ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Problem Solving

Artificial Intelligence – *Inteligência Artificial*

Bachelor in Computer Systems Engineering , 2022-23

Licenciatura em Engenharia de Sistemas Informáticos

Contents

- Topics included
  - Problem formulation
  - Solution search
  - Uninformed Search
  - Informed Search

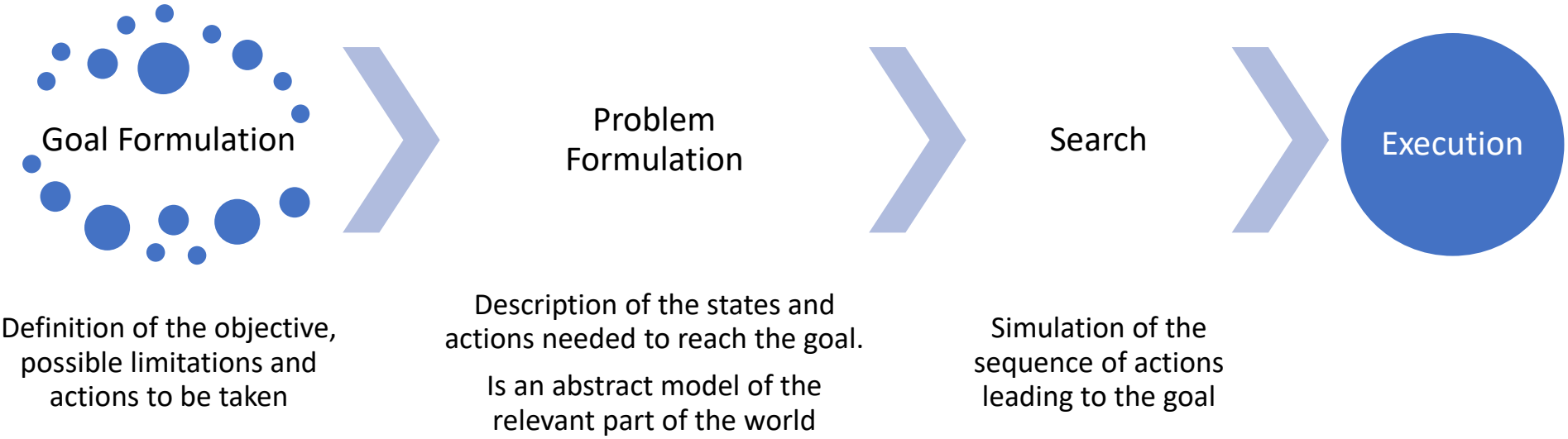
- These slides were based essentially on the following bibliography:
  - Norvig, P, Russell, S. (2021). Artificial Intelligence: A Modern Approach, 4th Edition. Pearson, ISBN-13: 978-1292401133
  - Most of the slides were adapted from materials prepared by Prof. Joaquim Gonçalves

Problem formulation

Problem Solving

We will assume our agents always have access to information about the world.

The agent looks forward to finding a sequence of actions that eventually can achieve the defined goal.



Goals organize formulation behavior by limiting the objectives and hence the actions to be considered

Problem Formulation

- What are the possible actions?
  - What is its effect on the state of the world?
- What are the possible states?
  - How to represent them?
- How to evaluate states
  - The formulation of the problem results in a mathematical abstraction that describes the phenomenon
- In an observable, deterministic and known environment, the solution is a fixed sequence of actions
  - If the model is correct, then once the agent has found a solution, it can ignore its percepts while it is executing the actions — that is an **open-loop** system
  - If there is a chance that the model is incorrect, or the environment is nondeterministic, then the agent would be safer using a **closed-loop** approach that monitors the percepts

Problems type

- Problem Solving Types
  - Search Algorithms
  - Adversarial Search
  - Constraint Satisfaction

- Standardized vs Non-Standardized Problems
  - Standardized – It allows to describe several methods being suitable as a benchmark for comparing the performance of algorithms
  - Non-Standardized – Many real-world problems are unique and therefore have idiosyncratic formulation

Search Problems

- Many problems in computer science can be defined as:
  - A set of states (Stace Space)
  - The initial state
  - A set of goal states
  - A set of actions available
  - A transition model that describe what each action does

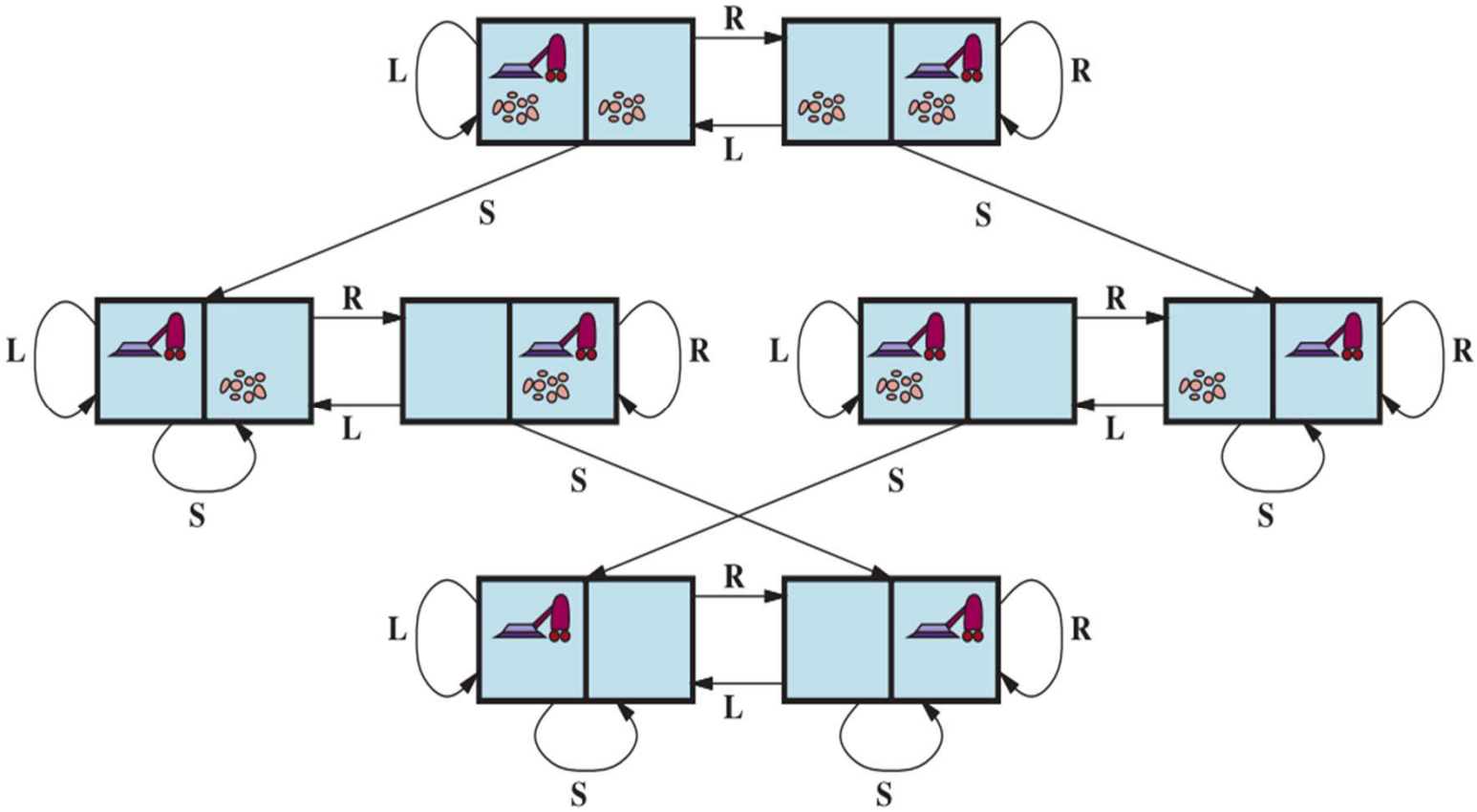
- Problem can be represented by a graph, where the nodes represent states and the arcs the pairs of the transition relation
  - A sequence of actions forms a path
  - The solution is a path between the initial state and an goal state

Example 01 – Vacuum agent

- States: A state of the world says which objects are in which cells.
  - the agent can be in either of the two cells, and each cell can either contain dirt or not, so, there are 2 * 2 * 2 = **8 states**
  - In general, a vacuum environment with n cells has n * 2n states.
- Initial state:
  - Any state can be designated as the initial state.
- Actions:
  - In the two-cell world we defined three actions: Suck, move Left, and move Right.
- Transition model
  - Suck removes any dirt from the agent's cell
  - TurnRight and TurnLeft change the direction it is facing by 90◦.
- • Goal states: The states in which every cell is clean.
- Action cost: Each action costs 1.

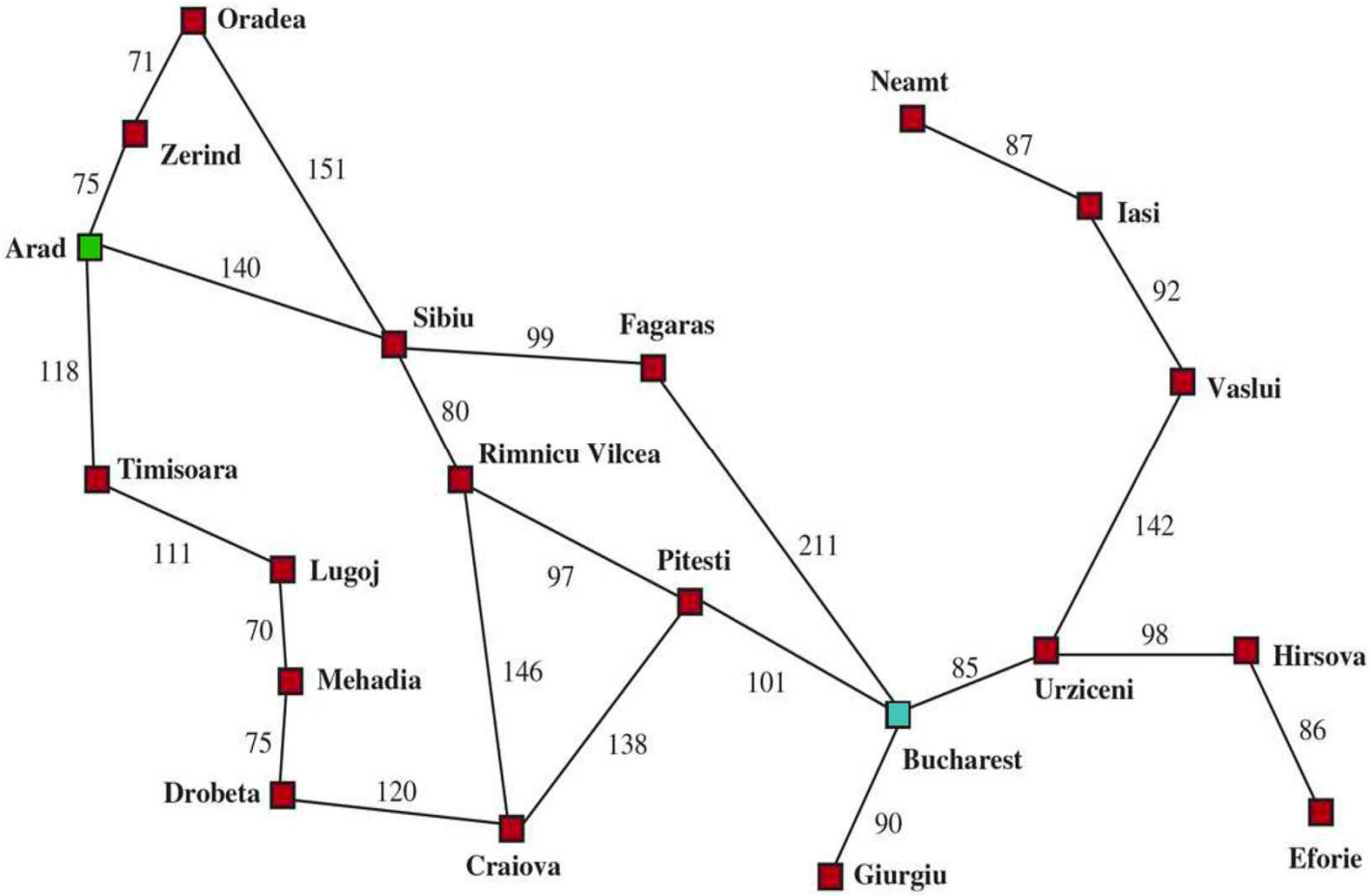
Example 01 – Two-cell vacuum world

- There are 8 states and three actions for each state:
  L = Left,
  R = Right,
  S = Suck



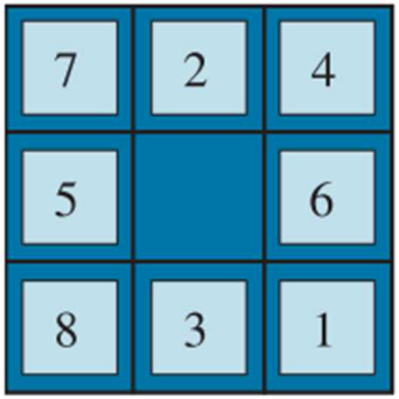
Example 02 – Traveling salesman problem

- Initial State:
  Arad
- Objective State:
  Bucharest
- Operators:
  Arches with respective costs
- Transition model:
  Maps a state and action to a resulting state
- Goal test:
  Verifies if the state corresponds to the goal
- Solution Cost
  The solution cost is the sum of cost of the steps/arches



Example 03 – Puzzle

- States
  - Specifies the position of the individual pieces and the empty space (various representations are possible)
- Initial state
  - Represented in the figure
- Operators - successors
  - Generate the valid states that result from the execution. There are four actions to move empty space: left, right, up or down
  - Some states may accept as valid only 2 or 3 of these actions
- Goal test
  - Verifies if the state corresponds to the goal configuration (represented in the figure)
- Solution Cost
  - Each step costs 1, and the solution cost is the number of steps to solve the problem

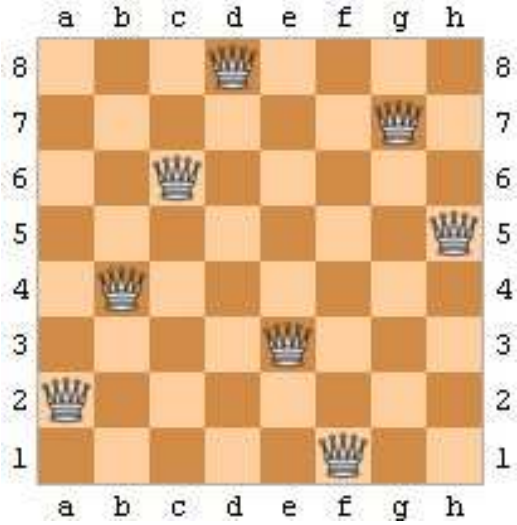


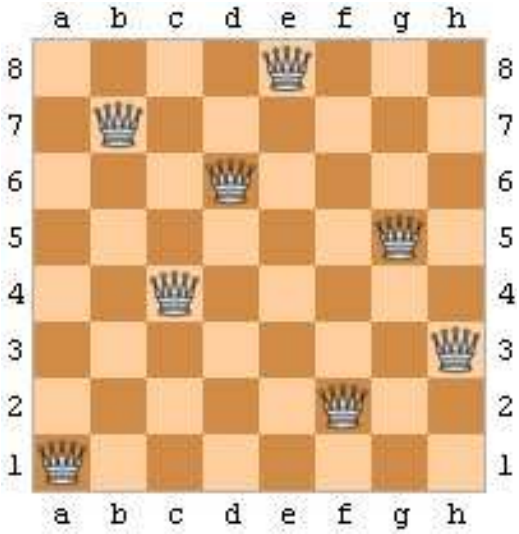
Start State

Goal State

Example 04 – Eight Queens Puzzle

- Test Objective: 8 queens on the board without any attacks
- Formulation 1:
  - State: Any arrangement of **0 to 8 Queens** on the board
  - Operator: Add a queen on **any square**, no attacks
- Formulation 2:
  - State: Arrangement of **0 to 8 Queens**, **one on each column, no attacks**!
  - Operator: Add a Queen on the **leftmost empty column**, no attacks
- Formulation 3:
  - State: Arrange **8 queens** on the board, **one on each column**!
  - Operator: Move attacked queen to a square in the same column
- The eight queens puzzle has 92 distinct solutions (include rotations), **12 unique**.



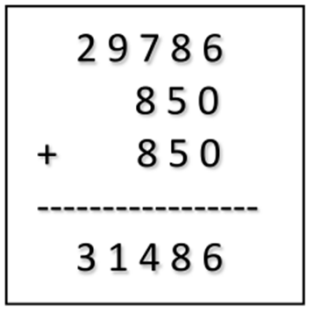


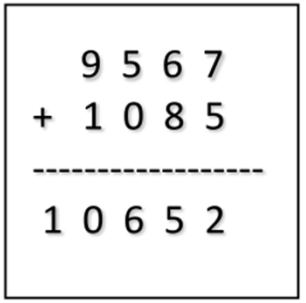
Retrieved in 2022.09.27 from: https://math.info/Misc/Eight_Queens_Puzzle/

Example 05 – Cryptogram

- Find digits (all different), one for each letter so that the sum is correct!

- States
  - Puzzle with some letters replaced by numbers
- Operators
  - Replace all occurrences of a letter with a digit
- Test Objective
  - Puzzle only contains digits and the sum is correct!
- Solution Cost
  - 0 (all solutions are equal)





Exercise 01

Two bucket puzzle

You have an endless source of water, and two buckets. One can hold 3 litres, and the other can hold 5 litres, there are no markings on the buckets. It is necessary to obtain exactly 4 litres.

- Formulate the problems as a search problem:
  - the state representation (space)
  - the initial state
  - the operators (preconditions and effects)
  - the objective test
  - the cost of the solution.

- Solve the problem using a tree search

Problem formulation

- State representation (space)
  - 4 (0L, 1L, 2L, 3L) * 6 (0L, 1L, 2L, 3L, 4L, 5L) = 24 states
- Initial state: (0L, 0L)
- Operators
  - Fill
  - Empty
  - Move until is full
- Objective test
  - (Any*, 4L) – several solutions!
- Cost of the solution:
  - Each step costs 1, and the solution cost is the number of steps to solve the problem

Retrieved on 2022.10.01 from
https://algorithmsinsight.wordpress.com/graph-theory-2/ida-star-algorithm-in-general/

Solution Search

Search Algorithms

- How can an agent look ahead to find a sequence of actions that will eventually achieve its goal?
- A search algorithm takes a search problem as input and returns a solution, or an indication of Search algorithm failure
- Tree search
  - A tree is model used to represent hierarchical data: nodes, leaves and branches
  - A tree search starts at the root, exploring the nodes from there, looking for a specific node that satisfies the conditions of the problem
  - There are several algorithms that use different strategies to select a node

- Search algorithms superimpose a **search tree** over the state space graph, forming various paths from the **initial state**, trying to find a path that reaches a **goal state**
  - Each node in the search tree corresponds to a state in the state space and the **edges nodes in the search tree correspond to actions or final/goal states**
  - The root of the tree corresponds to the initial state of the problem

Solution Search

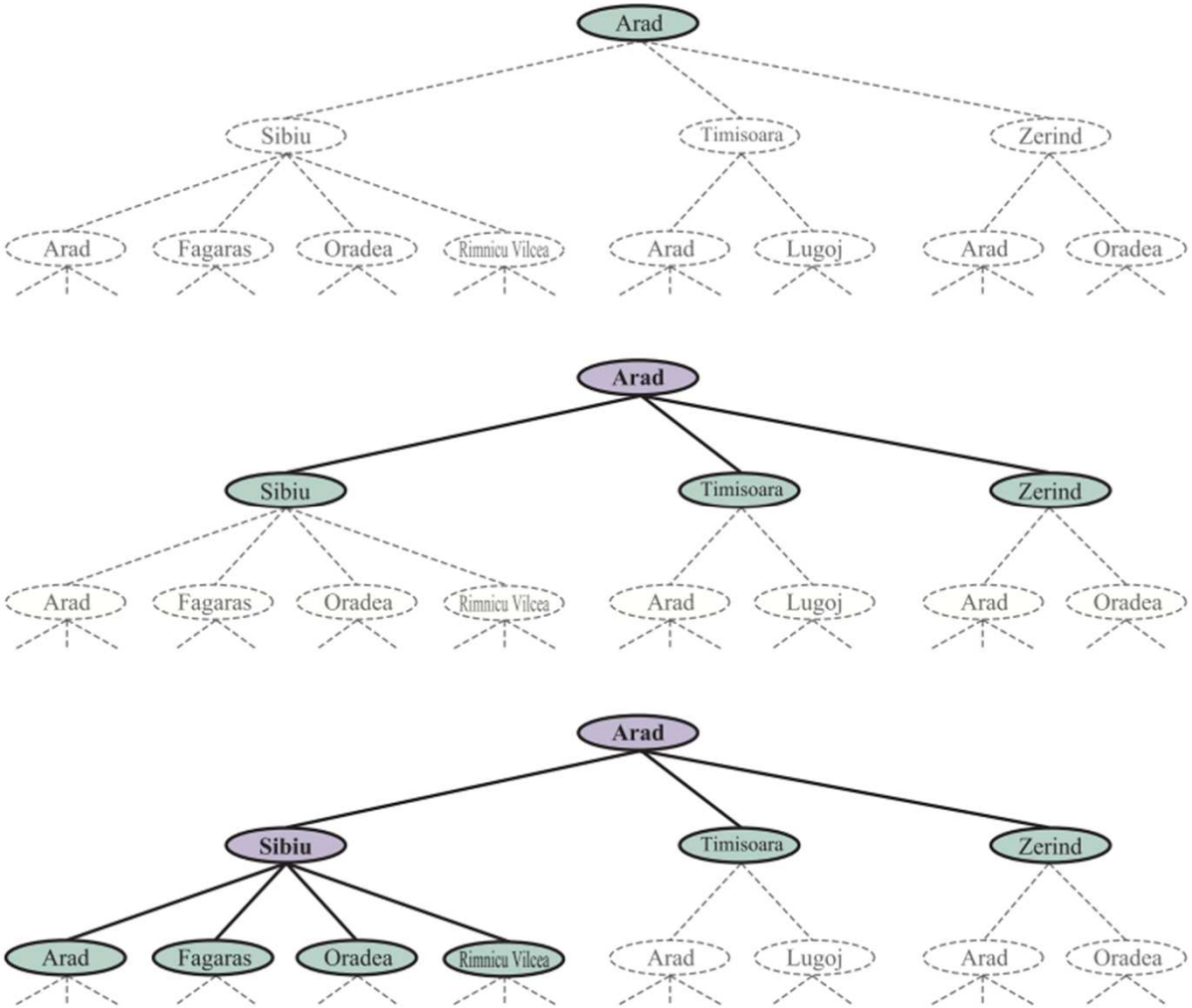
Method to perform the Solution Search

1. Start with the initial state
2. Perform the goal test
3. If the solution was not found, according to the search strategy, select a new state or use operators to expand the current state by generating new states (successor states) and back to 2
4. End

- Search tree composed of nodes.
  - **Leaf nodes** either have no successors or have not yet been expanded!
  - Important to distinguish between the search tree and the state space!
- Tree node (five components)
  - State it corresponds to
  - Node that originated it (parent)
  - Operator applied to generate it
  - Depth of the node
  - Path cost from the start node
- Boundary: Set of nodes waiting to be expanded

State space and search tree

- The **state space** describes the set of
  - states in the world (possibly infinite), and the
  - actions that allow transitions from one state to another.
- The **search tree** describes paths between these states, reaching towards the goal.
- The search tree may have multiple paths to any given state, but each node in the tree has a unique path back to the root
- Figure at left shows the first few steps in finding a path from Arad to Bucharest



Search Algorithms Properties

- Completeness
  - If there is a solution, whatever the initial state, it will be found
- Optimality
  - The algorithm will always find the least cost solution
- Time Complexity
  - The time is taken by the algorithm to complete a task
- Space Complexity
  - The required storage space at any point during the search

Complexity

- Time and space complexity can be measured in terms of the depth or number of actions in an optimal solution

- Henceforth we will consider the following notation:
  - m – represents the **maximum number of actions/steps** in any path (the path max length)
  - b – represents **branching factor** or number of successors of a node that need to be considered
  - d – represents the **depth or number of actions** in a optimal solution
  - l – represents is the established **limit for the depth**

Problem representation

- The model created to represent the problem is a determining factor in the effort required to solve it. The model allows you to
  - Visualize the problem
  - Specify the structures
  - Control the resolution process

- A good problem model should possess the following characteristics:
  - **Clarity** – the model and the problem should be clearly related
  - **Accuracy** – the model should be true to reality in the relevant characteristics of the problem
  - **Completeness** – the model should represent all aspects relevant to solving the problem
  - **Efficiency** – the representation should be able to be used efficiently
  - **Conciseness** – only relevant aspects should be represented
  - **Utility** – the model's ability to solve the problem should be evaluated

Search Algorithms Classes

- Search algorithms are classified between informed algorithms, in which the agent can estimate how far it is from the goal, and uninformed algorithms, where no such estimate is available.

- **Uninformed**
  - They do not have any knowledge related to the objective other than being able to recognize whether a node is the objective node or not.
  - Nodes are searched without any prior knowledge - blind search algorithms.

- **Informed**
  - Contain information about the objective state, namely the knowledge about the proximity of the goal state from the current state, the cost of the path, the way to reach the goal, among others.
  - A heuristic function is defined to measure the proximity between the current state and the goal state.

Uninformed Search

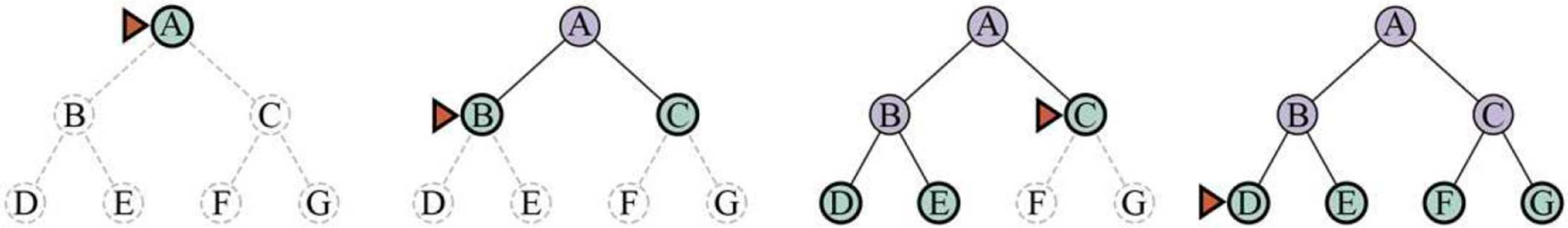
Uninformed Search Algorithms

Breadth First Search (BFS)

- The process starts from the root node of the tree and expands to all the successor nodes at the first level before moving to the next level nodes.
  - Systematic – If there is a solution it will be found the complete (if b is finite)) and optimal
  - Space complexity is same as time complexity because every node has to say in memory
  - The solution is found at the lowest depth node
  - Exponential complexity in space and time

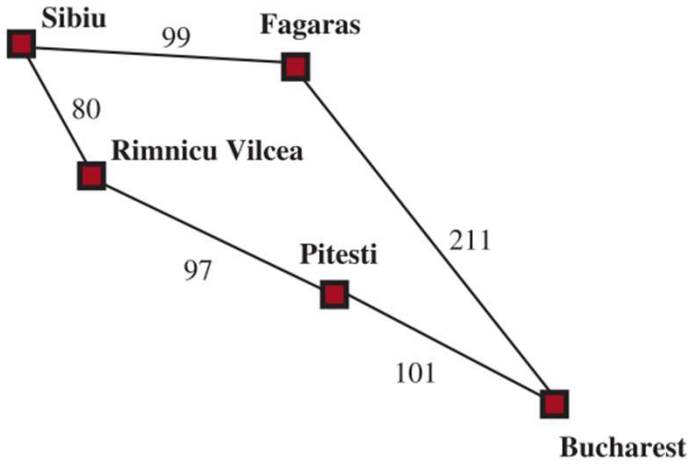
Time complexity: $\boldsymbol{O(b^d)}$
Space complexity: $\boldsymbol{O(b^d)}$

- In general, only **small problems** can be solved this way
  - Solution with depth *d* requires $1 + b + b^1 + \cdots + b^d$



Uniform Cost Search (Dijkstra's algorithm)

- When actions have different costs, nodes are expanded, starting from the root to the current node, according to the minimum cumulative cost
  - While BFS spreads out in waves of uniform depth (depth 1, depth 2, and so on) uniform-cost search spreads out in waves of uniform path-cost
  - The algorithm can be implemented as a call to BFS with PATH-COST as the evaluation function
- If we ensure that g(successor) ≥ g(n) then the first solution found is the cheapest
- Example: get from Sibiu to Bucharest (see example 2)
  - The successors of Sibiu are Rimnicu Vilcea (costs 80) and Fagaras (costs 99)
  - The least cost node, Rimnicu Vilcea, is expanded, adding Pitesti (cost 80+97=177)
  - Now , the least-cost node Fagarasis expanded, adding Bucharest (cost 99+211=310)
  - Bucharest is the goal, but the algorithm **tests for goals only when it expands a node**, not when it generates a node, so it has not yet detected that this is a path to the goal.
  - Next chooses Pitesti for expansion and adds a second path to Bucharest (cost 80+97+101=278)
  - It has a lower cost, so it replaces the previous path in reached and is added to the frontier. this node now has the lowest cost, so it is considered next, found to be a goal, and returned



Uniform Cost Search

- Uniform cost search considers all paths systematically in order of increasing cost
- **Avoids infinite paths**
  assuming that all action costs are > ϵ > 0
- BFS and Uniform Cost Search are equal if g(n) = Depth(n)
- In general, only **small problems** can be solved this way

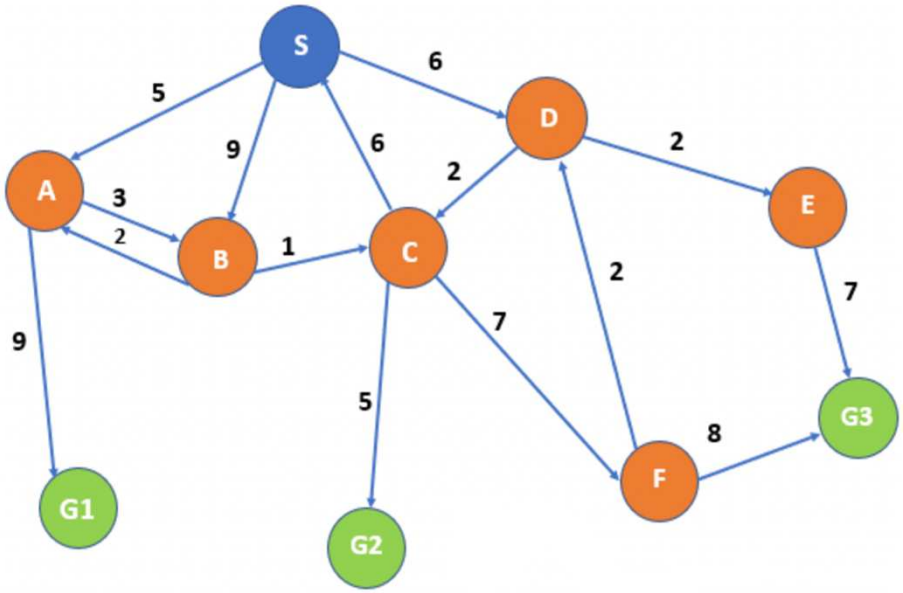
Time complexity: $O\left(b^{l+\left[\frac{C*}{\epsilon}\right]}\right)$

Space complexity: $O\left(b^{l+\left[\frac{C*}{\epsilon}\right]}\right)$

Legend

- $C*$ - the cost of the optimal solution;
- ϵ - a lower bound on the cost of each action, with ϵ > 0

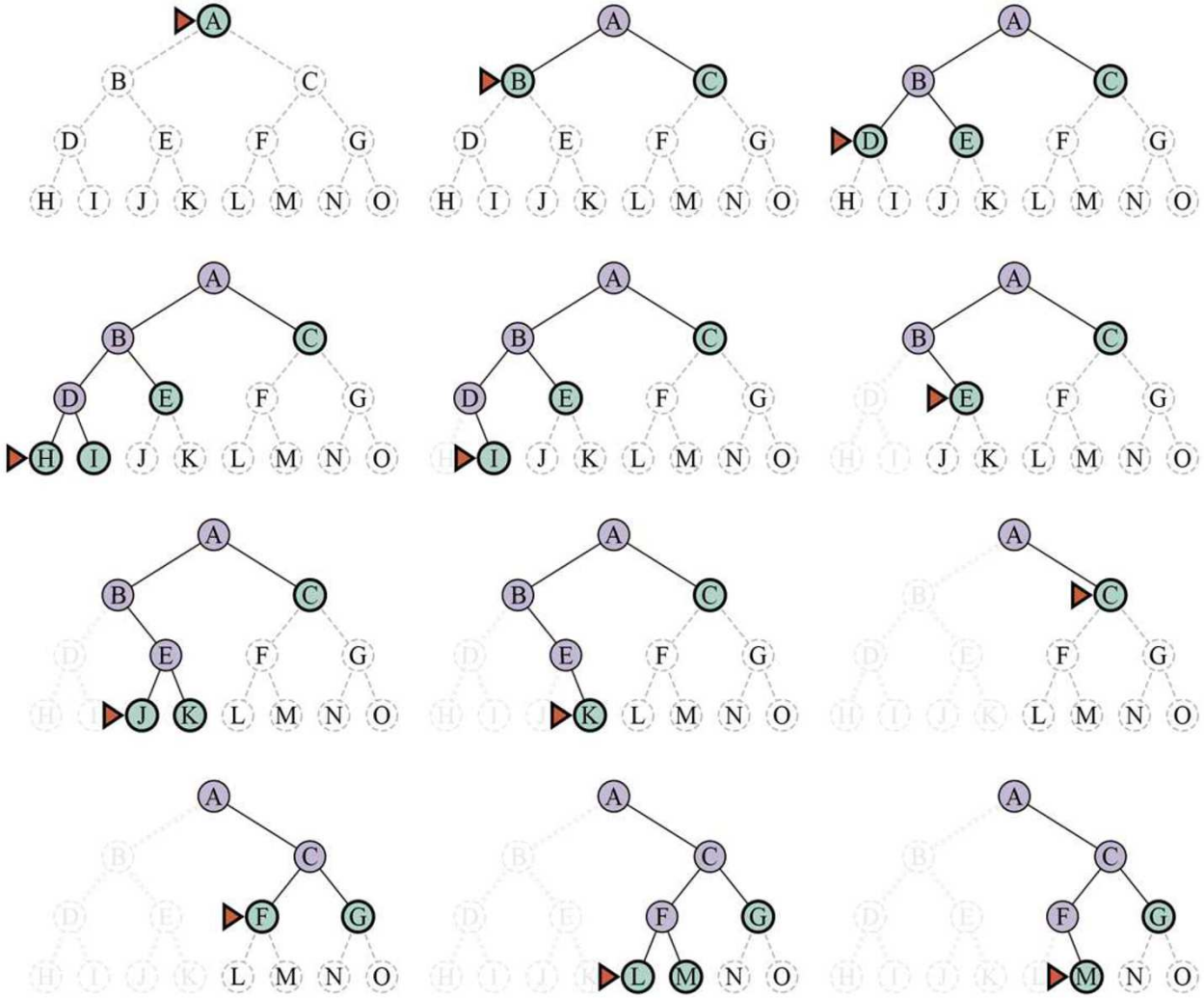
Pease, check this example of uniform cost search



Depth First Search (DFS)

- The searching procedure starts from the root node and follows each path to the greatest depth and then backtracks before entering the next path.
- Advantage: Low memory requirements, good for solution-intensive problems
- Disadvantage: Cannot be used for trees with infinite depth and can get stuck in an infinite loop

Time complexity: $O(b^d)$
Space complexity: $O(b \times m)$



Depth First Search (DFS)

- DFS is neither complete nor optimal
  - in case we have infinite depth, solution with depth d requires

    $1 + b + b^1 + \cdots + b^n$

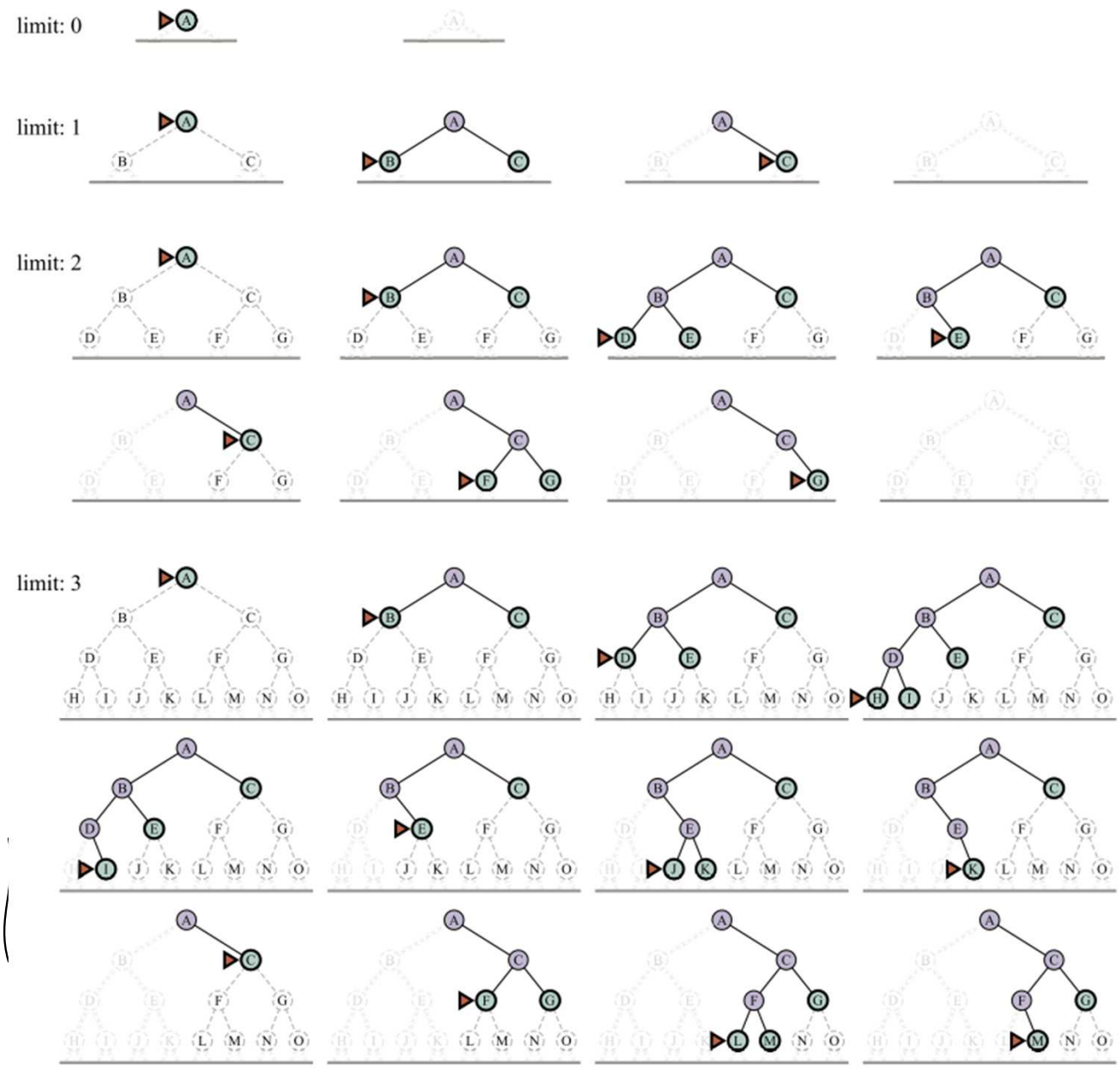
- For problems with multiple solutions can become much faster than BFS
- Inappropriate for problems that generate very deep or unlimited trees

- If you set a limit depth it becomes Depth-Limited Search

Depth-Limited Search (DLS)

- It is much similar to a DFS with a predetermined limit.
- The node at the limit is treated like it has no successor nodes further.
- DLS aims to remove the main disadvantage of DFS by avoiding the infinite path (**unlimited depth)**
- The idea is to admit that a solution should not be too far away from the start node, so you discard the parts of the graph that are not close enough to the start node.

- DLS is neither complete nor optimal.

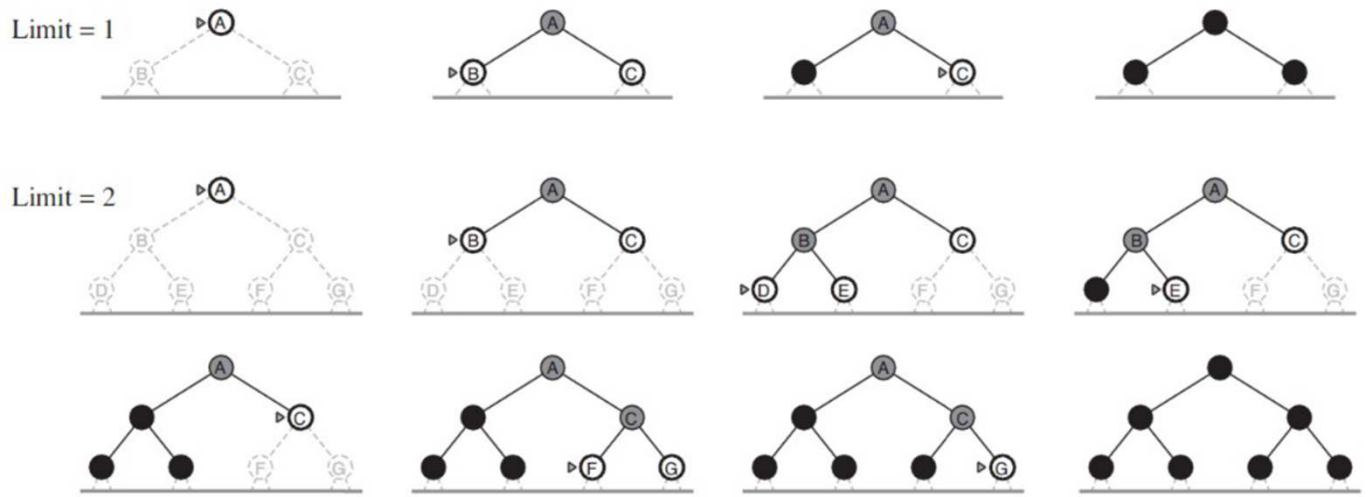


Iterative Deepening Depth First Search

- It is a **combination of both DFS and BFS**
- The algorithm performs a DFS to a certain depth limit and this depth limit is increased after each iteration until the goal is reached
- Iterative deepening combines many of the benefits of depth-first and breadth-first search.

Time complexity: $O(b^d)$
Space complexity: $O(b \times l)$

- Iterative Deepening DFS is complete and optimal
- In general, it is the best strategy for problems with a large search space and where the solution depth is not known



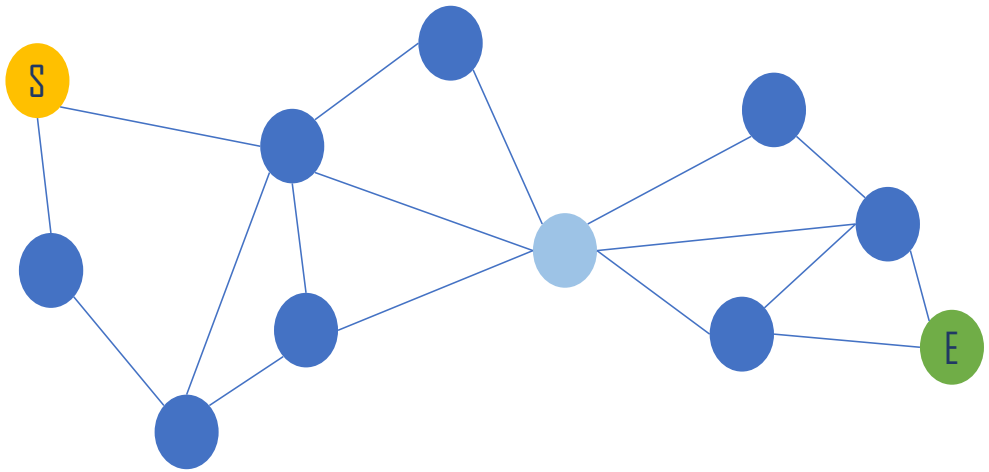
Bidirectional Search

- It performs two searches simultaneously:
  - One from the starting end is called **forward search**
  - other from the end is called **backward search**

- One single graph is replaced with two small subgraphs one from the initial vertex and the other from the final vertex.

- The searching procedure stops when these two graphs intersect each other

- It can use search techniques such as BFS, DFS, DLS, etc.



Avoid repeated States

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

1. Do not return to previous state
   - The node expansion function must be prevented from generating any node successor that is the same state as the node's parent.
2. Don't create paths with cycles in them
   - The node expansion function must be prevented from generating any node Trasuccessor that is the same state as any of the node's ancestors
3. There may be the same states in different sequences.
   - This requires that every state ever generated is remembered, potentially resulting in space complexity of $O(b^d)$.

- The **increased computational cost** to avoid repeated states must be analysed

Comparison of Search Algorithms / Method

Method	Breadth-First (BFS)	Uniform Cost	Depth-First (DFS)	Depth Limited (DLS)	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	$O(b^d)$	$O(b^{l+[\frac{C*}{\epsilon}]})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{l+[\frac{C*}{\epsilon}]})$	$O(b.m)$	$O(b.l)$	$O(b.d)$	$O(b^{d/2})$

Legend: b - branching factor; d - solution depth; m - maximum depth; l - depth limit
C∗ - the cost of the optimal solution; ϵ - a lower bound on the cost of each action, with ϵ > 0

Superscript caveats are as follows:
[1] complete if b is finite, and the state space either has a solution or is finite.
[2] complete if all action costs are ≥ ϵ > 0;
[3] cost-optimal if action costs are all identical;
[4] if both directions are breadth-first or uniform-cost.

Exercise 02

Hanoi towers

Move the stack of disks from the initial rod to another rod, where a disk cannot be placed on top of a smaller disk.

- Formulate the problems as a search problem:
  - the state representation (space)
  - the initial state
  - the operators (preconditions and effects)
  - the objective test
  - the cost of the solution.

- Solve the problem using a tree search

Problem formulation (3 rods, 3 disks)

- State representation (space)
  - Rod_A(disks), Rod_B (Ø), Rod_C(Ø)
- Initial state
  - A(3,2,1), B(), C()
- Operators (preconditions and effects)
  - Move disk into empty rod or over bigger disk
- Goal test
  - State = A(), B(), C(disks)
- Cost of the solution
  - Each step costs 1, and the solution cost is the number of steps to solve the problem

Exercise 03

Missionaries and cannibals' problem

Three missionaries and three cannibals must cross a river using a boat which can carry at most two people, under the constraint that, for both banks, if there are missionaries present on the bank, they cannot be outnumbered by cannibals

- Formulate the problems as a search problem:
  - the state representation (space)
  - the initial state
  - the operators (preconditions and effects)
  - the objective test
  - the cost of the solution.

- Solve the problem using a tree search

Problem formulation

- State representation (space)
- Initial state
- Operators (preconditions and effects)
- Objective test
- Cost of the solution

Informed Search

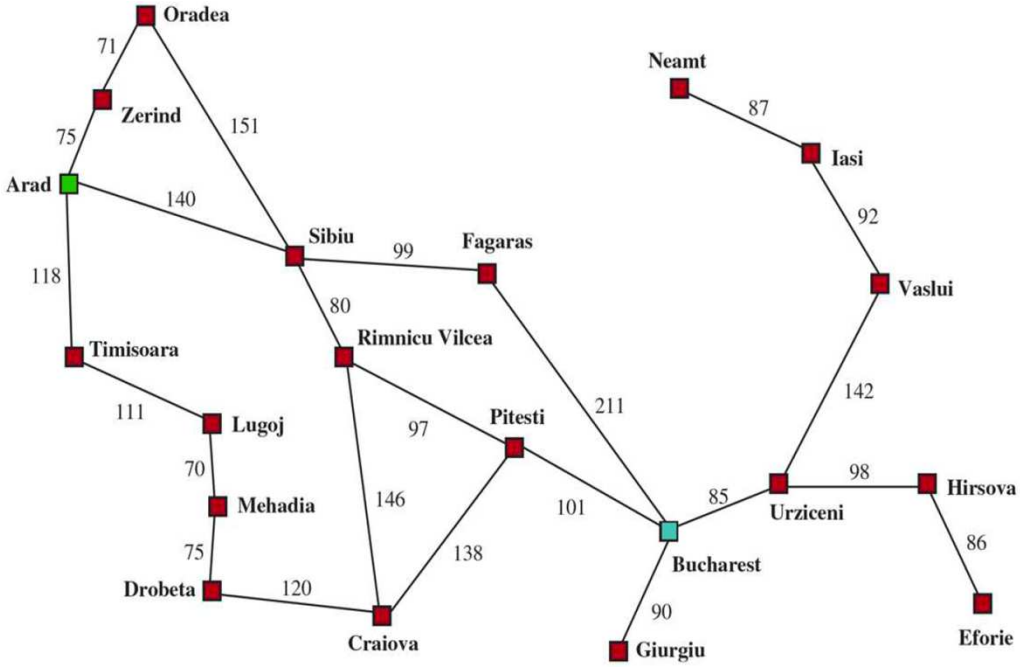
Informed Search

- Uses information to "guide" the search algorithm by choosing the order of expansion of nodes!
- informed search strategy uses domain-specific hints about the location of goals to find solutions more efficiently than an uninformed strategy.
- The hints come in the form of a heuristic function:

  h(n) = *estimated cost of the cheapest path from the state at node n to a goal state*

- For example, in route-finding problems, we can estimate the distance from the current state to a goal by computing the straight-line distance between the two points

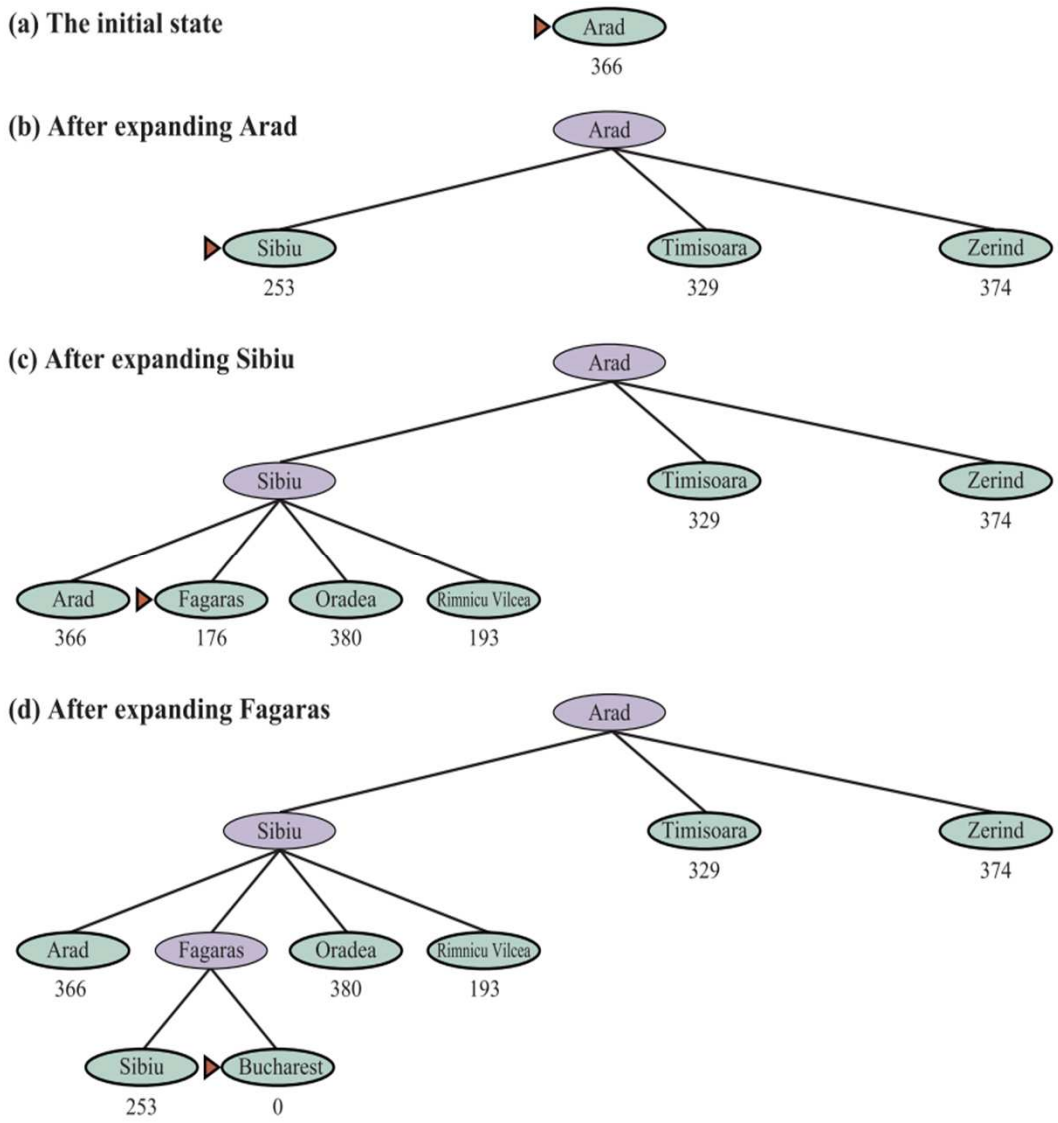
- Left figure shows the values of heuristic function
  - ***h(n)*** = straight-line distances from city *n* to Bucharest



Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Greedy-Search or Greedy best-first search

- Greedy best-first search is a form of best-first search that expands first the node with the **lowest h(n)** value.
  - Lowest h(n) corresponds to the node that appears to be closest to the goal, so it is likely to lead to a solution quickly
  - Uses an evaluation function that returns a number indicating the worth of expanding a node
  - **f(n) = h(n)** → estimates distance to goal

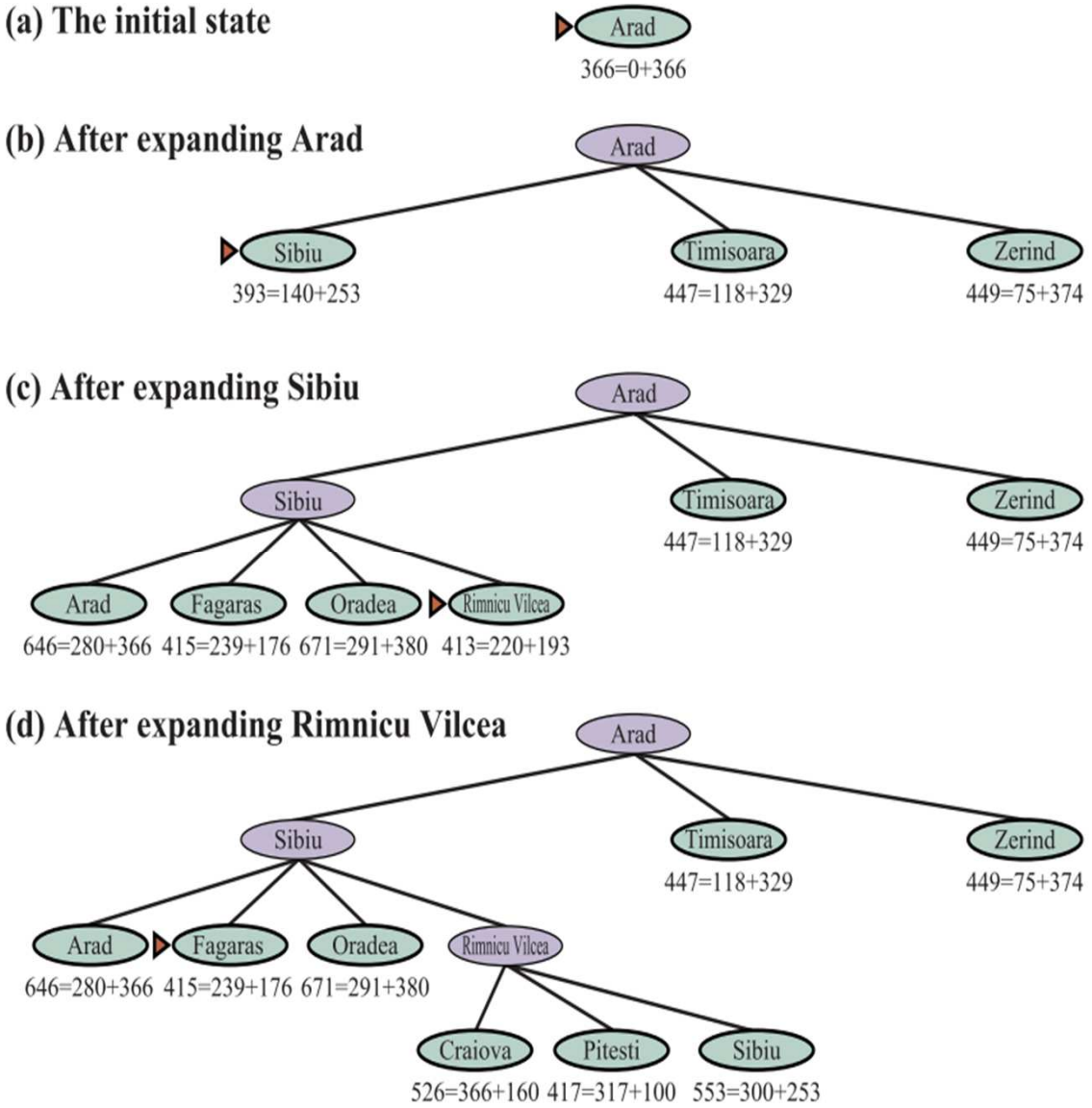


Greedy-Search or
Greedy best-first search

- It is **complete** in finite state spaces, but not in infinite ones.
  - It can go in cycles! (e.g., Iasi → Neamt → Iasi → Neamt → Iasi ...)
  - Susceptible to false starts
- The greedy-search does not always **not find optimal** cost
  - It does not always find the optimal solution – the path via Sibiu and Fagaras to Bucharest is 32 miles longer than the path through Rimnicu Vilcea and Pitesti.
  - On each iteration it tries to get as close to a goal as it can, but greediness can lead to worse results than being careful.
  - Does not (but need to) detect repeated states!
- The worst-case **time and space complexity** is O(|V|).
  - With a good heuristic function, the complexity can be reduced substantially to *O(b.m)*
  - With a perfect heuristic, it moves straight to the goal
  - Keeps all nodes in memory

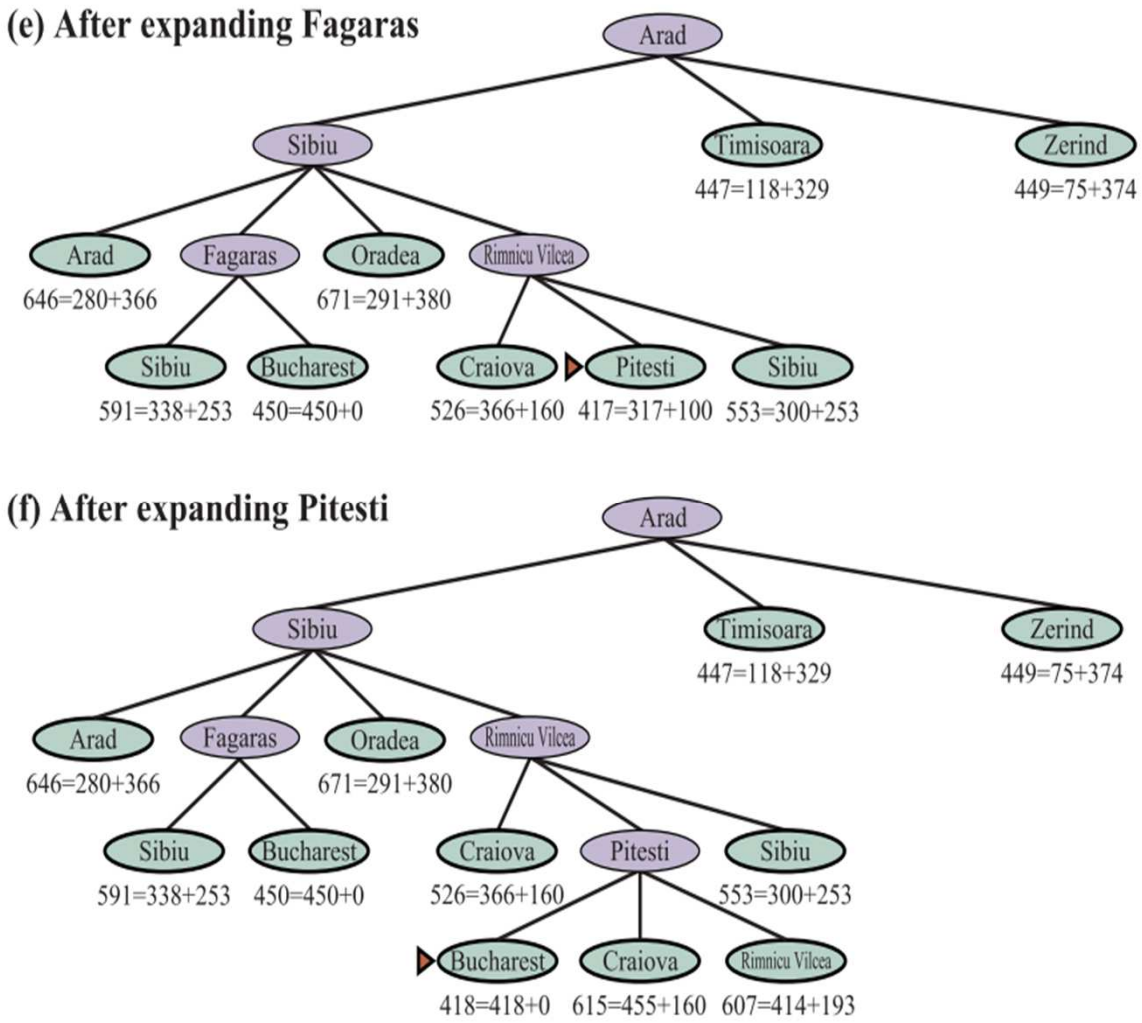
A* Search

- The algorithm A* combines the greedy search with the uniform cost
  - minimizes the sum of the path taken with the minimum expected missing to reach the solution
  - The idea is to avoid expanding nodes whose cost is too high
- heuristic function **f(n) = g(n) + h(n)**
  - Estimated cost of the best path that continues from n to a goal.
  - g(n) = total cost, so far, to reach state *n*
  - h(n) = estimated cost to reach the goal from *n*
  - f(n) = estimated cost of the cheapest solution that passes through node n
- Initial State: Arad; Goal: Bucharest
  - g(n) = cost to n;
  - h(n) = straight line distance to the goal



A* Search (2)

- Notice that Bucharest first appears on the frontier at step (e), but it is not selected for expansion
  - Thus, it is not detected as a solution
  - At f=450 it is not the lowest-cost node on the frontier — that would be Pitesti, at f=417.
- There is a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450.
- At step (f), a different path to Bucharest is now the lowest-cost node, at f=418, so it is selected and detected as the optimal solution.



A* Search – Admissibility of the Heuristic

- A∗ search is **complete**.
  - Assuming all action costs are > ϵ > 0, and the state space either has a solution or is finite

- A **heuristic h(n) is admissible** if and only if for each node $h(n) \leq h*(n)$ is verified, where
  - h(n) = estimated cost to reach the goal from *n*
  - $h*(n)$ = the actual path cost from n to the goal

- A* is **is optimal** if the heuristic *h(n)* **admissible** :
  - An admissible heuristic is one that never overestimates the cost to reach a goal
  - An admissible heuristic is therefore optimistic

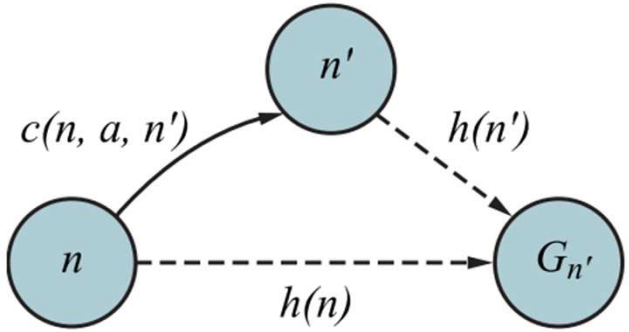
A* search is Optimal

- Suppose the optimal path has cost $C*$, but the algorithm returns a path with cost $C > C*$
  - Then there must be some node n which is on the optimal path and is **unexpanded**, otherwise we would have returned that optimal solution.
- using the notation

  $g*(n)$ = cost of the optimal path from the start to n, and
  $h*(n)$ = cost of the optimal path from n to the nearest goal,
  we have:
  - $f(n) > C*$, otherwise n would have been expanded
  - $f(n) = g(n) + h(n)$, by definition
  - $f(n) = g*(n) + h(n)$, because n is on an optimal path
  - $f(n) \leq g*(n) + h*(n)$, because of admissibility $h(n) \leq h*(n)$
  - $f(n) \leq C*$, by definition, $C* = g*(n)+h*(n)$
- The first and last lines form a contradiction, so the supposition that the algorithm could return a suboptimal path must be wrong — it must be that $A*$ returns only cost-optimal paths.

A* Search Consistency

- A **heuristic h(n) is consistent** if, for every node n and every successor n' of n generated by an action a → **h(n) ≤ c(n, a, n') + h(n)**
  - This is a form of the triangle inequality, which stipulates that a side of a triangle cannot be longer than the sum of the other two sides
  - An example of a consistent heuristic is the straight-line distance h that we used in getting to Bucharest.



- Every consistent heuristic is admissible (but not vice versa), so with a consistent heuristic, A∗ is cost-optimal
- With a consistent heuristic, the first time we reach a state it will be on an optimal path
  - No need to re-add a state to the frontier,
  - No need to change an entry after reached.

A* Search Consistency

- With an **inconsistent heuristic** we can have multiple paths reaching the same state
  - If each new path has a lower path cost than the previous one, then there are multiple nodes for that state in the frontier, costing both time and space
  - Some implementations of A∗ take care to only enter a state into the frontier once, and if a better path to the state is found, all the successors of the state are updated

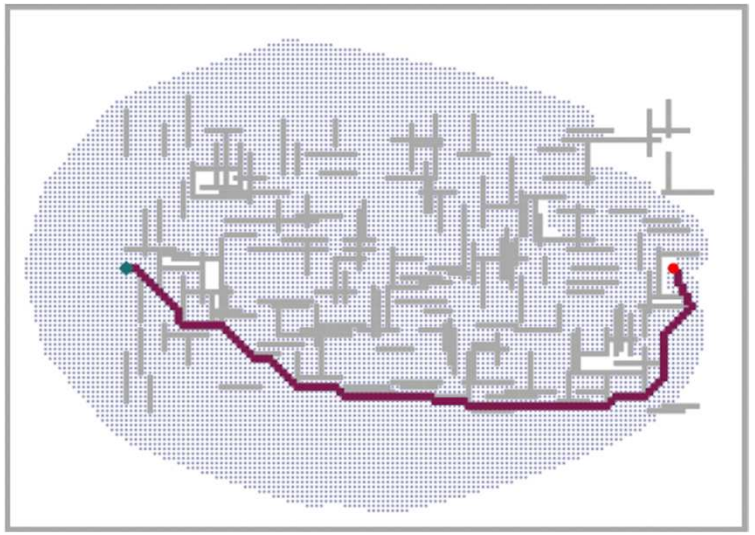
- With an **inadmissible heuristic**, A∗ may or may not be cost-optimal
  - If there is even one cost-optimal path on which h(n) is admissible for all nodes n on the path, then that path will be found, no matter what the heuristic says for states off the path.
  - If the optimal solution has cost C∗, and the second-best has cost C_2, and if h(n) overestimates some costs, but never by more than $C_2 - C*$, then A∗ is guaranteed to return cost-optimal solutions

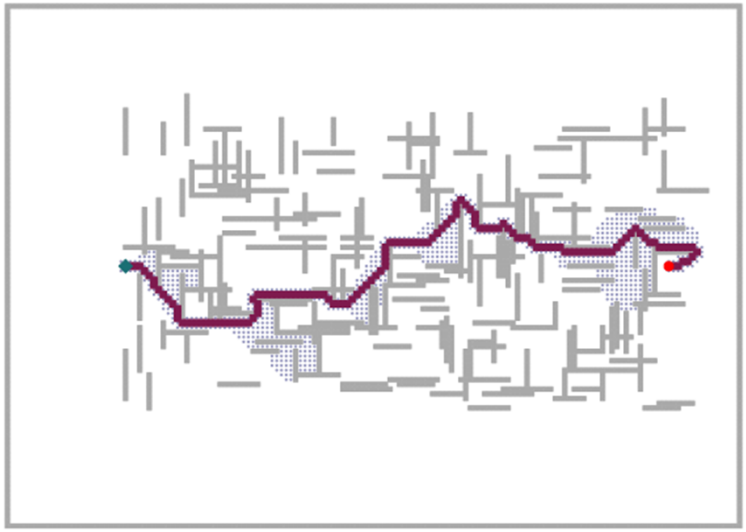
A* Search

- A∗ algorithm with a consistent heuristic is optimally efficient
  - **Expanded nodes** – If C∗ is the cost of the optimal solution path, then A∗ expands all nodes that can be reached from the initial state on a path where every node on the path has f(n) <C∗
  - Any algorithm that extends search paths from the initial state, and uses the same heuristic information, must expand all nodes that are surely expanded by A*
- A∗ with a consistent heuristic is optimally efficient
  - Any algorithm that extends search paths from the initial state, and uses the same heuristic information, must expand all nodes that are surely expanded by A∗
  - A∗ is efficient because it **prunes** away search tree nodes that are not necessary for finding an optimal solution
- That A∗ search is **complete**, **cost-optimal**, and o**ptimally efficient** among all such algorithms is rather satisfying. However, A∗ is the answer to all our searching needs
  - A∗ expands a lot of nodes and can explore fewer nodes if you accept solutions that are suboptimal
  - These solutions can be “good enough”— are called **satisficing solutions**

Weighted A∗ Search

- Weights the heuristic value more heavily
  - **f(n) = g(n) + W × h(n), for some W > 1.**
  - In top figure, A∗ search finds the optimal solution, but has to explore a large portion of the state space
  - In bottom figure, weighted A∗ search finds a solution that is slightly expensive, but the search time is much faster
- Explores fewer states
  - but if the optimal path ever strays outside of the weighted search's contour, then the optimal path will not be found.
- if the optimal solution costs C∗, Weighted A∗ search
  - Will find a solution that costs between C∗ and W×C∗;
  - Usually, it gets results much closer to C∗ than W×C∗.
- Weighted A∗ can be seen as a generalization of the others
  - A∗ search: g(n) + h(n) (W = 1)
  - Uniform-cost search: h(n) (W = 0)
  - Greedy best-first search: g(n) (W =∞)
  - Weighted A∗ search: g(n) + W×h(n) (1 <W <∞)



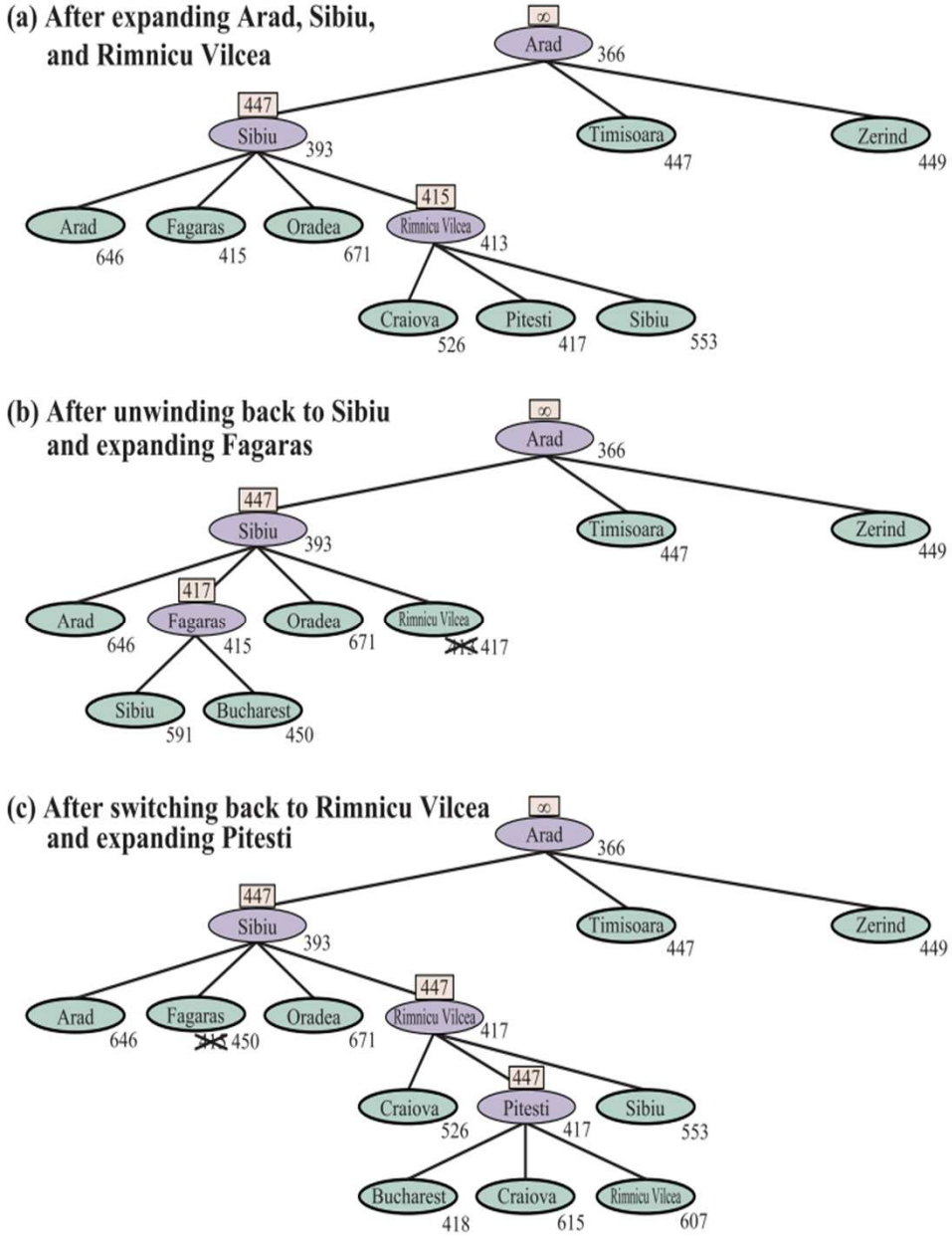


Iterative-deepening A∗ (IDA∗)

- A∗ iterative-deepening search is to depth-first
  - Gets benefits of A∗ but don't keep all reached states in memory, at a cost of visiting some states multiple times.
  - It is a very commonly used algorithm for **problems that do not fit in memory**
- Similar to A* but in each iteration the limit value of the depth is incremented
  - In each iteration the limit value of $f(n)$ is incremented
  - If the value of $f(n)$ is greater than the limit the node n is not explored, otherwise is expanded
  - In each iteration the limit value is updated with the smallest value of $f(n)$ for the nodes not explored in the previous iteration
- For problems where each path's *f-cost* is an integer, this works very well
  - There is a steady progress towards the goal each iteration
  - If the optimal solution has cost C∗, then there can be no more than C∗
- For a problem where every node has a different *f-cost*,
  - each new increment in the depth might contain only one new node
  - the number of iterations could be equal to the number of states

Recursive best-first search (RBFS)

- Similar to the standard best-first search,
  - Avoids continuing indefinitely down the current path
  - f-limit keeps track of the f-value of the best alternative path available from any ancestor of the current node.
  - If the current node exceeds this limit, the recursion unwinds back to the alternative path.
- As the recursion unwinds, RBFS replaces the f-value of each node along the path with the best f-value of its children
  - In this way, RBFS remembers the f-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth reexpanding the subtree at some later time.
- Each change corresponds to one iteration of IDA*
  - RBFS is somewhat more efficient than IDA∗
    => does not explore all nodes smaller than the iteration limit
  - Still suffers from excessive node regeneration
- RBFS is optimal if the heuristic function *h(n)* is admissible



Simplified memory-bounded A∗ (SMA∗)

- IDA∗ and RBFS suffer from using too little memory
  - Between iterations, IDA∗ retains only a single number: the current f-cost limit
  - RBFS retains more information in memory, but it uses only linear space even if more memory were available

- We need determine how much memory we have available, and allow an algorithm to use all of it

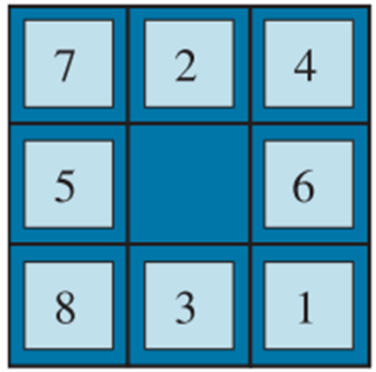
- SMA* proceeds just like A∗, expanding the best leaf until memory is full
  - At this point, it cannot add a new node to the search tree without dropping an old one
  - SMA∗ always drops the worst leaf node — the one with the highest f-value
  - Like RBFS, SMA∗ then backs up the value of the forgotten node to its parent.
  - The ancestor of a forgotten subtree knows the quality of the best path in that subtree.
  - SMA∗ regenerates the subtree only when all other paths have been shown to look worse than it.

Heuristic functions: 8 Puzzle example

- Typical solution in 20 steps with average branching factor: 3
  - Number of states: 320 = 3.5*109
  - No. of states = 9!/2 = 181 440 (15-puzzle → 16!/2 states > 10 trillion)

- Heuristics
  - h1 = No. of parts out of place
  - h2 = Sum of the distances the tiles from their goal positions
  - As expected, **neither of these overestimates** the true solution cost

- Problem **relaxation**, i.e., ignore some constraints, to find heuristics:
  - Original – a piece can move from A to B if A is adjacent to B and B is empty
  - Relaxed problem – some of this restrictions are removedover



Start State

Goal State

Problem relaxation

- h1 and h2 are estimates of the remaining path length for the 8-puzzle
  - they are also perfectly accurate path lengths for simplified versions of the puzzle.
  - If the rules of the puzzle were changed so that a tile could move any adjacent square (even it is occupied) then **h1** would give the **exact length of the shortest solution**.
  - Similarly, if a tile could move one square in any direction, even onto an occupied square, then **h2** would give the **exact length of the shortest solution**.

- A problem with fewer restrictions on the actions is called a relaxed problem
  - Its state-space graph is a supergraph of the original state space
  - The removal of restrictions creates added edges in the graph

- 8-puzzle problem relaxation
  - a) Piece can move from A to B if A is adjacent to B
  - b) Piece can move from A to B if B is empty
  - c) Piece can move from A to B

Exercise 04: Six matchsticks/squares

Suppose the following game for 1 player where the board consists of 6 squares and 6 matchsticks. The goal of the game is to place a matchsticks on each square. To do this, in each move, the player can

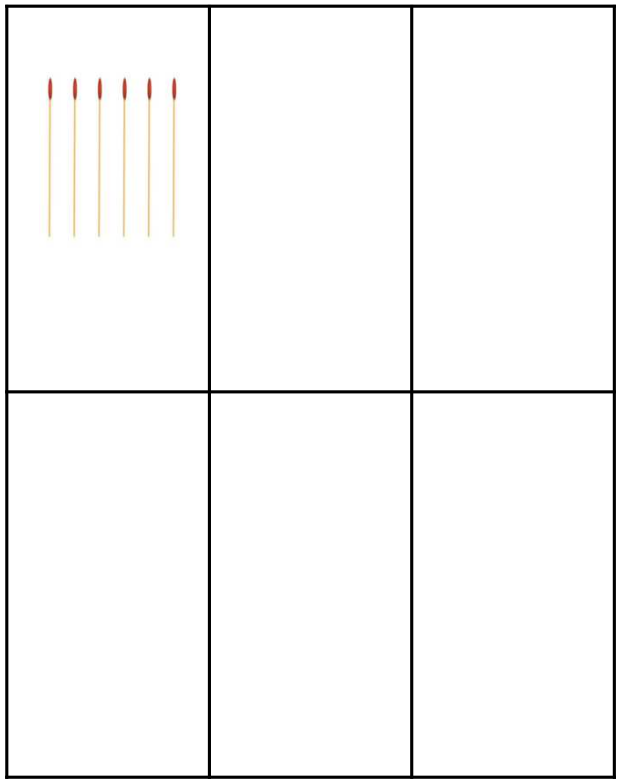
- Move 1 matchsticks from one square to another square that is to his right
- Move 2 matchsticks from one square to another square that is to his left
- Move 2 or 3 matchsticks from one square to another square that is above or below it

A. Formulate the problem as a search problem.

B. Starting from a given state, draw the search trees using the strategies first in width and first in depth (consider moves from square 1 first, then from square 2, and so on).

C. Represent the state space graphically (ignore states where a square has more than 3 matchsticks).

D. Define two heuristic functions that allow you to apply the A* algorithm to the problem and apply it to solve the problem.



Exercise 04 formulation

A. Formulate the problem as a search problem.

B. Starting from a given state, draw the search trees using the strategies first in width and first in depth
   - (consider moves from square 1 first, then from square 2, and so on).

C. Represent the state space graphically
   - ignore states w/ squares > 3 matches

D. Heuristics and solution
   - Heuristic functions (problem relaxation)
   - A* algorithm.

Problem formulation

- State representation (space)
  - All valid states, i.e., states that result from applying the operators to the initial state or another valid state
- Initial state
  - state (for test) = (6,0,0 | 0,0,0)
- Operators (preconditions and effects)
  - Right (R), Left(L), 2 Up (U2), 3 Up (U3), 2 Down (D2), 3 Down (D3)
- Goal test
  - state = (1,1,1 | 1,1,1)
- Cost of the solution
  - Each step costs 1
  - Solution cost = number of steps to solve the problem

Thank you!