Search Algorithms in AI (Chapter 3)

Dr. Nasima Begum
Assistant Professor
Dept. of CSE, UAP

Search Algorithms in AI

- Search algorithms are one of the most important areas of Artificial Intelligence. Search techniques are universal problem-solving methods in AI.

- ***Artificial Intelligence*** is the study of building agents that act rationally. Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.

- Problem-solving agents: **Rational agents** or **Problem-solving agents** in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result/solution.

- Problem-solving agents are the goal-based agents and use atomic representation. In this topic, we will learn various problem-solving search algorithms.

Problem-solving Agent

A simple Problem-solving agent is a goal-based agent. It decides what to do by finding different possible sequences of actions that lead to desirable states, and then choosing the best sequence.

This process of looking for such a sequence is called search. A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the execution phase.

Thus "formulate, search, execute" is the main parts in designing an agent.

Reflex Agents

- Reflex agents:
  - Choose action based on current percept (and maybe memory)
  - May have memory or a model of the world's current state
  - Do not consider the future consequences of their actions
  - Consider how the world IS

Planning Agents

- Planning agents:
  - Ask "what if"
  - Decisions based on (hypothesized) consequences of actions
  - Must have a model of how the world evolves in response to actions
  - Must formulate a goal (test)
  - Consider how the world WOULD BE

Search Algorithm Terminologies

- **Search:** Searching is a step by step procedure to solve a search-problem in a given search space.
  - **Search Space:** Search space represents a set of possible solutions, which a system may have.
  - **Start State:** It is a state from where agent begins **the search**.
  - **Goal Test:** It is a function which observe the current state and returns whether the goal state is achieved or not.
  - **Search Tree:** A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.

Search Algorithm Terminologies Cont..

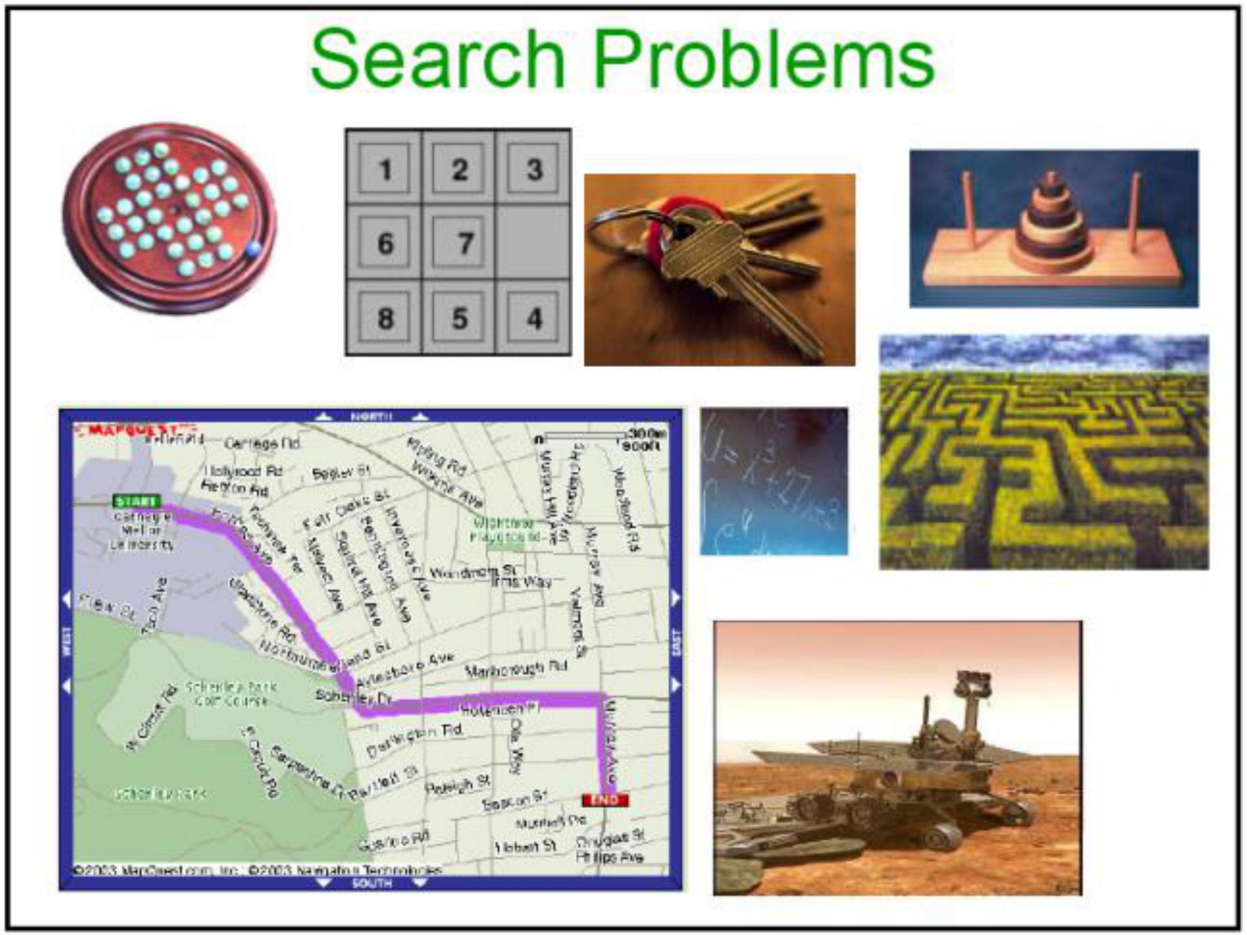
- **Actions:** It gives the description of all the available actions to the agent.
- **Transition Model:** A description of what each action do, can be represented as a transition model.
- **Path Cost:** It is a function which assigns a numeric cost to each path.
- **Solution:** It is an action sequence which leads from the start node to the goal node.
- **Optimal Solution:** If a solution has the lowest cost among all solutions.

Properties of Search Algorithms

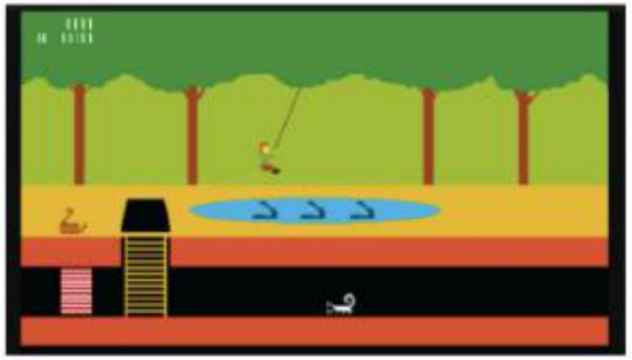
➢ Four essential properties of search algorithms to compare the efficiency of these algorithms:

- **Completeness:** A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.
- **Optimality:** If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution is said to be an optimal solution.
- **Time Complexity:** Time complexity is a measure of time for an algorithm to complete its task.
- **Space Complexity:** It is the maximum storage space required at any point during the search, as the complexity of the problem.

AI Solutions for Popular Search Problems

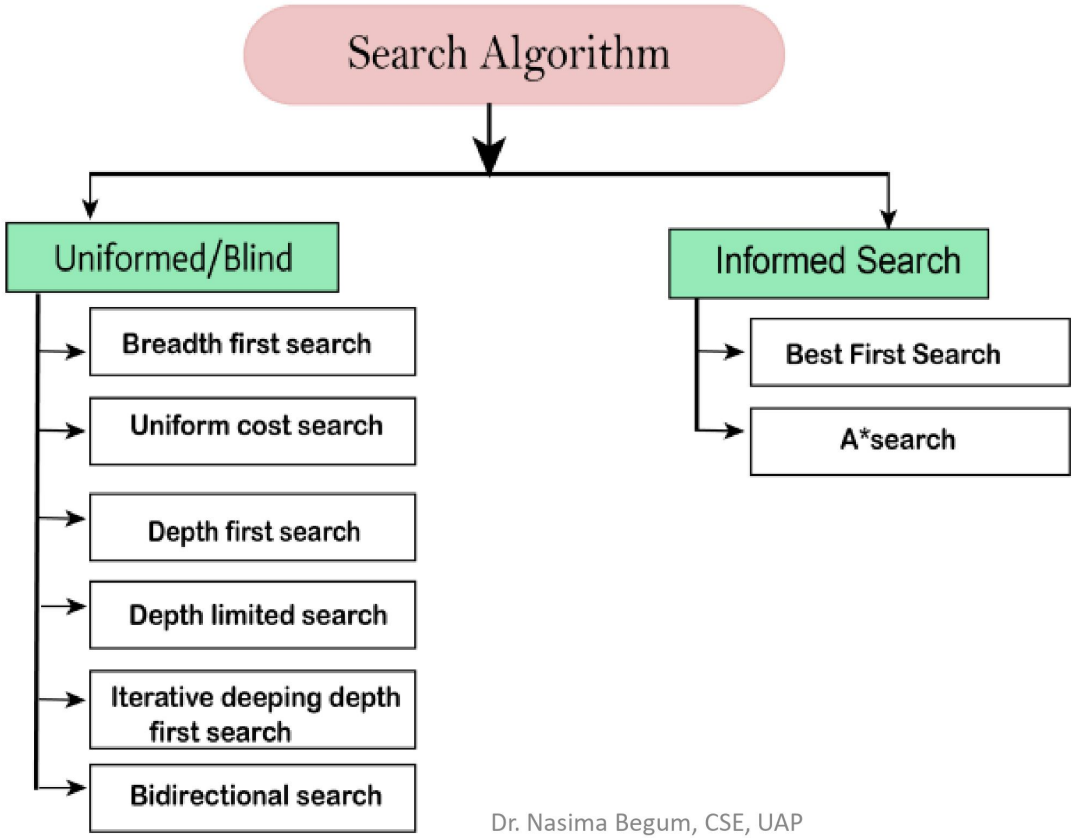


More Search Problems..



Types of search algorithms

- Based on the search problems, search algorithms can be classify into **uninformed (Blind search)** search and **informed search (Heuristic search)** algorithms.



Uninformed (Blind search)

- Have no additional information on the goal node other than the one provided in the problem definition.
- Uninformed (blind) search strategies use only the information available in the problem definition. All they can do is generate successor and distinguish a goal state from a non-goal state.
- The uninformed search does not contain any domain knowledge such as closeness, the location of the goal. It operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes.
- Uninformed search applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goal, so it is also called blind search. It examines each node of the tree until it achieves the goal node.
- Requirements of brute-force search strategy −
  - State description
  - A set of valid operators
  - Initial state
  - Goal state description

Informed (Heuristic search)

- Have information on the goal state, which helps in more efficient searching. This information is obtained by something called a *heuristic.*
- Strategies that know whether one non-goal state is 'more promising' than another are called informed or heuristic search strategies.
- It expands nodes in the order of their heuristic values. It creates two lists, a closed list for the already expanded nodes and an open list for the created but unexpanded nodes.
- In each iteration, a node with a minimum heuristic value is expanded, all its child nodes are created and placed in the closed list. Then, the heuristic function is applied to the child nodes and they are placed in the open list according to their heuristic value. The shorter paths are saved and the longer ones are disposed.
- This algorithm use domain knowledge. Here, problem information is available which can guide the search. It can find a solution more efficiently than an uninformed search strategy.
- A heuristic is a way which might not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.
- Can solve much complex problem which could not be solved in another way.

Formulating Search Problems (Road Map of Romania)

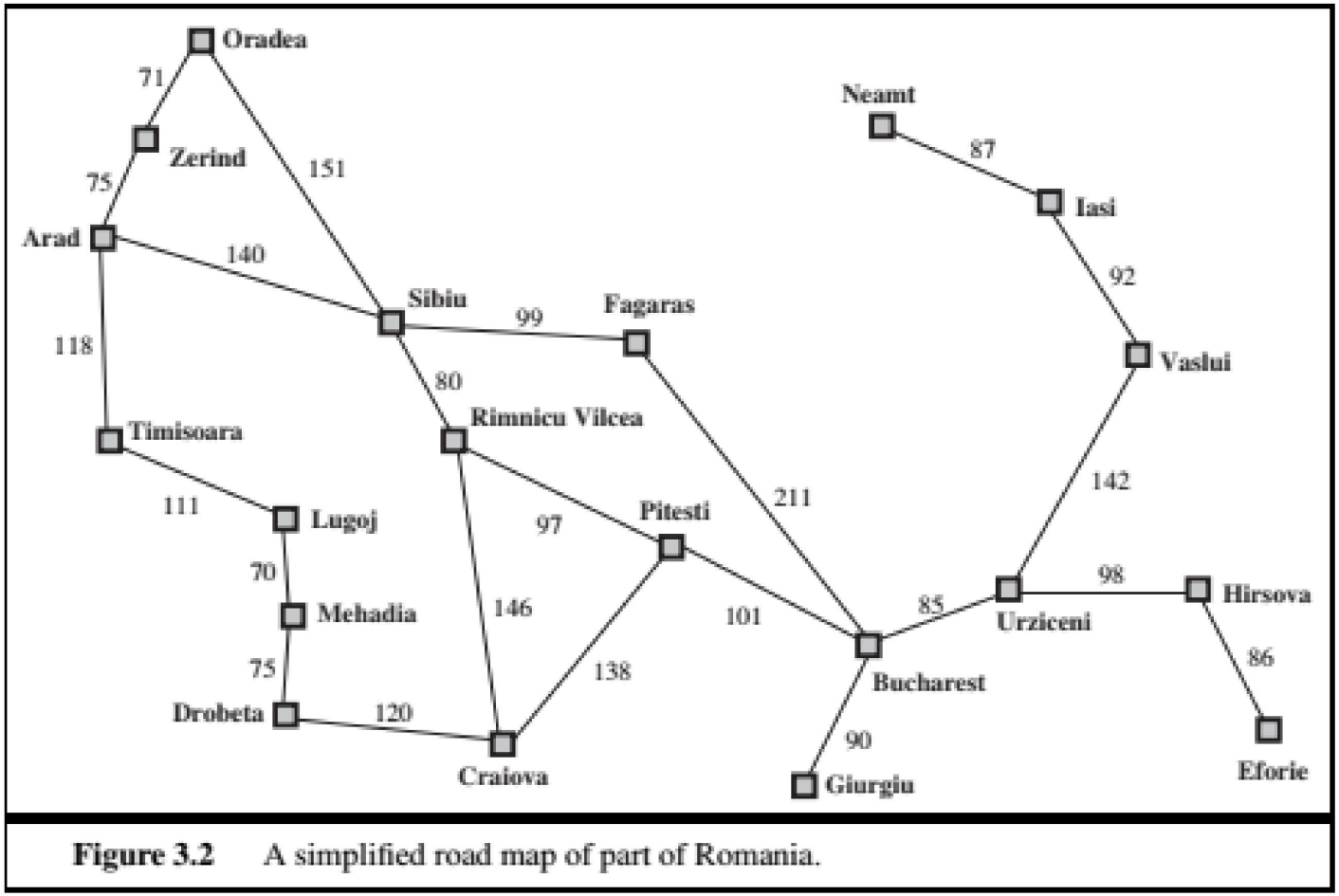


Figure 3.2 A simplified road map of part of Romania.

A simplified road map of part of Romania.

Q. Find the route from Arad to Bucharest.

Example Problem: Touring Holiday in Romania

- The agent on holiday in Romania; currently in Arad
- Have a nonrefundable flight ticket to fly out of Bucharest tomorrow
- Formulate goal:
  - be in Bucharest and catch the flight
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

A **problem** can be defined formally by **five components**:

1. The **initial state** that the agent starts in. For example, the initial state for our agent in Romania might be described as In(Arad).

2. A description of the possible **actions** available to the agent.
   - For example, from the state In(Arad), the applicable actions are {Go(Sibiu), Go(Timisoara), Go(Zerind)}.

3. **Transition model**: A description of what each action does
   RESULT(In(Arad), Go(Zerind)) = In(Zerind) .

Together, the initial state, actions, and transition model implicitly define the **state space/search space** of the problem—the **set of all states** reachable from the initial state by any sequence of actions.

The state space forms a directed network or **graph** in which the nodes are states and the links between nodes are actions.

Formulating Search Problems

4. The **goal test**, which determines whether a given state is a goal state. The agent's goal in Romania is the singleton set {In(Bucharest)}

5. A **path cost** function that **assigns a numeric cost to each path**. The problem solving agent chooses a cost function that reflects its own performance measure.

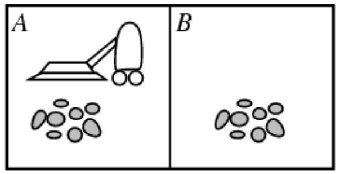
For the agent trying to get to Bucharest, the cost of a path might be its length in kilometers.

- In this chapter, we assume that:

  the cost of a path = the ***sum*** of the costs of the individual actions along the path.

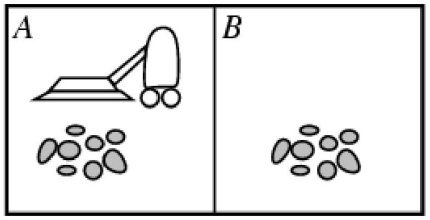
- The **step cost** of taking action ***a*** in state ***s*** to reach state ***s'*** is denoted by ***c(s, a, s')***.

Vacuum World



- **States/State Space**: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are 2 × (2 x 2) = 8 possible world states. A larger environment with **n locations** has **n x 2n states**.
- **Initial State**: Any state can be designated as the initial state.
- **Actions**: In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.
- **Transition Model**: The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Suck*ing in a clean square have no effect. The complete state space is shown in Figure 3.3.
- **Goal Test**: This checks whether all the squares are clean.
- **Path Cost**: Each step costs 1, so the path cost is the number of steps in the path.

Vacuum World



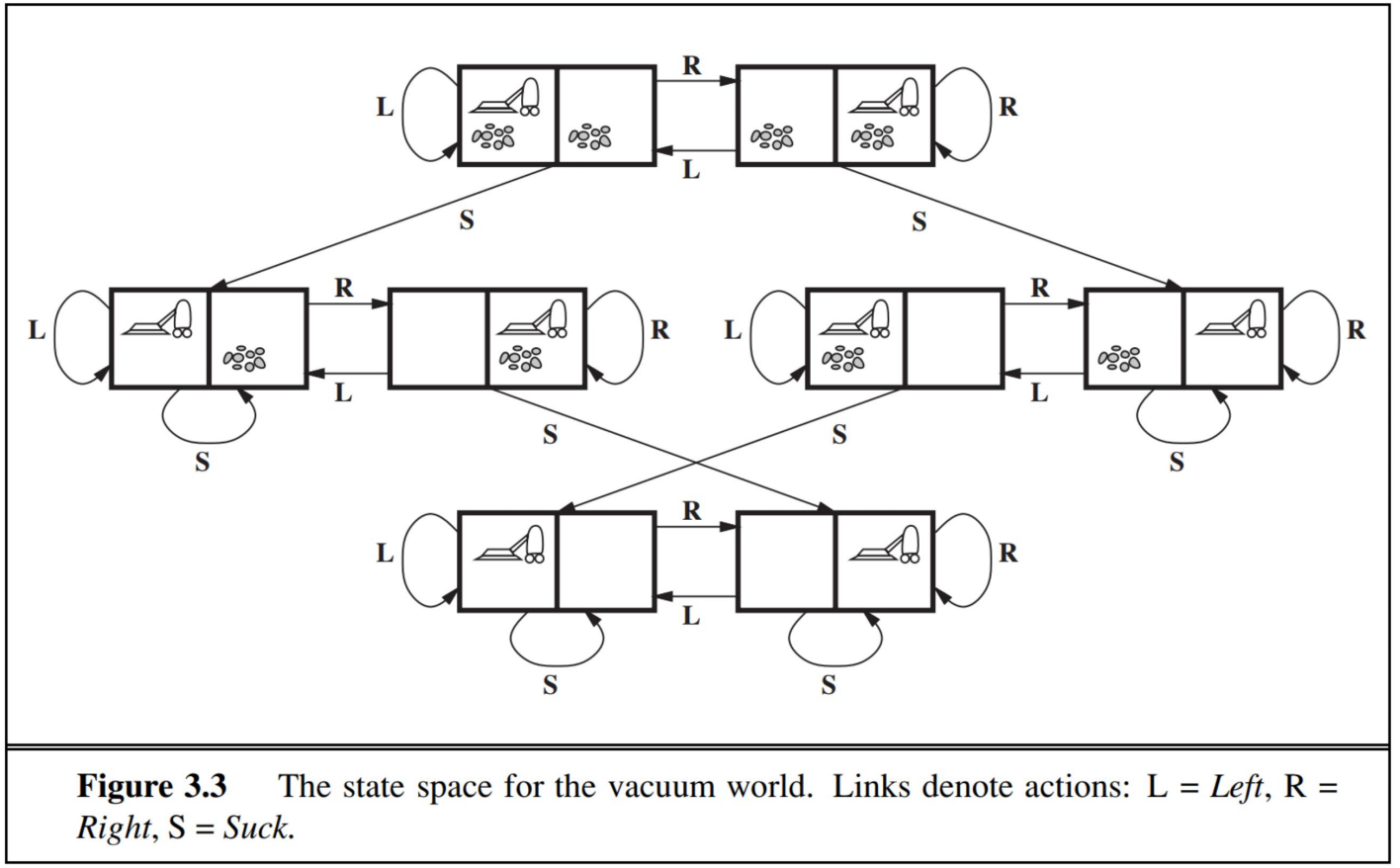


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

8-puzzle Game

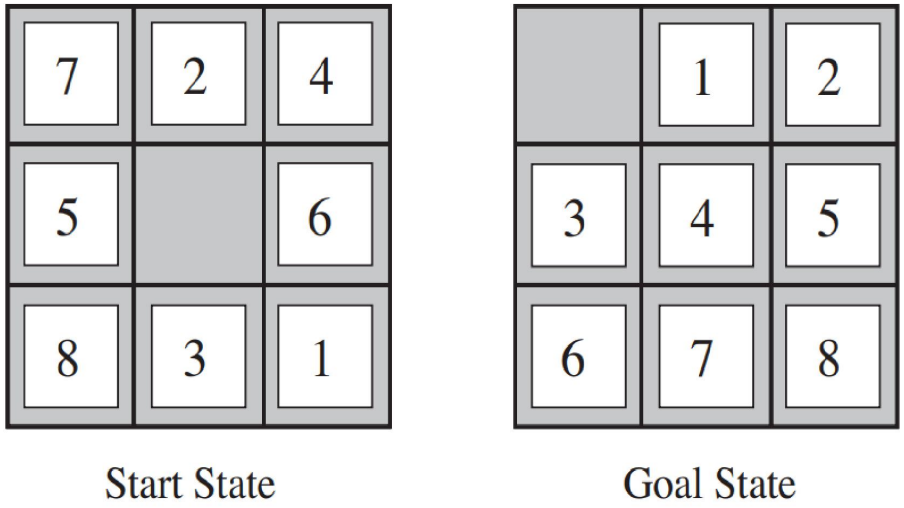
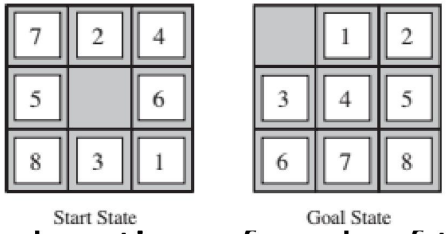


Figure 3.4 A typical instance of the 8-puzzle.

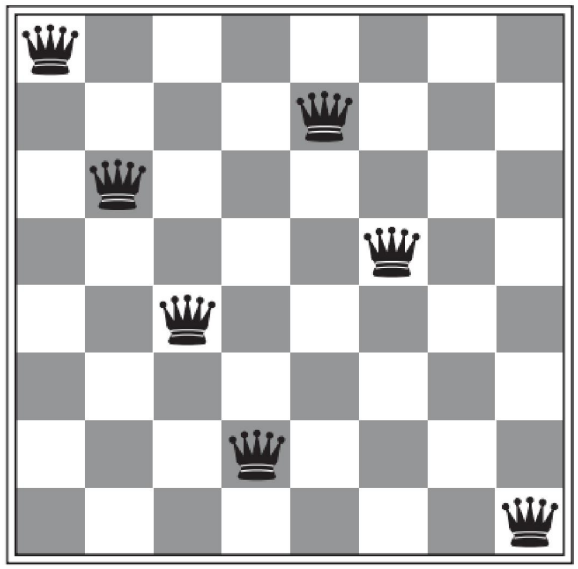
8-puzzle Game



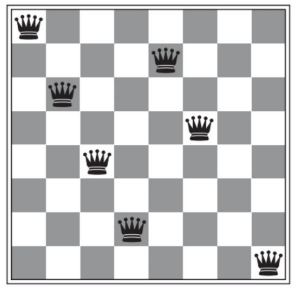
- **States/State Space**: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial State**: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states (Exercise 3.4).
- **Actions**: The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*. Different subsets of these are possible depending on where the blank is.
- **Transition Model**: Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state in Figure 3.4, the resulting state has the 5 and the blank switched.
- **Goal Test**: This checks whether the state matches the goal configuration shown in Figure 3.4. (Other goal configurations are also possible.)
- **Path Cost**: Each step costs 1, so the path cost is the number of steps in the path.

8-queens Problem

- The goal of the **8-queens problem** is to place eight queens on a chessboard such that no queen attacks any other.



8-queens Problem

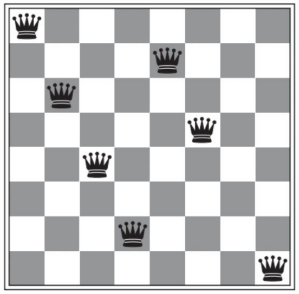


- **States**: Any arrangement of 0 to 8 queens on the board is a state.
- **Initial State**: No queens on the board.
- **Actions**: Add a queen to any empty square.
- **Transition Model**: Returns the board with a queen added to the specified square.
- **Goal Test**: 8 queens are on the board, none attacked.

In this formulation, we have 64 x 63 x · · · x 57 ≈ 1.8×10^{14} possible sequences to investigate.

Can we do better ?

8-queens Problem



- **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.

➢This formulation reduces the 8-queens state space from 1.8×10^{14} to just 2,057, and solutions are easy to find.

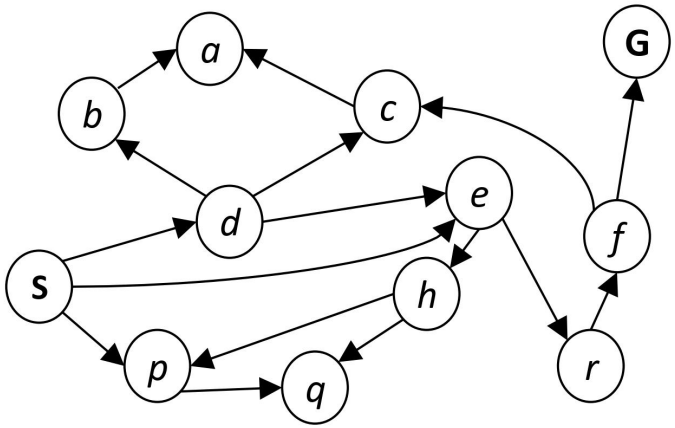
$8 \times 7 \times \cdots \times 1 \approx 2057$

State Space Graphs

- State Space Graph: A mathematical 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)

- In a search graph, each state occurs only once!

- We can rarely build this full graph in memory (it's too big), but it's a useful idea

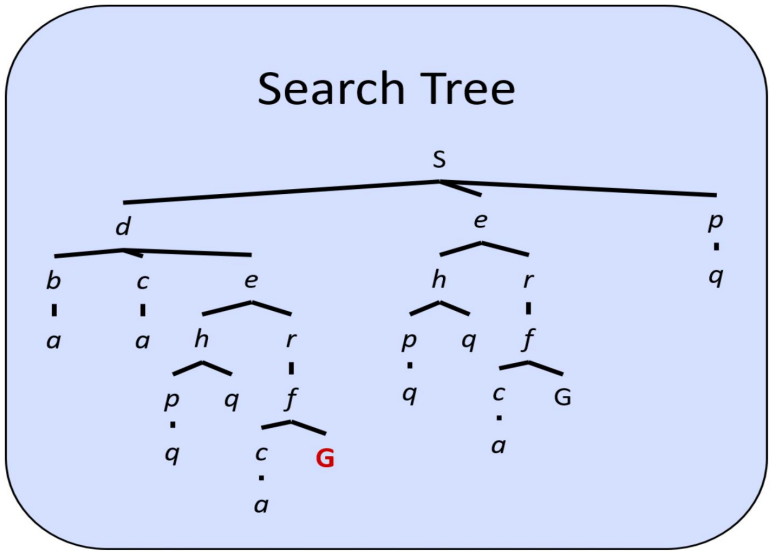


Tiny search graph for a tiny search problem

States vs. Nodes

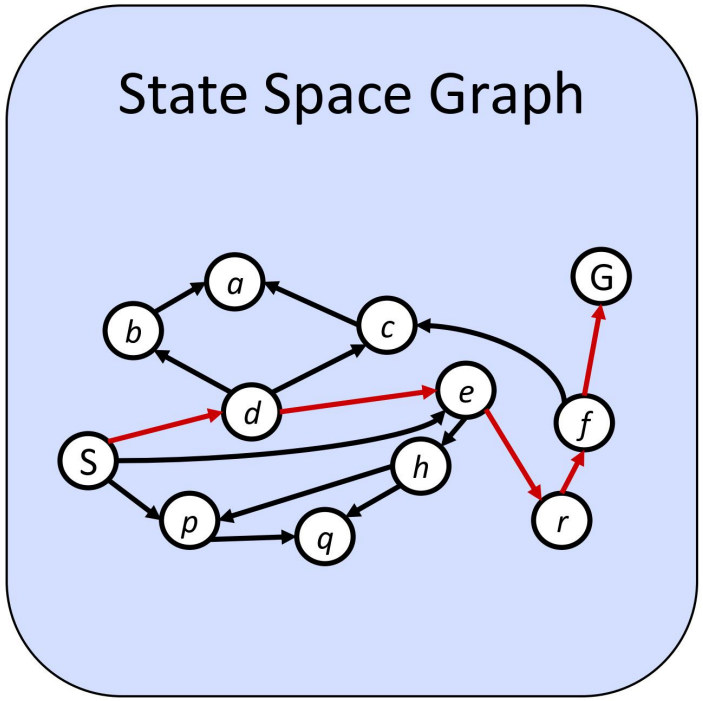
- A *state* is a (representation of a) physical configuration.
- A *node* is a data structure constituting part of a search tree includes *parent, children, depth, path cost g(n).*
- *States* do not have parents, children, depth, or path cost!

Search Trees



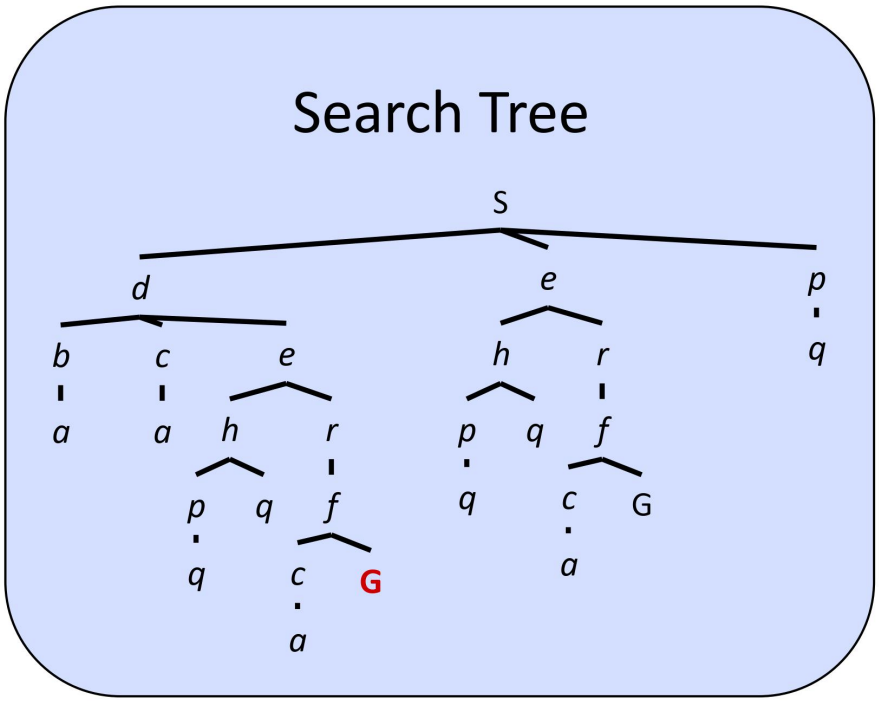
- A search tree:
  - A “what if” tree of plans and their outcomes
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those states
  - For most problems, we can never actually build the whole tree instead we build part by part,

State Space Graphs vs. Search Trees



Each NODE in in the search tree is an entire PATH in the state space graph.

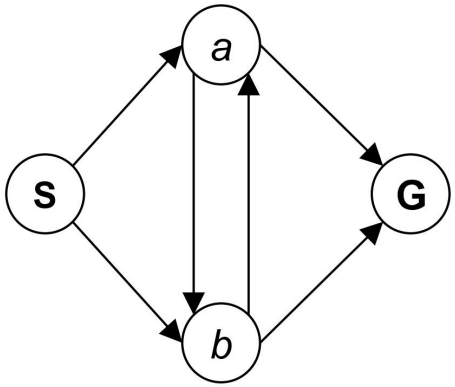
We construct both on demand – and we construct as little as possible (we build part by part).

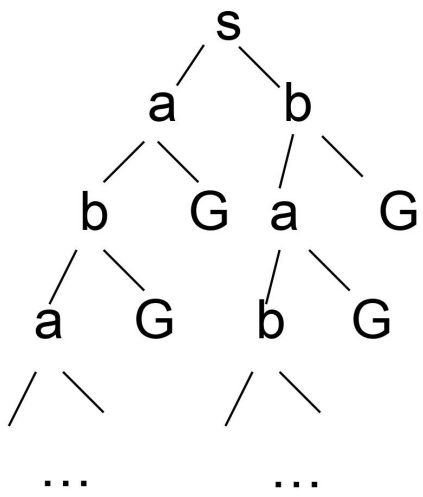


State Space Graphs vs. Search Trees

Consider this 4-state graph:

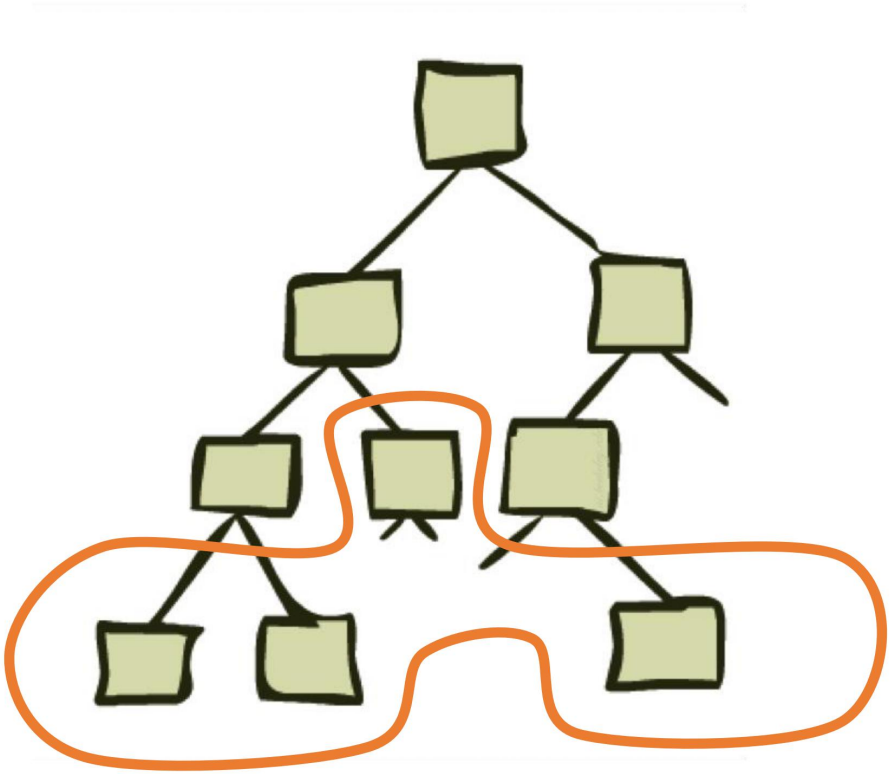
How big is its search tree (from S)?



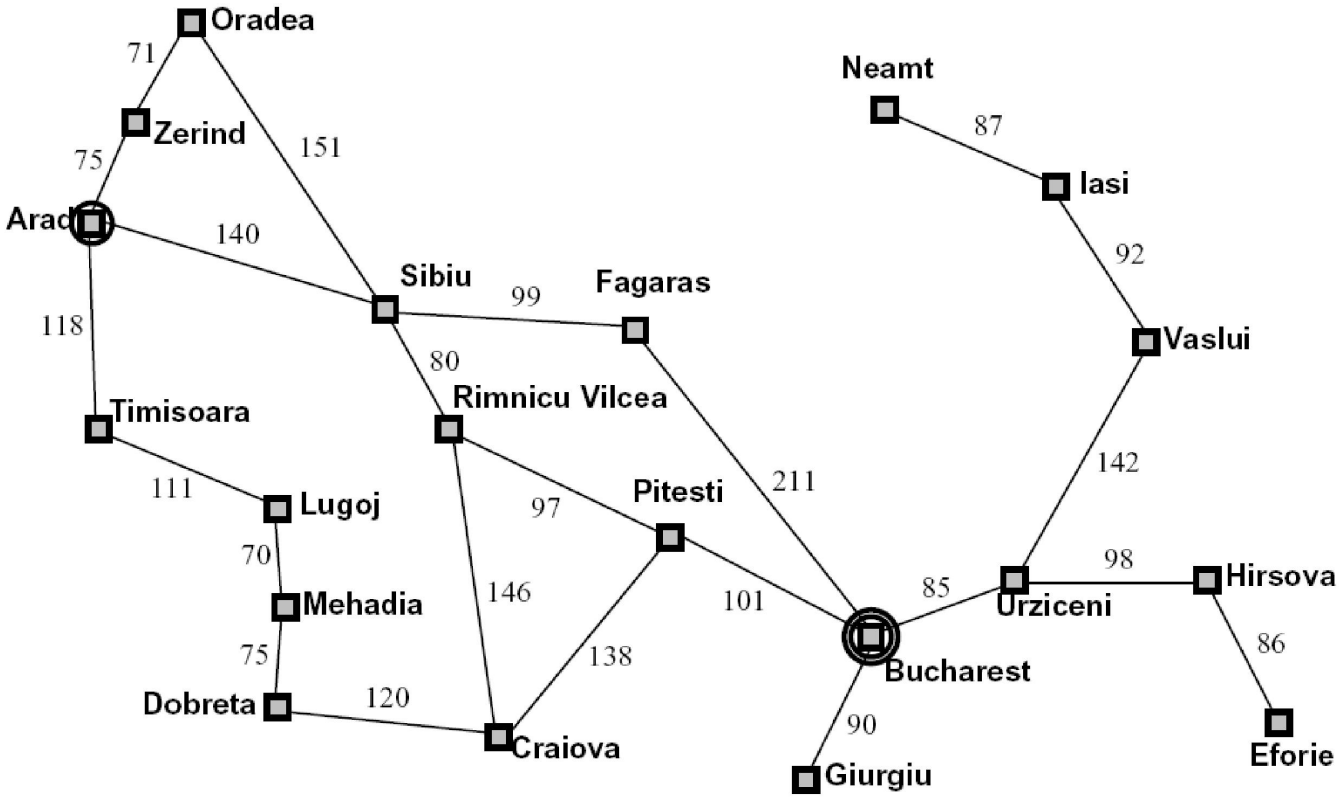


Important: Lots of repeated structure in the search tree!

Tree Search

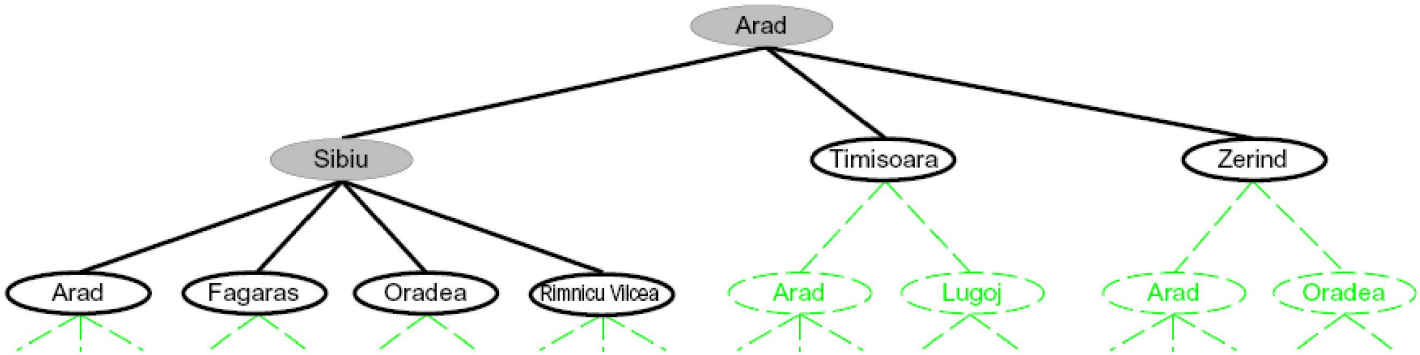


Search Example: Romania



Searching with a Search Tree

After expanding Sibiu



- Search:
  - Expand out potential plans (tree nodes)
  - Maintain a fringe of partial plans under consideration
  - Try to expand as few tree nodes as possible

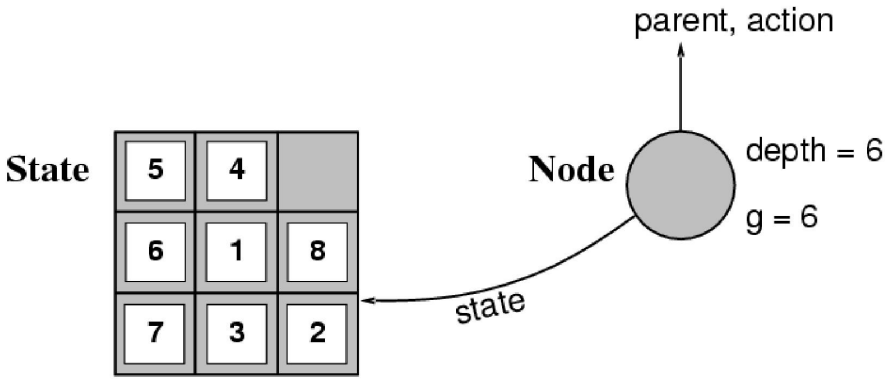
General Tree Search

```
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
```

- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy
- Main question: which fringe nodes to explore?

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



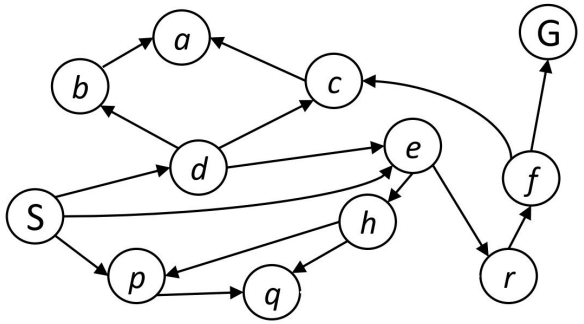
- The `Expand` function creates new nodes, filling in the various fields and using the `SuccessorFn` of the problem to create the corresponding states.

Implementation: general tree search

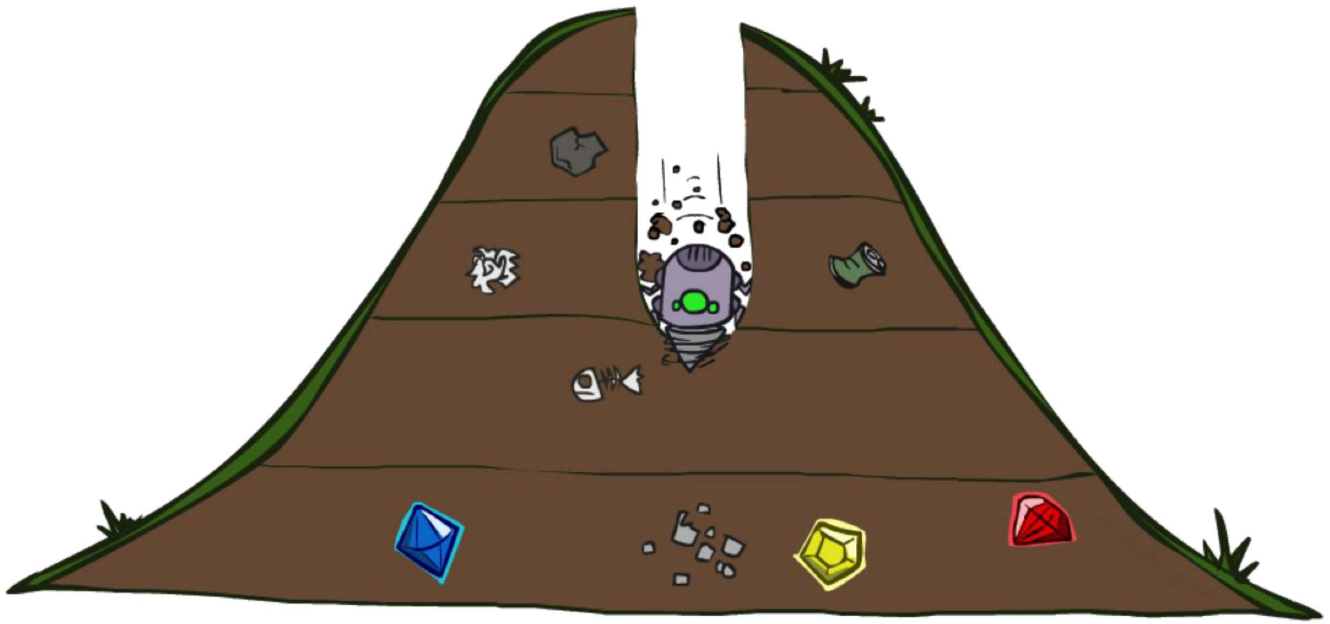
```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Example: Tree Search



Depth-First Search



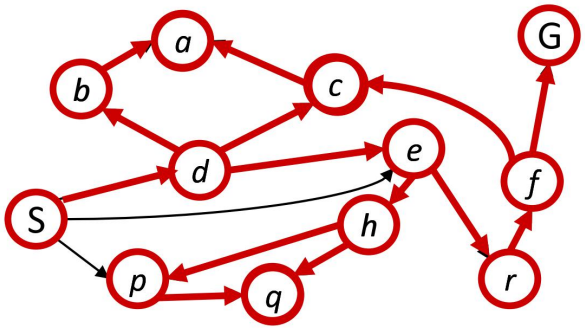
Depth-First Search

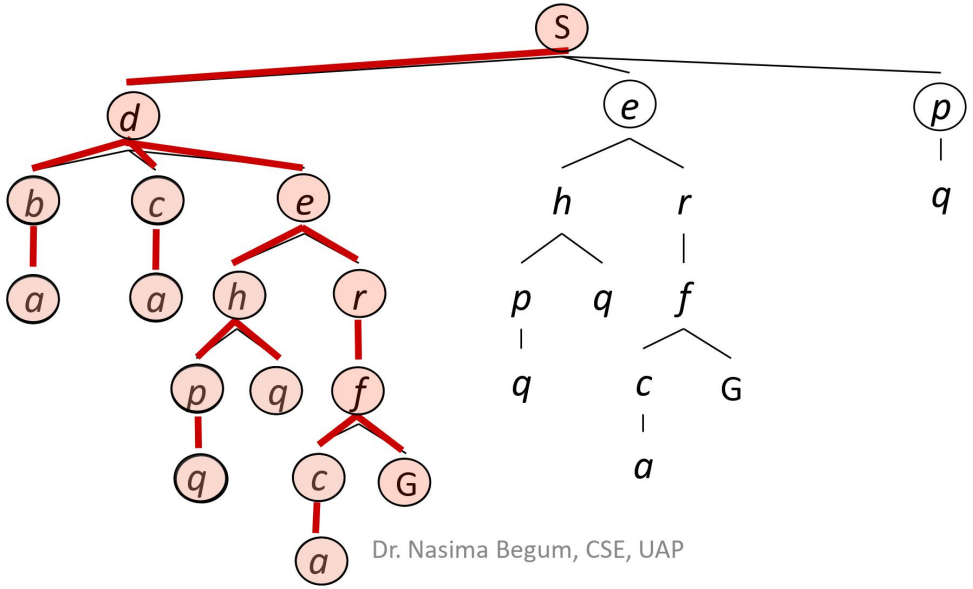
```
Function Depth-FIRST-SEARCH(problem) returns a solution, or failure
        node ←a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        frontier ← a LIFO queue with node as the only element
        explored ← an empty set
        loop do
                if EMPTY?( frontier) then return failure
                node ← POP( frontier) /* chooses the shallowest node in frontier */
                add node.STATE to explored
                for each action in problem.ACTIONS(node.STATE) do
                    child ← CHILD-NODE(problem, node, action)
                    if child.STATE is not in explored or frontier then
                        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                        frontier ← INSERT(child, frontier)
```

Depth-First Search

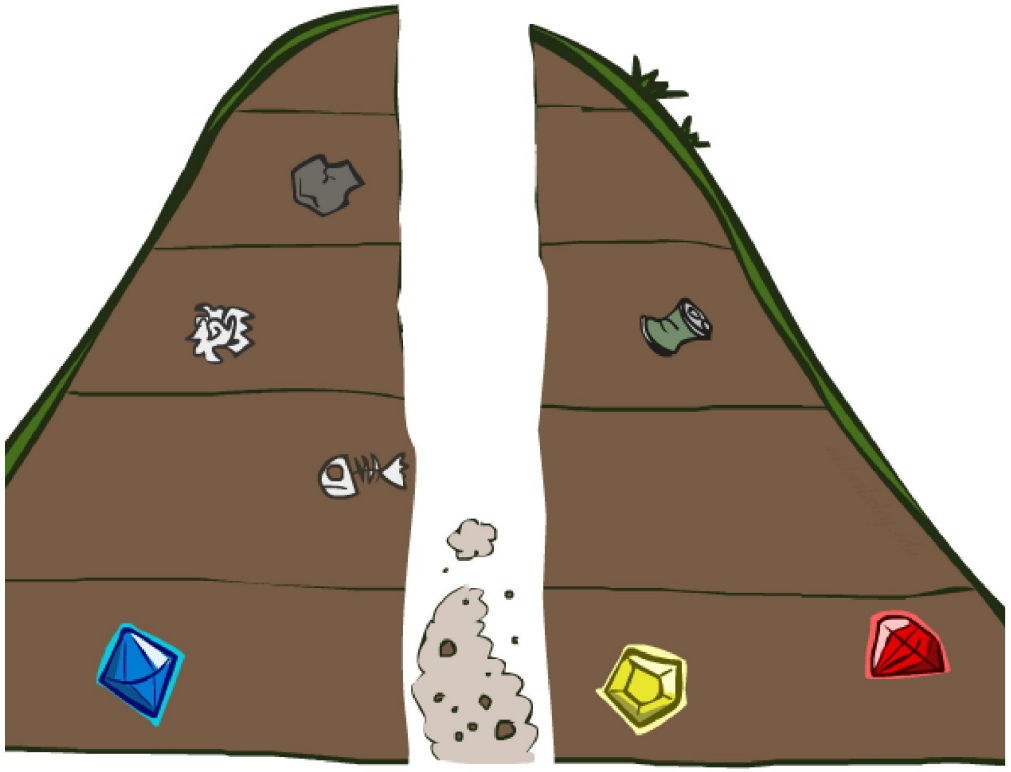
Strategy: expand a deepest node first

Implementation: Fringe is a LIFO stack





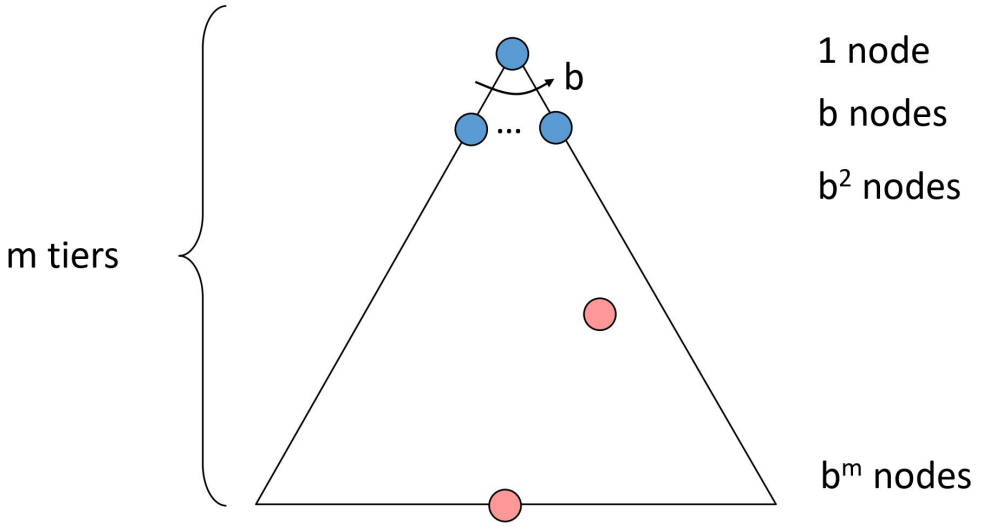
Search Algorithm Properties



Search Algorithm Properties

- Complete: Guaranteed to find a solution if one exists?
- Optimal: Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

- Cartoon of search tree:
  - b is the branching factor
  - m is the maximum depth
  - solutions at various depths

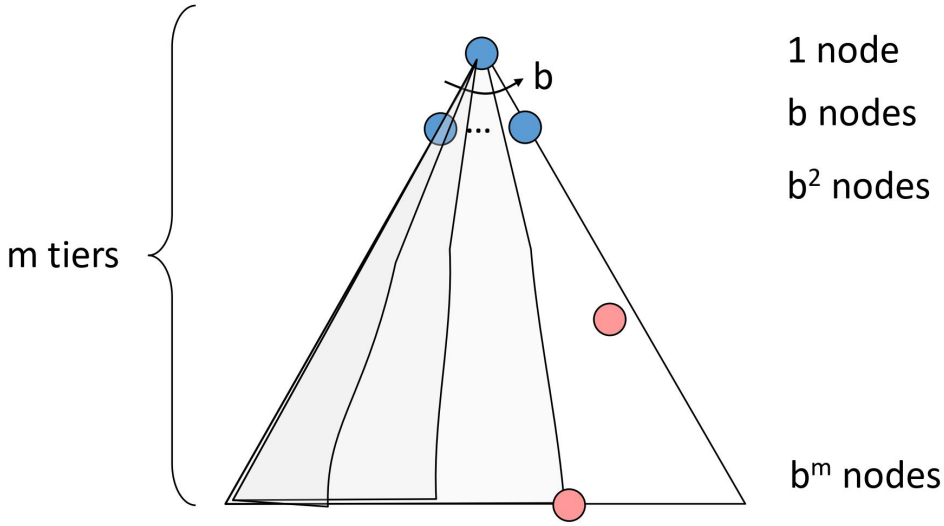


- Number of nodes in entire tree?
  - $1 + b + b^2 + \ldots. b^m = O(b^m)$

➢ **Big- O notation expresses the upper bound of growth function.**

Depth-First Search (DFS) Properties

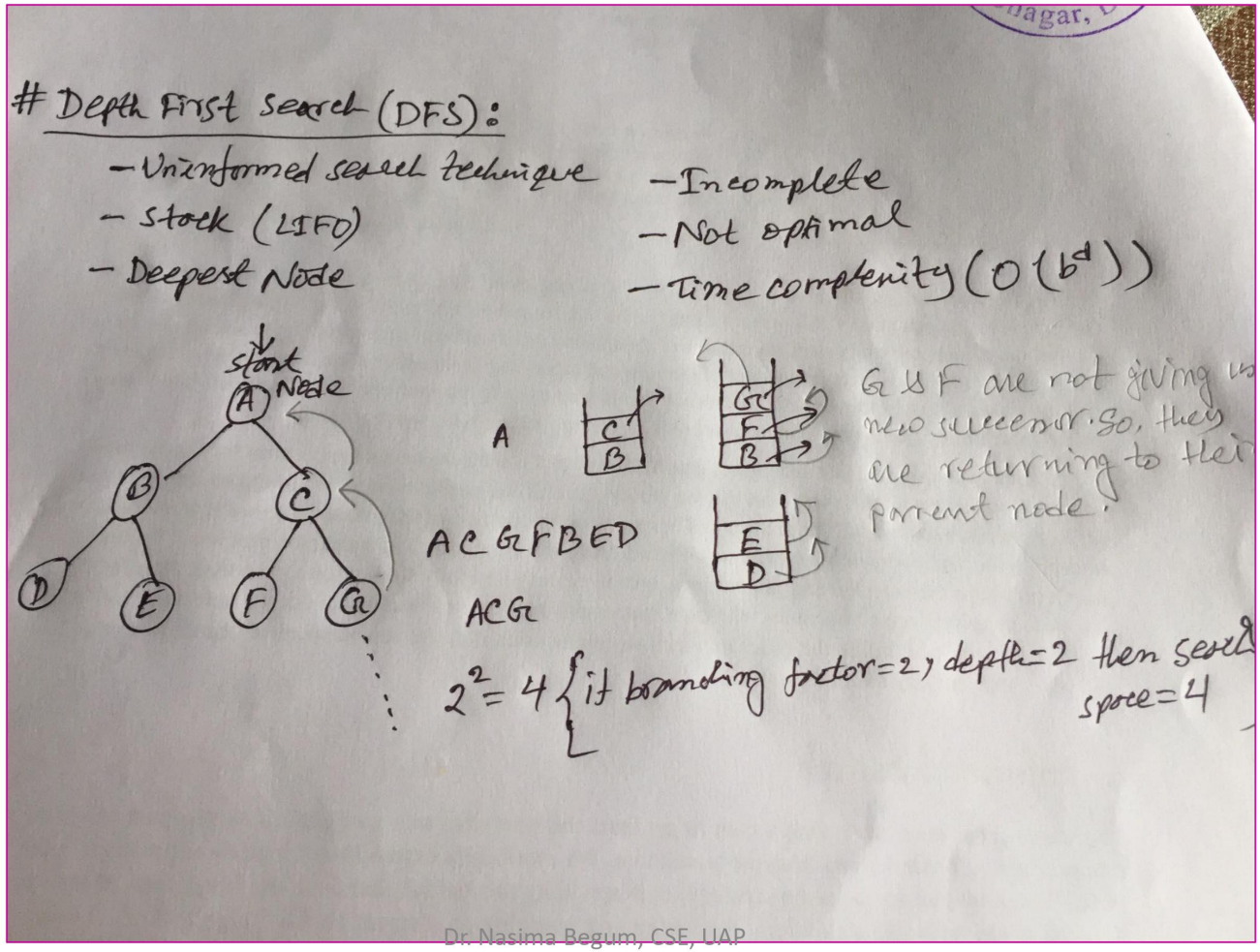
- What nodes DFS expand?
  - Some left prefix of the tree.
  - Could process the whole tree!
  - If m is finite, takes time $O(b^m)$
- How much space does the fringe take?
  - Only has siblings on path to root, so O(bm)
- Is it complete?
  - m could be infinite, so only if we prevent cycles (more later)
- Is it optimal?
  - No, it finds the "leftmost" solution, regardless of depth or cost



DFS

- **Complete** No: fails in infinite-depth spaces, spaces with loops Modify to avoid repeated states along path.
  ⇒ complete in finite spaces
- **Time** $O(b^m)$: $1 + b + b^2 + b^3 + \ldots + b^m = O(b^m)$ bad if m is much larger than d but if solutions are dense, may be much faster than BFS.
- **Space** $O(bm)$ linear space complexity!(needs to store only a single path from the root to a leaf node, **along with the remaining unexpanded sibling nodes for each node on the path, hence the m factor**.)
- **Optimal** No
- **Implementation:** fringe: LIFO (Stack)

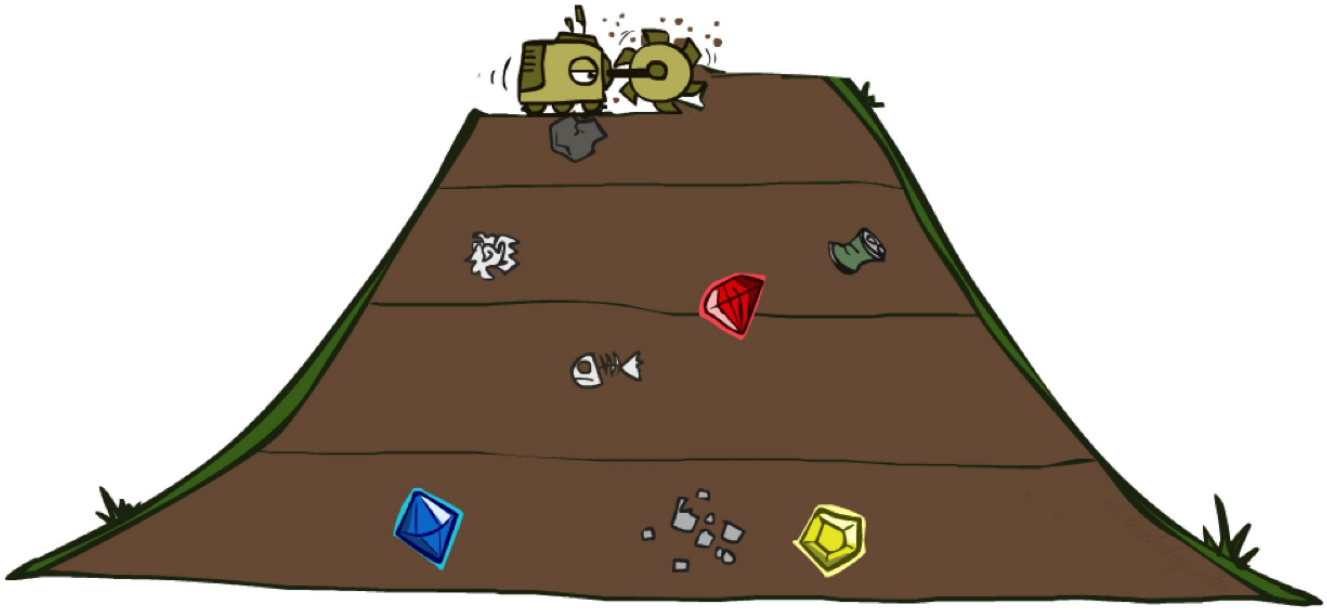
Example of Depth-First Search



Application of Depth-First Search (DFS)

- Path Finding
- Detecting Cycle in a Graph
- Topological Sorting
- Finding Strongly Connected Components
- To test if a Graph is Bipartite
- Solving puzzles with only one solution, such as ‘mazes’

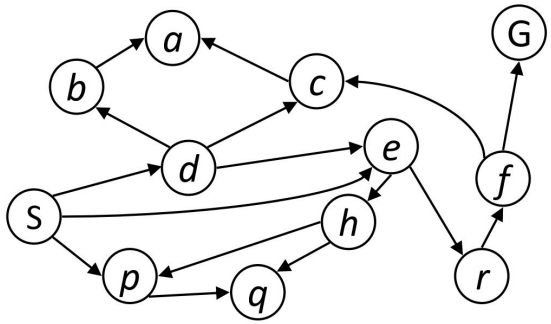
Breadth-First Search

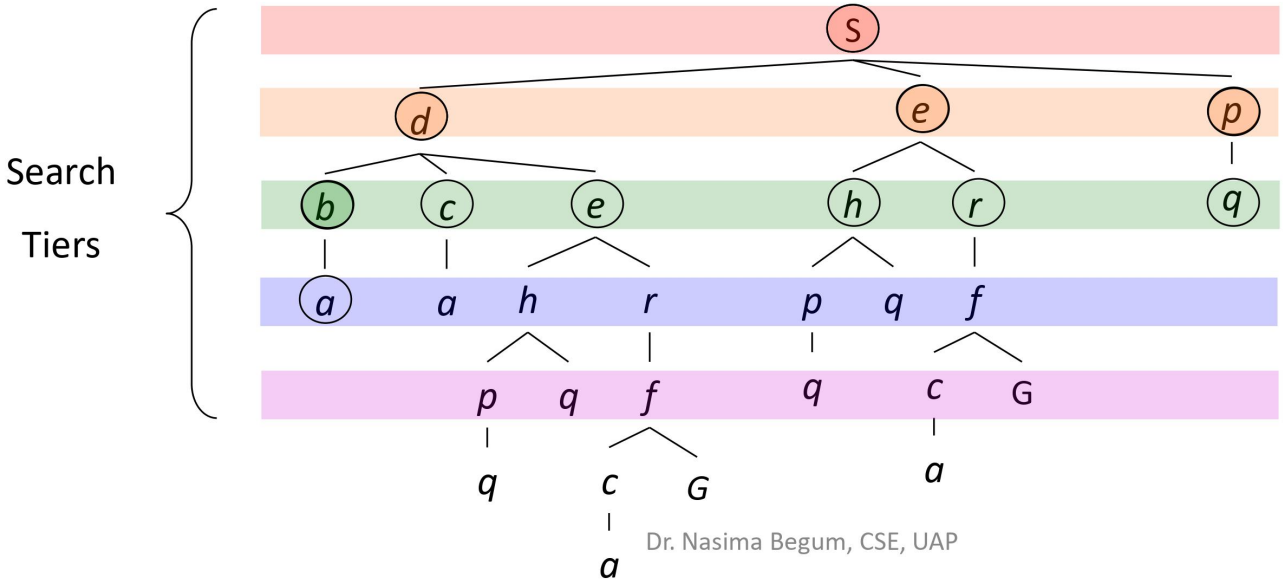


Breadth-First Search

Strategy: expand a shallowest node first

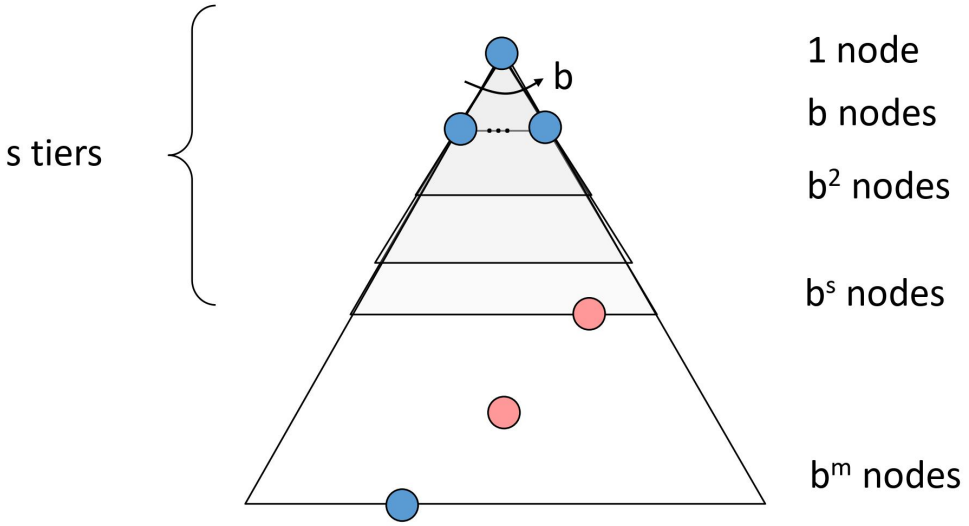
Implementation: Fringe is a FIFO queue



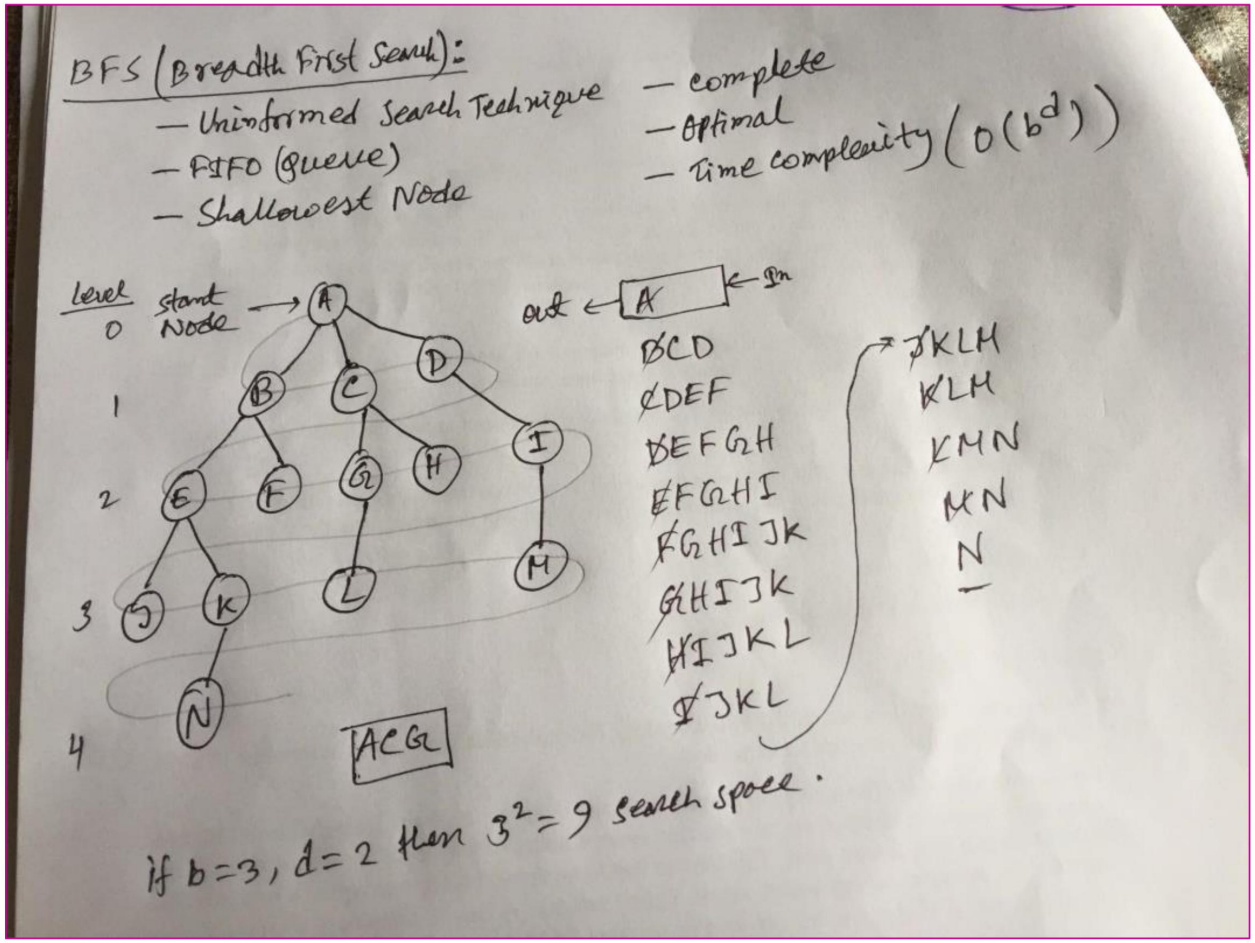


Breadth-First Search (BFS) Properties

- What nodes does BFS expand?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be s
  - Search takes time $O(b^s)$
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^s)$
- Is it complete?
  - s must be finite if a solution exists, so yes!
- Is it optimal?
  - Only if costs are all 1 (more on costs later)



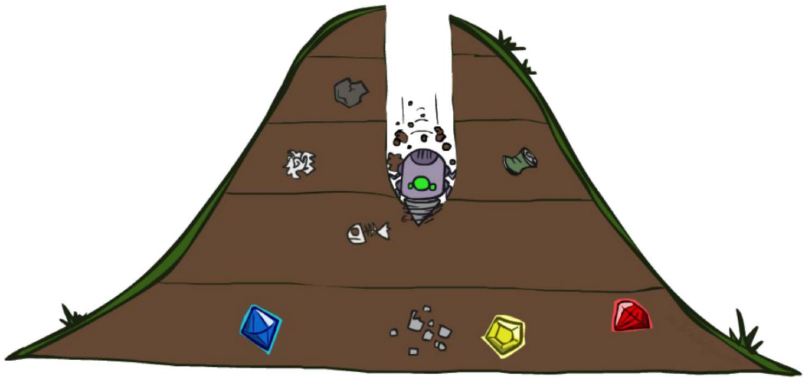
Example of Breadth-First Search

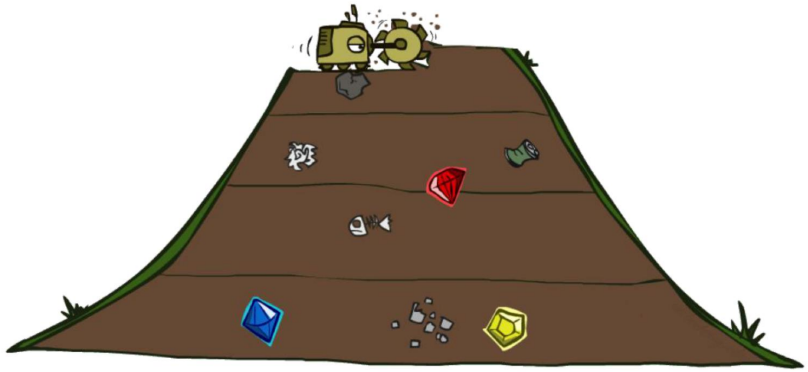


Application of Breadth-First Search

- Shortest Path in a Graph
- Social Network
- Web Crawler
- Cycle Detector
- To test if a graph is Bipartite
- Broad casting in a Network
- Ford-Fulkerson Algorithm

Quiz: DFS vs BFS





Quiz: DFS vs BFS

- When will BFS outperform DFS?

- When will DFS outperform BFS?

When should we use BFS instead of DFS, and so on?

- In general, *usually*, we would like to:
- *Use **BFS*** - when we want to find the ***shortest*** path from a certain source node to a certain destination. (Or more generally, the smallest number of steps to reach the end state from a given initial state.)
- *Use **DFS*** - when we want to ***exhaust*** all possibilities, and check which one is the best/count the number of all possible ways.
- *Use either **BFS or DFS*** - when we just want to check ***connectedness*** between two nodes on a given graph. (Or more generally, whether we could reach a given state to another.)

When should we use BFS instead of DFS, and so on?

- As we know, DFS is a recursive algorithm, we can write it using a stack, but it does not change its recursive nature, so DO NOT USE it on infinitely deep graphs. DFS is not a complete algorithm for infinitely deep graphs (it does not guarantee to reach the goal if there is any).

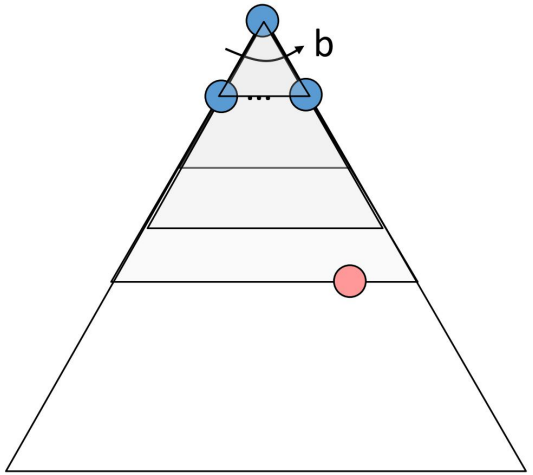
- Even if our graph is very deep but we have the prior knowledge that our goal is a shallow one, using DFS is not a very good idea.

- Also BFS needs to keep all the current nodes in the memory, so DO NOT USE BFS on the graphs with high branching factor, or our machine will run out of memory very quickly.

- If we are facing an infinitely deep graph with a high branching factor, we can use Iterative Deepening (IDS) algorithm.

Iterative Deepening Search (IDS)

- Idea: get DFS's space advantage with BFS's time / shallow-solution advantages
  - Run a DFS with depth limit 1. If no solution…
  - Run a DFS with depth limit 2. If no solution…
  - Run a DFS with depth limit 3. …..
- Isn't that wastefully redundant?
  - Generally most work happens in the lowest level searched, so not so bad!



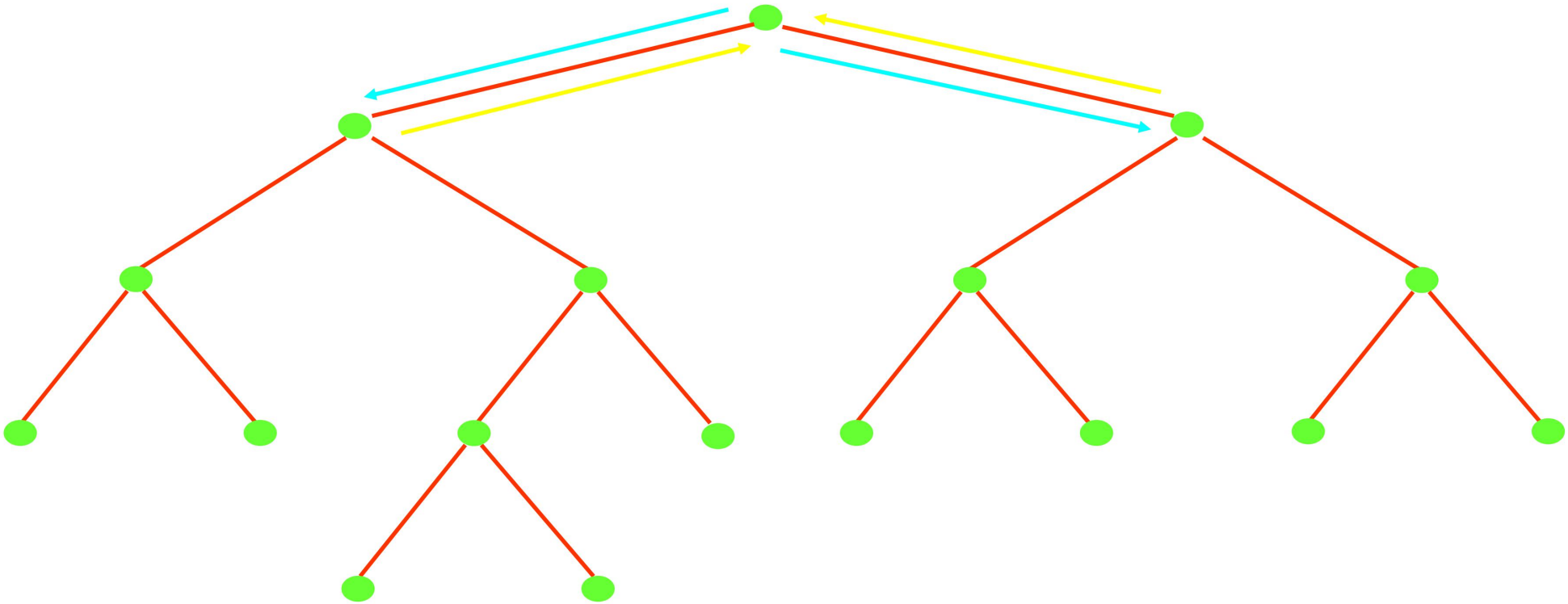
Iterative Deepening Search (IDS)

- Iterative deepening is an interesting **combination** of breadth-first and depth-first strategies:
- Space complexity for search depth n is O(n).
- Is guaranteed to find the shortest path to a solution without searching unnecessarily deep.

- How does it work?

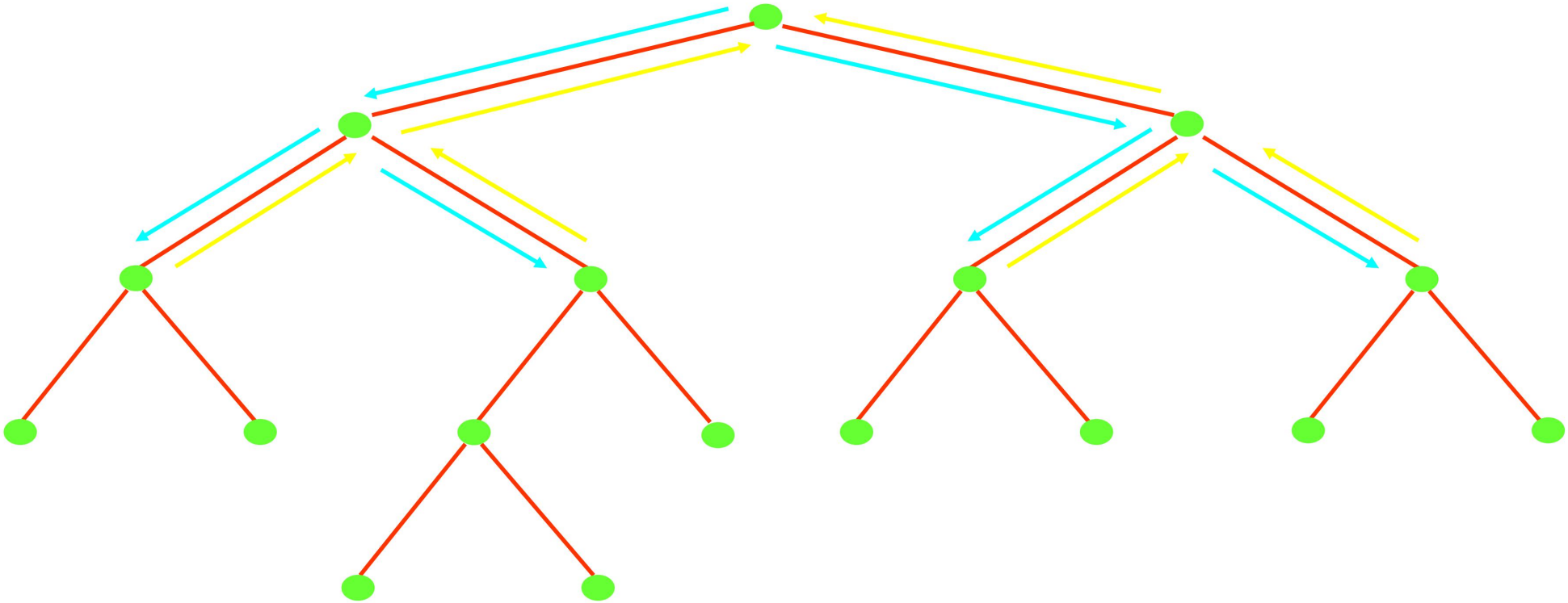
- The idea is to successively apply depth-first searches with **increasing depth bounds** (maximum search depth).

Iterative Deepening Search (IDS)



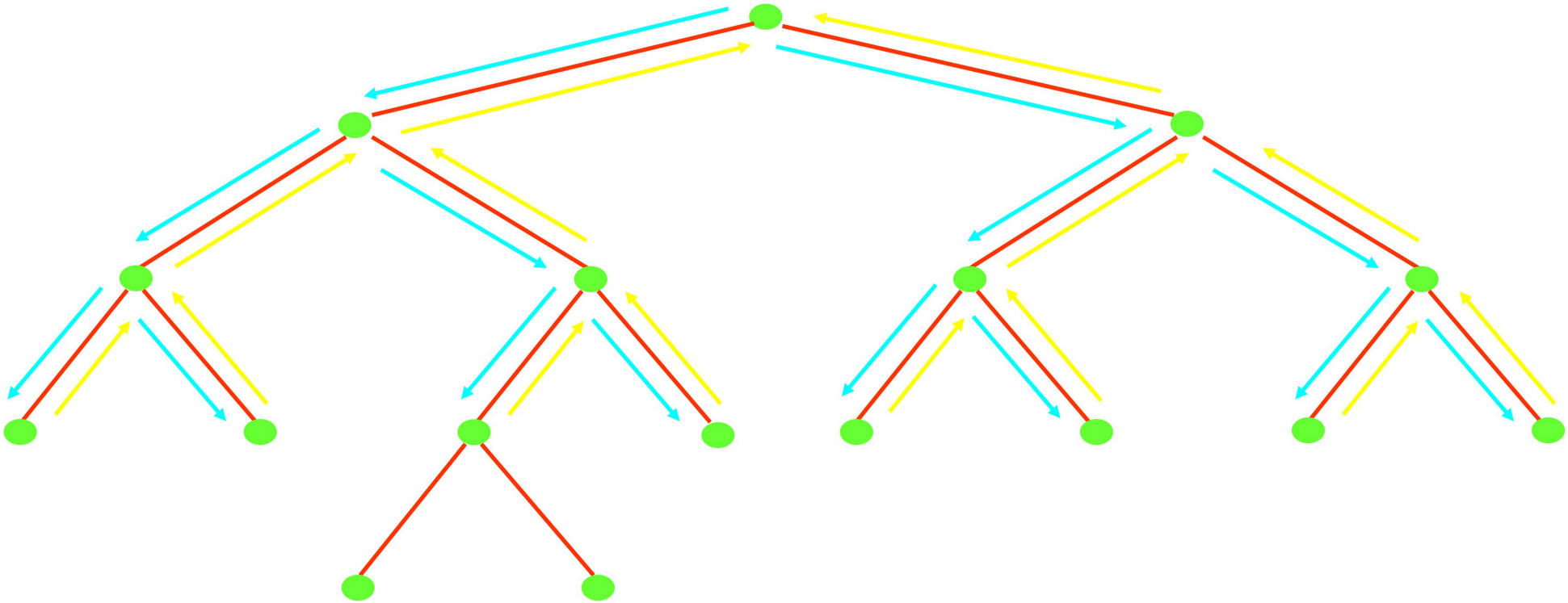
maximum search depth = 1

Iterative Deepening Search (IDS)



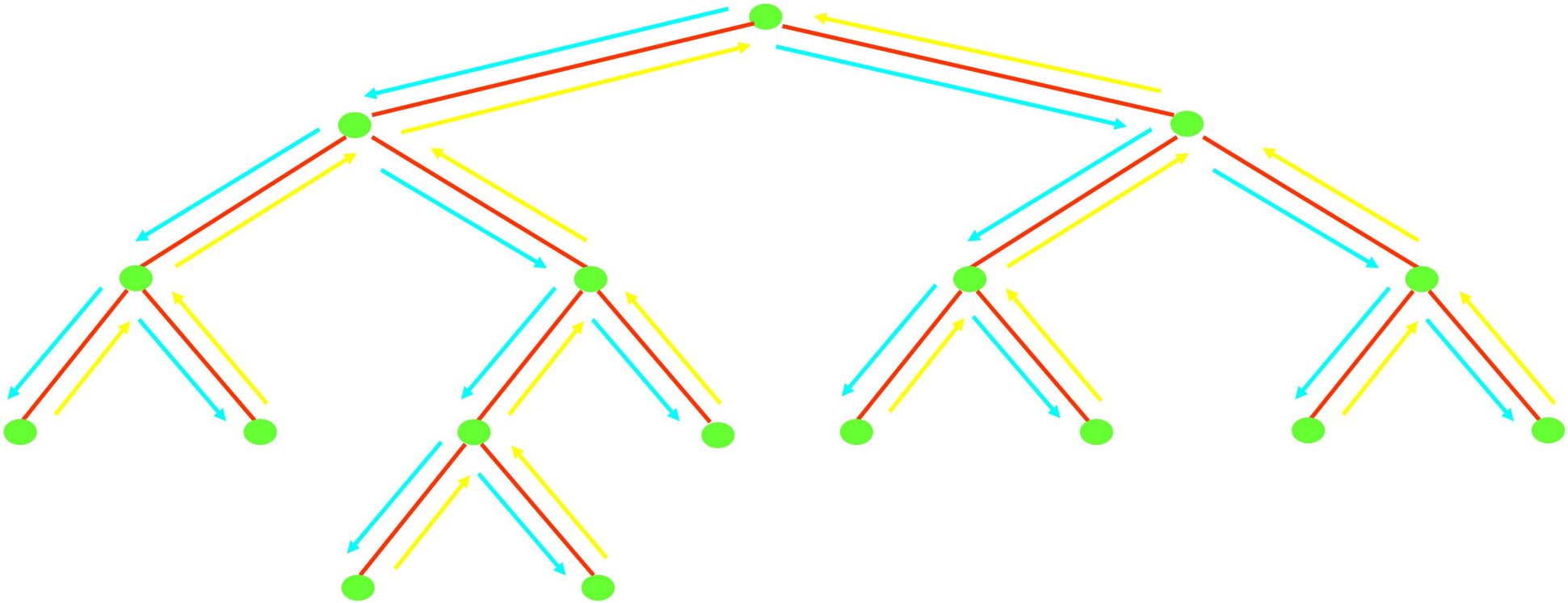
maximum search depth = 2

Iterative Deepening Search (IDS)



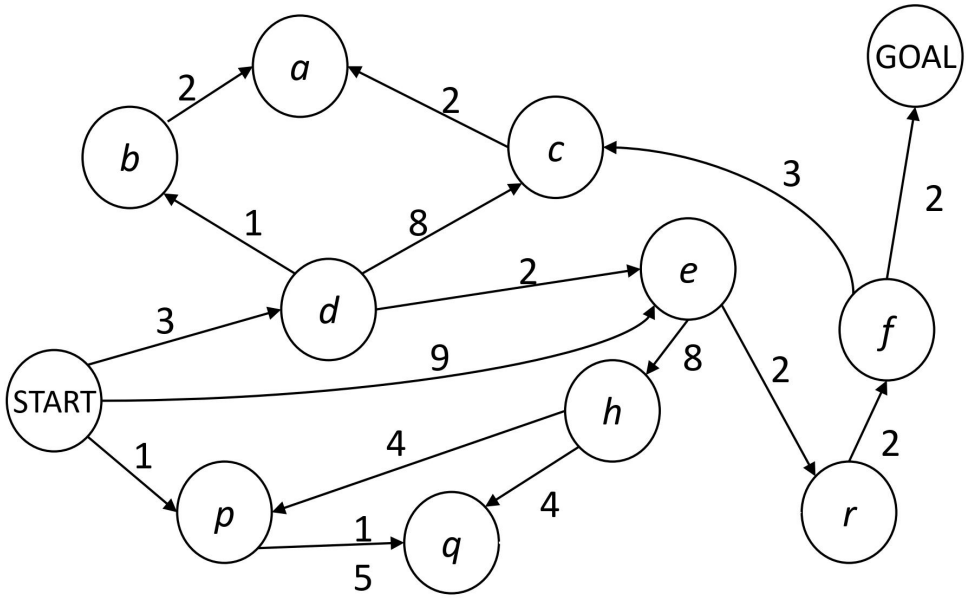
maximum search depth = 3

Iterative Deepening Search (IDS)



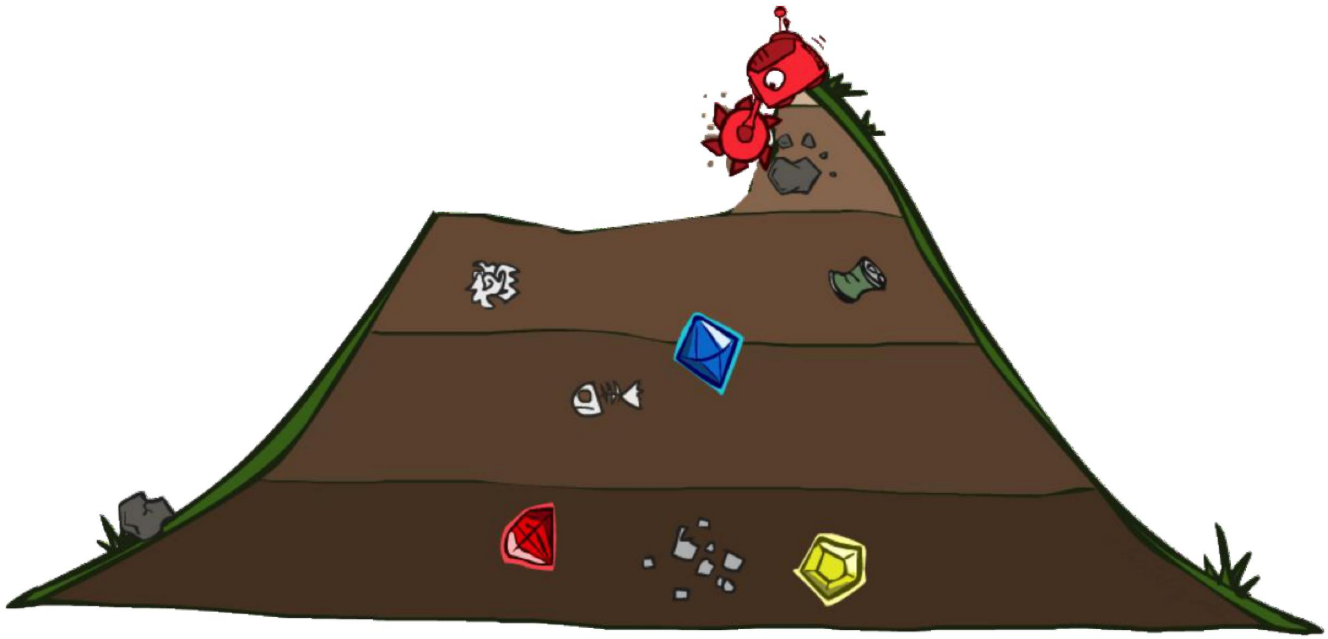
maximum search depth = 4

Cost-Sensitive Search



- BFS finds the shortest path in terms of number of actions. It does not find the least-cost path. We will now cover a similar algorithm which does find the least-cost path.
- We want the cheapest not shallowest solution.
- Modify BFS: Prioritize by cost not depth → Expand node n with the lowest path cost g(n).

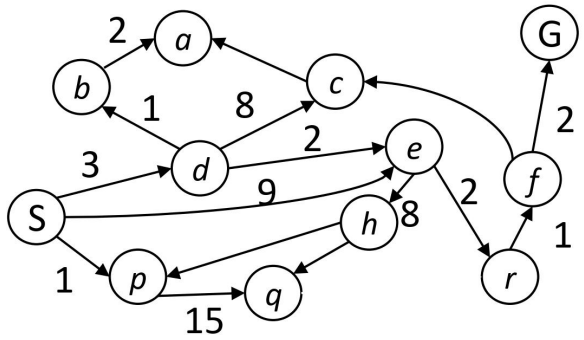
Uniform Cost Search (UCS)

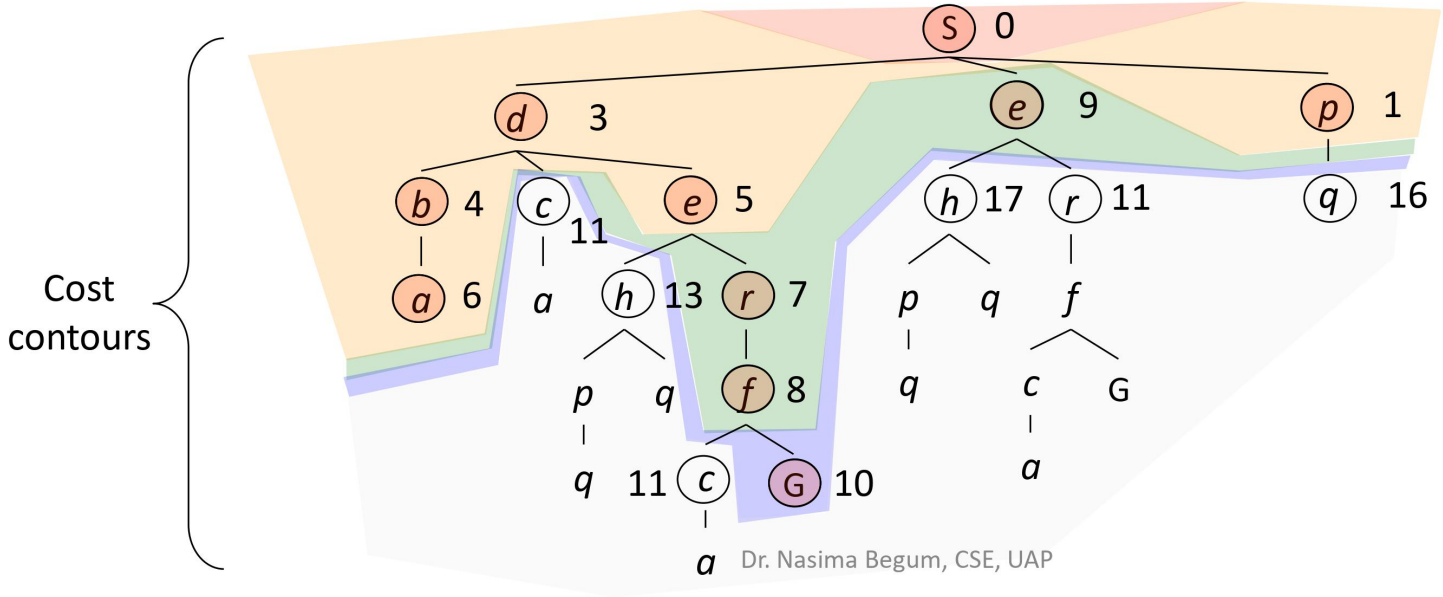


Uniform Cost Search (UCS)

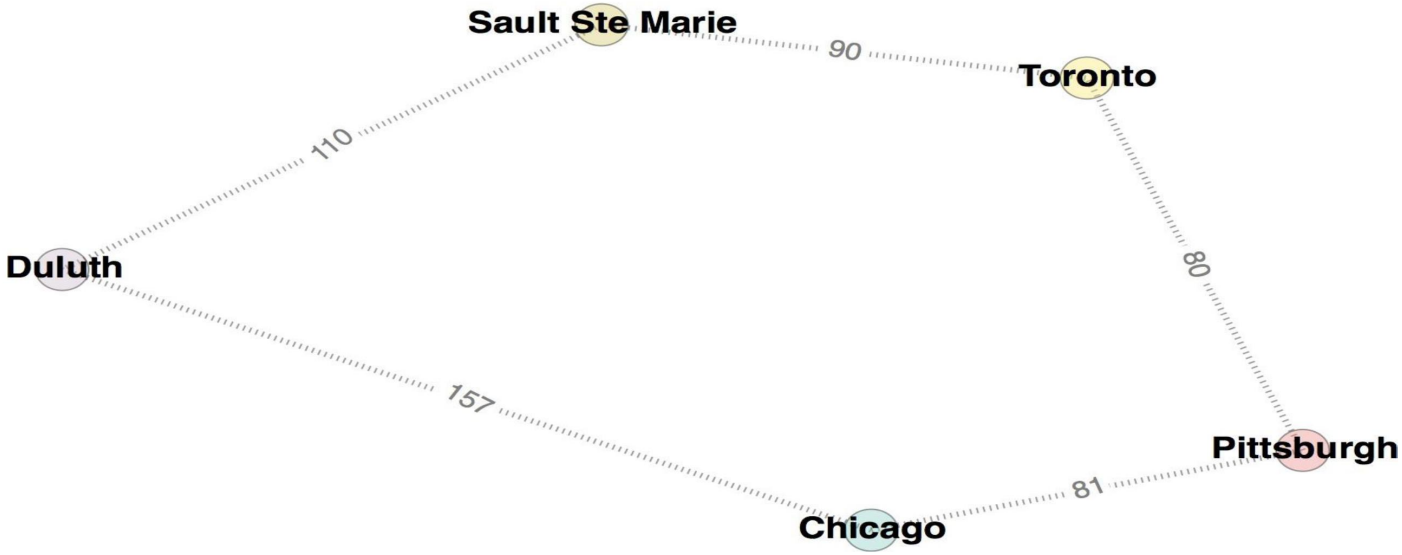
Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)





Uniform-cost Search



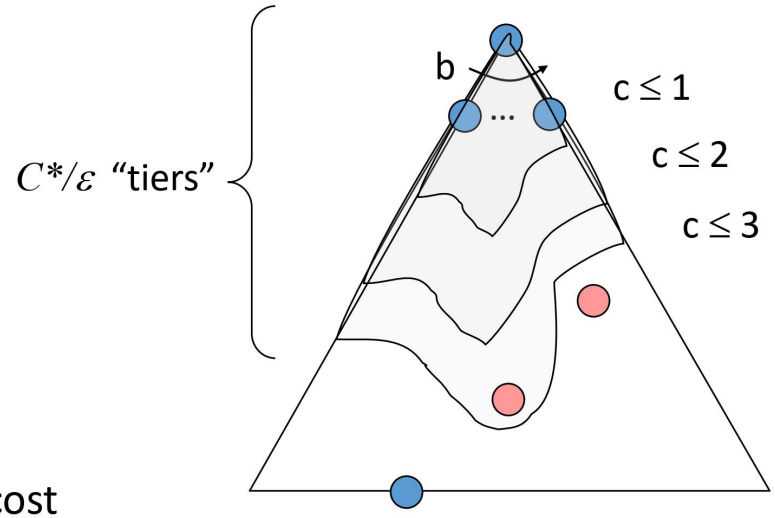
- Go from Chicago to Sault Ste Marie. Using BFS, we would find:

  Chicago → Duluth →Sault Ste Marie.

- However, using UCS, we would find Chicago → Pittsburgh → Toronto → Sault Ste Marie, which is actually the shortest path!

Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$
- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!
- Is it optimal?
  - Yes! (Can proof via A*)

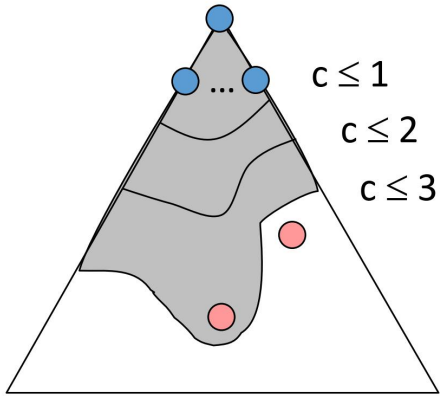


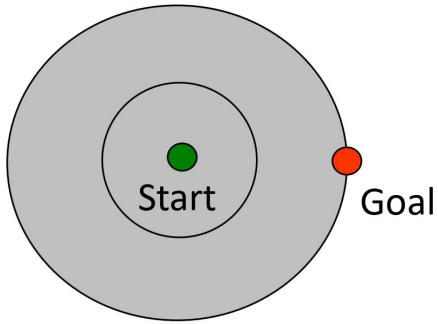
Uniform-cost Search

- **Complete** Yes, if solution has a finite cost.
- **Time**
  - Suppose C^*: cost of the optimal solution
  - Every action costs at least s (bound on the cost)
  - The effective depth is roughly C^*/s (how deep the *cheapest* solution could be).
  - $O(b^{C^*/s})$
- **Space** # of nodes with $g \leq$ cost of optimal solution, $O(b^{C^*/s})$
- **Optimal** Yes
- **Implementation**: fringe = queue ordered by path cost $g(n)$, lowest first = Heap!

Uniform Cost Issues

- Remember: UCS explores increasing cost contours
- The good:
  - UCS is complete and optimal!
  - Expand least cost node
- The bad:
  - Explores options in every “direction”
  - No information about goal location
- We’ll fix that soon!

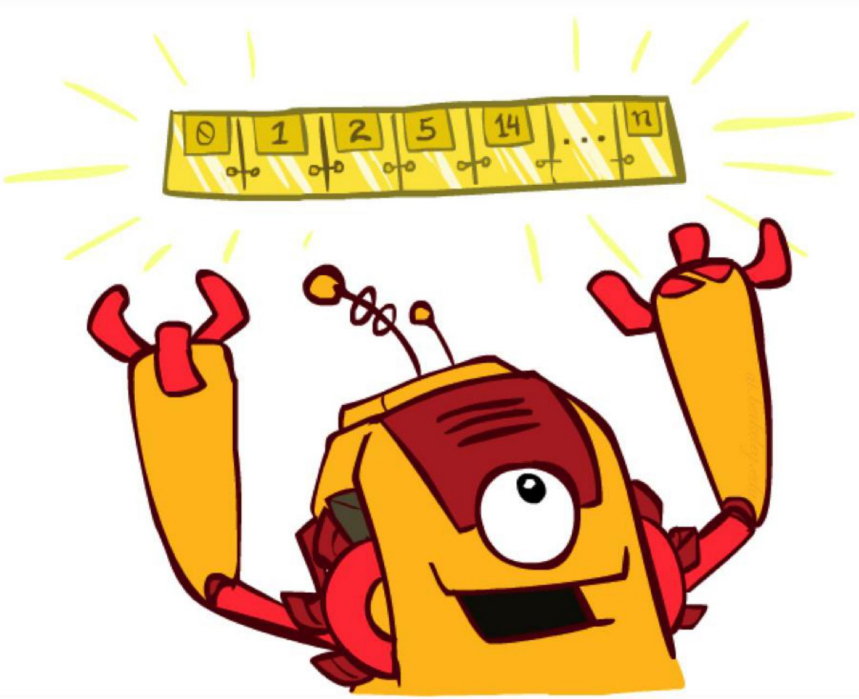




The One Queue

DFS -> Death
BFS -> Level
IDS -> level + depth
UCS -> cummitive cost

- All these search algorithms are the same except for fringe strategies
  - Conceptually, all fringes are priority queues (i.e. collections of nodes with attached priorities)
  - Practically, for DFS and BFS, we can avoid th log(n) overhead from an actual priority queue, by using stacks and queues
  - Can even code one implementation that takes a variable queuing object



Acknowledgement

- AIMA = Artificial Intelligence: A Modern Approach by Stuart Russell and Peter Norving (3rd edition)
- UC Berkeley (Some slides were created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley)
- U of toronto
- Other online resources

Thank You