Chapter 3: Problem Solving Agents

Dr Ammar Masood

Department of Cyber Security,

Air University Islamabad

Contents

- Problem solving by searching
- Problem Formulation
- Search strategies
- Uninformed search strategies

Context

- The simplest agents discussed in Chapter 2 were the reflex agents, which base their actions on a direct mapping from states to actions.
- Such agents cannot operate well in environments for which this mapping would be too large to store and would take too long to learn.

Cont..

Goal-based agents, on the other hand, consider future actions and the desirability of their outcomes. This chapter describes one kind of goal-based agent called a problem-solving agent. Problem-solving agents use atomic representations

Goal based AI agents

Goal-based AI agents represent a sophisticated approach in artificial intelligence (AI), where agents are programmed to achieve specific objectives. These agents are designed to plan, execute, and adjust their actions dynamically to meet predefined goals.

Goal-based Agents

- Choose an action that at least leads to a state that is closer to a goal than the current one is.

Making that work can be tricky

- What if one or more of the choices you make turn out not to lead to a goal?
- What if you're concerned with the best way to achieve some goal?
- What if you're under some kind of resource constraint?

Problem Solving as Search

One way to address these issues is to view goal-attainment as problem solving, and viewing that as a search through a state space.

In chess, e.g., a state is a board configuration

Important concepts within the context

- *The solution to any problem is a fixed sequence of* ***actions***
- *The process of looking for a sequence of actions that reaches the goal 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***
- *After formulating a goal and a problem to solve, the agent calls a* ***search procedure*** *to solve it*

Contents

- Problem solving by searching
- Problem Formulation
- Search strategies
- Uninformed search strategies

Problem solving by searching

A problem can be defined formally by five components:

1. The **initial state** that the agent starts in.
2. A **description of the possible actions** available to the agent.
3. A description of what each action does; the formal name for this is the **transition model**
4. The **goal test**, which determines whether a given state is a goal state.
5. A **path cost function** that assigns a numeric cost to each path.

A simple problem solving agent

function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an ***action***

persistent:

  seq, an action sequence, initially empty

  state, some description of the current world state

  goal, a goal, initially null

  problem, a problem formulation

state ← ***UPDATE-STATE(state, percept)***

if seq is empty then

  goal ← ***FORMULATE-GOAL(state)***

  problem ← ***FORMULATE-PROBLEM(state, goal)***

  seq ← ***SEARCH(problem)***

  if seq = failure then

    return a null action

action ← ***FIRST(seq)***

seq ← ***REST(seq)***

return ***action***

End goal?

- A solution to a problem is an action sequence that leads from the initial state to a goal state.
- Solution quality is measured by the path cost function, and an optimal solution has the lowest path cost among all solutions.

Contents

- ~~Problem solving by searching~~
- Problem Formulation
- Search strategies
- Uninformed search strategies

Example Problems

- Toy problems (but sometimes useful)
  - Illustrate or exercise various problem-solving methods
  - Concise, exact description
  - Can be used to compare performance
  - *Examples*: 8-puzzle, 8-queens problem, Cryptarithmetic, Vacuum world, Missionaries and cannibals, simple route finding
- Real-world problem
  - More difficult
  - No single, agreed-upon description
  - *Examples*: Route finding, Touring and traveling salesperson problems, VLSI layout, Robot navigation, Assembly sequencing

Problem Formulation

- ***A toy problem*** is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is usable by different researchers to compare the performance of algorithms.
- ***A real-world problem*** is one whose solutions people actually care about. Such problems tend not to have a single agreed-upon description, but we can give the general flavor of their formulations

Toy Problem

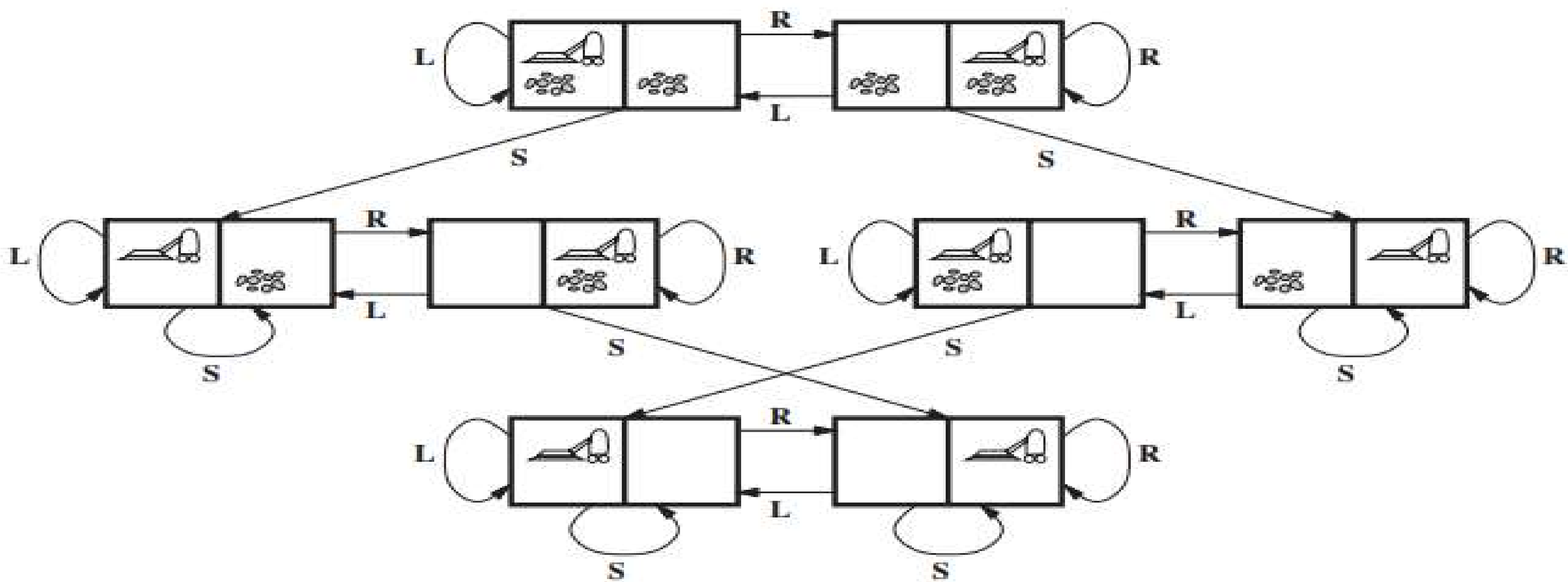


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

Vacuum World example

- This can be formulated as a problem as follows:
- ***States:*** 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 × 2x2 = 8 possible world states. A larger environment with n locations has n · 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 Sucking in a clean square have no effect.
- ***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.

Real-world problems

- Route finding
  - Specified locations and transition along links between them
  - *Applications*: routing in computer networks, automated travel advisory systems, airline travel planning systems
- Touring and traveling salesperson problems
  - "Visit every city on the map at least once and end in Bucharest"
  - Needs information about the visited cities
  - *Goal*: Find the shortest tour that visits all cities
  - *NP-hard*, but a lot of effort has been spent on improving the capabilities of TSP algorithms
  - *Applications*: planning movements of automatic circuit board drills

Contents

- ~~Problem solving by searching~~
- ~~Problem Formulation~~
- Search strategies
- Uninformed search strategies

Example: Romania

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest
- Formulate goal:
  - be in Bucharest
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

What is a Solution?

- A sequence of actions that when performed will transform the initial state into a goal state (e.g., the sequence of actions that gets the passenger safely to destination)
- Or sometimes just the goal state (e.g., infer molecular structure from mass spectrographic data)

Example: Romania

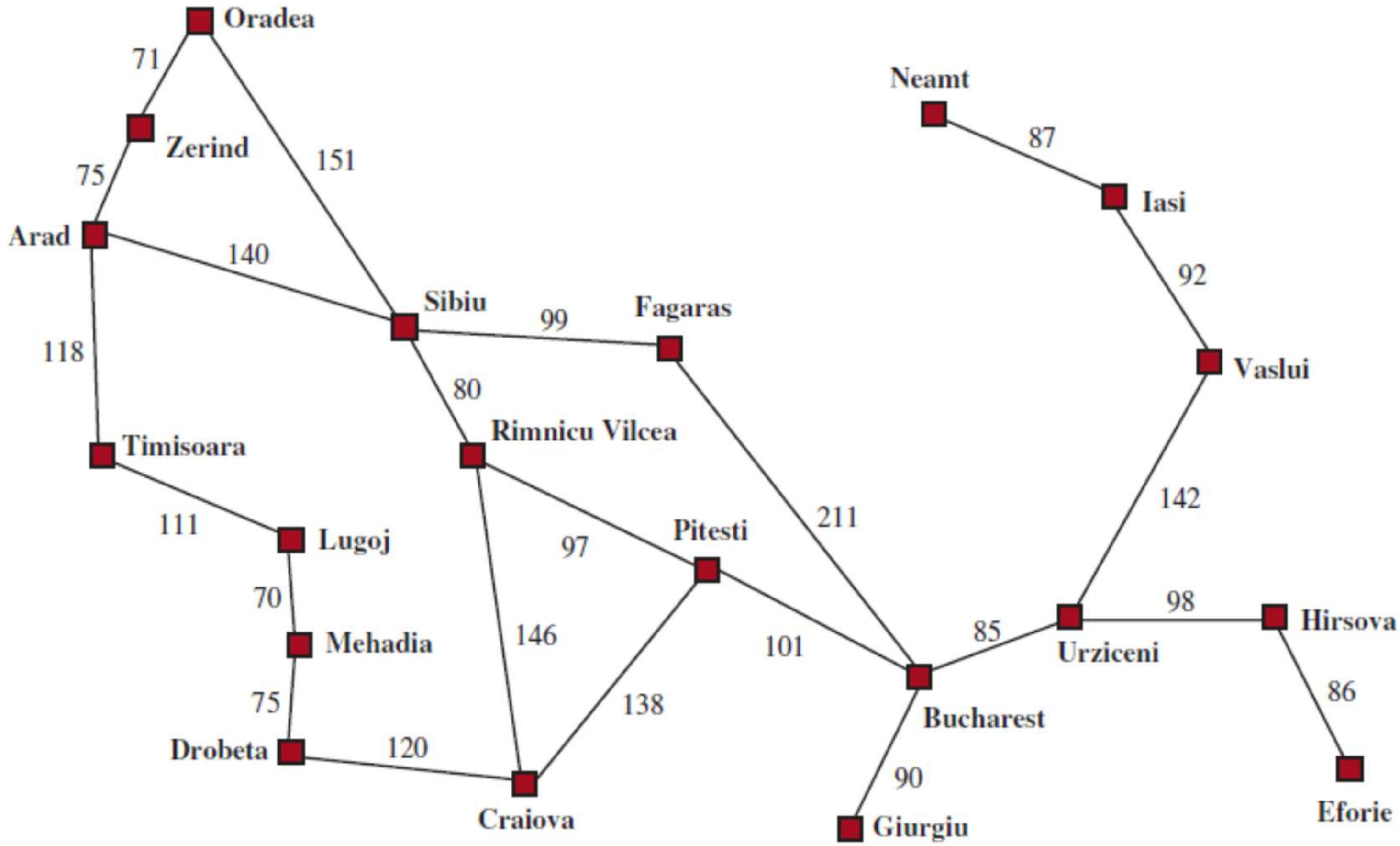


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

Selecting a state space

- Real world is absurdly complex
  → state space must be abstracted for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad“ must get to some real state "in Zerind"
- (Abstract) solution =
  - set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem

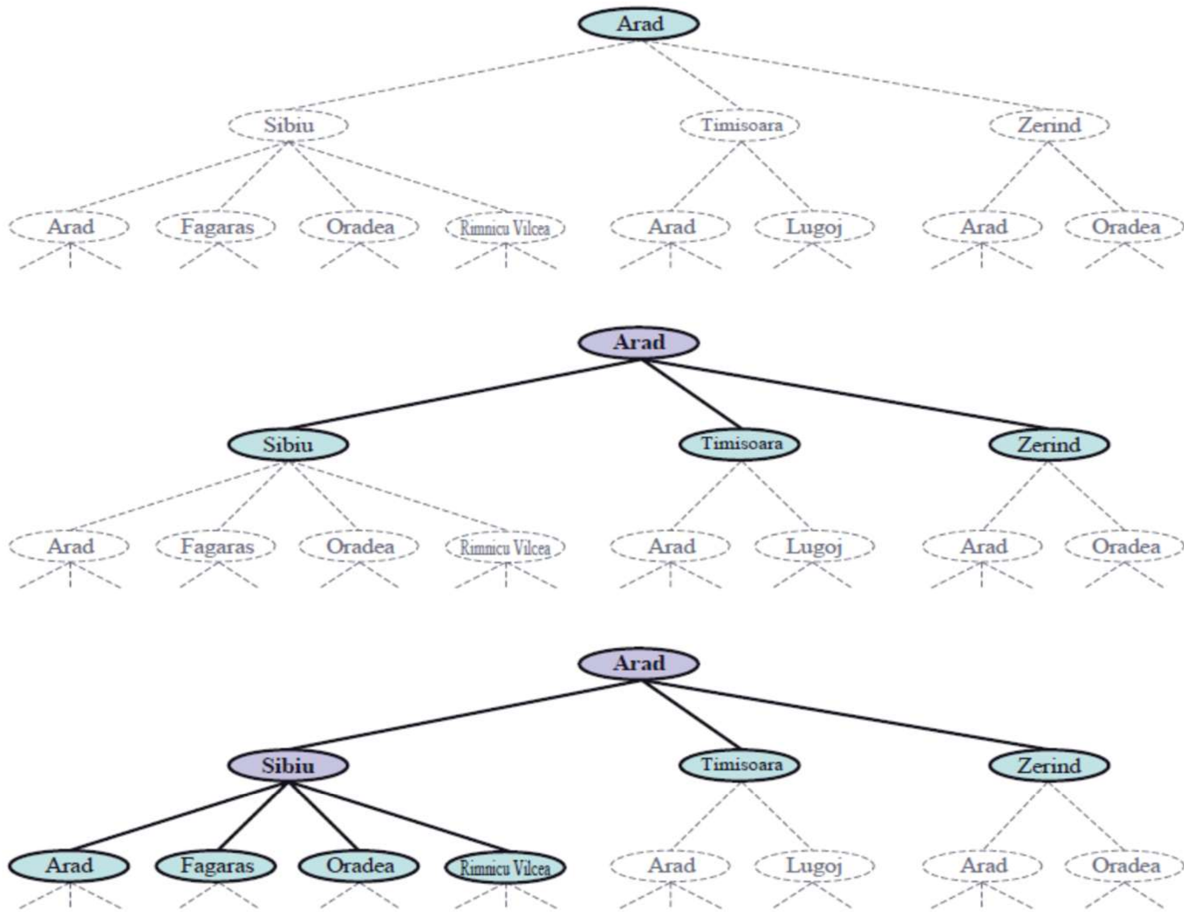
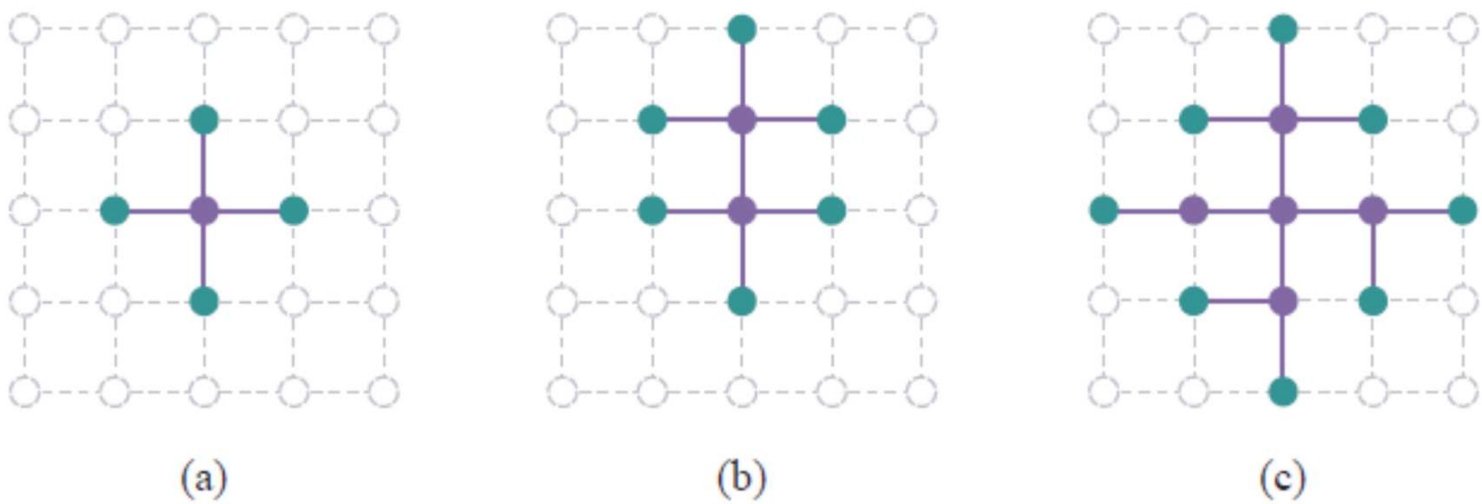


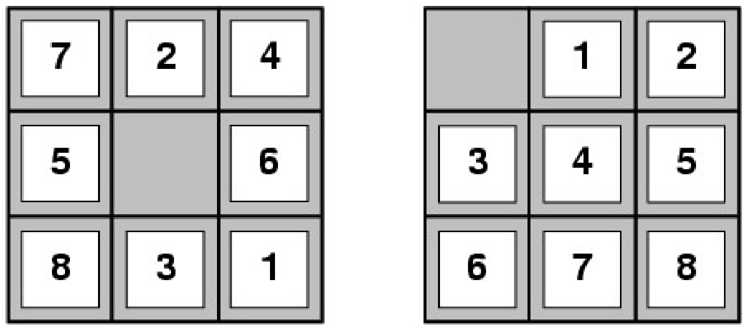
Figure 3.4 Three partial search trees for finding a route from Arad to Bucharest. Nodes that have been *expanded* are lavender with bold letters; nodes on the frontier that have been *generated* but not yet expanded are in green; the set of states corresponding to these two types of nodes are said to have been *reached*. Nodes that could be generated next are shown in faint dashed lines. Notice in the bottom tree there is a cycle from Arad to Sibiu to Arad; that can't be an optimal path, so search should not continue from there.

Expanded Nodes vs Frontier

The **frontier** is the set of nodes (and corresponding states) that have been reached but not yet expanded; the **interior** is the set of nodes (and corresponding states) that have been expanded; and the **exterior** is the set of states that have not been reached.



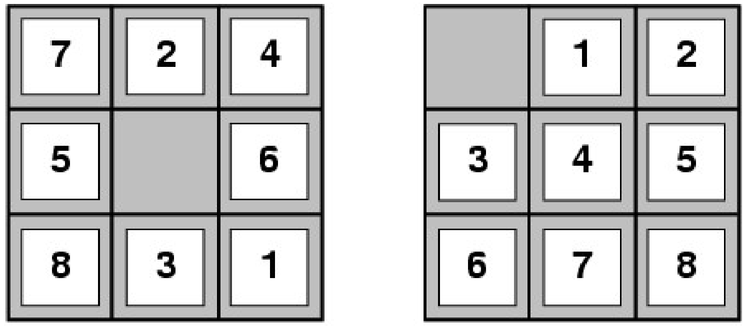
Example: The 8-puzzle



Start State Goal State

- states?
- actions?
- goal test?
- path cost?

Example: The 8-puzzle

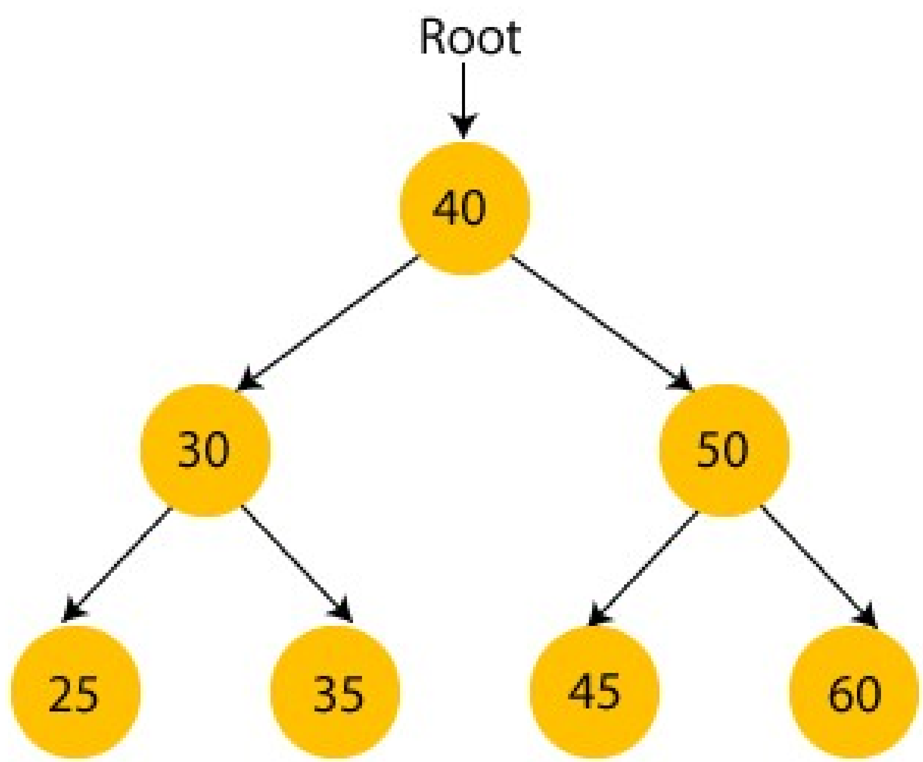


Start State Goal State

- states? locations of tiles
- actions? move left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move

[Note: optimal solution of *n*-Puzzle family is NP-hard]

Solution by searching



Elements/Structure

Elements of a search tree include:

- **Root Node**: The starting state of the problem.
- **Branches (Edges)**: Possible actions leading to new states.
- **Nodes**: Represent states in the search space.
- **Leaf Nodes**: Terminal states with no further expansion.
- **Goal Node**: A leaf node that satisfies the goal condition.
- **Path Cost**: The cumulative cost of reaching a node from the root.

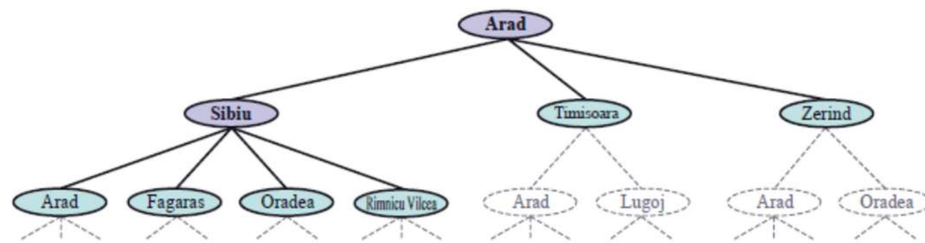
Searching

The possible action sequences starting at the initial state form a search tree with the initial state at the root; the branches are actions and the nodes correspond to states in the state space of the problem.

Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next—the **so called search strategy**.

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```



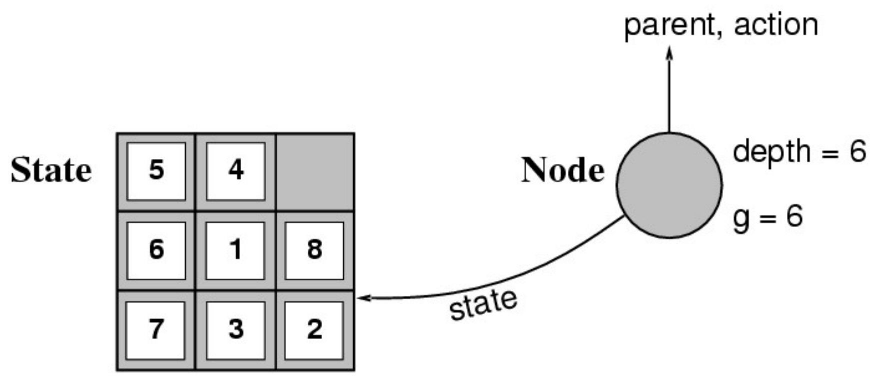
yield: a function that contains the keyword yield is a generator that generates a sequence of values, one each time the yield expression is encountered. After yielding, the function continues execution with the next statement.

Infrastructure for search algorithms

- Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we have a structure that contains four components:
- ***n.STATE:*** the state in the state space to which the node corresponds;
- ***n.PARENT:*** the node in the search tree that generated this node;
- ***n.ACTION:*** the action that was applied to the parent to generate the node;
- ***n.PATH-COST:*** the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

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

Where to store?

We need a data structure to store the frontier. The appropriate choice is a queue of some kind, because the operations on a **frontier** are:

- IS-EMPTY(*frontier*) returns true only if there are no nodes in the frontier.
- POP(*frontier*) removes the top node from the frontier and returns it.
- TOP(*frontier*) returns (but does not remove) the top node of the frontier.
- ADD(*node*, *frontier*) inserts node into its proper place in the queue.

Three kinds of queues are used in search algorithms:

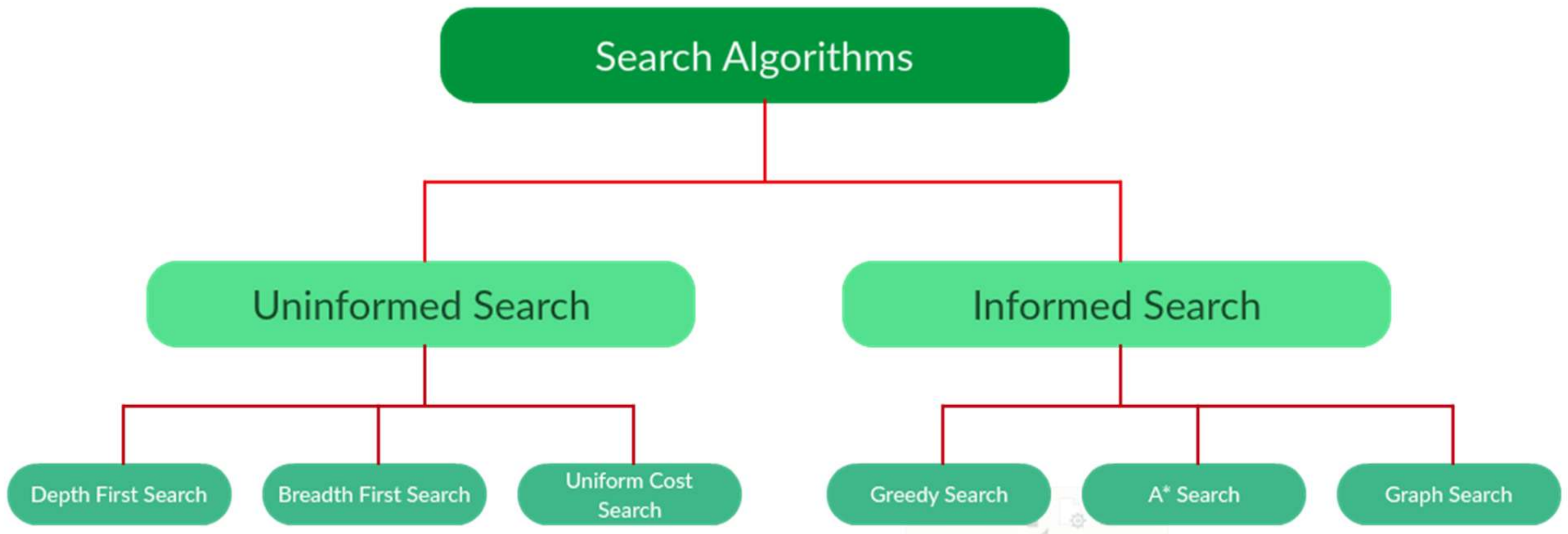
- Priority queue A priority queue first pops the node with the minimum cost according to some evaluation function, f . It is used in best-first search.
- FIFO queue A FIFO queue or first-in-first-out queue first pops the node that was added to the queue first; we shall see it is used in breadth-first search.
- LIFO queue A LIFO queue or last-in-first-out queue (also known as a stack) pops first the most recently added node; we shall see it is used in depth-first search.

Search strategies

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)

Contents

- ~~Problem solving by searching~~
- ~~Problem Formulation~~
- ~~Search strategies~~
- Uninformed search strategies


Search Algorithms
Uninformed Search
Informed Search
Depth First Search
Breadth First Search
Uniform Cost Search
Greedy Search
A* Search
Graph Search

Uninformed search strategies

Uninformed search strategies explore the search space without prior knowledge about the goal's location, only using the problem definition.

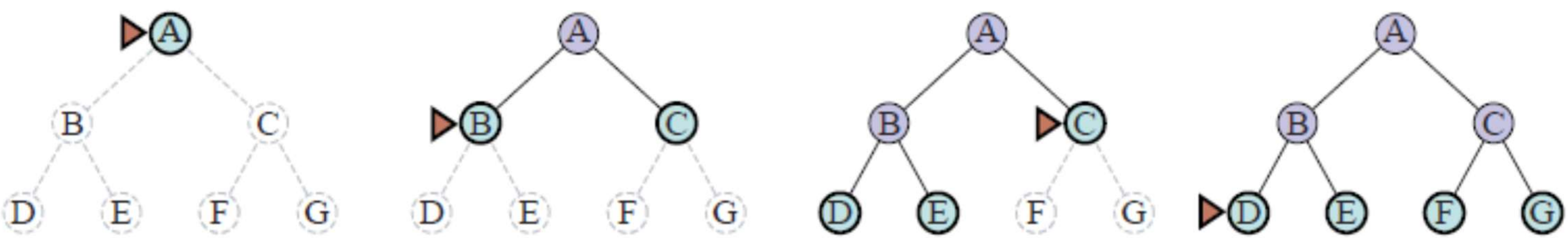
Types:

1. **Breadth first search**
2. **Uniform Cost Search**
3. **Depth first search**
4. **Depth-Limited Depth-First Search (DLS)**
5. **Iterative Deepening Depth-First Search (IDDFS)**

Breadth first search (BFS)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

- Explores all nodes at the current depth before moving deeper.
- Uses a **queue (FIFO)** for node expansion.
- Guaranteed to find the **shortest path** in an unweighted graph.
- High memory consumption as it stores all nodes at a given depth.



Properties of breadth-first search

- Branching factor **b**: Number of successors per node
- Each level multiplies nodes by branching factor
- Level progression: Root → b nodes → b^2 nodes → b^3 nodes
- Total nodes at depth d= $1 + b + b^2 + b^3 + \ldots + b^d$
- Time and space complexity: $O(b^d)$
- Complete? Yes (if b is finite)
- Optimal? Yes (if cost = 1 per step)
- Space is the bigger problem (more than time)

Real-World Example Breakdown

- Branching factor b = 10
- Processing speed: 1 million nodes/second
- Memory per node: 1 KB
- At depth d = 10
  - Time: < 3 hours
  - Memory required: 10 terabytes
- At depth d = 14 : Processing time ≈ 3.5 years
- Breadth-first search particularly affected by memory limitations
- Exponential complexity makes uninformed search impractical for:
  - Large datasets
  - Deep searches
  - High branching factors
- **Solution: Need for informed search strategies for real-world applications**

Uniform Cost Search

- Called **Dijkstra's algorithm** by the theoretical computer science community, and uniform-cost search by the AI community.
- Idea: While breadth-first search spreads out in waves of uniform depth — first depth 1, then depth 2, and so on —uniform-cost search spreads out in waves of path cost

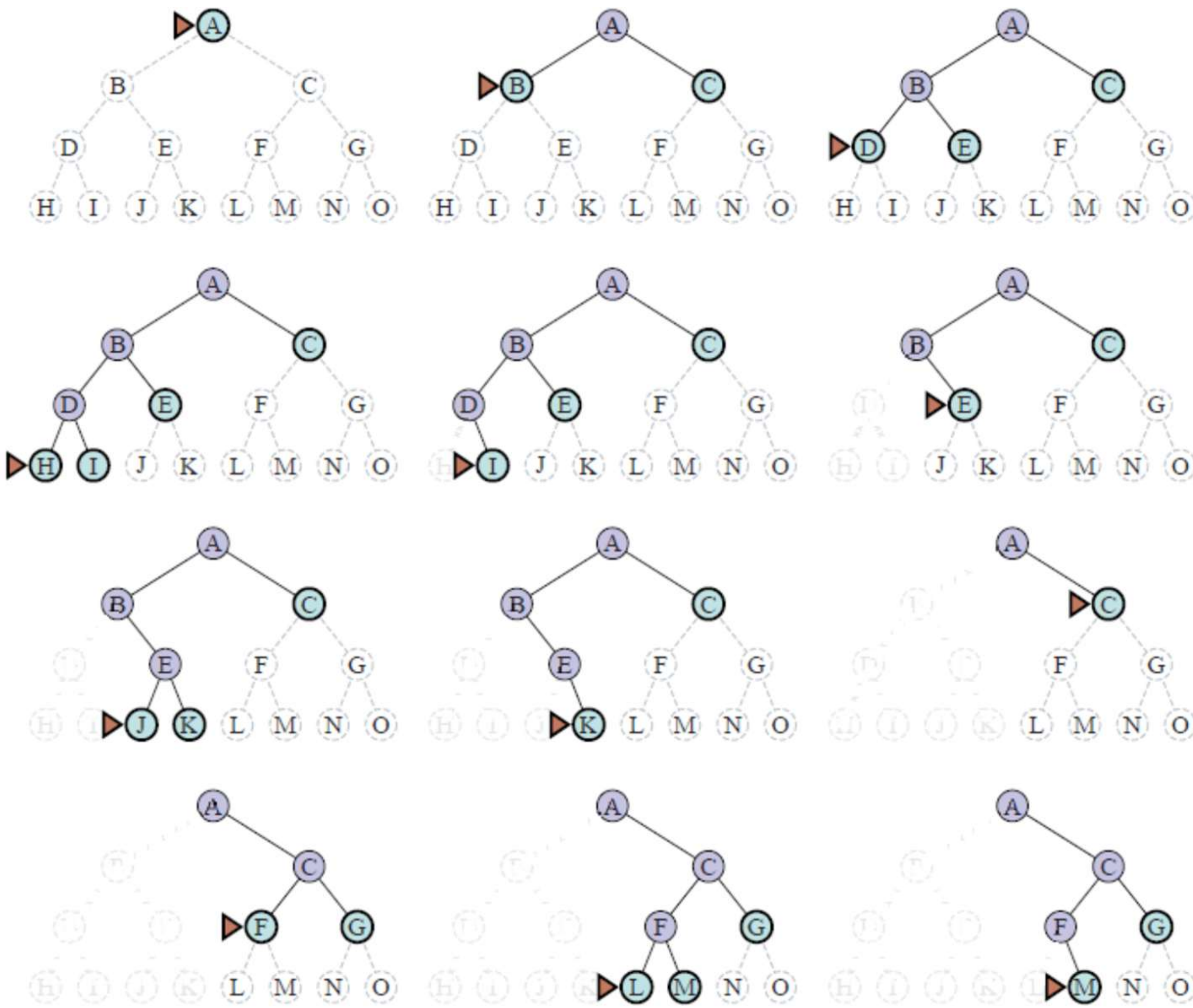
function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)

Depth first search (DFS)

- Explores as deep as possible along a branch before **backtracking**
- Uses a **stack (LIFO)** for node expansion.
- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
    → complete in finite spaces
- Optimal? No
  - Not guaranteed to find the shortest path.

Time Complexity: $O(b^m)$ Where b is the branching factor, and m is the max depth.

Space Complexity: O(bm) Where *m* is the maximum depth of the tree



In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next. In this way, only O(m) memory is needed rather than O(bm).

Depth-Limited Depth-First Search (DLS)

DFS with a predefined depth limit L.

Prevents infinite loops by stopping at the depth limit.

If the goal is beyond L, it **may fail**.

Time Complexity: $O(b^L)$
Space Complexity: $O(bL)$

Sometimes, depth limits can be based on knowledge of the problem. For example, on the map of Romania there are 20 cities. Therefore, we know that if there is a solution, it must be of length 19 at the longest, so L= 19 is a possible choice

Algorithm

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

Iterative Deepening Depth-First Search (IDDFS)

Repeatedly applies DLS with increasing depth limits.

Combines **DFS's low memory use** with **BFS's completeness**.

Guaranteed to find the **shortest path**.

Time Complexity: $O(b^d)$ (re-explores nodes but remains efficient).

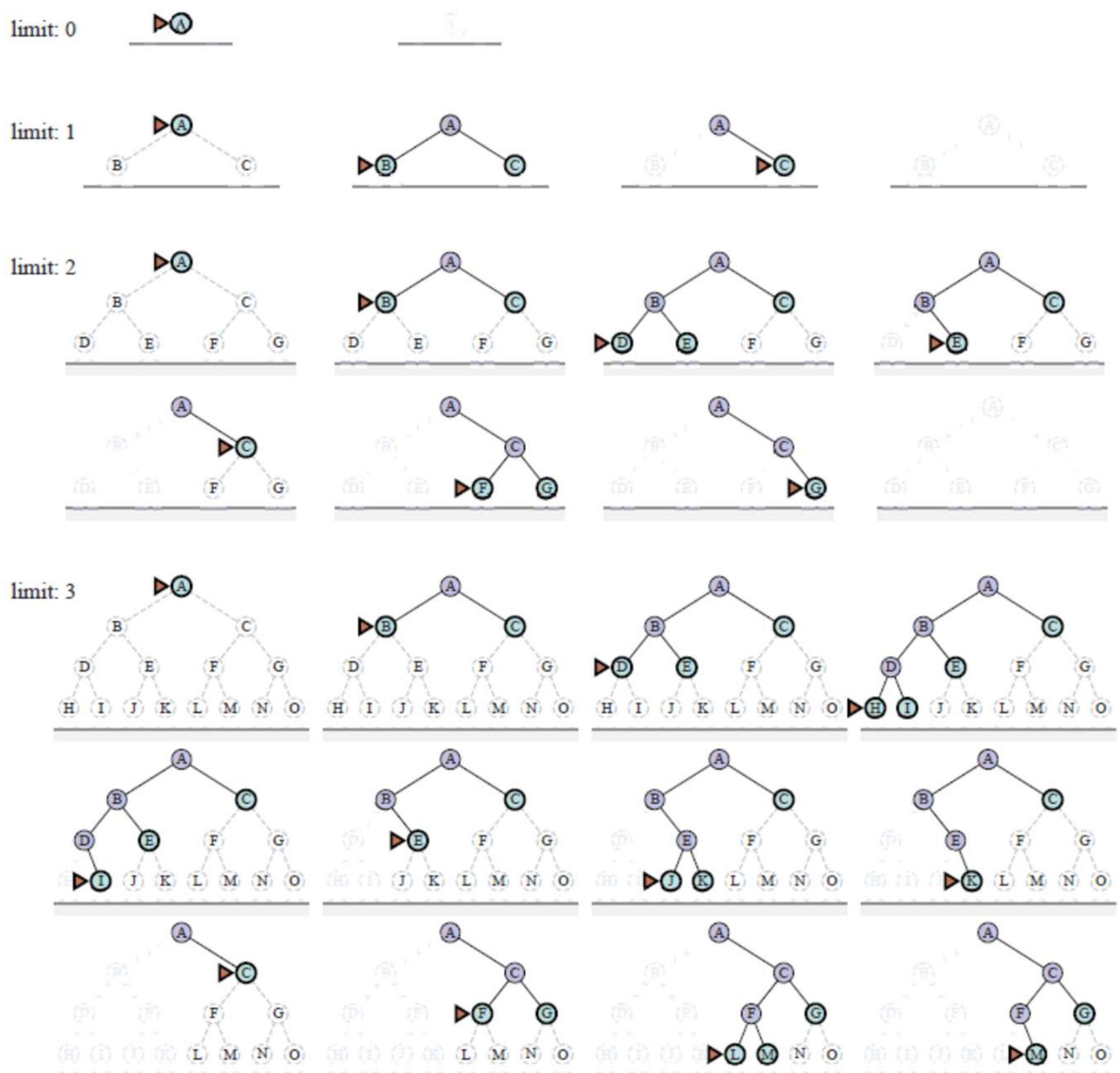
Space Complexity: $O(bd)$

IDDFS is commonly used in large search spaces where BFS's memory usage is impractical.

Algorithm

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

The iterative deepening search algorithm, which repeatedly applies depth limited search with increasing limits. It terminates when a solution is found or if the depth limited search returns failure, meaning that no solution exists.


limit: 0
limit: 1
limit: 2
limit: 3

Iterative deepening search

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

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

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

$$\mathrm{N}_{\mathrm{IDS}} = (\mathrm{d}+1)\mathrm{b}^0 + \mathrm{d}\ \mathrm{b}^1 + (\mathrm{d}-1)\mathrm{b}^2 + \ldots + 3\mathrm{b}^{\mathrm{d}-2} + 2\mathrm{b}^{\mathrm{d}-1} + 1\mathrm{b}^{\mathrm{d}}$$

- For $b = 10$, $d = 5$,
  - $\mathrm{N}_{\mathrm{DLS}} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $\mathrm{N}_{\mathrm{IDS}} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$

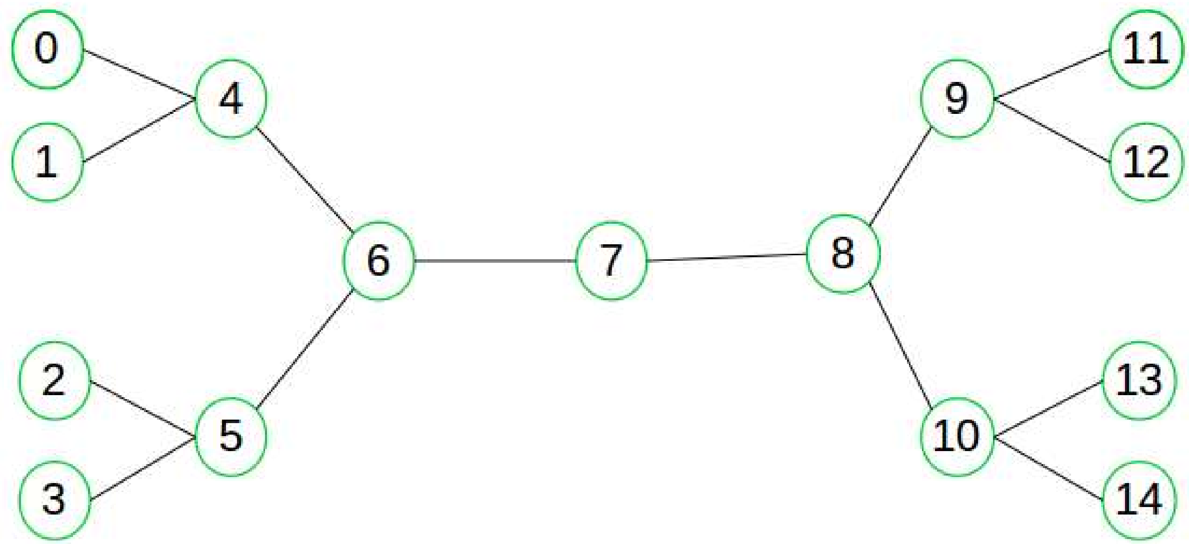
- Overhead = (123,456 - 111,111)/111,111 = 11%

Properties of iterative deepening search

- Complete? Yes
- Time? $(d+1)b^0 + d\, b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

Bi-directional Search

- An alternative approach called bidirectional search simultaneously searches forward from the initial state and backwards from the goal state(s), hoping that the two searches will meet. The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d (e.g., 50,000 times less when b=d=10).



Suppose we want to find if there exists a path from vertex 0 to vertex 14. Here we can execute two searches, one from vertex 0 and other from vertex 14. When both forward and backward search meet at vertex 7, we know that we have found a path from node 0 to 14 and search can be terminated now. We can clearly see that we have successfully avoided unnecessary exploration.

Bi-directional best first search

```
function BIBF-SEARCH(problem_F, f_F, problem_B, f_B) returns a solution node, or failure
  node_F ← NODE(problem_F.INITIAL)                    // Node for a start state
  node_B ← NODE(problem_B.INITIAL)                    // Node for a goal state
  frontier_F ← a priority queue ordered by f_F, with node_F as an element
  frontier_B ← a priority queue ordered by f_B, with node_B as an element
  reached_F ← a lookup table, with one key node_F.STATE and value node_F
  reached_B ← a lookup table, with one key node_B.STATE and value node_B
  solution ← failure
  while not TERMINATED(solution, frontier_F, frontier_B) do
    if f_F(TOP(frontier_F)) < f_B(TOP(frontier_B)) then
      solution ← PROCEED(F, problem_F frontier_F, reached_F, reached_B, solution)
    else solution ← PROCEED(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution

function PROCEED(dir, problem, frontier, reached, reached_2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached_2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← JOIN-NODES(dir, child, reached_2[s]))
        if PATH-COST(solution_2) < PATH-COST(solution) then
          solution ← solution_2
  return solution
```

Bidirectional best-first search keeps two frontiers and two tables of reached states. Although there are two separate frontiers, **the node to be expanded next is always one with a minimum value of the evaluation function, across either frontier.**

Bi-directional best first search

```
function BIBF-SEARCH(problem_F, f_F, problem_B, f_B) returns a solution node, or failure
  node_F ← NODE(problem_F.INITIAL)                         // Node for a start state
  node_B ← NODE(problem_B.INITIAL)                         // Node for a goal state
  frontier_F ← a priority queue ordered by f_F, with node_F as an element
  frontier_B ← a priority queue ordered by f_B, with node_B as an element
  reached_F ← a lookup table, with one key node_F.STATE and value node_F
  reached_B ← a lookup table, with one key node_B.STATE and value node_B
  solution ← failure
  while not TERMINATED(solution, frontier_F, frontier_B) do
    if f_F(TOP(frontier_F)) < f_B(TOP(frontier_B)) then
      solution ← PROCEED(F, problem_F frontier_F, reached_F, reached_B, solution)
    else solution ← PROCEED(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution

function PROCEED(dir, problem, frontier, reached, reached_2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached_2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← JOIN-NODES(dir, child, reached_2[s]))
        if PATH-COST(solution_2) < PATH-COST(solution) then
          solution ← solution_2
  return solution
```

When a path in one frontier reaches a state **that was also reached in the other half of the search**, the two paths are joined (by the function JOIN-NODES) to form a solution. The first solution we get is not guaranteed to be the best; the function TERMINATED determines when to stop looking for new solutions.

Comparison

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms