



# COL333/671: Introduction to AI

Semester I, 2024-25

## Solving Problems by Searching

### Uninformed Search

Rohan Paul

# This Class

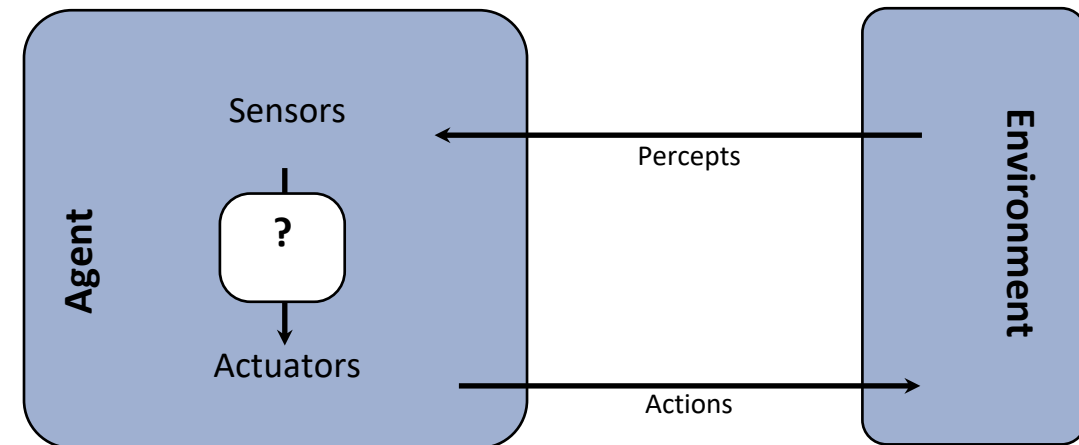
- Reflex Agents
- Problem Solving as search
  - Uninformed Search
- Reference Material
  - AIMA Ch. 3

# 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.**

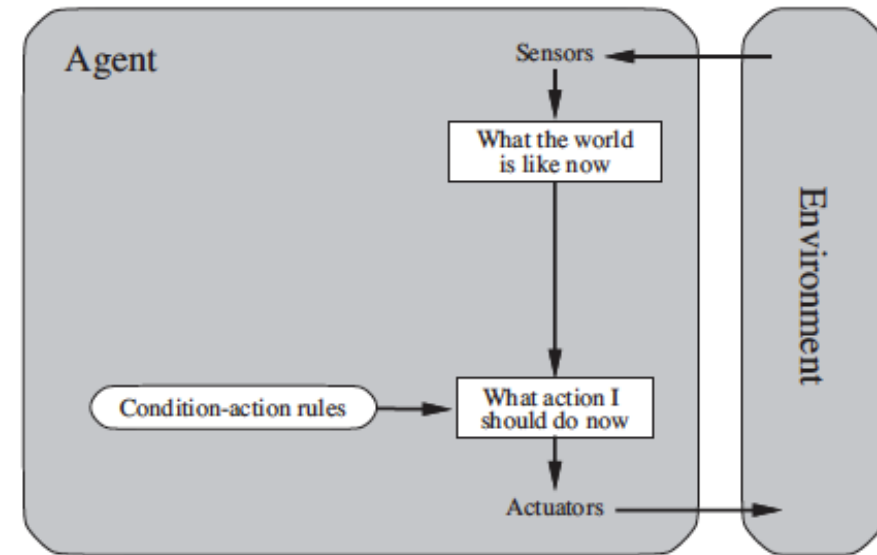
# Last time: Agent View of AI

- 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
  - .....



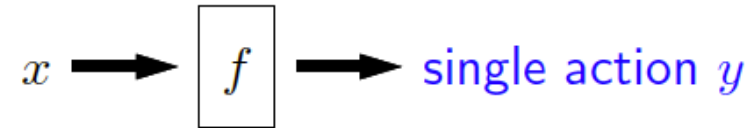
# Simple Reflex Agents

- 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.

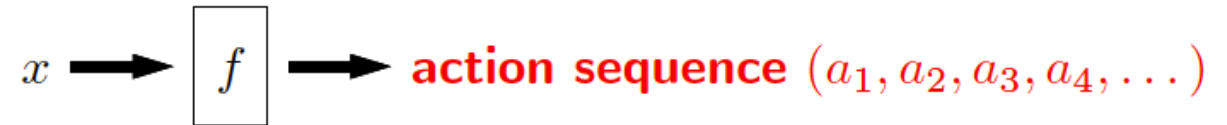


# From Reflex to Problem Solving Agents

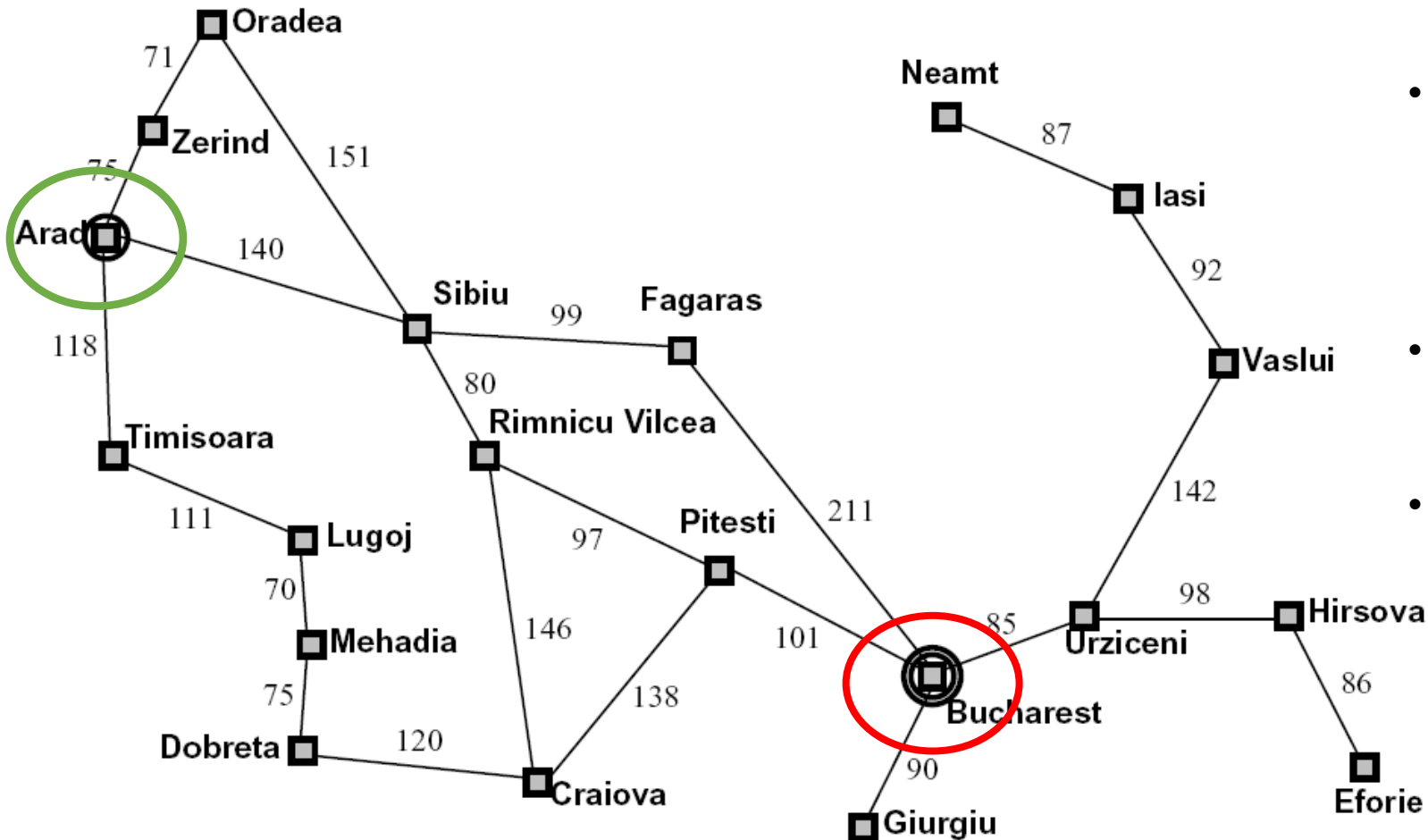
- Reflex agents
  - Directly map states to actions.
  - No notion of goals. Do not consider action consequences.



- 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.



# 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.

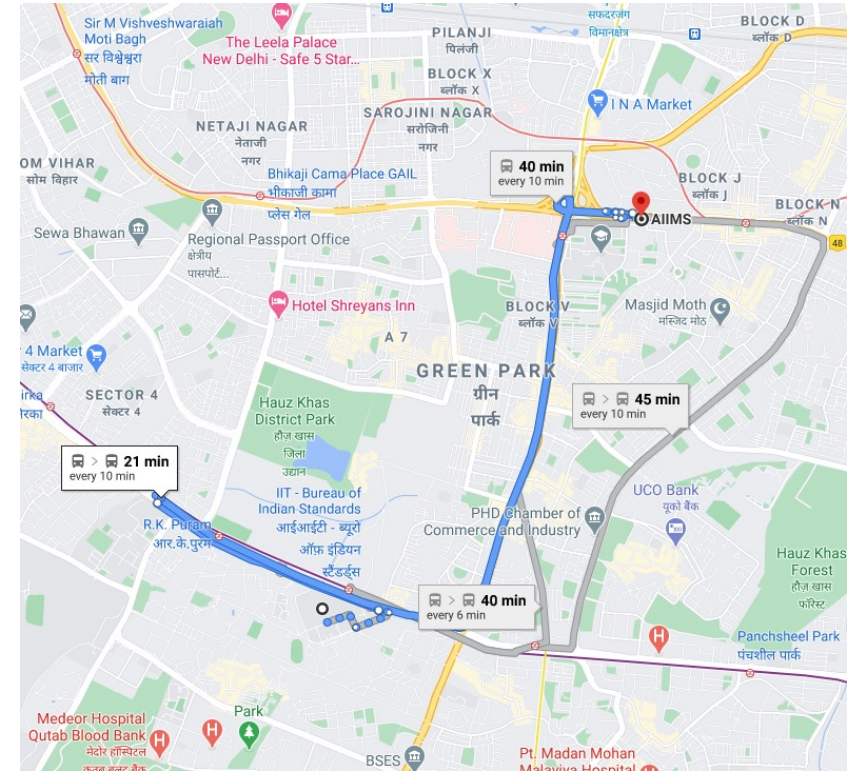
- *State space*  $S$ : all possible configurations of the domain of interest
- *An initial (start) state*  $s_0 \in S$
- *Goal states*  $G \subset S$ : the set of end states
  - Often defined by a *goal test* rather than enumerating a set of states
- *Operators*  $A$ : the actions available
  - Often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

Transition model or successor function



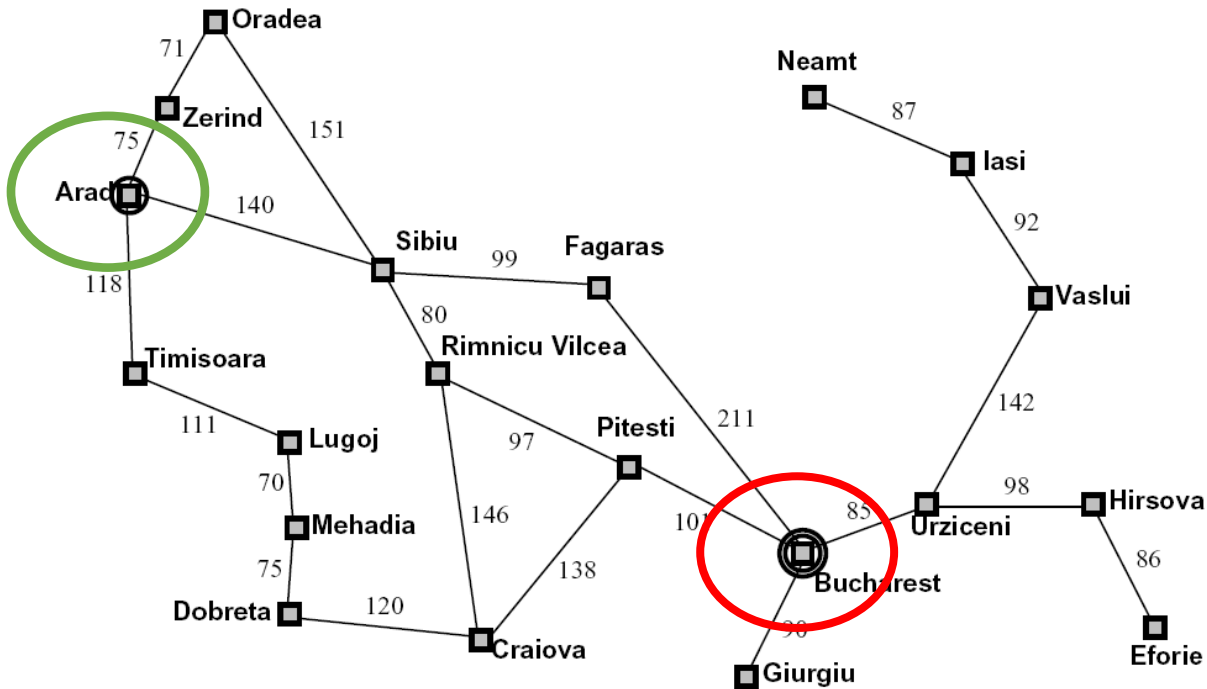
# Formulating a Search Problem

- **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 its 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 states are reached by each action.
  - Action has exactly one outcome when applied to a state.

# Example – The Eight Puzzle

5	4	
6	1	8
7	3	2

Start State

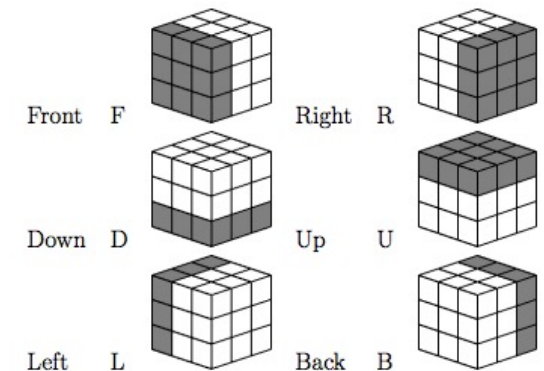
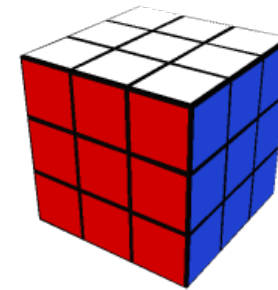
1	2	3
8		4
7	6	5

Goal State

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

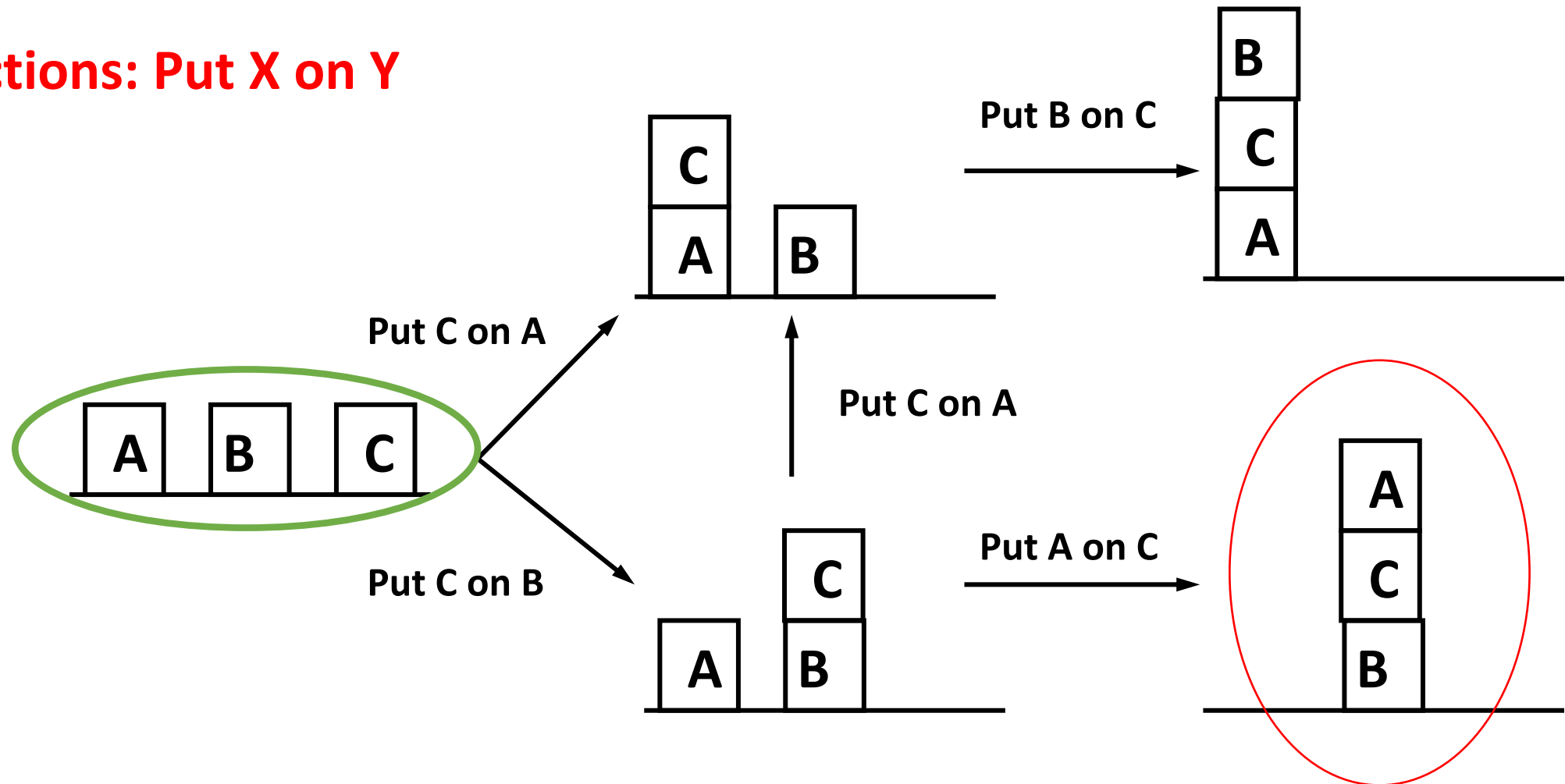
# Example: Rubric's Cube

- Setup
  - Single goal state (correction)
  - Large number of possible states ( $4.3 \times 10^{19}$  states).
  - Randomly exploring is unlikely to give a solution.
  - Different ways to represent actions.
    - <https://solvethecube.com/notation>
  - Modern attempts at solving (advanced techniques)
    - <https://www.nature.com/articles/s42256-019-0070-z>
    - Combination of classical search and learning based approaches.
  - There is fundamental symmetries and structure



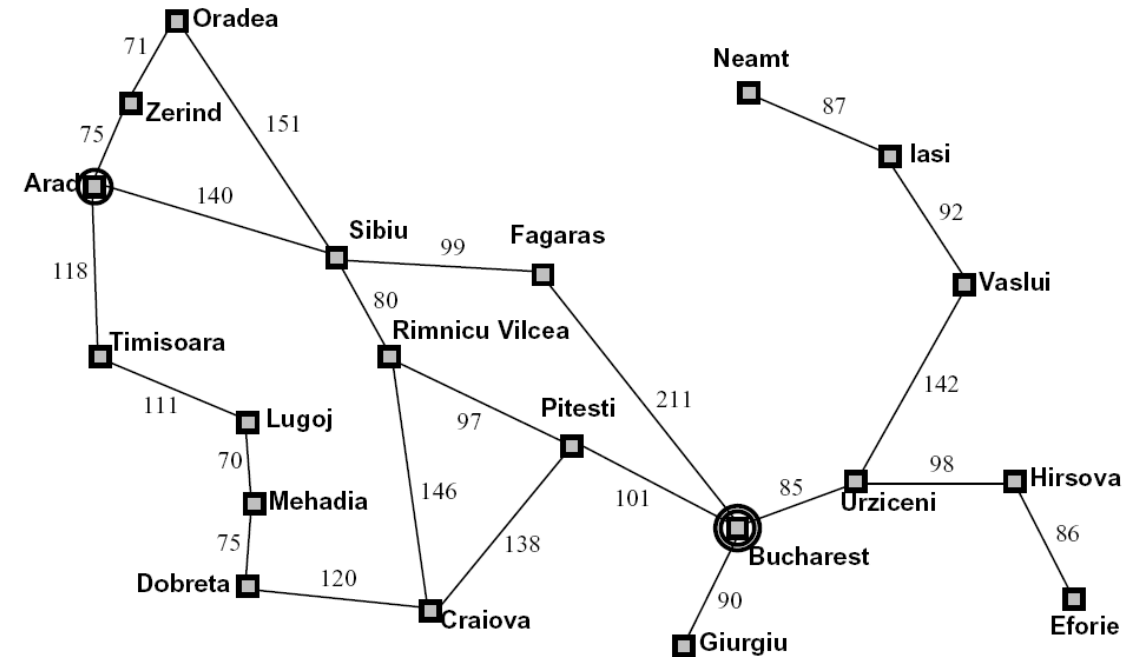
# Example – Block Manipulation

**Actions: Put X on Y**



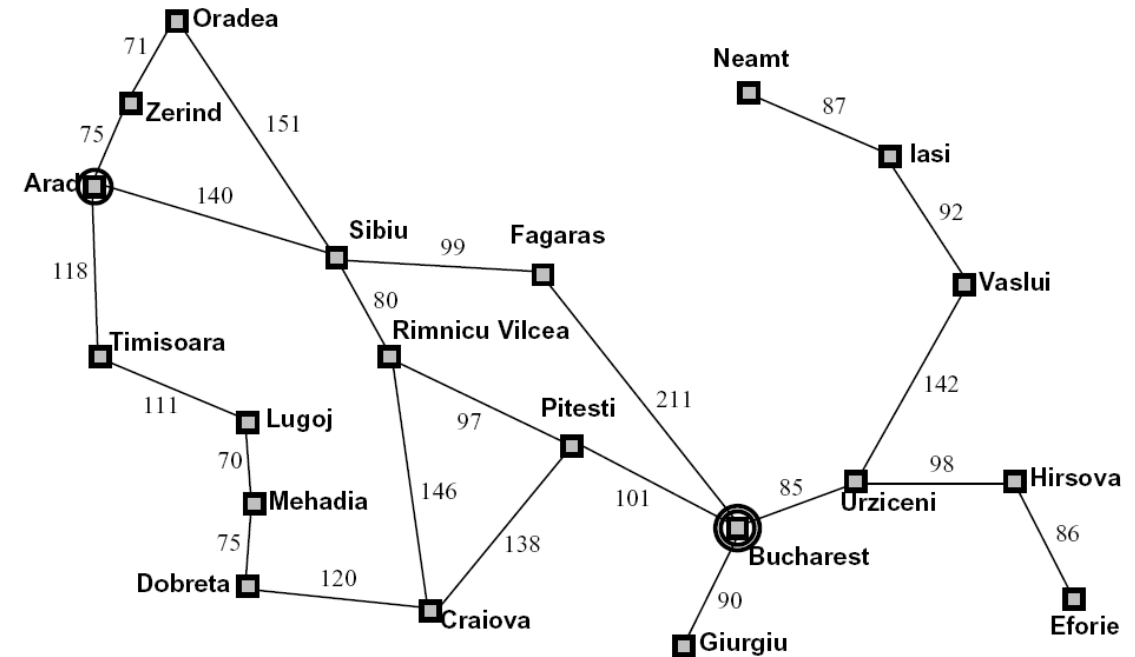
# State Space Graphs

- 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.



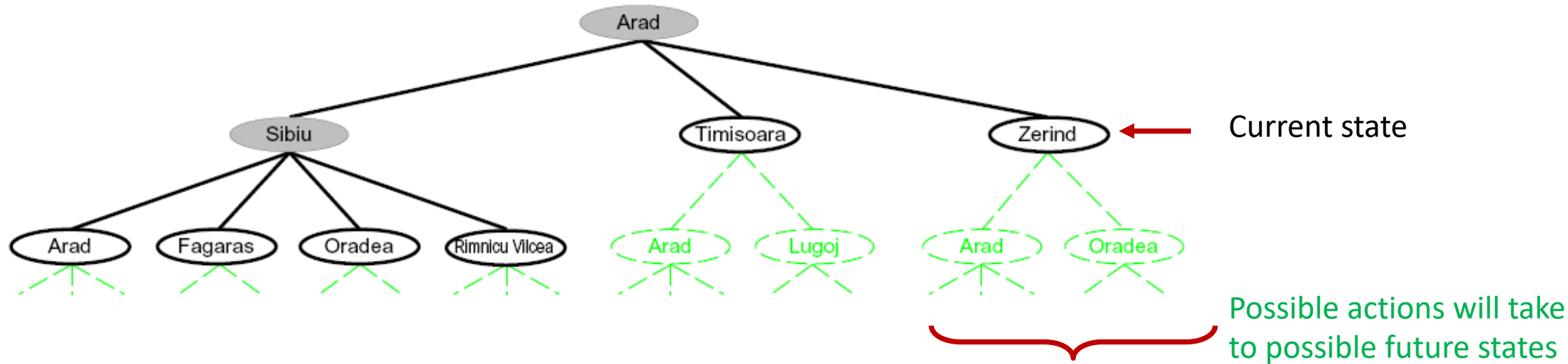
# Searching for a solution

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





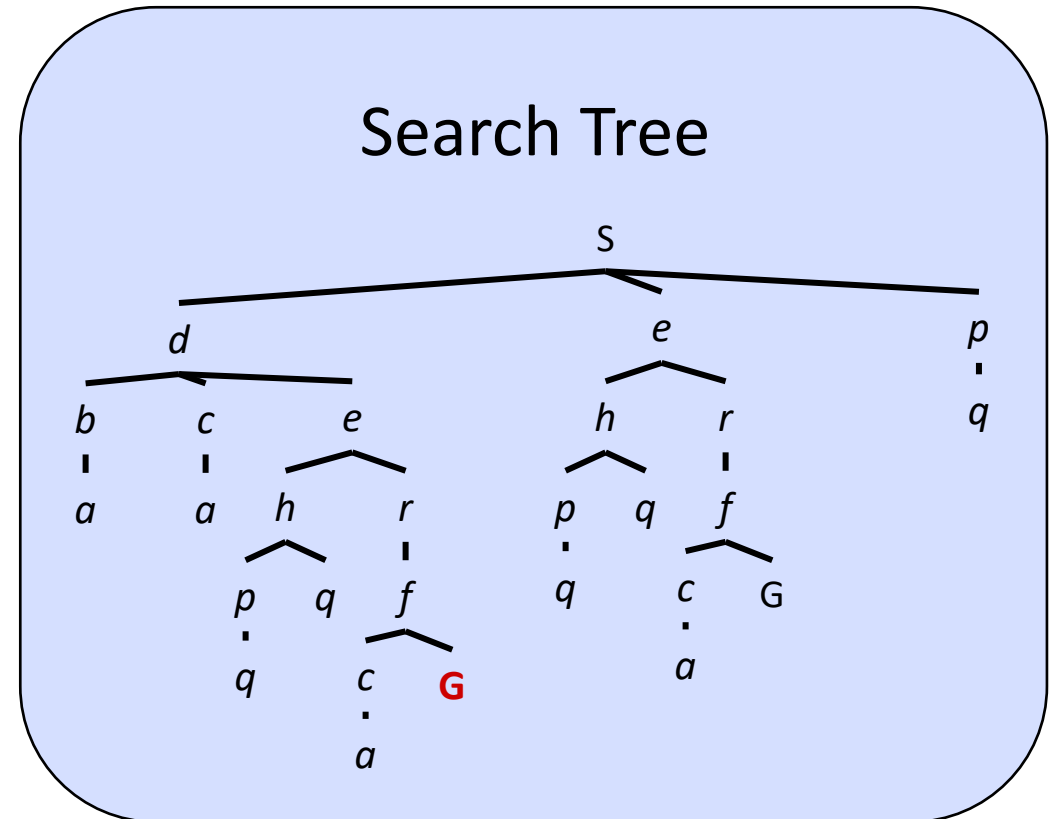
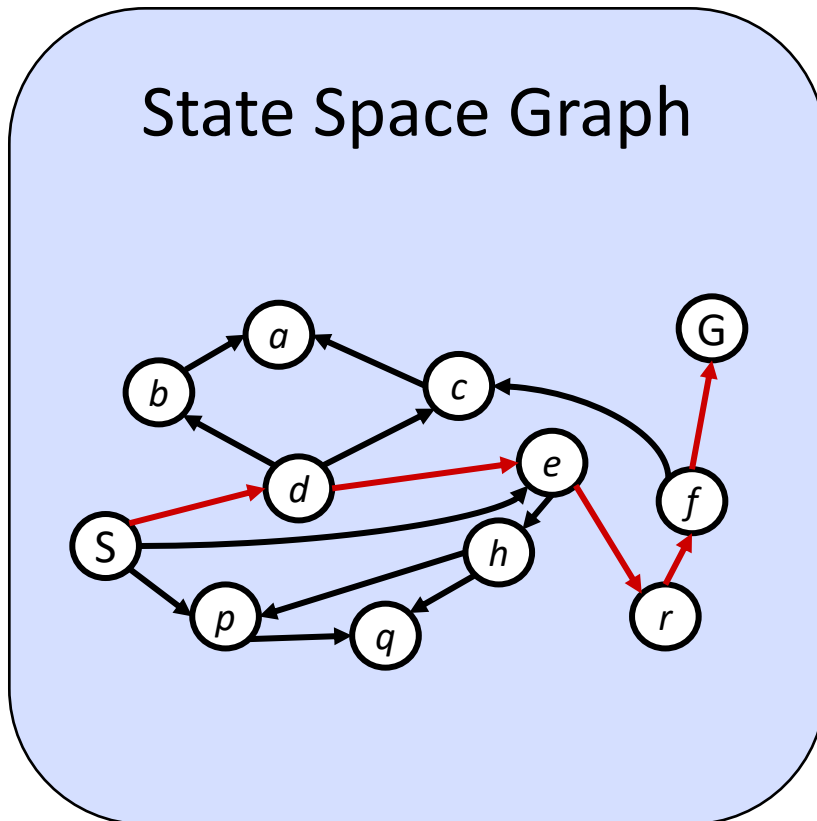
# Search Trees



- 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.

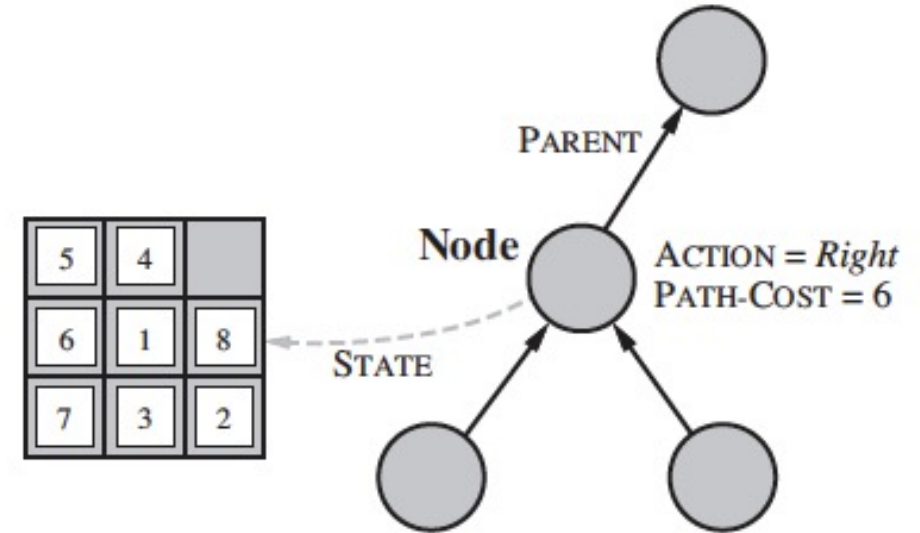


# Tree Search

```
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

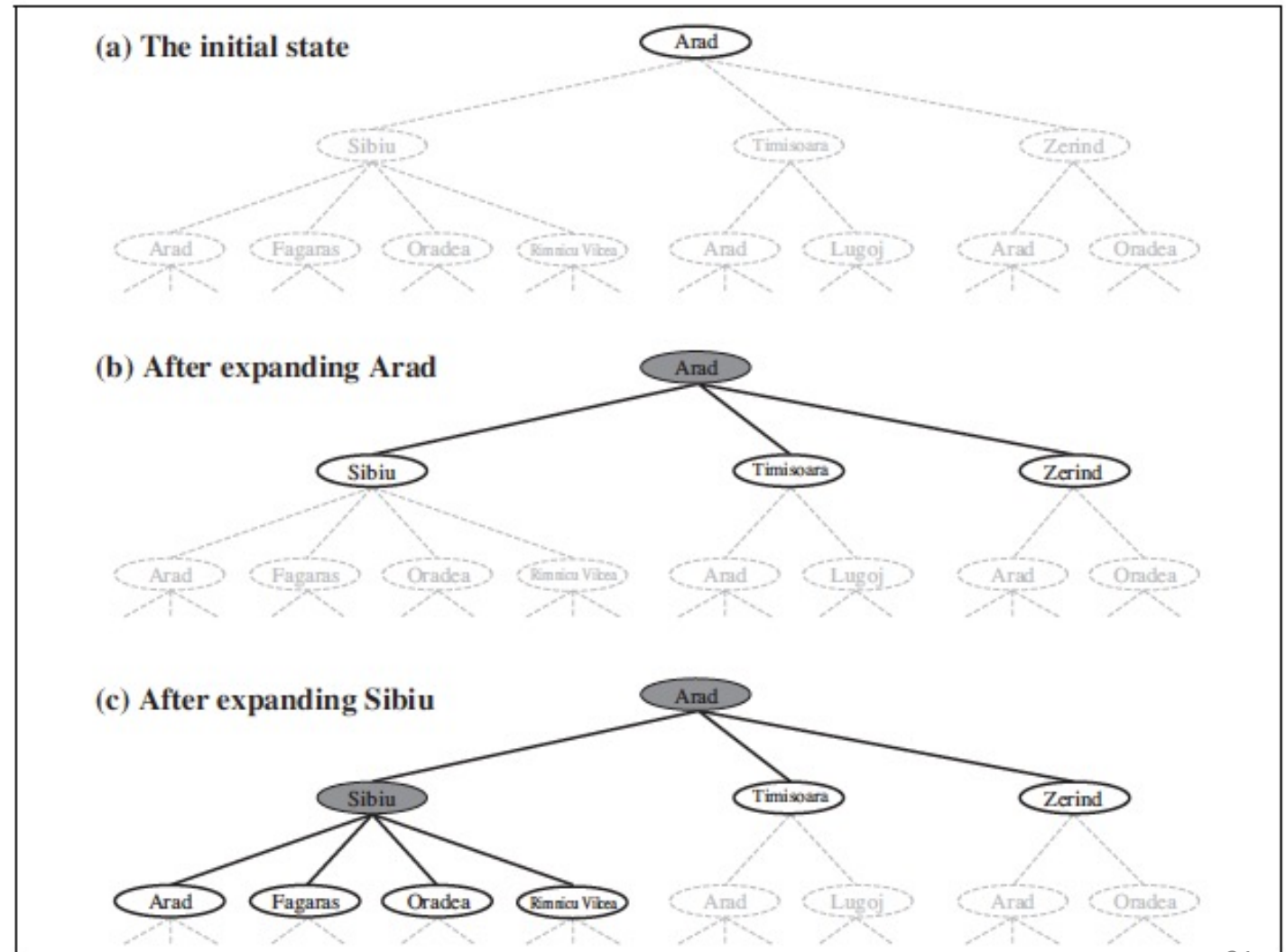
- *Defining a search node:*
  - Each node contains a state
  - Node also contains additional information, e.g.:
    - \* The parent state and the operator used to generate it
    - \* Cost of the path so far
    - \* Depth of the node
- *Expanding a node:*
  - Applying all legal operators to the state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



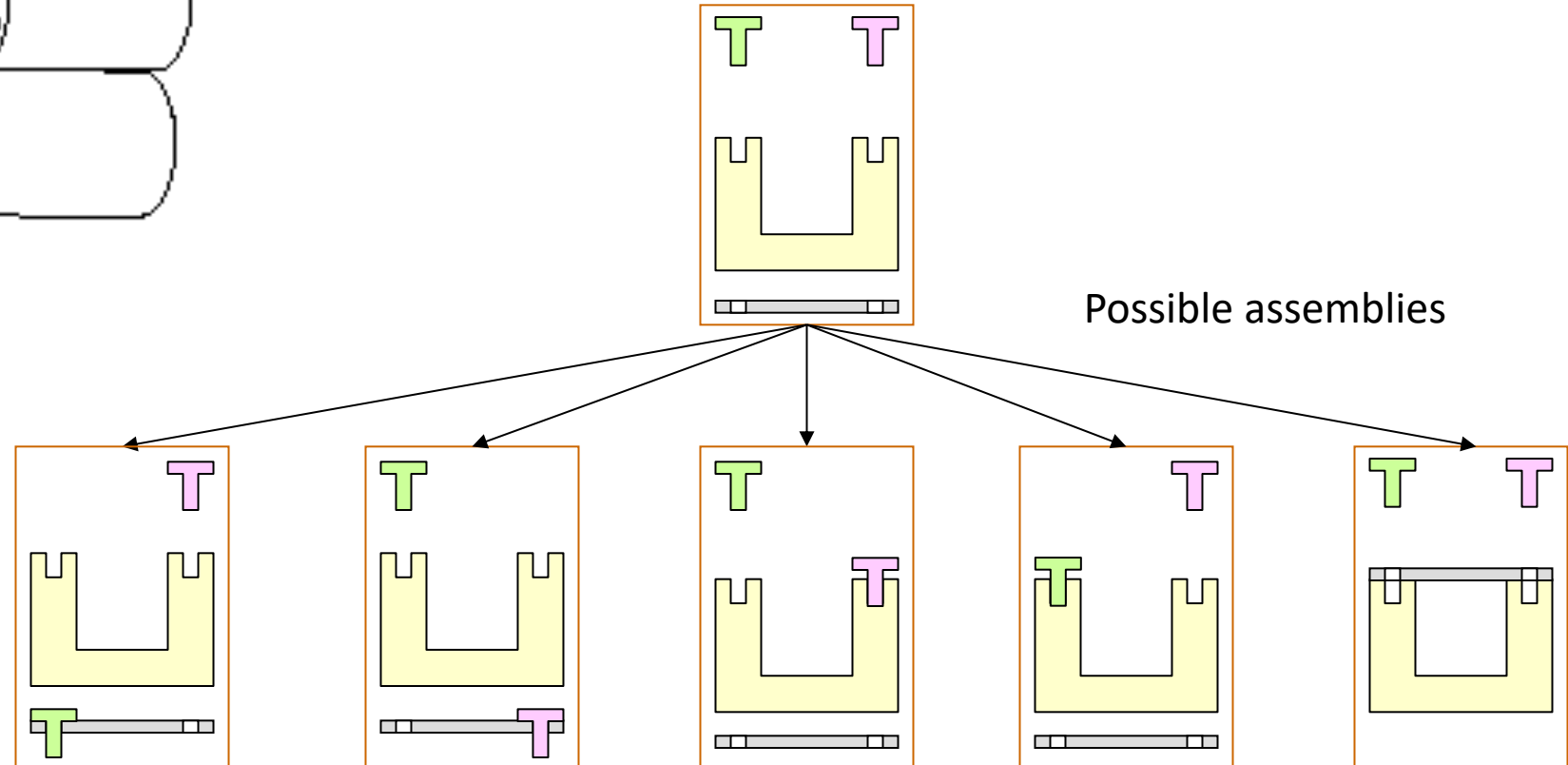
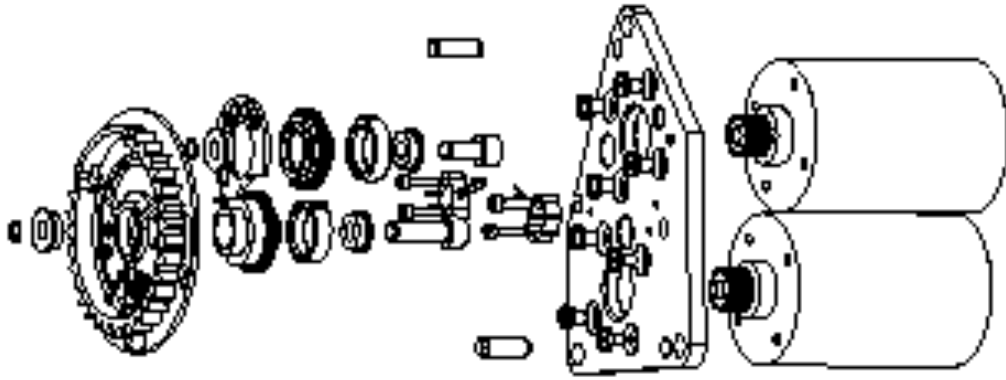
```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
    STATE = problem.RESULT(parent.STATE, action),
    PARENT = parent, ACTION = action,
    PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

# Search Tree

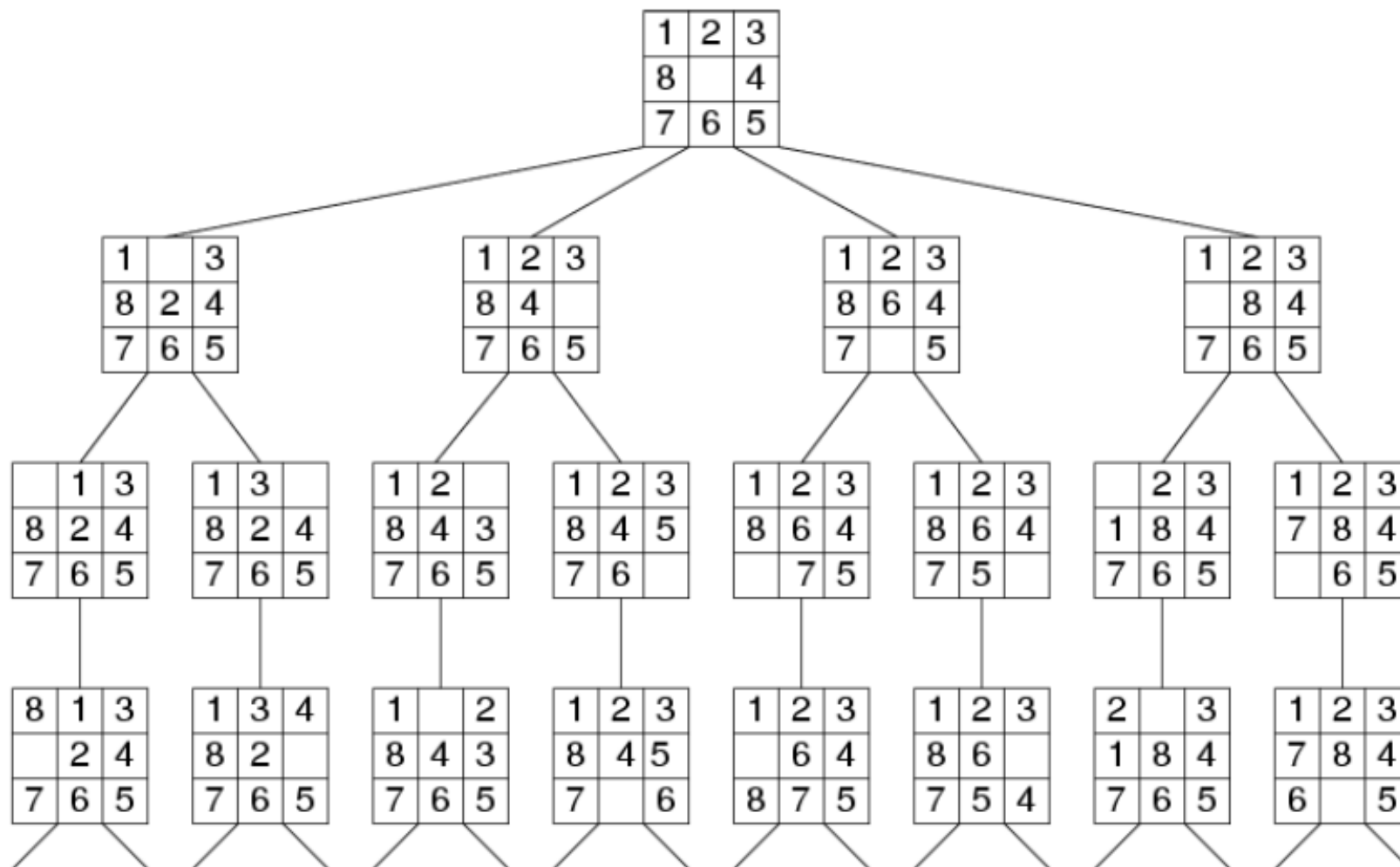
- The process of expansion while constructing the search tree.
  - Note that Arad appears again.
- Loops lead to redundancy
  - Why? Path costs are additive.
- Can we remember which nodes were expanded?



# Examples – Assembly Planning



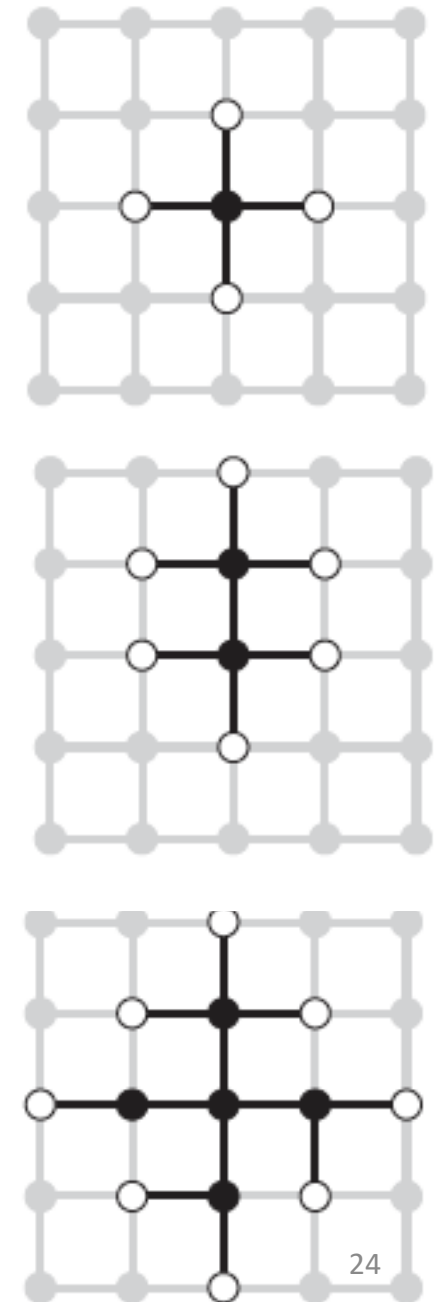
# The Eight Puzzle – State Space (Fragment)



# 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 in 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

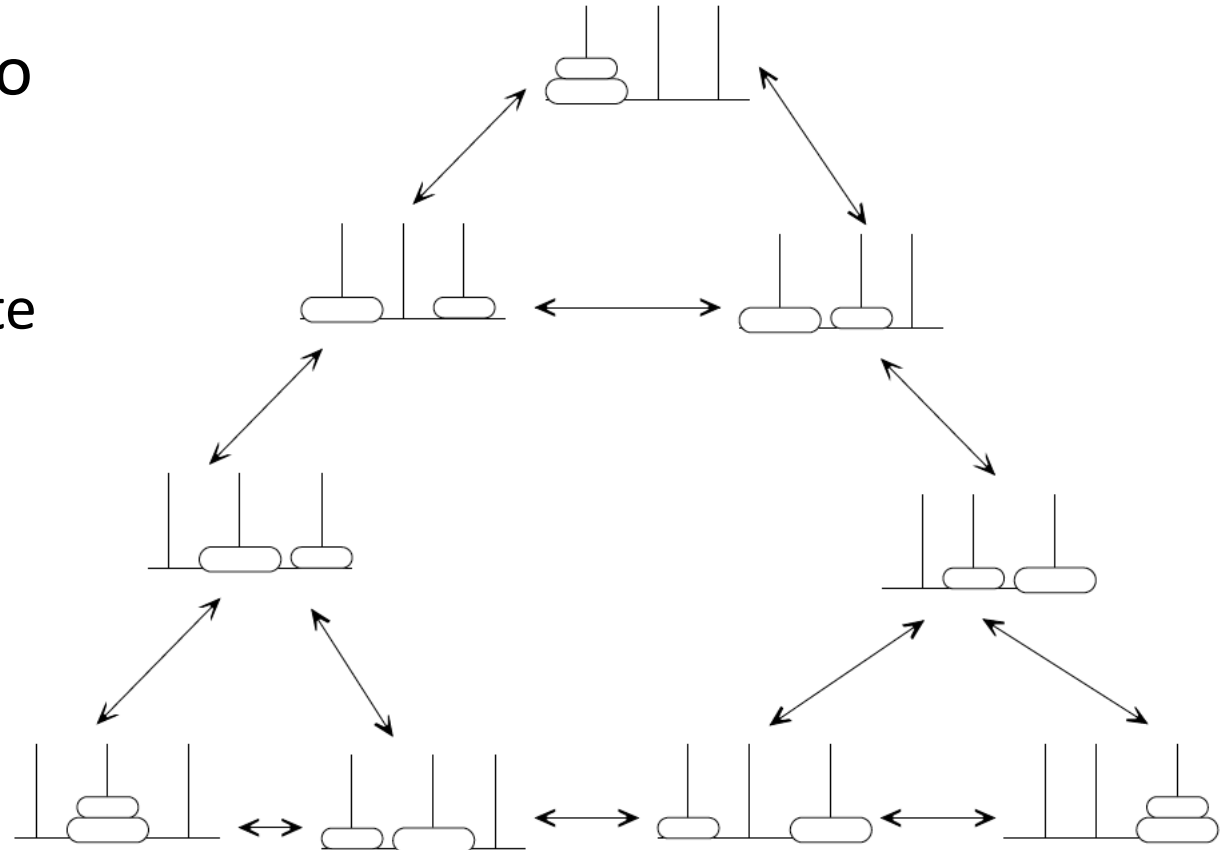




# Repeated States

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

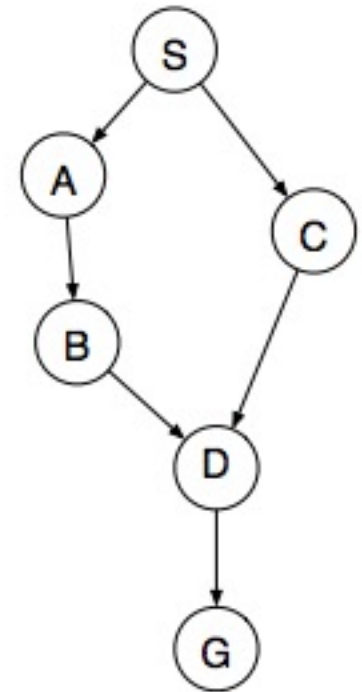
**Towers of Hanoi problem**



\* Each additional path found is going to be longer.

# 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*

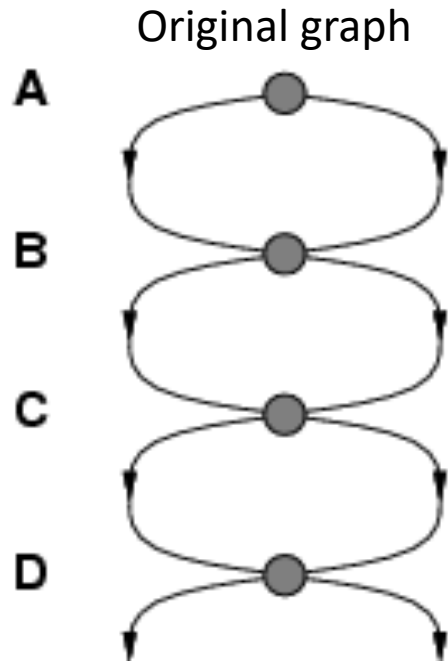
        expand the chosen node, adding the resulting nodes to the frontier

*only if not in the frontier or explored set*

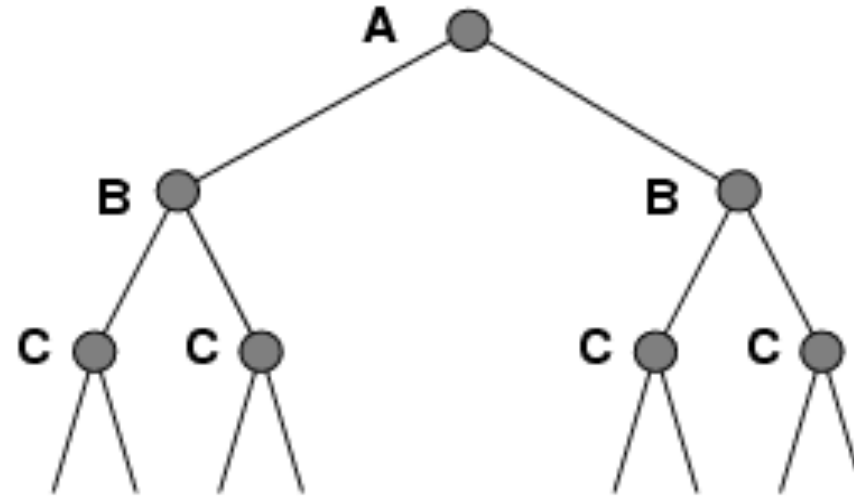
Newly generated nodes that match the frontier nodes or expanded nodes are discarded.



# Importance of detecting repeated states



Search tree without checking revisited states.



If we did not check for duplicate states, then the tree size is exponential in the number of states. If we do check for repeated states, then our tree is much smaller (linear).

# Handling Repeated States – Remember all the visited nodes

- **Never generate states that have already been generated before.**
- Maintain an explored list (Graph search)
- Optimal approach
- Memory inefficient, why?
  - Exponential number of nodes in the tree
    - E.g., 8-puzzle problem, we have  $9! = 362,880$  states.
  - Duplicate checking of states also adds time.

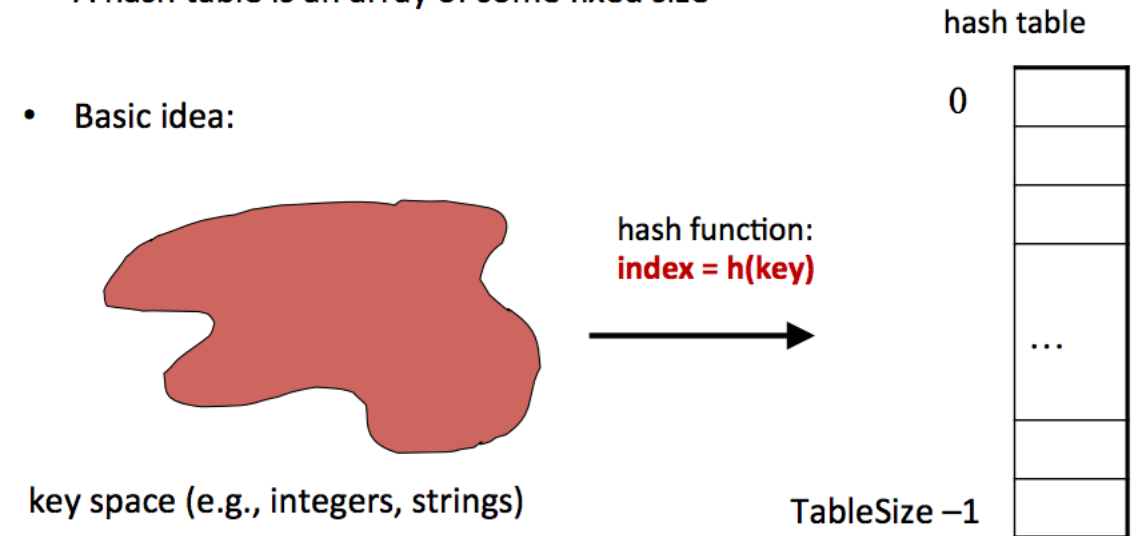
```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains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

# Handling Repeated States – Use efficient data structures

- Use efficient data structures to keep the explored nodes.
- Hash Tables
  - Insertion and look up in constant time.
- Duplicate checking
  - 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:

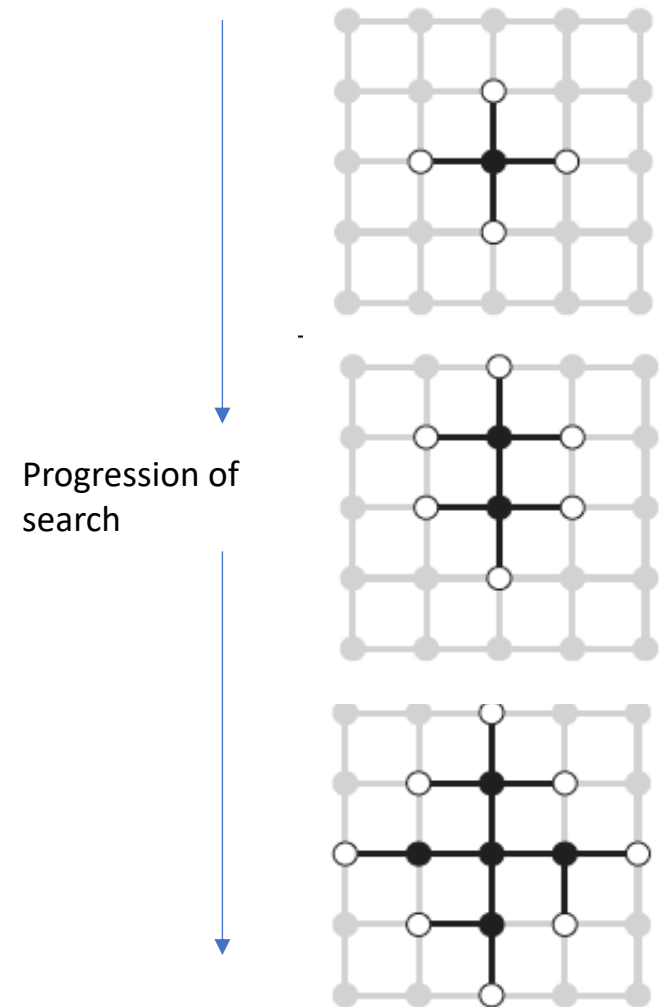


# Handling Repeated States – Check for some of the cases

- **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
- **Practical techniques but sub-optimal**

# 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





# 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

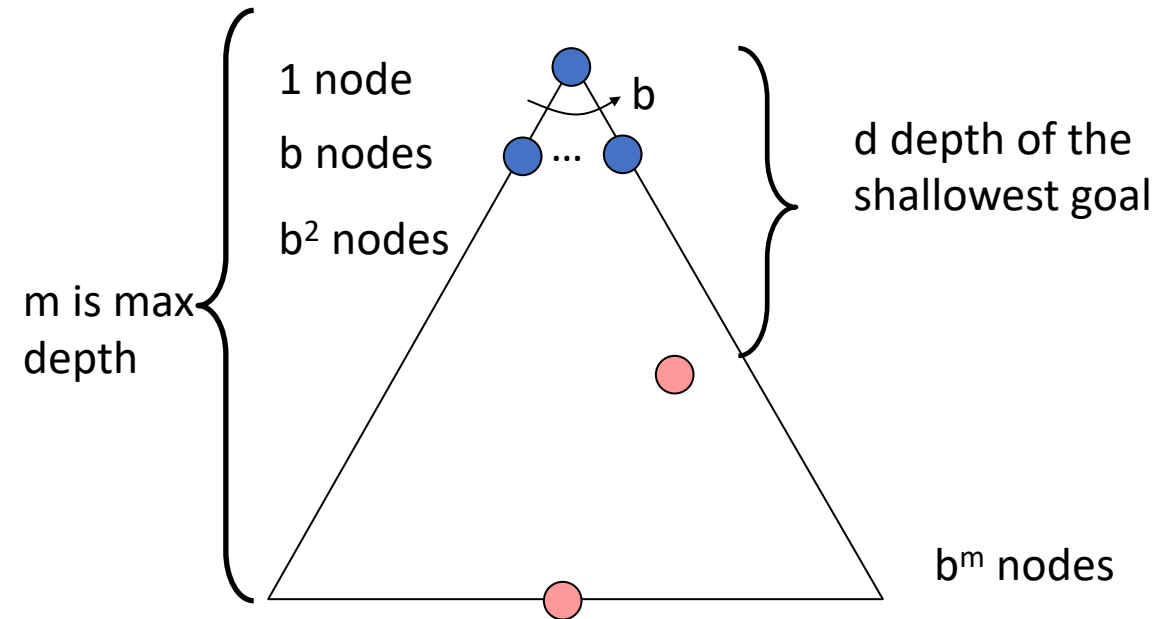
- The number of nodes generated during search.

- Space Complexity

- The maximum number of nodes stored in memory during search.

# 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



# Search Algorithms

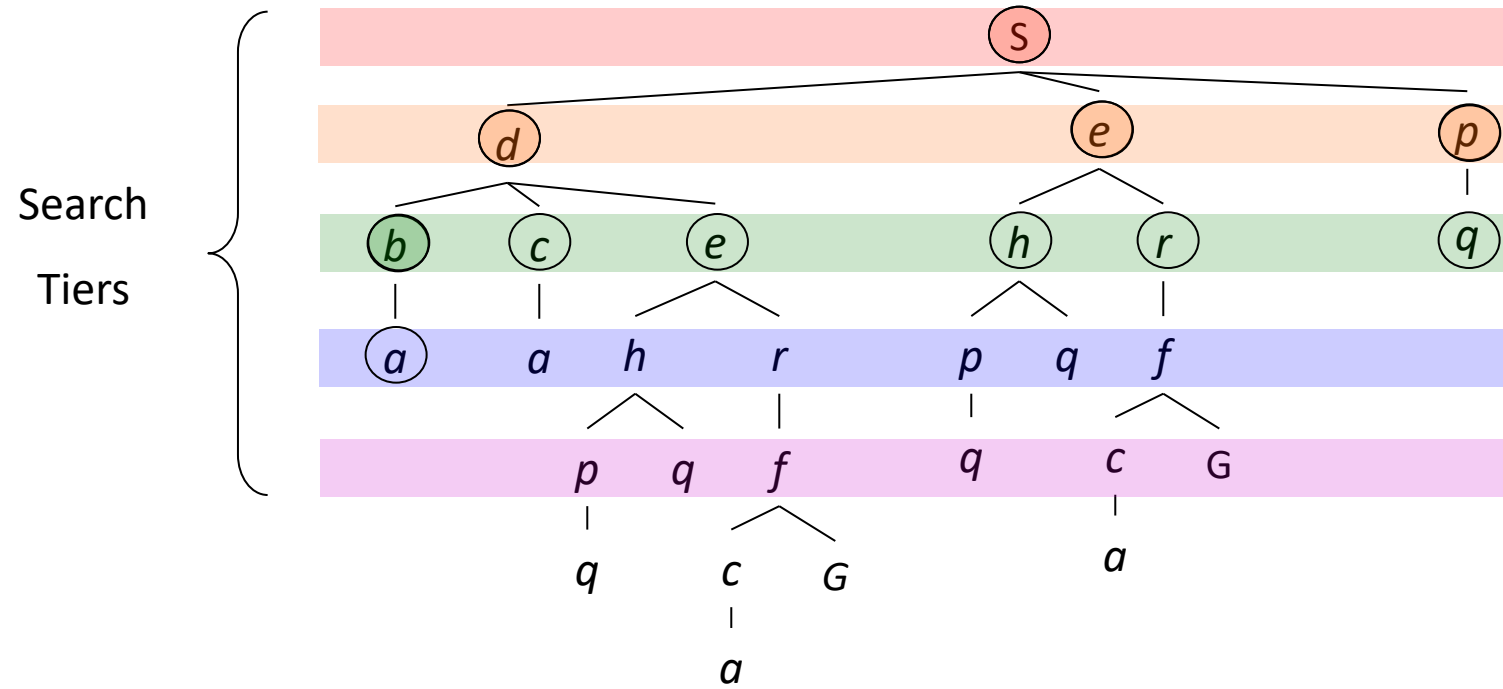
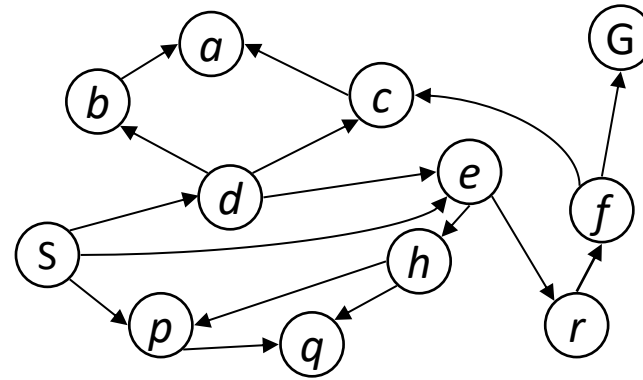
- **The strategy for exploration of nodes leads to a variety of search algorithms**
- Uninformed Search
  - Only use information about the state in the problem definition.
  - Generate successors and distinguish goal states from no-goal states.
- Informed Search
  - Use problem-specific knowledge beyond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

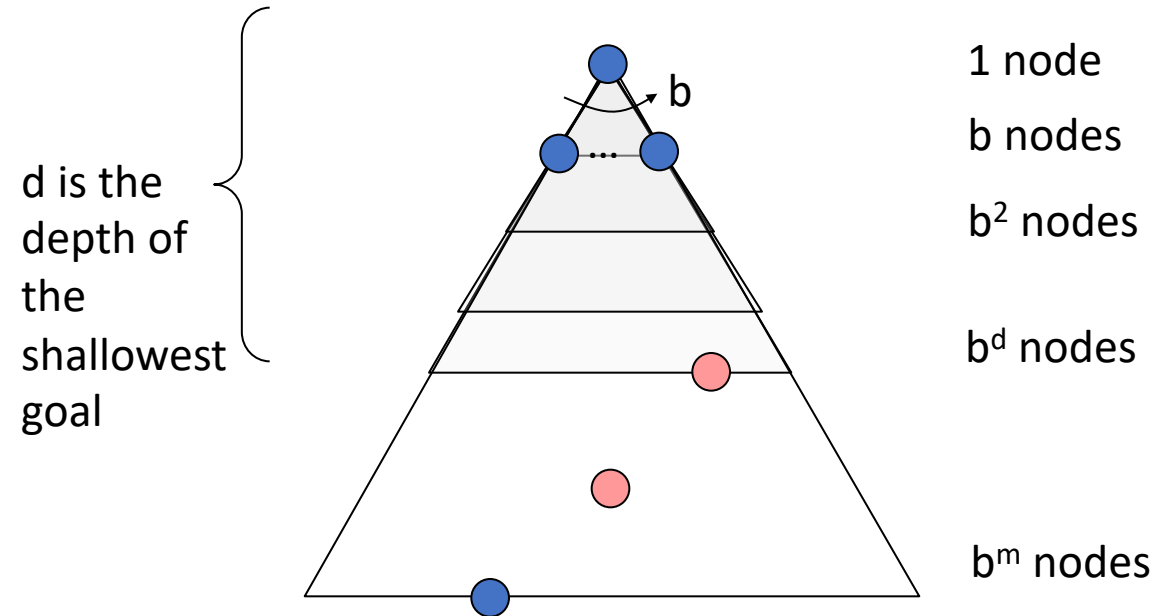
- Expands the **shallowest** unexplored node in the frontier of the search tree.

- **Time Complexity**

- Search takes time  $O(b^d)$

- **Space Complexity**

- Frontier nodes  $O(b^d)$
- Explored nodes  $O(b^{d-1})$
- **Memory requirement is a problem.**



# Breadth First Search (BFS) Properties

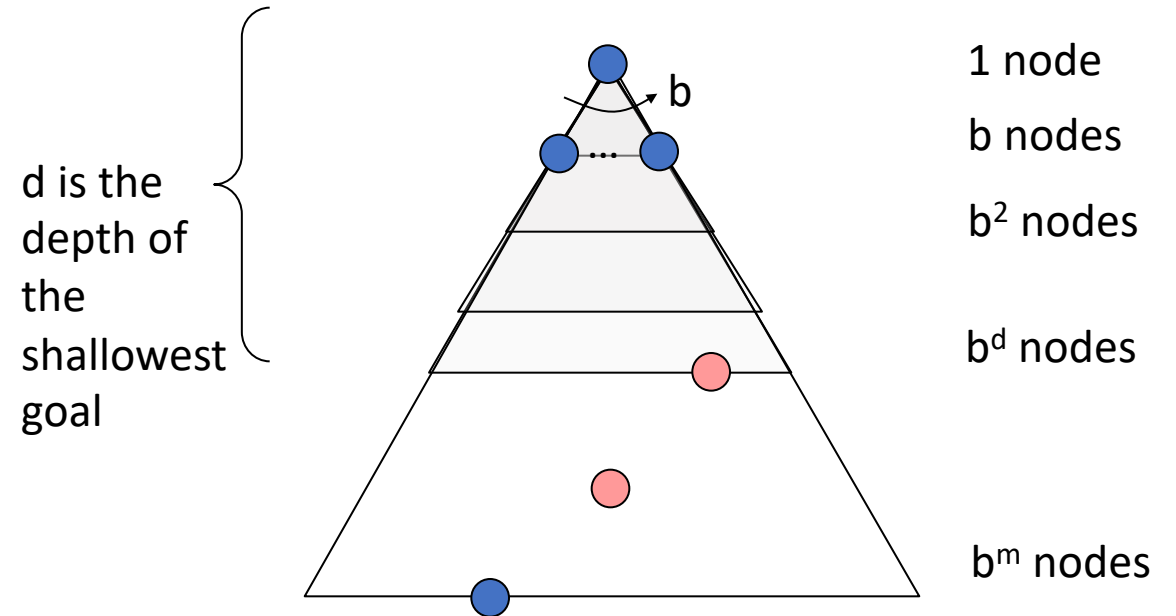
Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	$10^6$	18 minutes	111 megabytes
8	$10^8$	31 hours	11 gigabytes
10	$10^{10}$	128 days	1 terabyte
12	$10^{12}$	35 years	111 terabytes
14	$10^{14}$	3500 years	11,111 terabytes

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

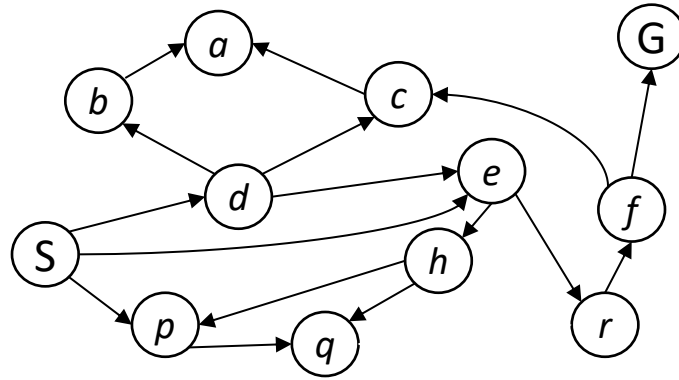
Take away - Memory requirement is a big problem for BFS.

# 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)

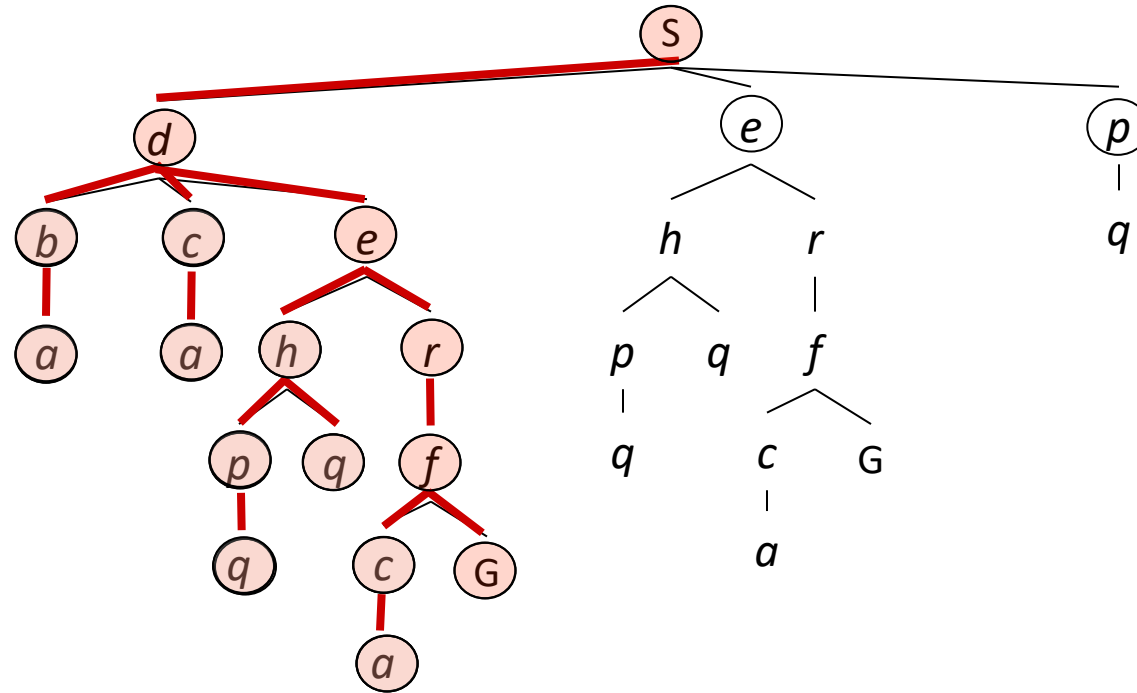
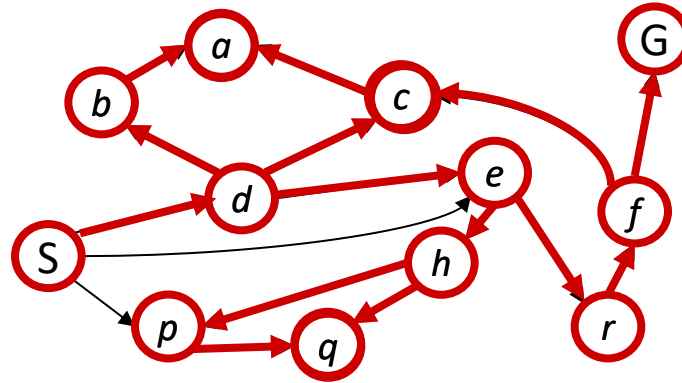




# Depth-First Search (DFS)

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**

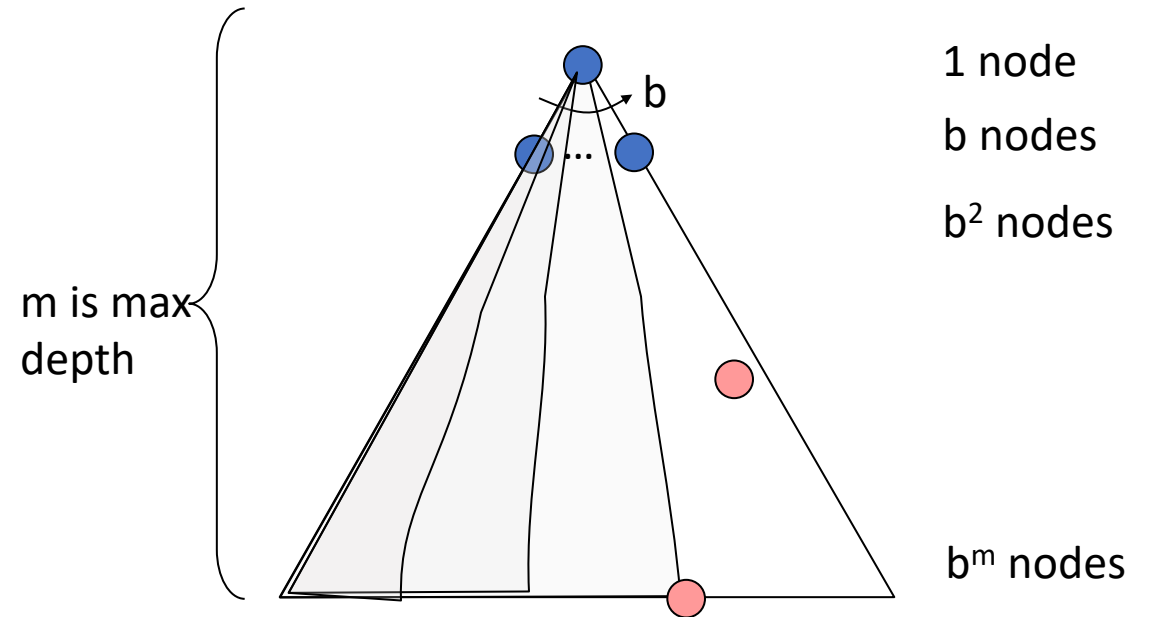
- Frontier stores:
  - Single path from the root to the leaf node.
  - Sibling nodes on the path that are unexplored.
- **Memory requirement is low  $O(bm)$**

- **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.

# Reducing DFS memory requirements

- 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 for a domain
- 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 level  $l$  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**

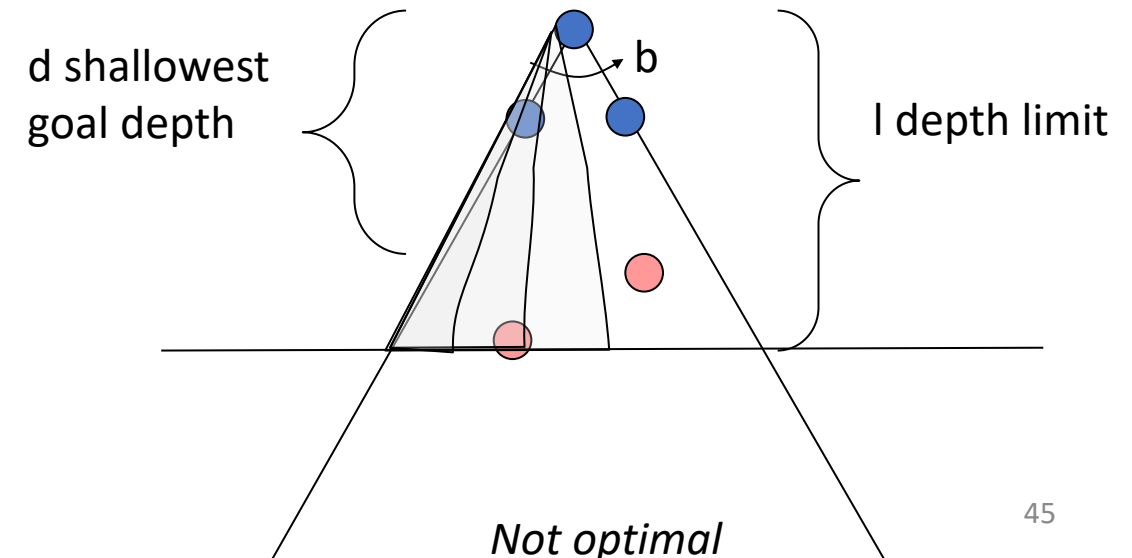
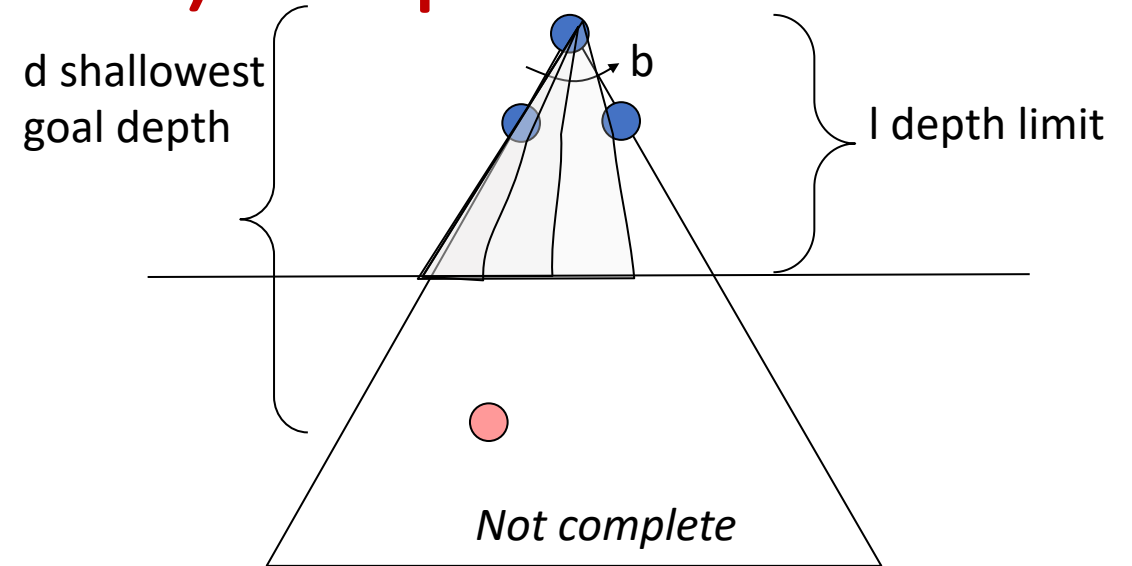
- Frontier is managed like Depth First Search.
- Memory  $O(bl)$ .

- **Is it complete?**

- **Not complete** when goal depth is greater than the limit ( $d > l$ )

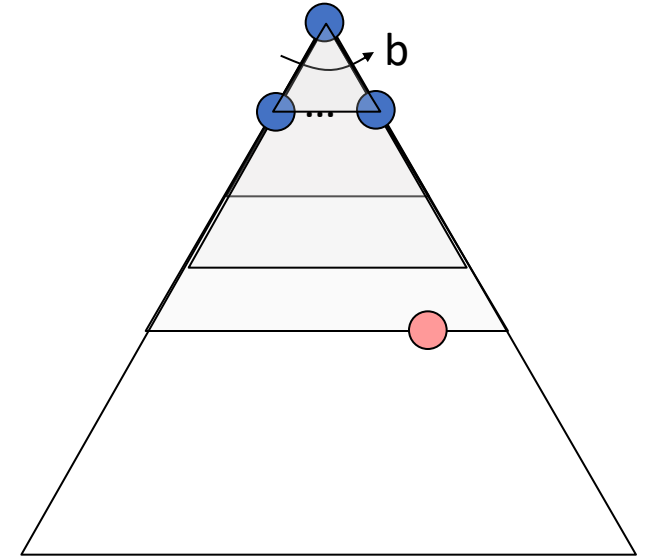
- **Is it optimal?**

- **Not optimal** when the limit is greater than the goal depth ( $l > d$ )



# Iterative Deepening Search

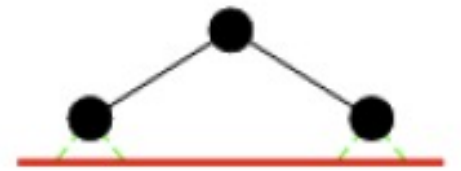
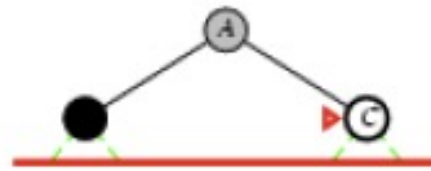
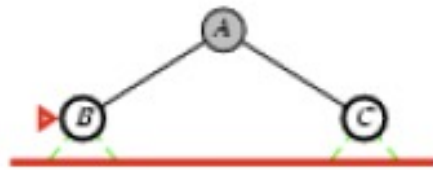
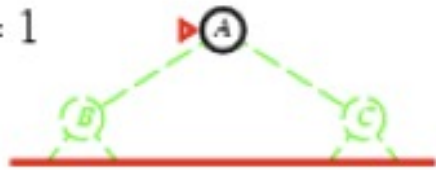
- Combine DFS's space advantage with BFS's shallow-solution advantages
  - Run a DLS with depth limit 1. If no solution...
  - Run a DLS with depth limit 2. If no solution...
  - Run a DLS with depth limit 3. ....



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

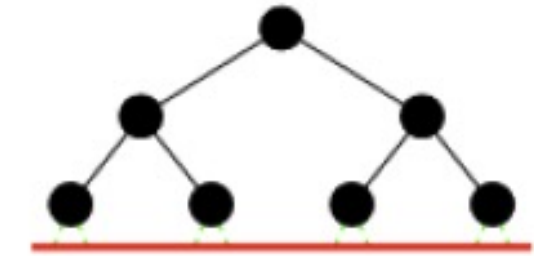
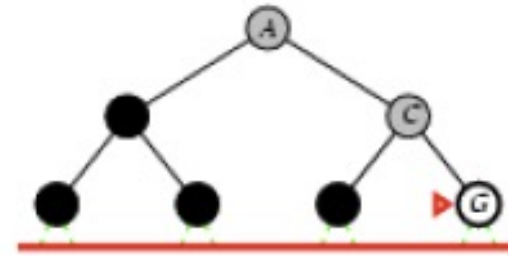
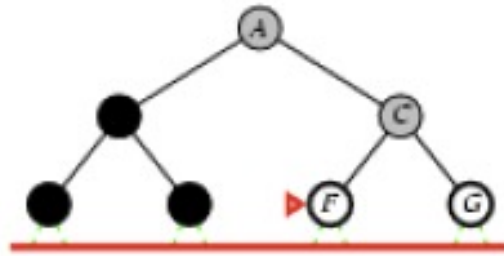
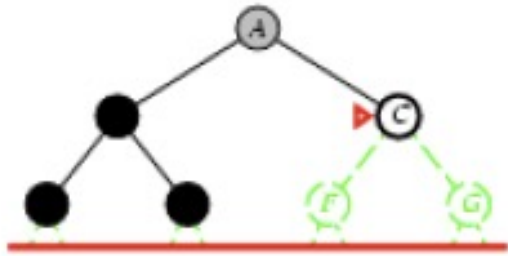
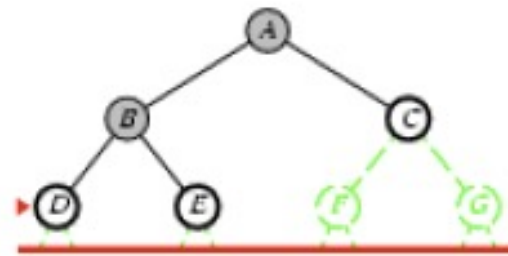
# Iterative Deepening: Example

Limit = 1



# Iterative Deepening: Example

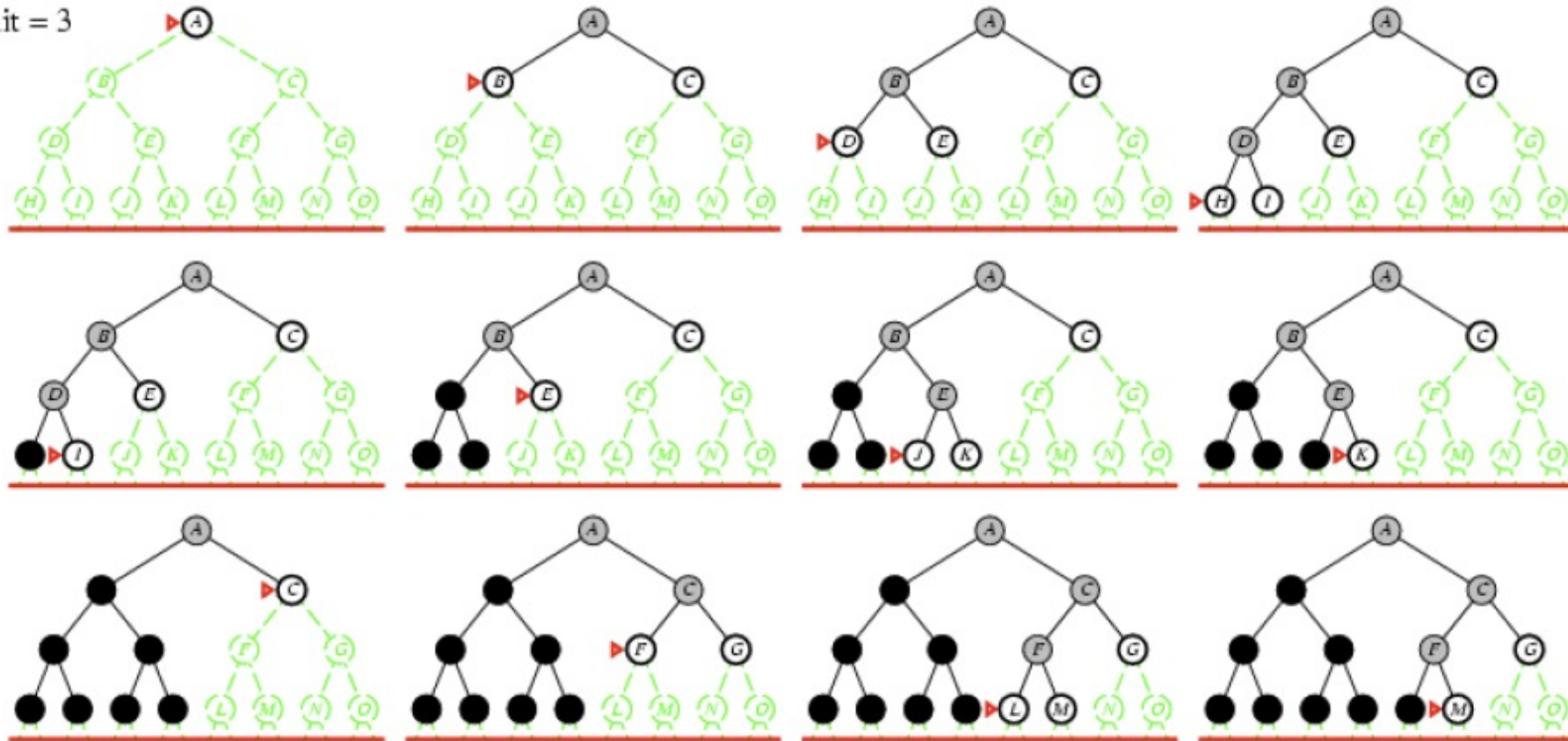
Limit = 2





# Iterative Deepening: Example

Limit = 3



# Iterative Deepening: Pseudo-code

---

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

function DEPTH-LIMITED-SEARCH(problem,  $\ell$ ) returns a node or failure or cutoff
  frontier  $\leftarrow$  a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result  $\leftarrow$  failure
  while not IS-EMPTY(frontier) do
    node  $\leftarrow$  POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) >  $\ell$  then
      result  $\leftarrow$  cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

**Figure 3.12** Iterative deepening and depth-limited tree-like search. Iterative deepening repeatedly applies depth-limited search with increasing limits. It returns one of three different types of values: either a solution node; or *failure*, when it has exhausted all nodes and proved there is no solution at any depth; or *cutoff*, to mean there might be a solution at a deeper depth than  $\ell$ . This is a tree-like search algorithm that does not keep track of *reached* states, and thus uses much less memory than best-first search, but runs the risk of visiting the same state multiple times on different paths. Also, if the IS-CYCLE check does not check *all* cycles, then the algorithm may get caught in a loop.

# 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  $b$ :

- $N_{DLS} = b^0 + b^1 + b^2 + \dots + b^{d-2} + b^{d-1} + b^d$

- Number of nodes generated in an iterative deepening search to depth  $d$  with branching factor  $b$ :

- $N_{IDS} = (d+1)b^0 + d b^1 + (d-1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + 1b^d$

- Asymptotic ratio:  $(b+1)/(b-1)$

No. of times generated.

- 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 =  $(123,456 - 111,111)/111,111 = 11\%$

# Iterative Deepening Properties

- **Time Complexity**

- Time  $O(b^d)$

- **Space Complexity**

- Memory  $O(bd)$
- Linear memory requirement like **DFS**

- **Is it complete?**

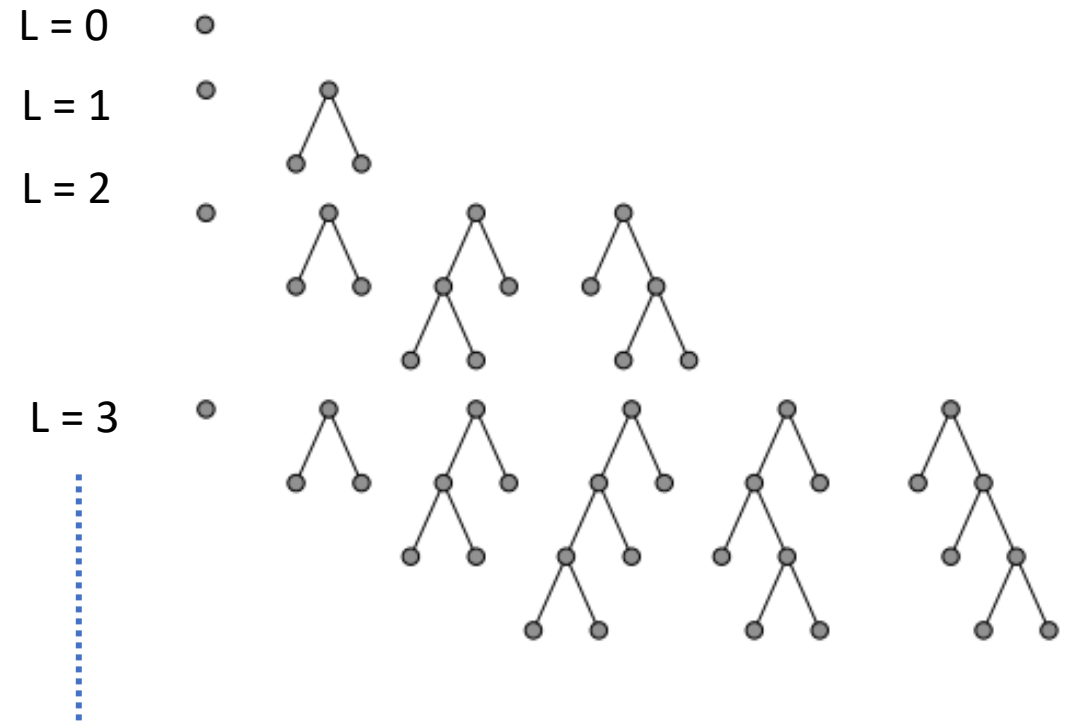
- **Yes.** Complete like **BFS**

- **Is it optimal?**

- **Yes.** Optimal like BFS (if costs are non-decreasing function of path length)

- **Relevance**

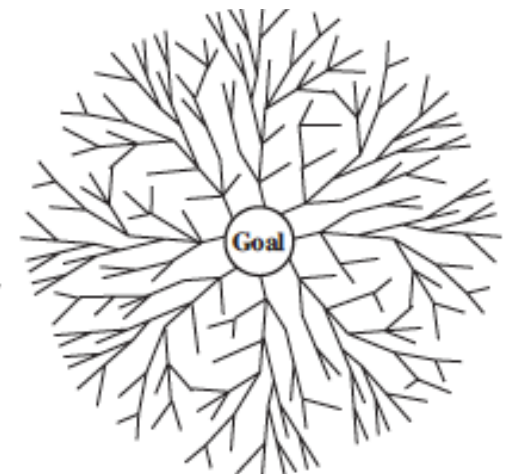
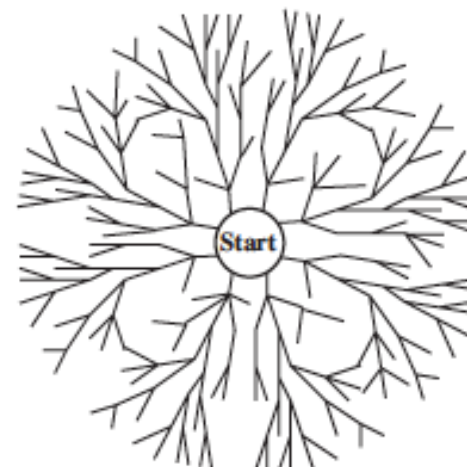
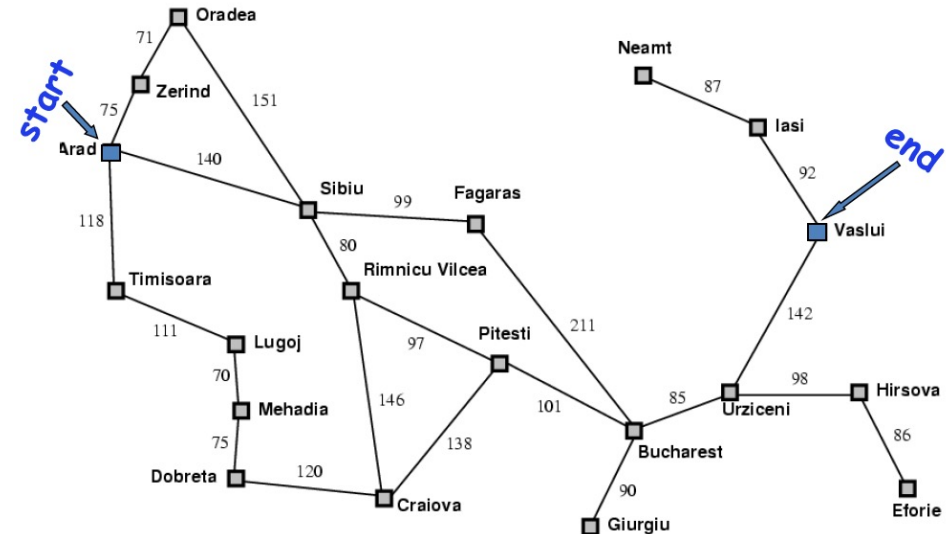
- Preferred method for large state spaces where maximum depth of a solution is unknown



# Two small search trees instead of one large?

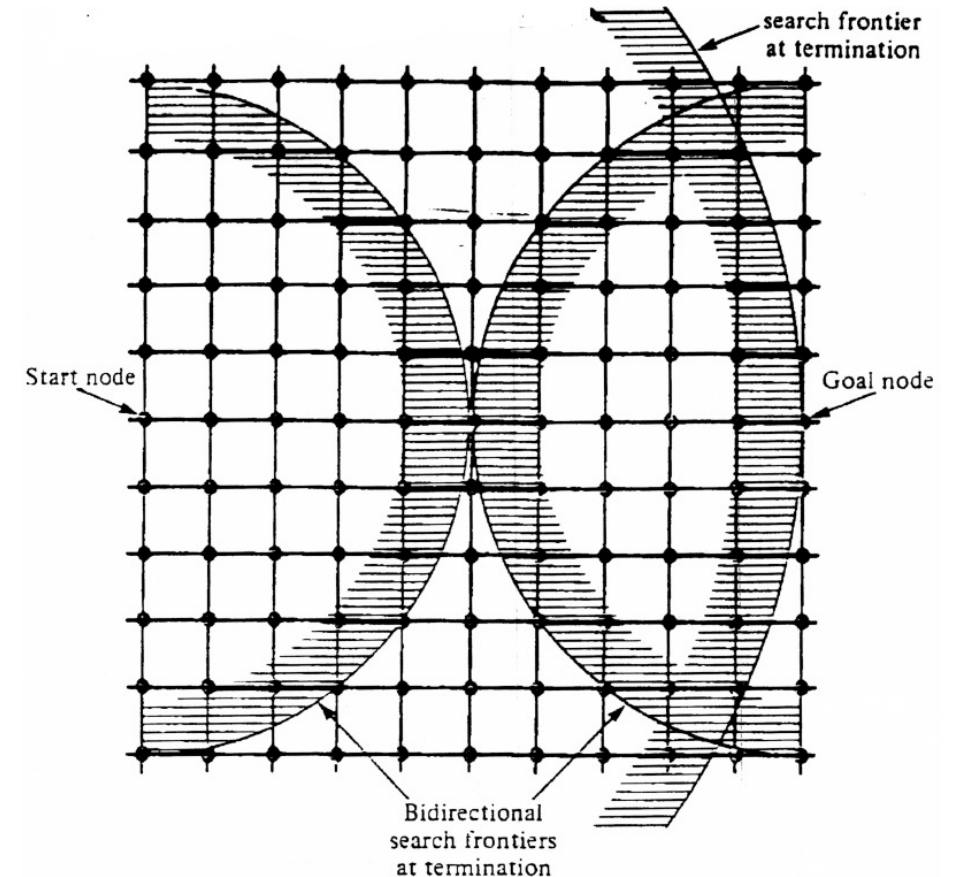
## Bi-directional Search

- 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

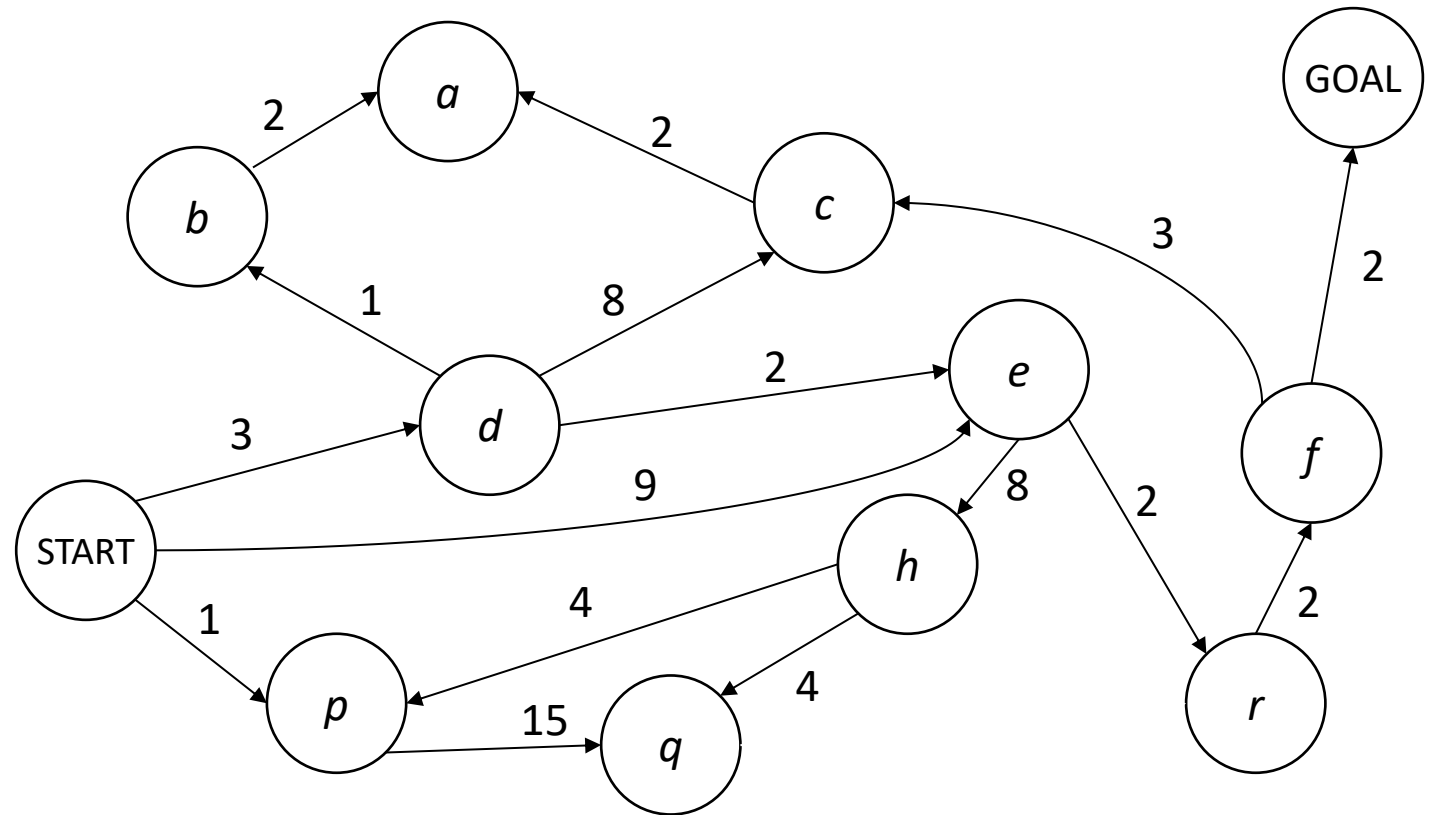
- Space and time complexity
  - $O(b^{d/2})$
  - $b^{d/2} + b^{d/2}$  is smaller than  $b^d$
  - $10^8 + 10^8 = 2 \cdot 10^8 \ll 10^{16}$
- Needs an efficiently computable *Predecessor()* function
  - Difficult: e.g., predecessors of checkmate in chess?
- What if there are several goal states?
  - Create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





# Guiding search by costs instead of depth

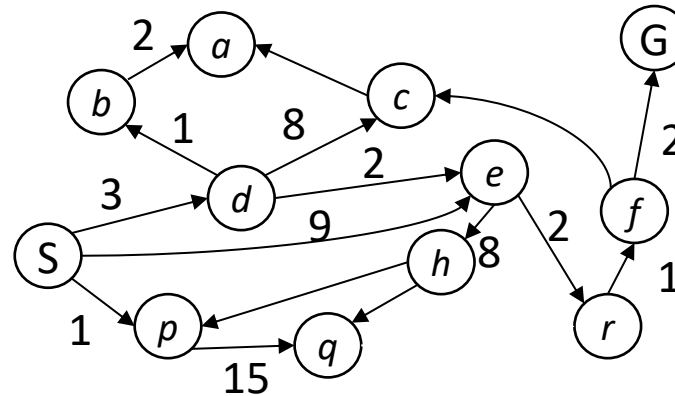
- Till now, the cost on the edges was not considered
  - Worked with solution depths.
- Solution was found in terms of number of actions.
  - Did not find the least-cost path.
  - BFS, DFS ....
- Next
  - Cost-sensitive search



# Uniform Cost Search (UCS)

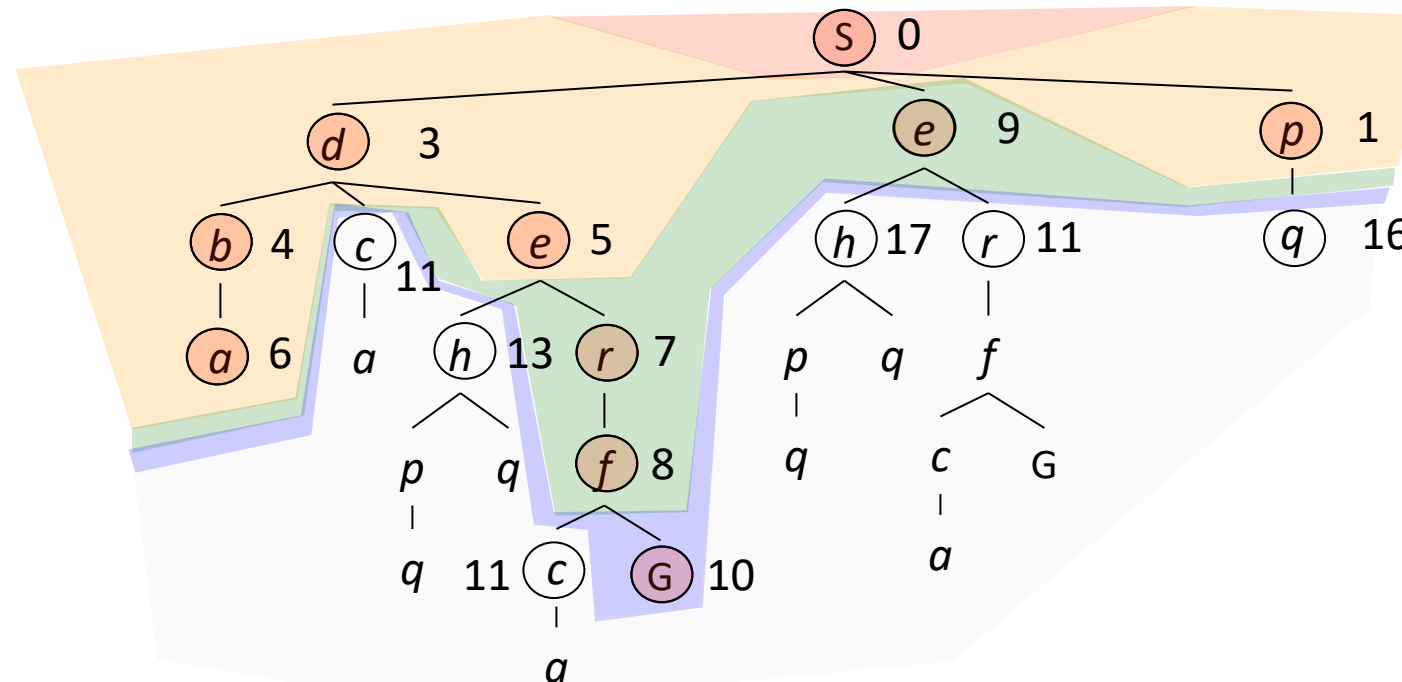
Strategy: expand a “cheapest” cost node first:

**Frontier is a Priority Queue (Priority: cumulative cost so far)**



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

Cost contours during search





# Uniform Cost Search (UCS)

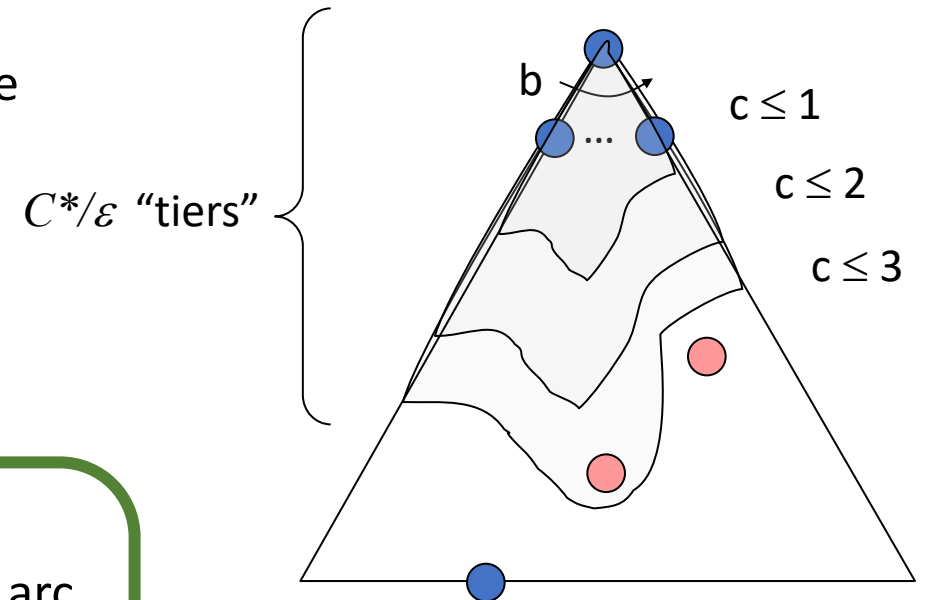
- The first goal node generated may be on the sub-optimal path.
- If the new path is better than the old path, then discard the old one.

```
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  $\epsilon$ , then the “effective depth” is roughly  $C^*/\epsilon$
  - Takes time  **$O(b^{C^*/\epsilon})$**  (exponential in effective depth)
- How much space does the frontier take?
  - **$O(b^{C^*/\epsilon})$**

- 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)



# What if we bound the frontier size?

## Beam Search

- Keep a *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.
- Source: AIMA

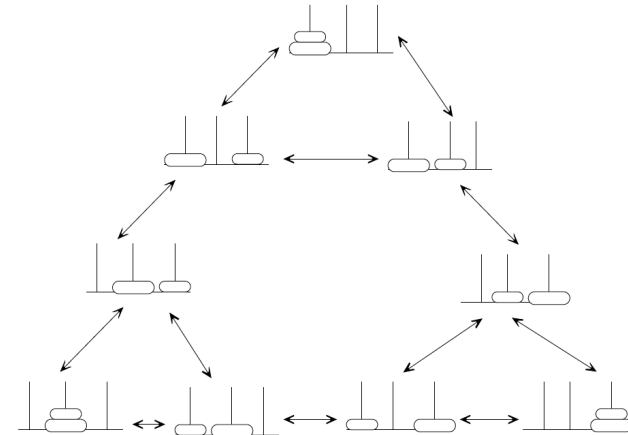
# Search Algorithms

- 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

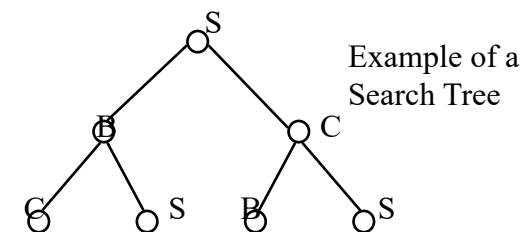
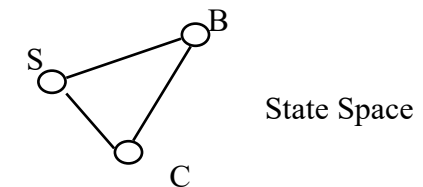
# Repeated States Burden Search Algorithms

- What causes states to appear again during search?
  - Reversible actions can lead to repeated states.
    - Reversible actions, e.g., the 8 puzzle, tower of hanoi or route finding.
  - Genuine multiple paths to the goal.
- Lead to more than one way to arrive at a node.  
Adds additional plans to explore within the search process.
- However, they are longer and hence redundant hence not useful.
- How to not consider redundant paths in the tree search.
  - Remember all the nodes that have been explored. In addition to these nodes, also check nodes on the frontier before adding a new solution path for consideration.
- Let's think: What else can be done to either not generate solution candidates that we know will be longer or in general make the process of checking for loops faster?

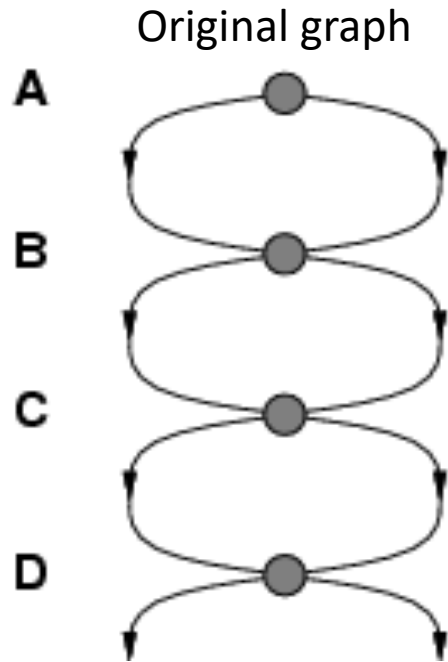
Towers of Hanoi problem



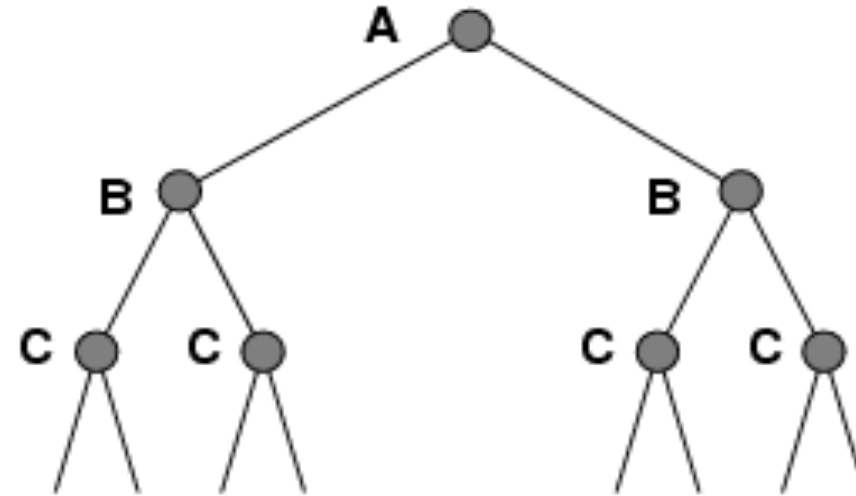
Route traversal problem



# Revisited states can significantly increase search complexity



Search tree without checking revisited states.



If we did not check for duplicate states, then the tree size is exponential in the number of states. If we do check for repeated states, then our tree is much smaller (linear).

# Handling Repeated States – Remember all the visited nodes

- **Never generate states that have already been generated before.**
- Maintain an explored list (Graph search)
- Optimal approach
- Memory inefficient, why?
  - Exponential number of nodes in the tree
    - E.g., 8-puzzle problem, we have  $9! = 362,880$  states.
  - Duplicate checking of states also adds time.

```
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
```

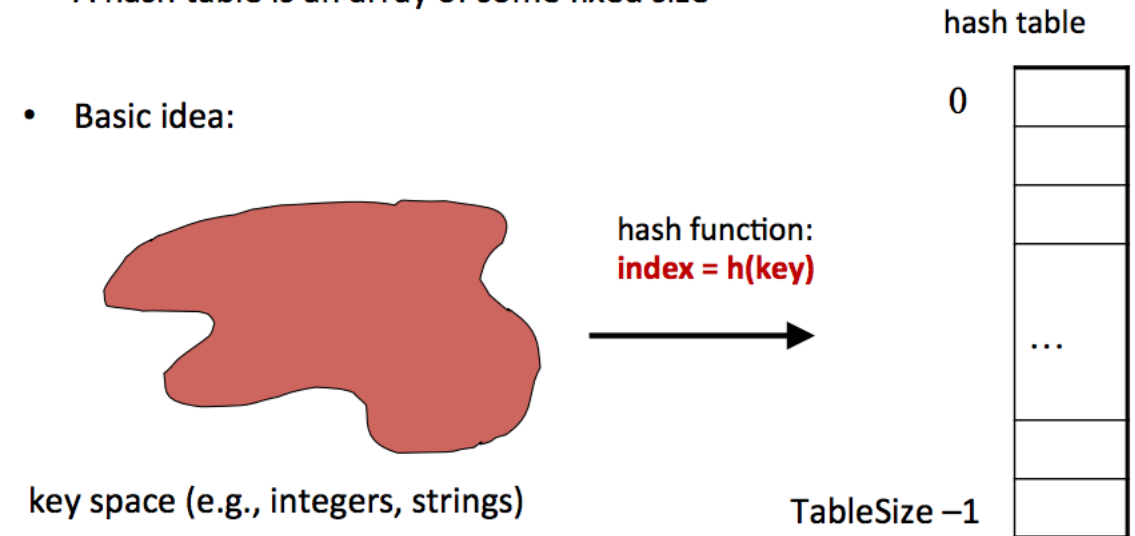


# Handling Repeated States – Use efficient data structures

- Use efficient data structures to keep the explored nodes.
- Hash Tables
  - Insertion and look up in constant time.
- Duplicate checking
  - 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:

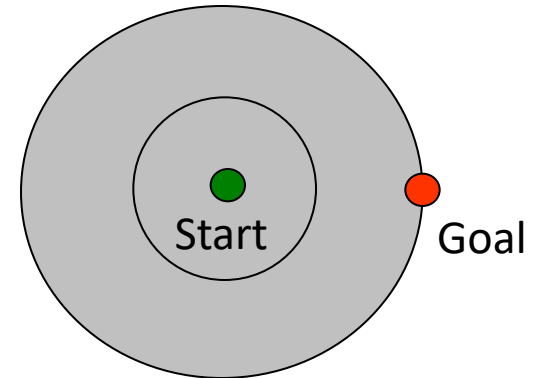
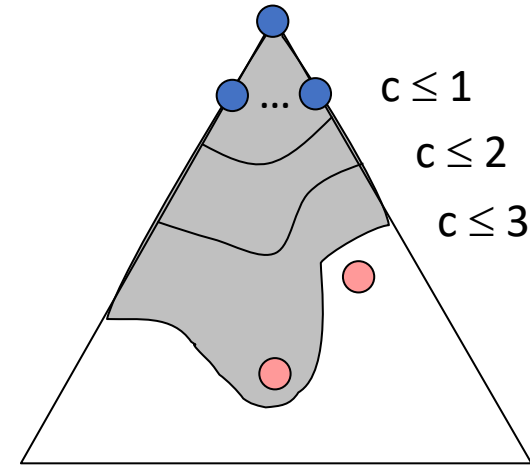


# Handling Repeated States – Check for some of the cases

- **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
- **Practical techniques but sub-optimal**

# Problem with uninformed search

- Uninformed search explores options in every “direction”
  - For example, UCS explores increasing cost contours
- **Does not make use the goal information.**



# 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.