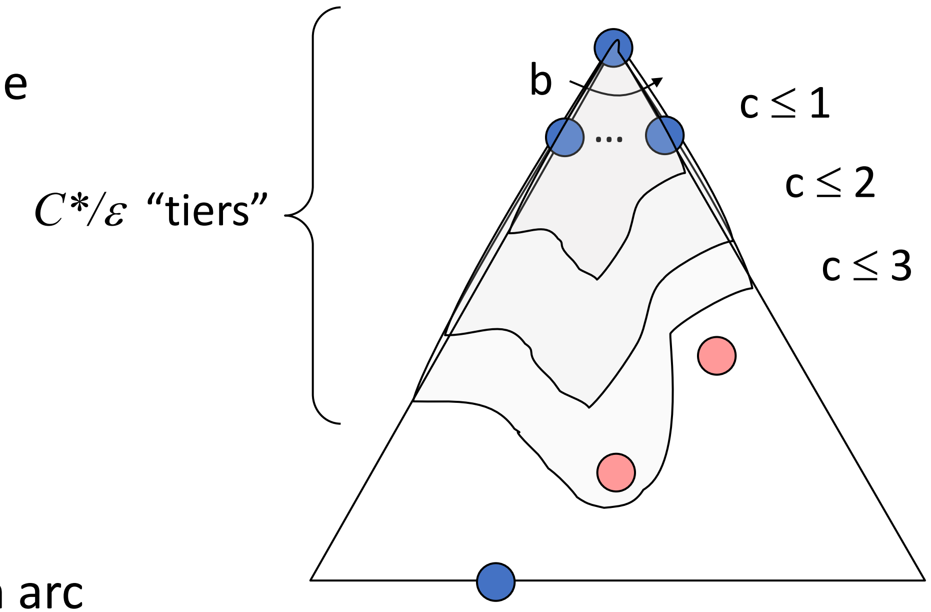
# Uniform Cost Search (UCS)

- The first goal node generated may be on the sub-optimal path.
- If the new path is better than the old path, then discard the old one.

```
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
     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
```

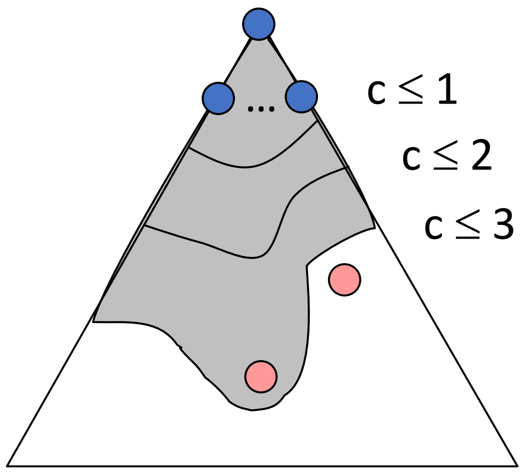
# Uniform Cost Search (UCS) Properties

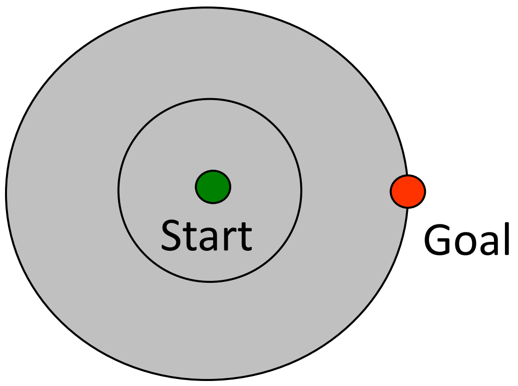
- What nodes does UCS expand?
  - Guided by costs and not the depth.
  - If that solution costs $C^*$ and action cost at least $\varepsilon$, then the “effective depth” is roughly $C^*/\varepsilon$
  - Takes time **O(b$^{C^*/\varepsilon}$)** (exponential in effective depth)
- How much space does the frontier take?
  - **O(b$^{C^*/\varepsilon}$)**
- Is it complete?
  - **Yes.** Assuming best solution has a finite cost and minimum arc cost is positive, yes!
- Is it optimal?
  - **Yes.** (Proof via contradiction)



# Uniform Cost Search (UCS) Issues

- UCS explores increasing cost contours
- Explores options in every “direction”
- **Does not use the goal information.**



# Beam Search

- Bound the *maximum size* of the frontier.
- Only keep the *k* best candidates for expansion, discard the rest.
- Advantage:
  - More space efficient
- Disadvantage
  - May throw away a node that is on the solution path
- Complete? **No.**
- Optimal? **No.**
- Very popular in practice

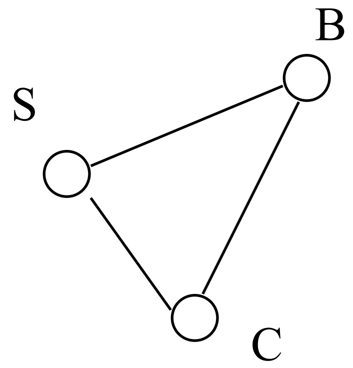
# Summary Table

| 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 |

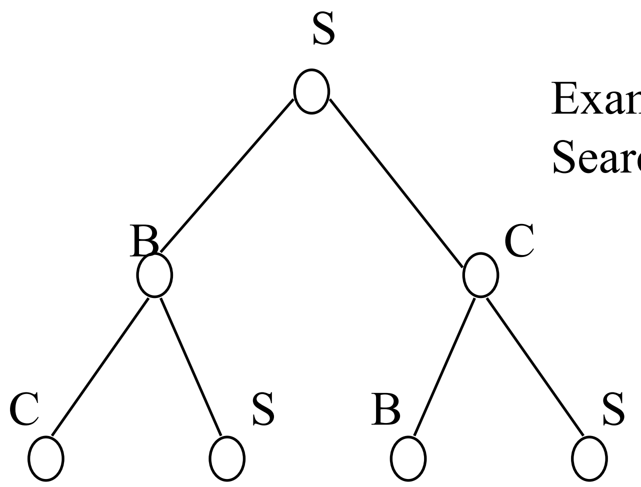
- The comparison is for the tree-search version of the algorithms.
- For graph searches, DFS is complete for finite state spaces and that the space time complexities are bounded by the size of the state space.

# Repeated States

**Route traversal problem**

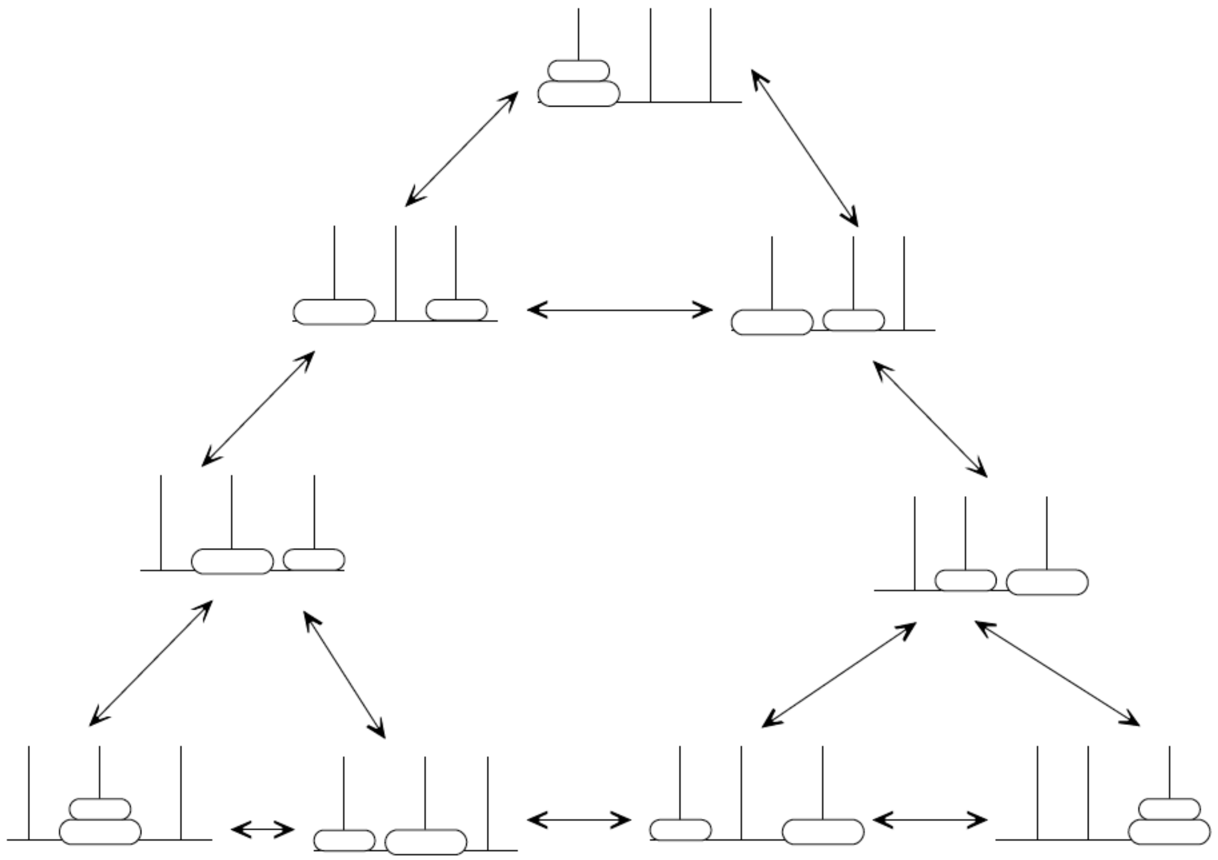



State Space




Example of a Search Tree

**Towers of Hanoi problem**



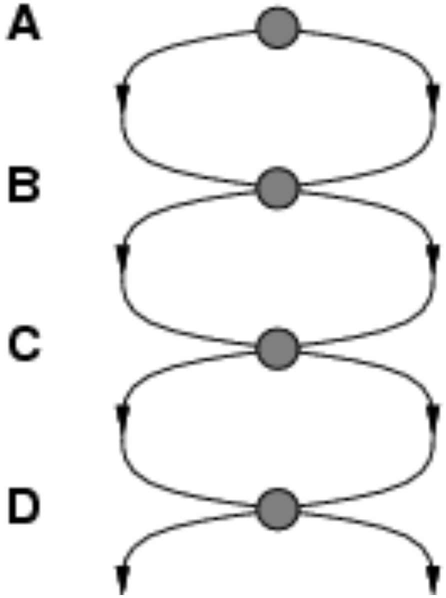
# Repeated States

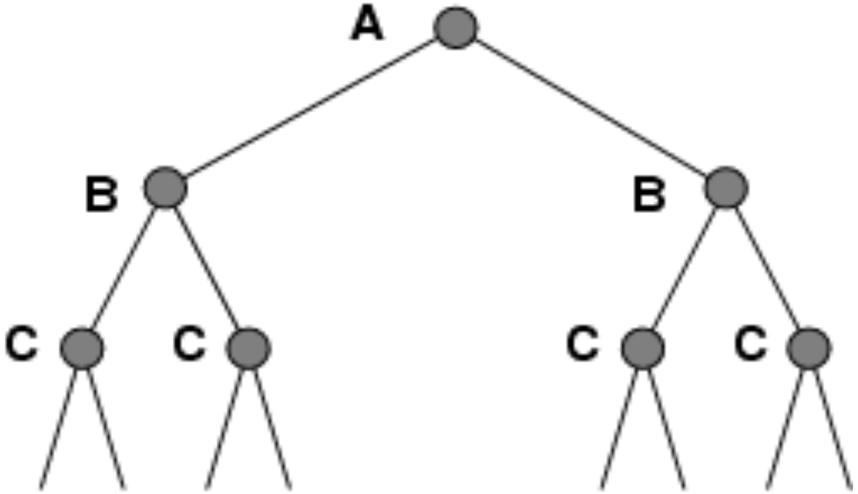
- Reversible actions can lead to repeated states.
  - Reversible actions, e.g., the 8 puzzle or route finding.
- Lead to loopy or redundant* paths in the tree search.

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  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
      expand the chosen node, adding the resulting nodes to the frontier
```

* Each additional path found is going to be longer.

# Importance of detecting repeated states




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

# Solutions to Repeated States – I(a)

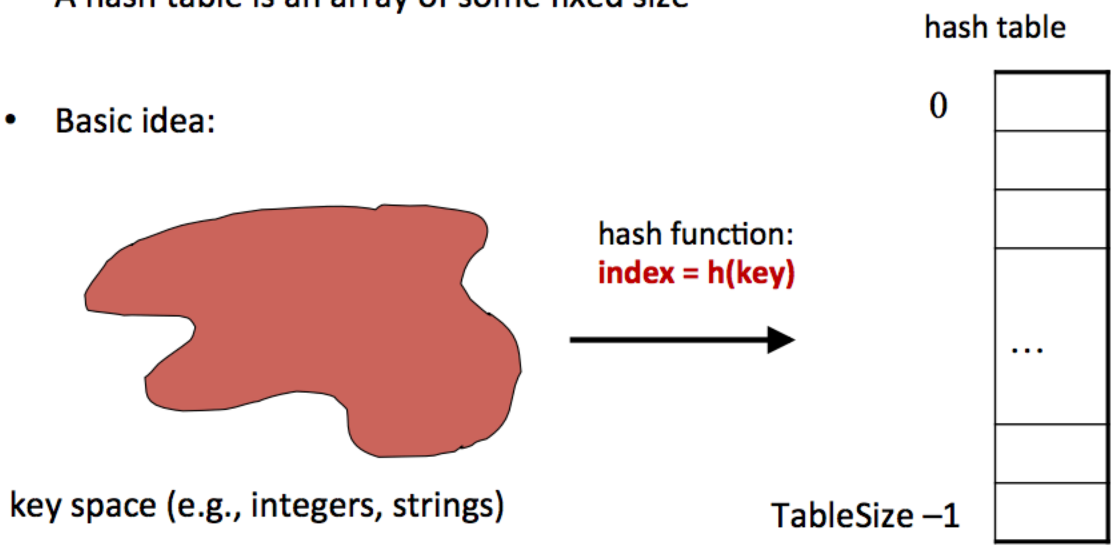
- **Never generate states that have already been generated before.**
- Maintain an explored list (Graph search)
- **Optimal but memory inefficient**
  - Exponential number of nodes
  - e.g., 8-puzzle problem, we have 9! = 362,880 states

```
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
```

# Approaches to handling repeated states – I(b)

- **Use efficient data structures to keep the explored nodes.**
- Hash Tables
  - Insertion and look up in constant time.
- Duplicate checking – adds time
  - Canonical form, sorted list or other efficient methods.

- Aim for constant-time (i.e., $O(1)$) **`find`**, **`insert`**, and **`delete`**
  - "On average" under some often-reasonable assumptions
- A hash table is an array of some fixed size
- Basic idea:



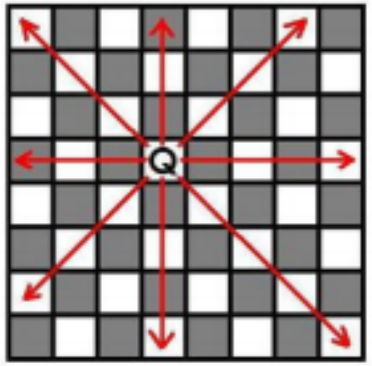
# Approaches to handling repeated states – II

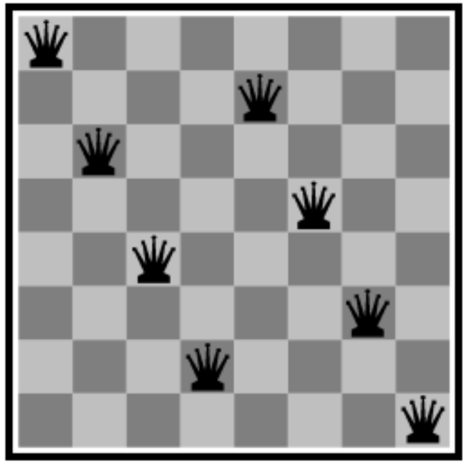
- **Never return to the state you have just come from**
  - Prevent the node expansion function from generating any node successor that is the same state as the node's parent.
- **Never create search paths with cycles in them**
  - The node expansion function must be prevented from generating any node successor that is the same state as any of the node's ancestors
- **Sub-optimal but practical**

# Examples – The 8-Queens Problem

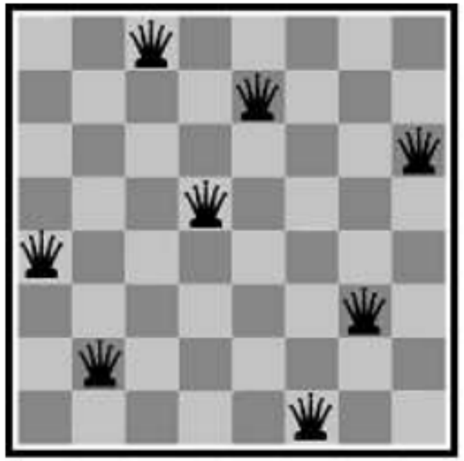
- Find a placement of 8 queens on a chessboard so that no queen can capture another queen.

- A queen can move any number of spaces horizontally, vertically, or diagonally



Almost a solution



A solution

# Examples – The 8-Queens Problem

- Incremental Problem Formulation

  - **States:** Any arrangement of 0 to 8 queens on the board is a state.
  - **Initial state:** No queens on the board.
  - **Actions:** Add a queen to any empty square.
  - **Transition model:** Returns the board with a queen added to the specified square.
  - **Goal test:** 8 queens are on the board, none attacked.

- Possible sequences to investigate? $64 \cdot 63 \cdots 57 \approx 1.8 \times 10^{14}$
- Many actions will violate the problem constraints.

# Examples – 8 Queens Problem

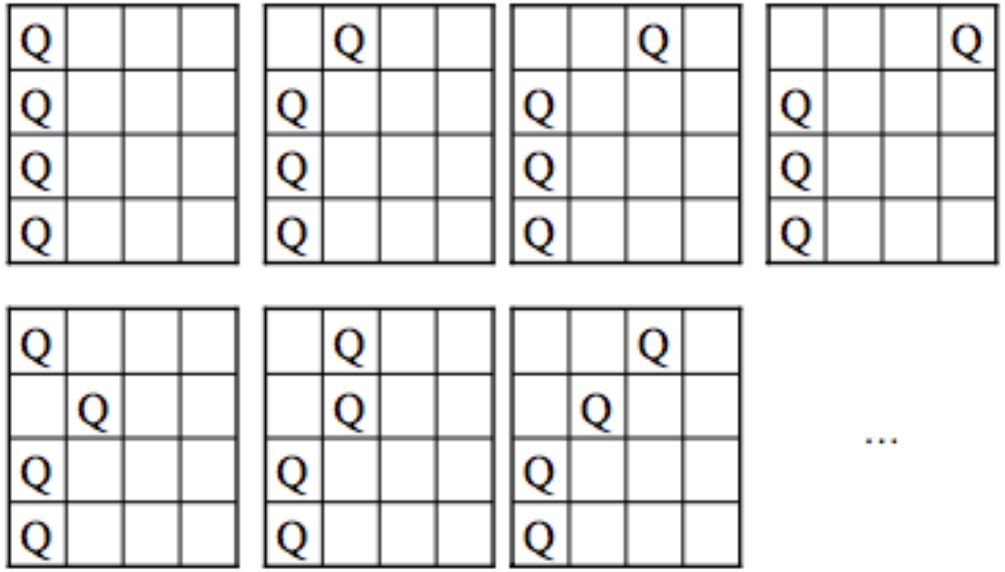
- Incremental formulation (alternative)

  - **States:** All possible arrangements of $n$ queens ($0 \leq n \leq 8$), one per column in the leftmost $n$ columns, with no queen attacking another.
  - **Actions:** Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

- Possible sequences to investigate?
  - 2,057 for the 8-queen state space.
- For 100 queens:
  - A reduction from $10^{400}$ to $10^{52}$
- Constraint is modeled while checking the applicability of an action on a state.

# Examples – The 8-Queens Problem

- Complete state formulation



State includes the all the queens on the board.
Action move the queens to empty squares.

- For 4-Queens there are 256 different configurations.
- For 8-Queens there are 16,777,216 configurations.
- For 16-Queens there are 18,446,744,073,709,551,616 configurations.
- This would take about 12,000 years on a fast modern machine.
- In general we have $N^N$ configurations for N-Queens.

# Uninformed Search: Summary

- Algorithms that are given no information about the problem other than its definition.
  - No additional information about the state beyond that provided in the problem definition.
  - Generate successors and distinguish goal from non-goal states.
  - Hence, all we can do is move systematically between states until we stumble on a goal
  - Search methods are distinguished by the order in which the nodes are expanded.
- *Next time: Informed (heuristic) search uses a guess on how close to the goal a state might be*

# Search Algorithms

- The strategy for exploration of nodes leads to a variety of search algorithms
- Uninformed Search
  - Only use information about the state in the problem definition.
  - Generate successors and distinguish goal states from no-goal states.
- **Informed Search**
  - Use problem-specific knowledge beyond the problem definition
  - Heuristics for more efficient search

# Summary

## This Module

- **Uninformed Search**
    - Depth-First Search
    - Breadth-First Search
    - Depth-limited Search
    - Iterative Deepening
    - Bi-directional Search
    - Uniform Cost search
    - Beam Search

## Next Module

- **Informed Search**