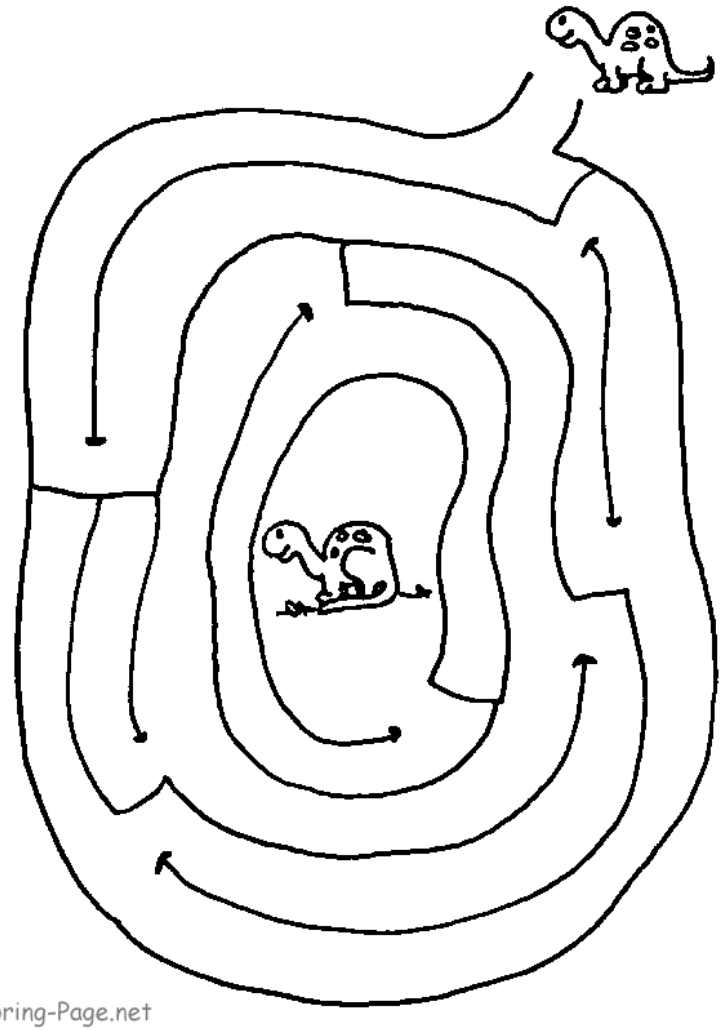
Problem Solving by Searching

Example Problem



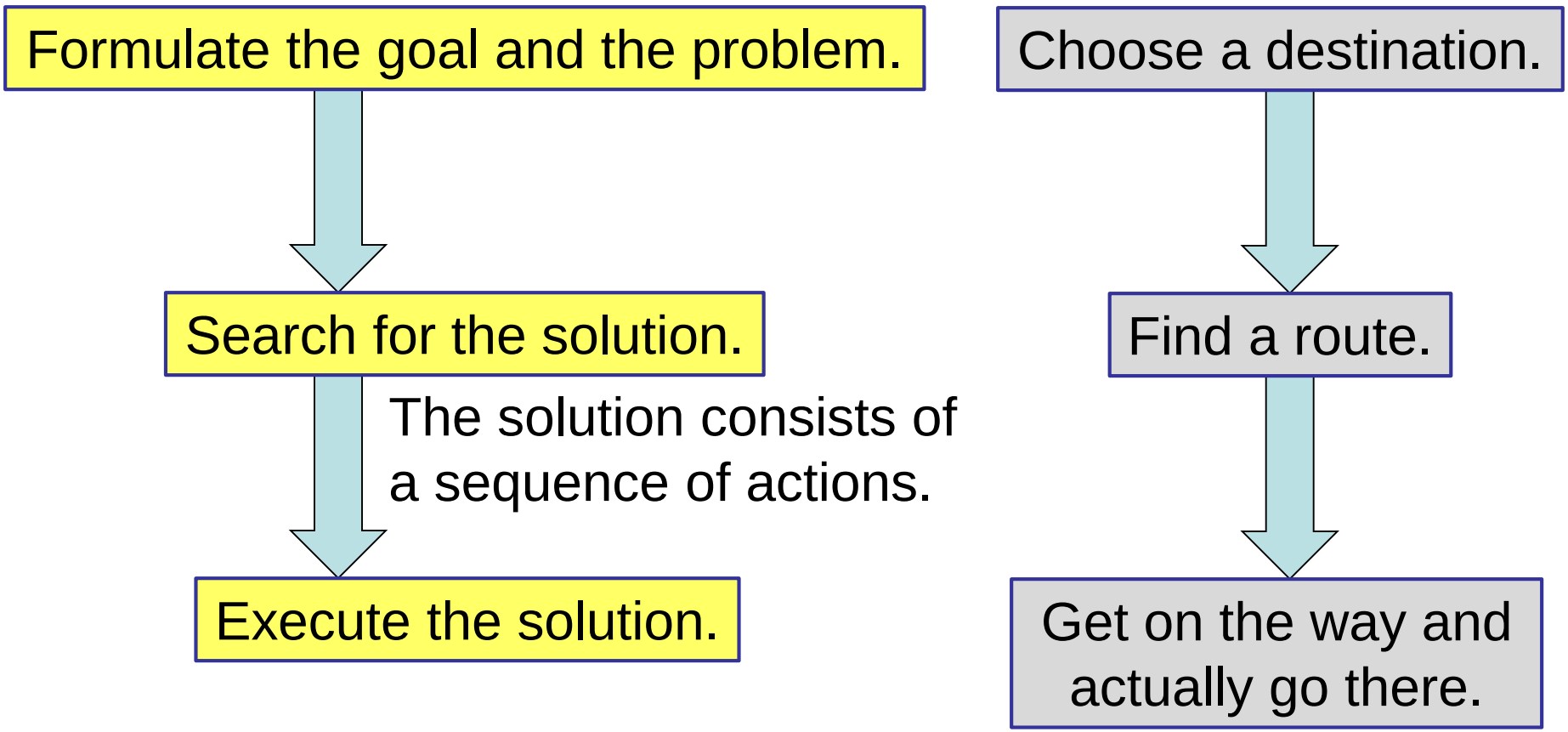
The dinosaur wants to find his friend. Some points to consider:

- Does the dinosaur have the map of the maze?
- Does the dinosaur know the location of his friend?

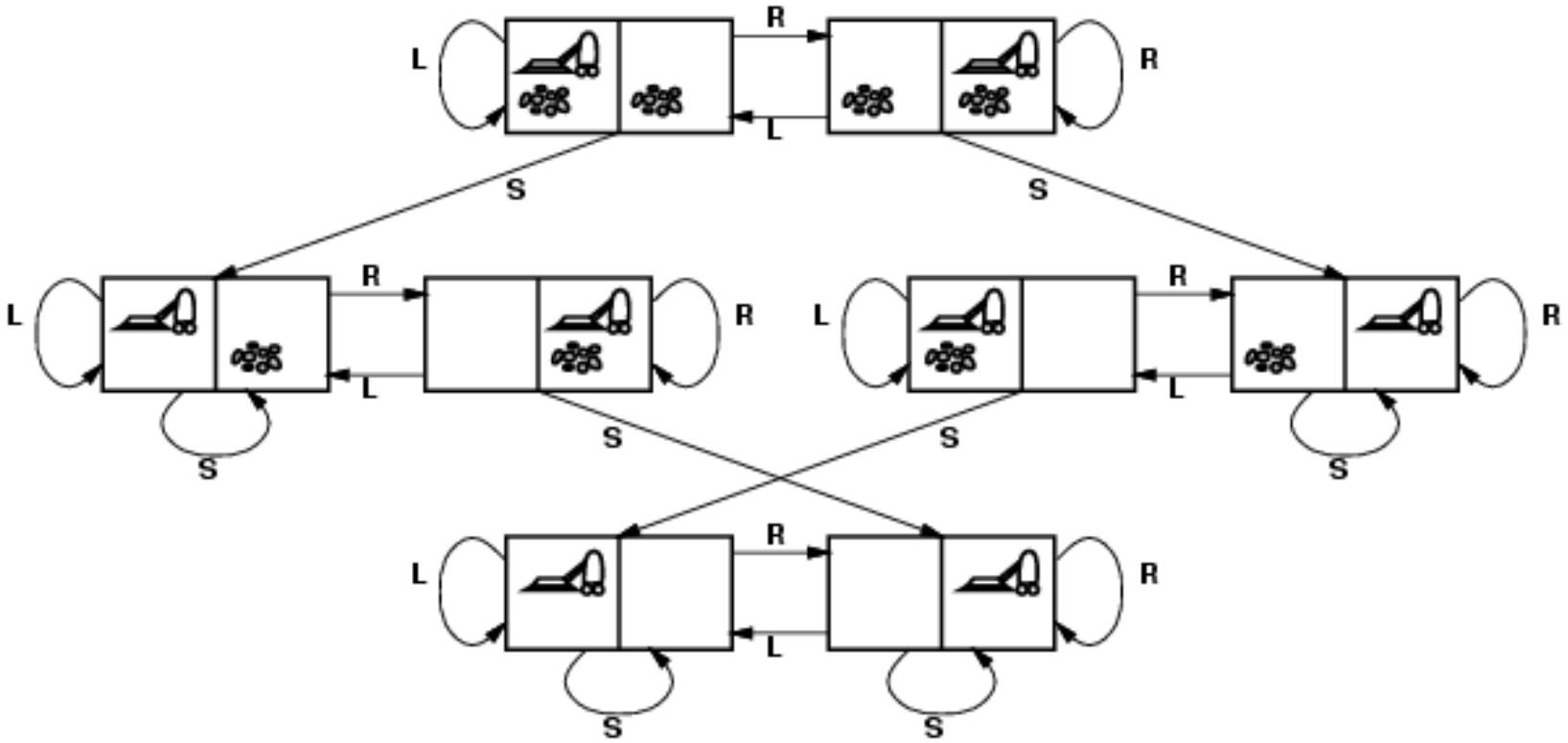
Problem Formulation

- **Initial State**:
- **Actions**: Actions(s) is the set of actions applicable in state s.
- **Transition model**: Result(s,a) is the state reached by doing action a in state s.
- **Goal test**: Whether the state is a goal state.
  - Example (explicit): { *In*(*MyDesiredDestination*) }
  - Example (implicit): *checkmate*(x)
- **Solution**:
  - A sequence of actions (e.g., a route)
  - The goal state found
- **Path cost**: (This should reflect the performance measure.)
  - Additive cost (assumed here): sum of step costs $c(s,a,s')$.

To Start: Offline Problem Solving

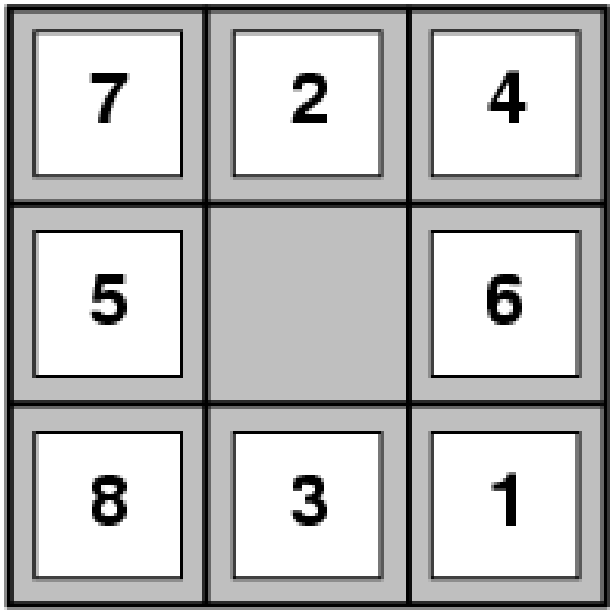


Example: Vacuum-Cleaner World

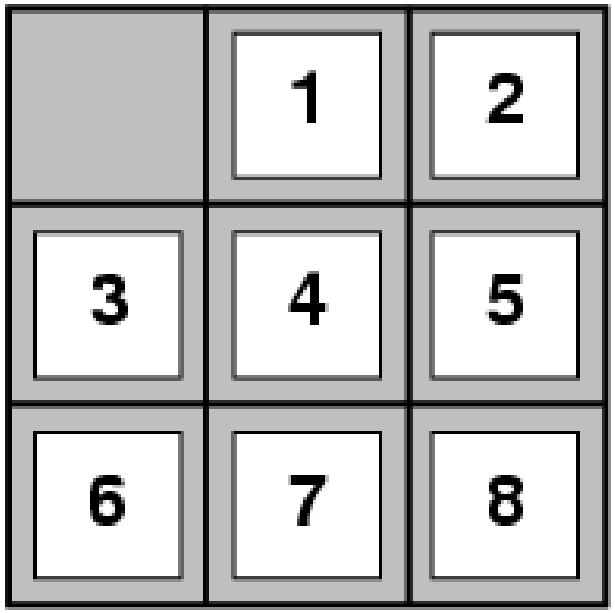


- states: existence of dirt, robot location
- actions: *Left*, *Right*, *Suck*
- transition model: (see above)
- goal test: no dirt at either locations
- path cost: 1 per action

Example: 8-Puzzle



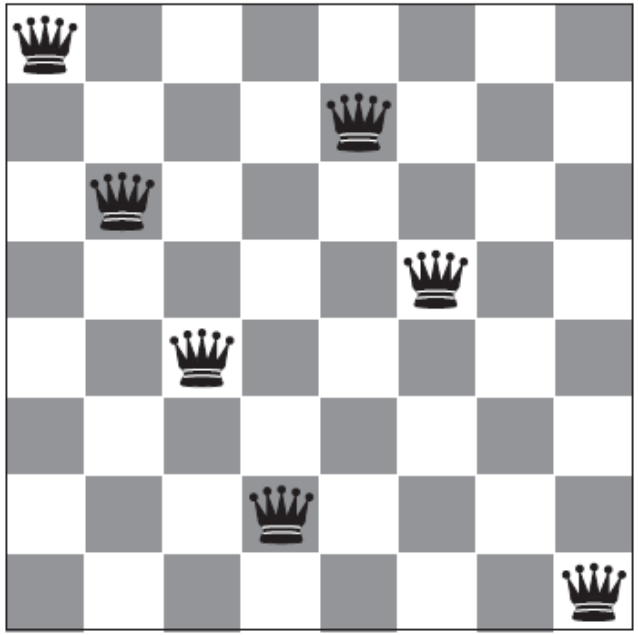
Start State



Goal State

- states: arrangements of tiles
- actions: *Left*, *Right*, *Up*, *Down* (the blank)
- transition model:
- goal test: given (a particular arrangement of tiles)
- path cost: 1 per move

Example: 8-Queens



Incremental formulation:

- states: arrangements of 0-8 queens on the board
- initial state: empty board
- actions: add a queen to any empty square
- transition model:
- goal test: 8 queens on the board, none attacked by others
- path cost: (who cares)

Example Problem: Romania

Search Tree

- The process of searching for a solution can be viewed as **growing** a tree.
  - Each node corresponds to a state.
  - A state might correspond to multiple nodes (if the state is visited multiple times during the search).
- The initial tree contains only a node (the initial state).
- **Node expansion**: Adding child nodes to a leaf node.
  - Add a child node for each state that results from applying a legal action at the state of the node being expanded.
- **Frontier**: The set of leaf nodes available for expansion, but not yet expanded.

A Tree-Search Algorithm

function Tree-Search(*problem*) **returns** a solution, or failure

initialize the frontier using the initial state of the problem

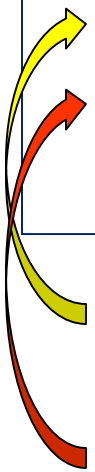
loop do

if the frontier is empty **then return** failure

choose a leaf node and remove it from the frontier

if the node contains a goal state **then return** the corresponding solution

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

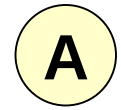


How this node is selected is called the **searching strategy**.

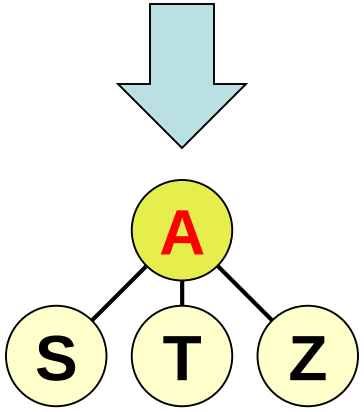
The goal test can happen at node expansion (as in here) or at node generation, depending on the searching strategy.

Example Search Tree

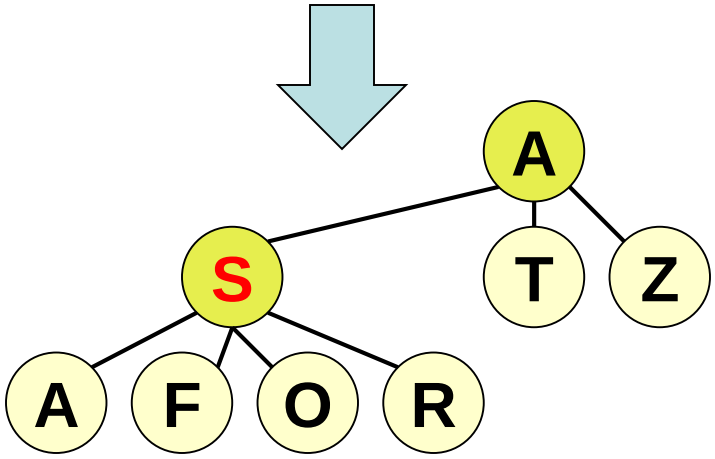
Initial state:



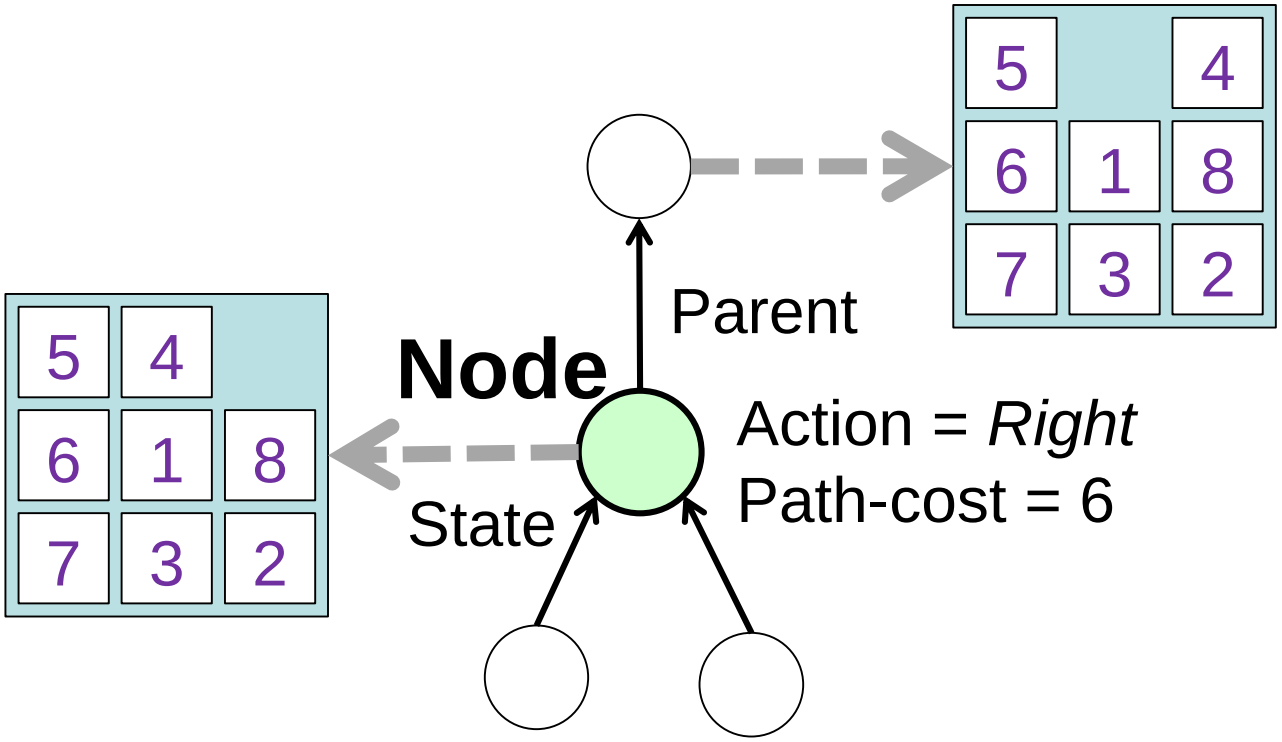
Expand **A**:



Expand **S**:



Information Kept by Nodes

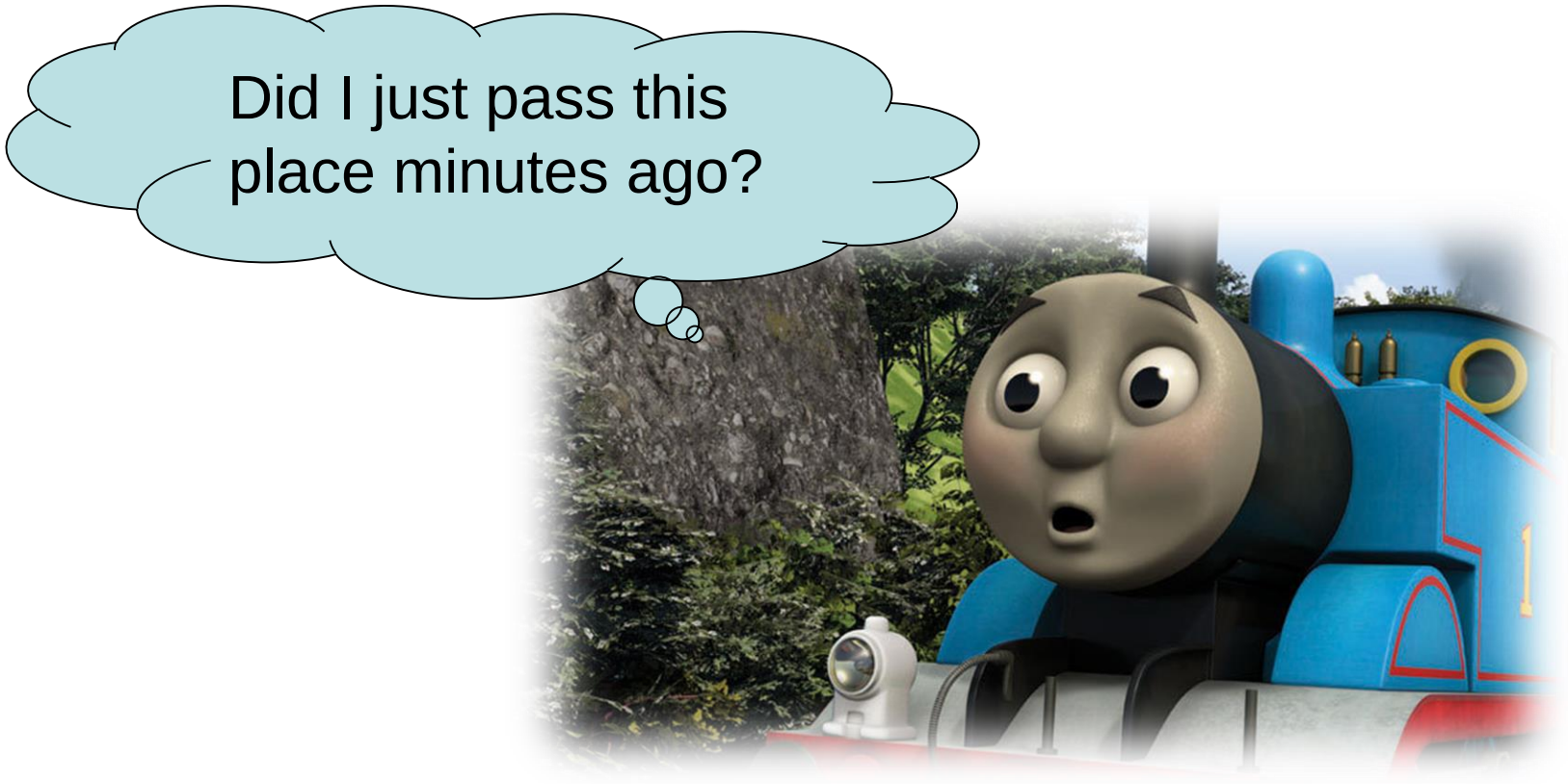


The *parent* pointer (and hence the actual tree structure) is needed only if

- We need to back-track (some searching algorithms), or
- The desired solution is a path to the goal state.

Repeated States

- “Algorithms that forget their history are doomed to repeat it.”
- Repeated states can make a simple problem intrackable.



Graph Search

- Search in the state-space (a graph), keeping only one node for each state.
- To do this, we add an **explored set** to remember the states of all the nodes that are already **expanded**.

function Graph-Search(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of the problem
  initialize the explored set to be empty
loop do
  if the frontier is empty **then return** failure
  choose a leaf node and remove it from the frontier
  if the node contains a goal state **then return** the corresponding solution
  add the node (actually, the state) to the explored set
  expand the chosen node, adding the resulting nodes to the frontier
    only if its state is not in any nodes in the frontier or explored set

Data Structures for Searching

- **Frontier**:
  - FIFO (queue)
  - FILO (stack)
  - Priority queue
- **Explored Set**:
  - Hash table
  - Look-up table (for finite state spaces that are sufficiently small)
  - For some problems, it is impractical to remember all the explored states.

Searching Strategies

- A strategy is how we decide the **order of node expansion**.
- Considerations for choosing a strategy:
  - Completeness: Does it always find a solution if one exists?
  - Time complexity: Number of nodes generated/expanded
  - Space complexity: Maximum number of nodes in memory
  - Optimality: Does it always find a least-cost (optimal) solution?
- Time and space complexity are measured in terms of
  - b: Maximum branching factor of the search tree
  - d: Depth of the optimal solution
  - m: Maximum depth of the state space (may be infinite).

Uninformed Searching Strategies

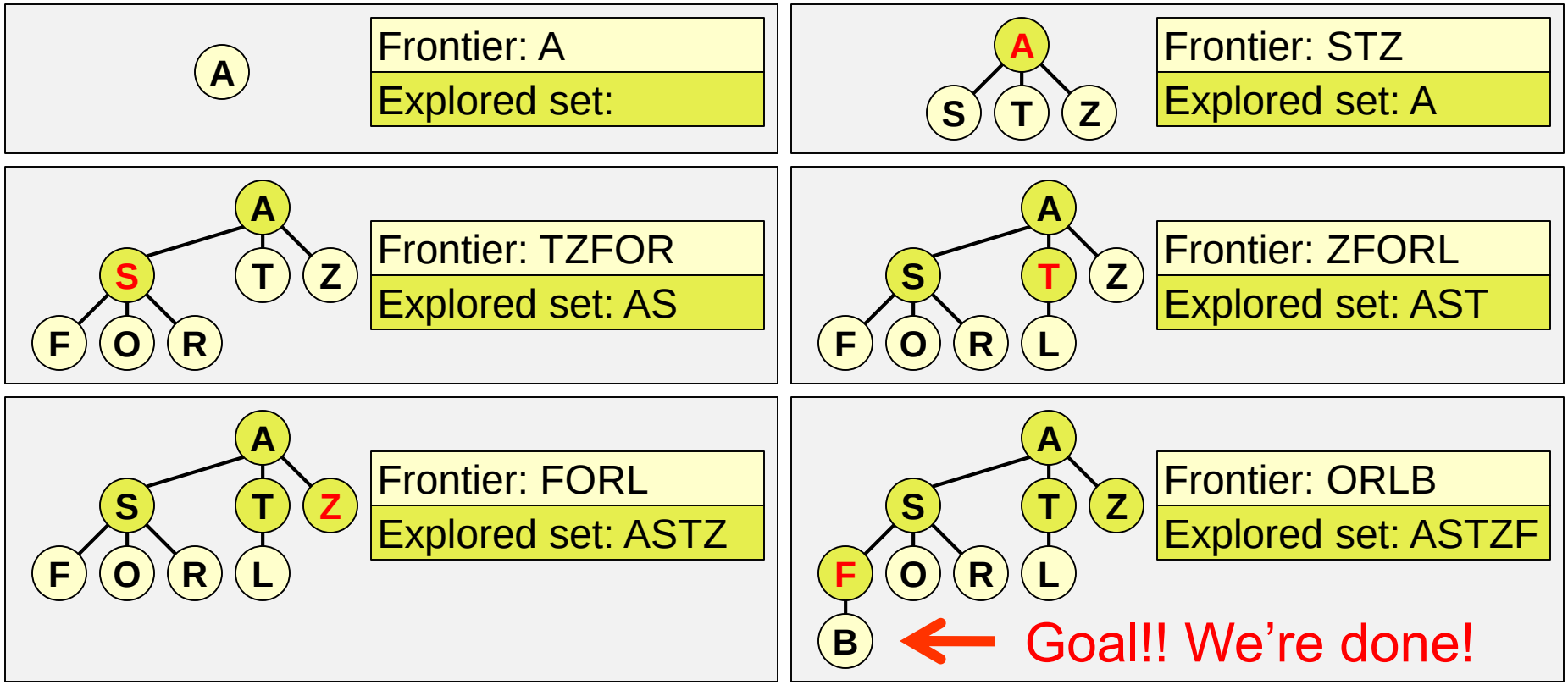
Uninformed searching strategies use only the information available in the problem definition.

- Use step counts as path costs (unit step costs):
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
  - Depth-limited search
  - Iterative deepening search (IDS)
  - These searching strategies have **tree-search** and **graph-search** versions.
- Step costs considered:
  - Uniform-cost search (Dijkstra's algorithm)

Breadth-First Search (BFS)

- Strategy: Expand shallowest unexpanded node.
- Frontier: FIFO queue
- Goal test at node generation

This example is based on graph-search of the Romania map:



Breadth-First Search Properties

Tree-search:

- Completeness: Yes (if d is finite)
- Time complexity: $O(b^d)$
- Space complexity: $O(b^d)$ The main problem
- Optimality: Yes (for unit step costs)

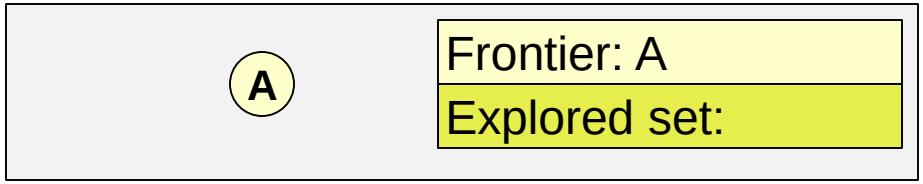
Graph-search:

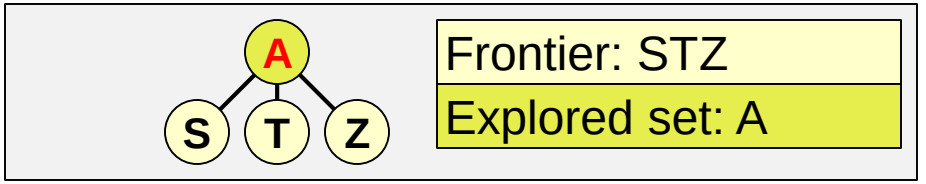
- Time and space complexities are bounded by the size of the state space.

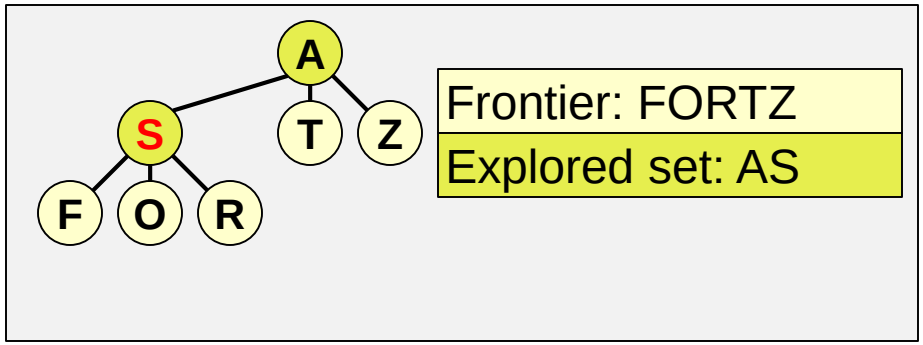
Depth-First Search (DFS)

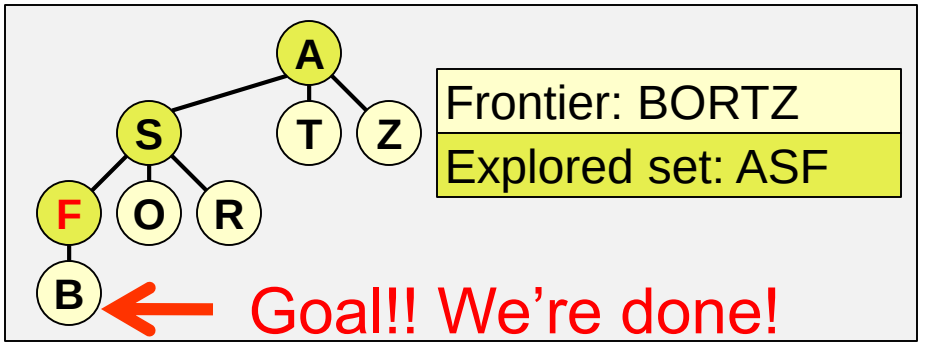
- Strategy: Expand the deepest (most recent) unexpanded node.
- Frontier: FILO (stack)
- Goal test at node generation
- DFS can be implemented recursively, and therefore can be used in online search.

This example is based on graph-search of the Romania map:









Depth-First Search Properties

Tree-search:

- Completeness: No
- Time complexity: $O(b^m)$
- Space complexity: $O(mb)$
- Optimality: No

Note: Repeated states can cause infinite searches when there are loops in the state space.

Graph-search:

- Complete for finite state spaces.
- Time and space complexities are bounded by the size of the state space.

Depth-Limited Search Properties

Tree-search:

- Completeness: No
- Time complexity: $O(b^l)$
- Space complexity: $O(bl)$
- Optimality: No

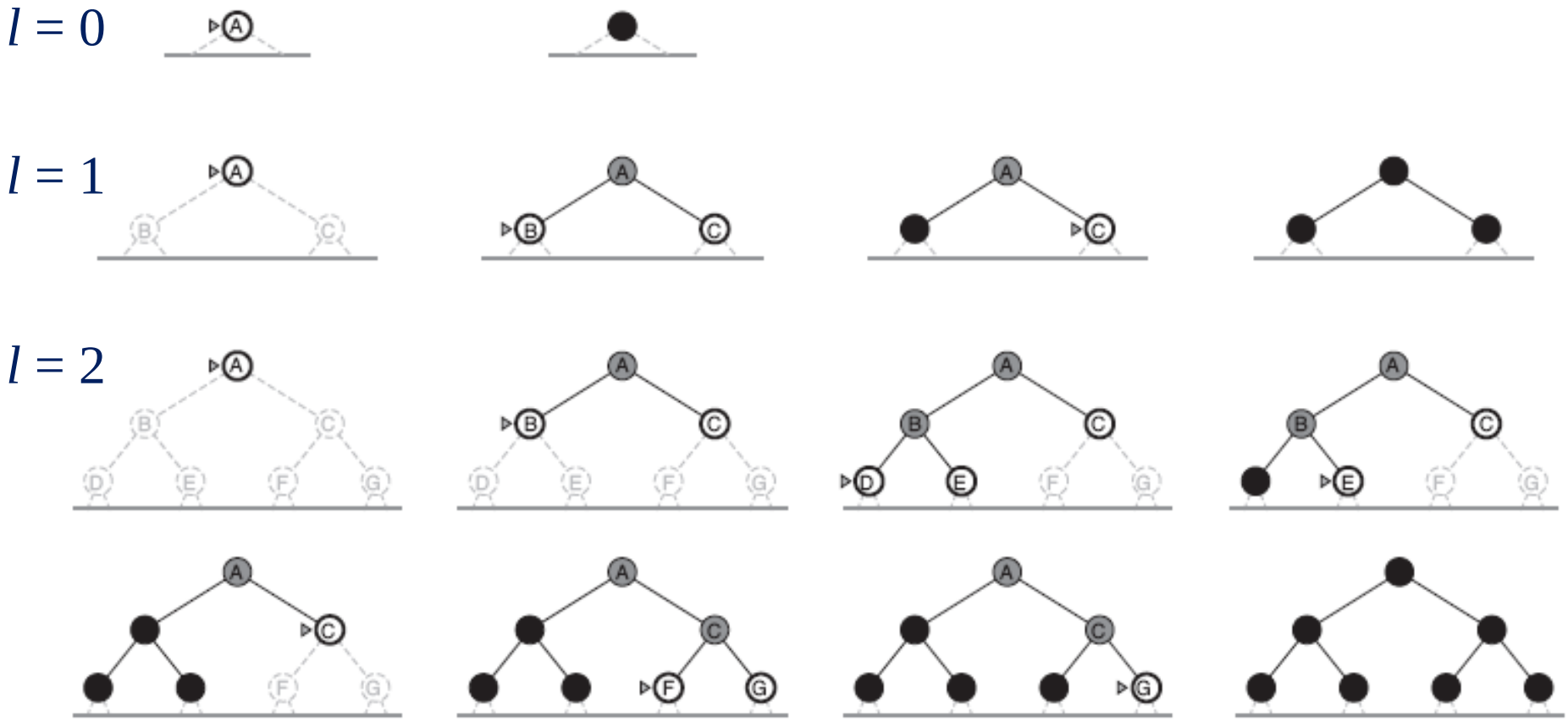
This is DFS with a maximum depth l. Benefit: To avoid infinite loops and overly long paths.

Graph-search:

- Time and space complexities are bounded by the size of the state space.

Iterative Deepening Search (IDS)

Idea: Iteratively apply depth-limited search with increasing maximum depth l.

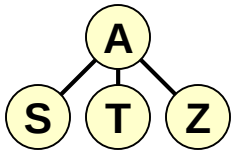


Iterative Deepening Search Example

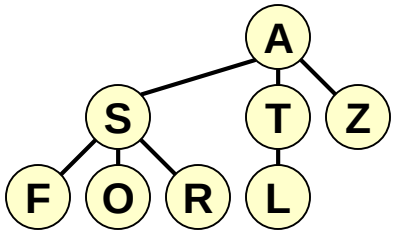
This example is based on graph-search of the Romania map. Each step here only shows all the nodes at the given depth limit.

$l = 0$

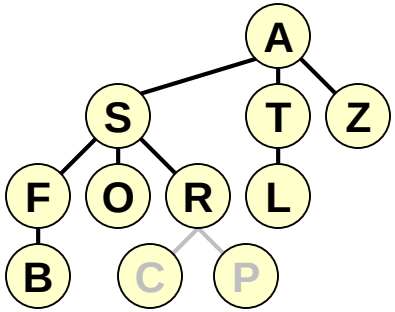
$l = 1$



$l = 2$



$l = 3$



Iterative Deepening Search Properties

Tree-search:

- Completeness: Yes (if d is finite)
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$ — The main advantage over BFS
- Optimality: Yes (for unit step costs)

Graph-search:

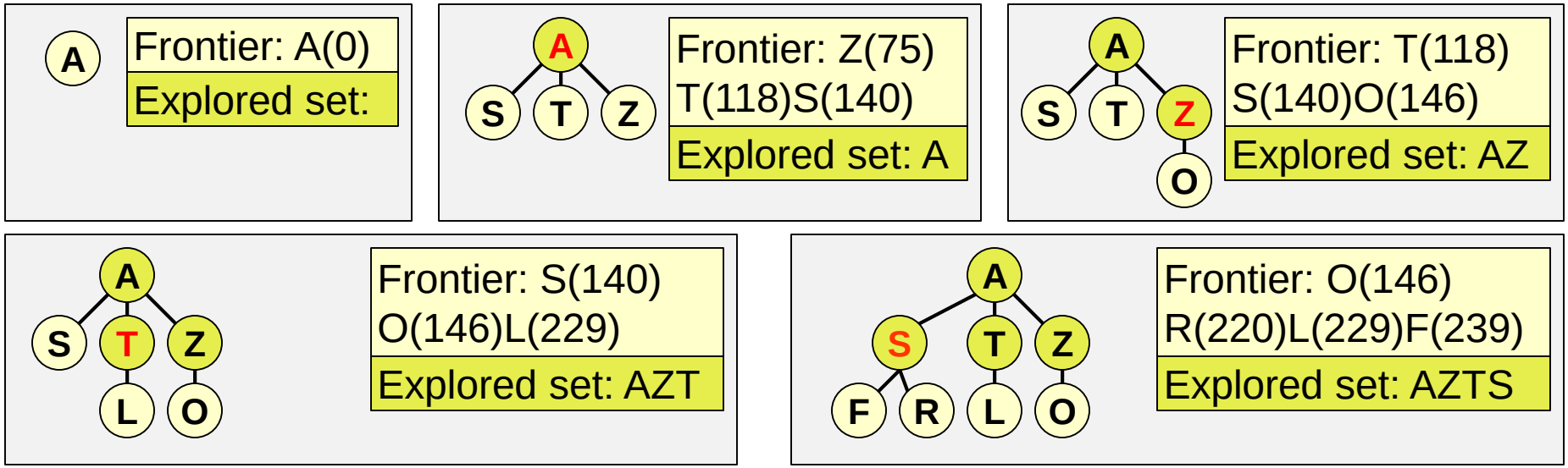
The apparent "waste" of generating nodes of the same state multiple times is actually insignificant in general cases.

- Time and space complexities are bounded by the size of the state space.

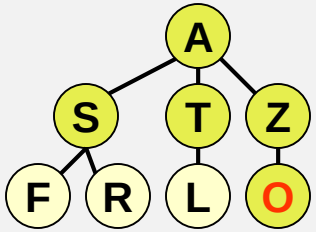
Uniform-Cost Search

- Strategy: Expand the least-cost unexpanded node.
- Frontier: priority queue (lowest cost at front)
- Goal test at node expansion (**why?**)
- This is basically the single-source-single-goal version of **Dijkstra's Algorithm**.

This example is based on graph-search of the Romania map (duplicated states checked only against the explored set):

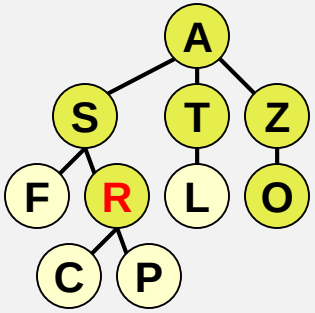


Uniform-Cost Search Example (Cont.)



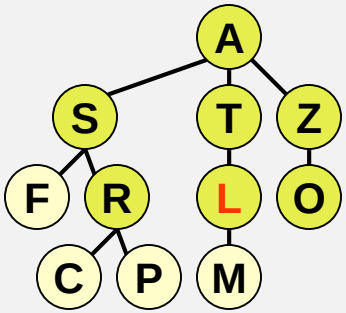
Frontier:
R(220)L(229)F(239)

Explored set:
AZTSO



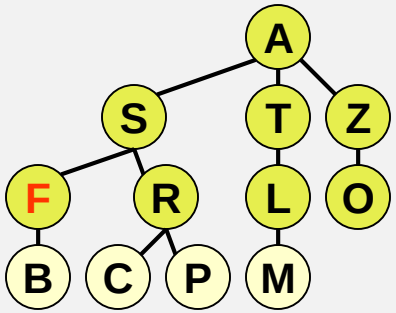
Frontier: L(229)
F(239)P(317)C(366)

Explored set:
AZTSOR



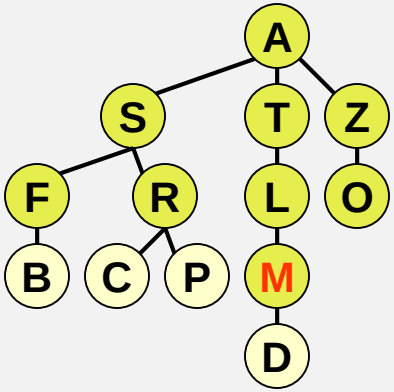
Frontier: F(239)
M(299)P(317)C(366)

Explored set:
AZTSORL



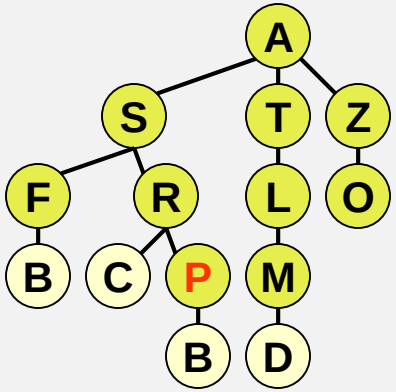
Frontier: M(299)
P(317)C(366)B(460)

Explored set:
AZTSORLF



Frontier: P(317)
C(366)D(374)B(460)

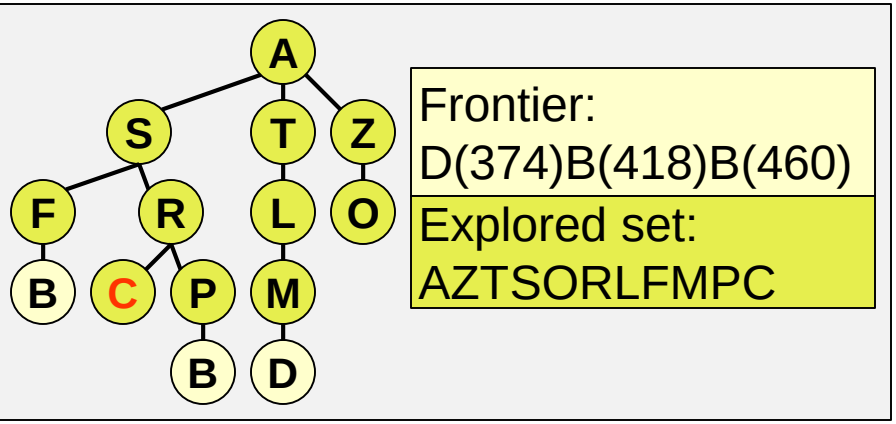
Explored set:
AZTSORLFM

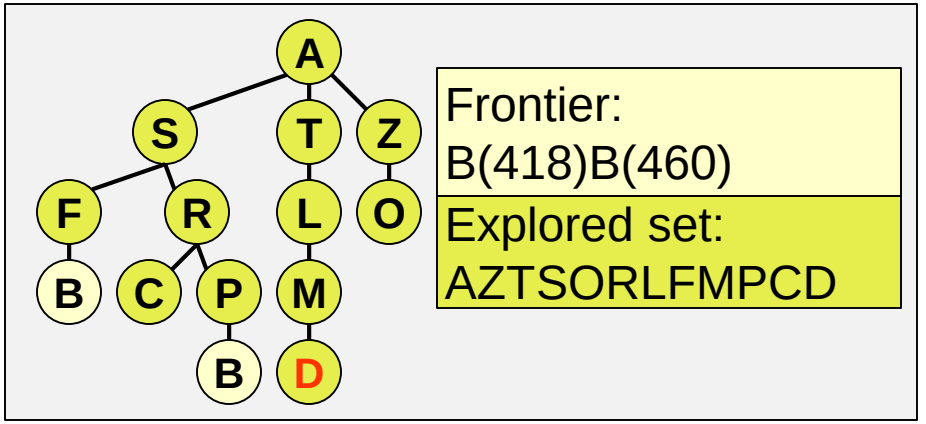


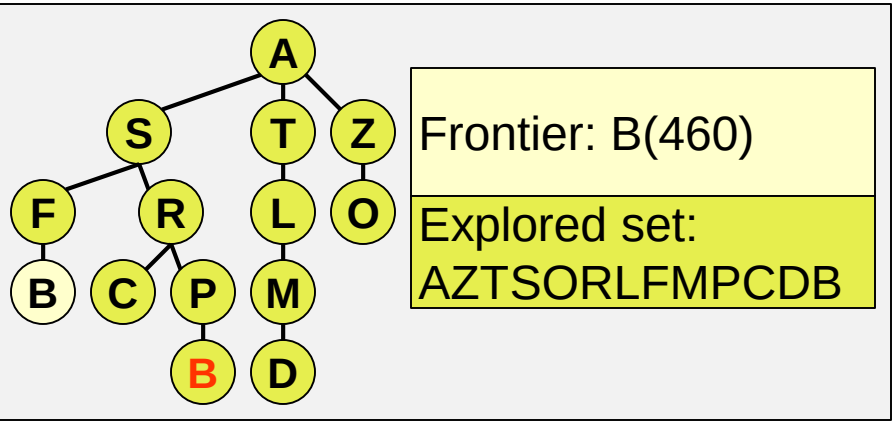
Frontier: C(366)
D(374)B(418)B(460)

Explored set:
AZTSORLFMP

Uniform-Cost Search Example (Cont.)







If using a priority queue that supports the priority-update operation, there will be at most one instance of each state in the frontier. (Consider the multiple nodes for state **B** in the example here.)

Uniform-Cost Search Properties

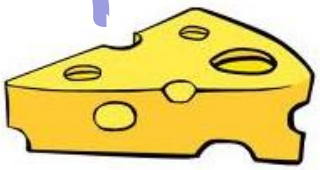
- Completeness: Yes (if finite goal cost and all step costs $\geq \varepsilon$)
- Time complexity:
- Space complexity:
- Optimality: Yes

The optimal depth d for BFS is replaced here by $1+C^*/\varepsilon$. (C^*: cost of optimal solution; ε: smallest step cost.)

Q: Is it a good idea to do iterative-deepening uniform-cost search?

Informed Searching Strategies

Which path should he choose?



Informed Searching Strategies

Informed strategies use problem-specific knowledge beyond the problem definition in order to find the solution more efficiently.

We will discuss the following variants:

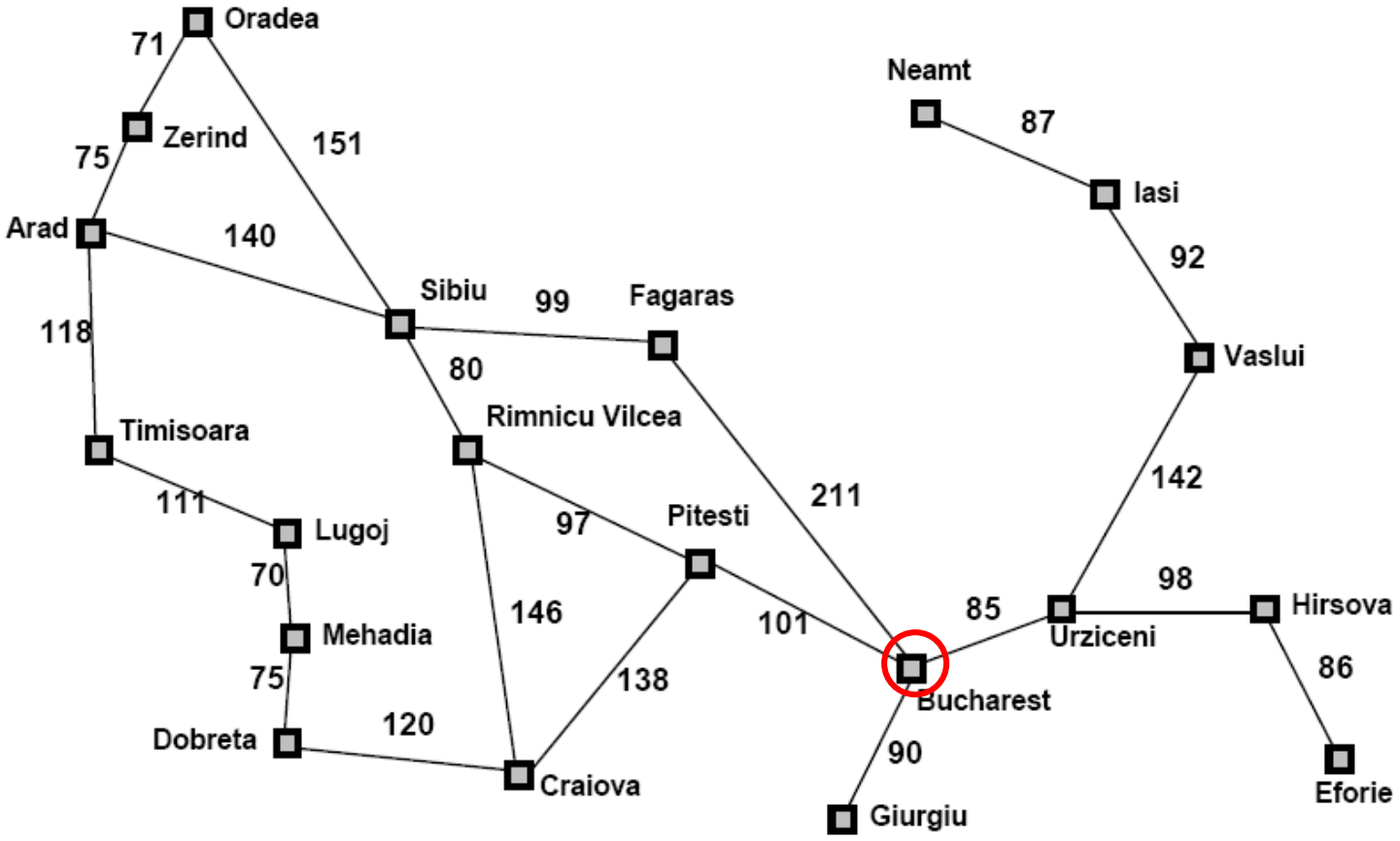
- Greedy best-first search
- A* search
- Iterative deepening A* (IDA*)

Best-First Search and Heuristics

- **Best-first search**: In tree or graph searches, expand the node that appears most likely to lead to the (optimal) solution.
- **Evaluation function** $f(n)$: Used to select the best node for expansion.
- Frontier: priority queue (lowest $f(n)$ at front).
- Goal test at node expansion (reason: the same as uniform-cost search)
- **Heuristic function** $h(n)$: An estimated cost of reaching the solution from node n. This is where the "problem-specific knowledge" comes in.

Example Problem: Romania

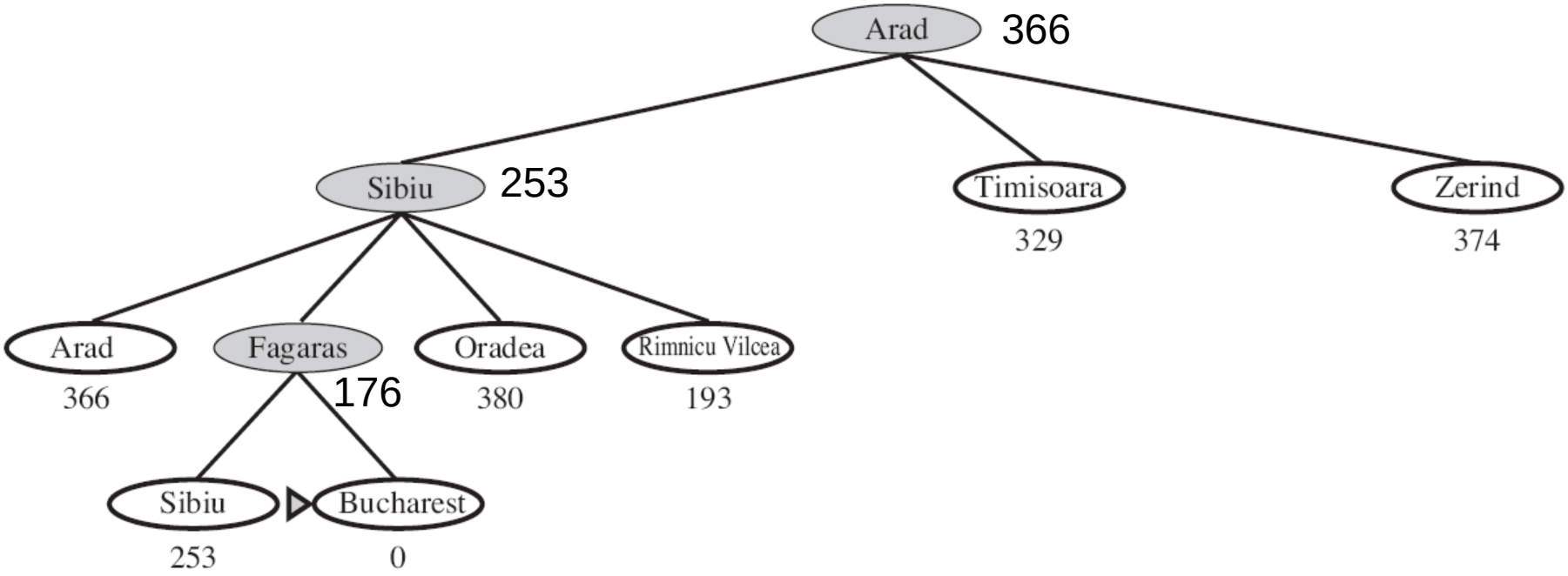
Heuristic: straight-line distance to goal



Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Best-First Search

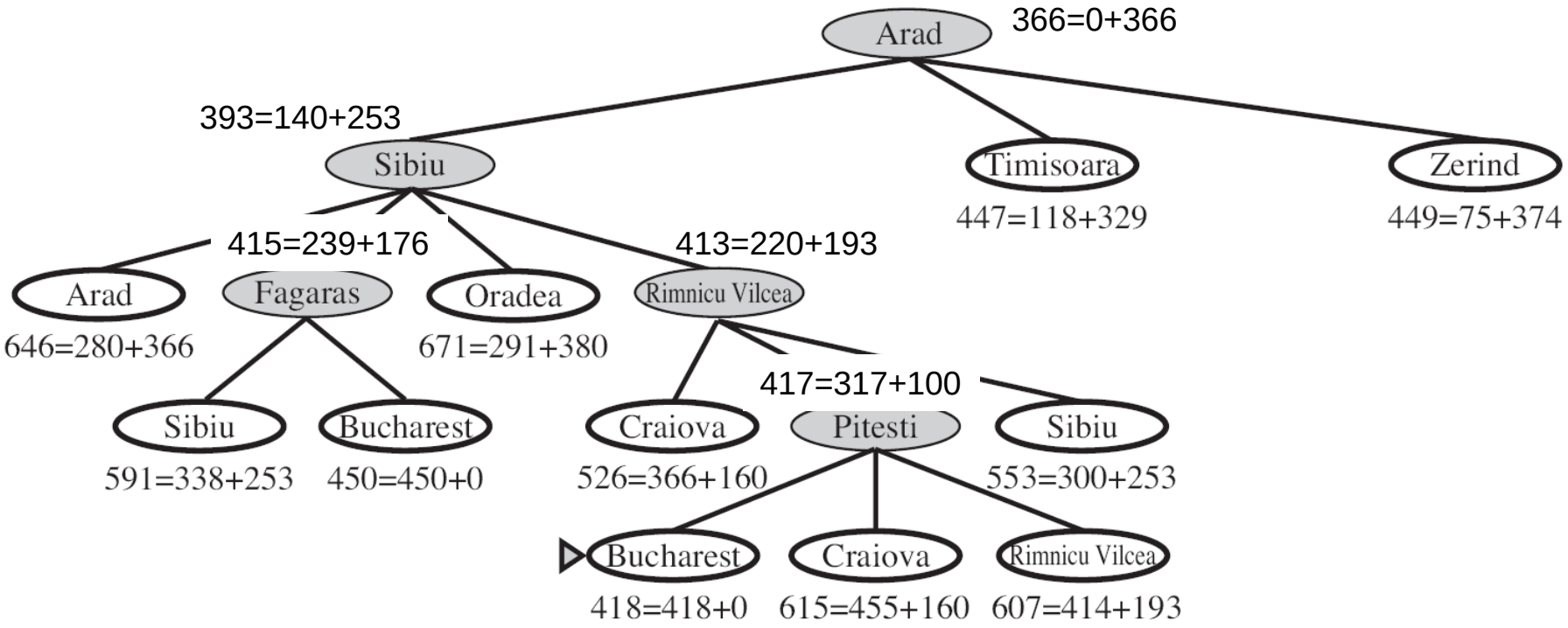
- Strategy: Expand the unexpanded node with the lowest $h(n)$, i.e., $f(n) = h(n)$.



Greedy best-first search is neither complete (except for its graph-search version in finite state spaces) nor optimal. Why?

A* Search

- Strategy: Expand the unexpanded node with the lowest estimated total path cost, i.e., $f(n) = g(n)+h(n)$. Here $g(n)$ is the current path cost (from the root to node n).



A* Search

- Completeness: (same as uniform-cost search) Yes, unless there are infinitely many nodes with $f < C^*$. (Here C^* is the optimal path cost to a goal state.)
- Time complexity: Exponential in [relative error in $h \times$ length of solution]
- Space complexity: Keeps all nodes in memory
- Optimality: Yes, assuming some conditions on the heuristic.
  - A* expands all nodes with $f(n) < C^*$.
  - A* expands some nodes with $f(n) = C^*$.
  - A* expands no nodes with $f(n) > C^*$.

Optimality of A* Search

- A* tree-search is optimal if the heuristic is admissible.
  - **Admissible heuristic**: $h(n)$ never overestimates the true cost from node n to the goal.
- A* graph-search is optimal if the heuristic is consistent.
  - **Consistent heuristic**: $h(n)$ satisfies triangle inequality:

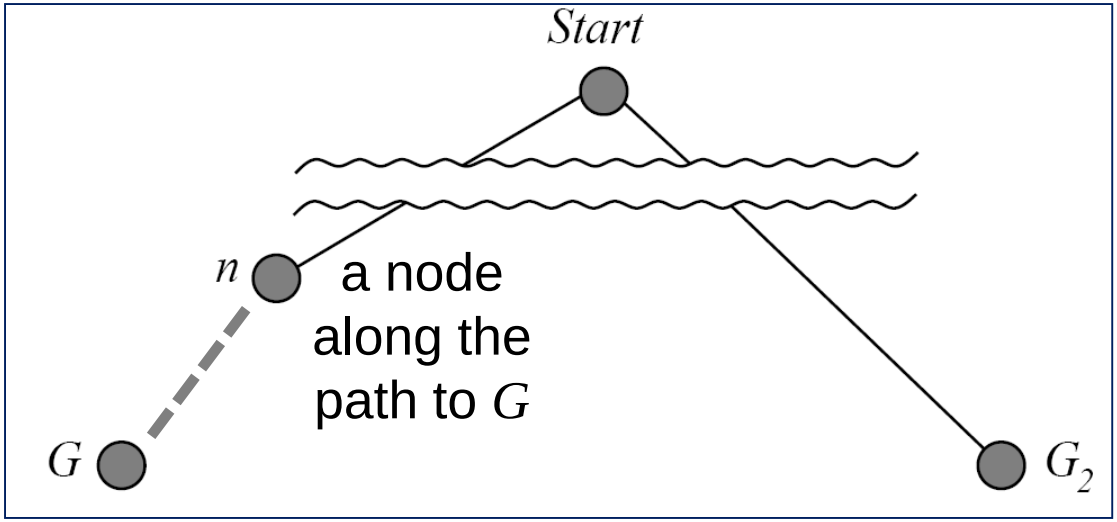
    $h(n) \leq c(n,a,n') + h(n')$ for any successor n' and action a

Note: Consistent heuristic functions are always admissible.

Optimality of A* Tree Search

Idea of proof (similar to uniform-cost search): A suboptimal goal state is never expanded before the optimal one.

optimal goal state



suboptimal goal state

$f(G_2) = g(G_2)$ since $h(G_2) = 0$

$> g(G)$ since G_2 is suboptimal

$\geq g(n) + h(n)$ since $h(n)$ is admissible

$= f(n)$

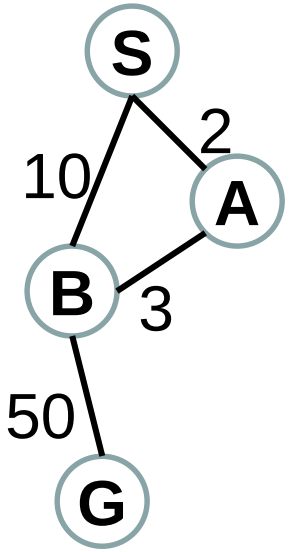
Optimality of A* Graph Search

Consistent heuristic: The value of $f(n)$ along any path is non-decreasing. To prove, let n' by a sucessor of n:

$$
\begin{aligned}
f(n') &= g(n') + h(n') \\
&= g(n) + c(n,a,n') + h(n') && \text{by some action } a \\
&\geq g(n) + h(n) = f(n) && \text{since } h(n) \text{ is consistent}
\end{aligned}
$$

This assures that the nodes are expanded in the order of increasing $f(n)$, which is $g(n)$ if n is a goal state. Therefore, the optimal goal state is always expanded first.

An example demonstrating why the graph-search A* algorithm is not optimal when the heuristic function is not consistent



Let h(A)=50 and h(B)=5.
➔ h(A) =50 > c(A,B)+h(B) = 3+5=8
This violates the triangular inequality, so the heuristic is not consistent, although it is still admissible.

Now when S is expanded, both A and B are put in the frontier. We have

f(A) = g(A) + h(A) = 2 + 50 = 52
f(B) = g(B) + h(B) = 10 + 5 = 15

As a result, B is selected next for expansion before A.
After B is expanded, G (the goal state) is put in the frontier.
A is selected for expansion next. However, since B is already expanded, no new node for B is generated (graph-search).
G is expanded next. A solution is now found: S→B→G. It is not optimal because the link from A to B is not checked.

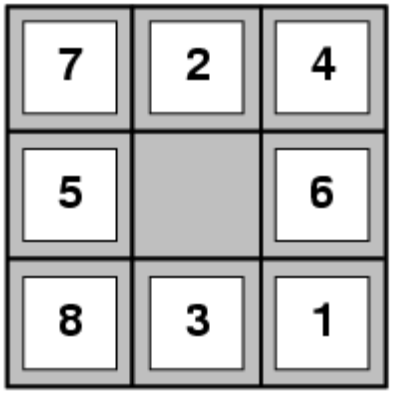
If we use tree-search, when A is expanded, a new node for B (with parent A) can be generated, with new f(B) = g(B) + h(B) = 5+5 = 10. This new node for B will be expanded before the current node for G. This eventually results in a new path that is optimal: S→A→B→G.

Iterative-Deepening A* Search (IDA*)

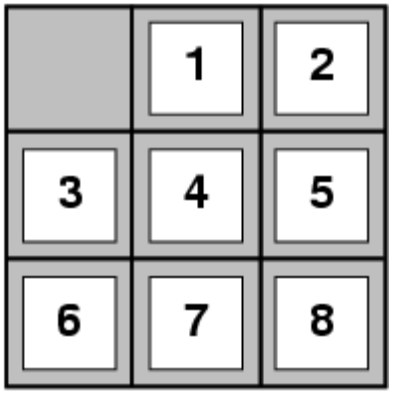
- The main drawback of A* search is the space complexity. This is similar to the problem of BFS.
- The solution to the problem of BFS is to use IDS. The same can be applied to A* search, resulting in IDA* search.
- For problems with unit step costs, IDA* is a good choice compared to A*. (e.g., 8-puzzle)
- For problems with real-valued step costs, IDA* becomes impractical. This is similar to the case of iterative-deepening uniform-cost search. (e.g., route finding in the real world)

More on Heuristic Functions

In order to gain more insights on heuristics, we will look at the 8-puzzle problem again.



Start State



Goal State

Two possible admissible heuristics:

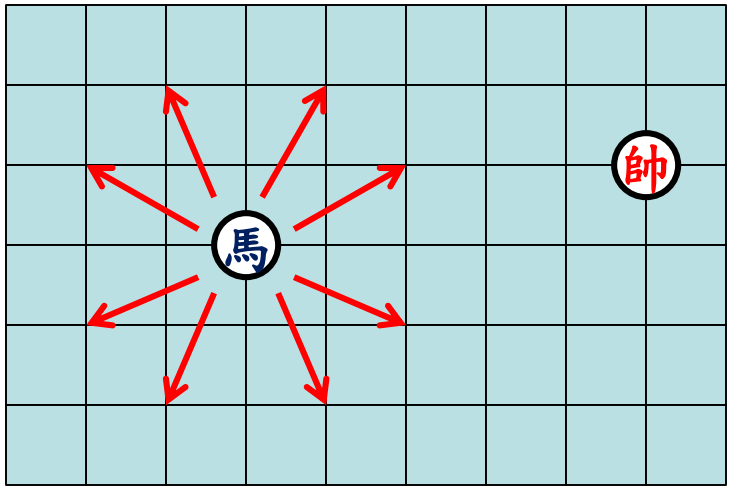
- $h_1(n)$: number of misplaced tiles.
- $h_2(n)$: total Manhattan distances of all the tiles to their goal-state locations.

What are their values for the initial state?

More on Heuristic Functions

How do we design heuristic functions? One approach is to use a **relaxed problem** (with some constraints of the original problem removed) where the path costs can be computed directly without searching. Examples:

Heuristic Function	Constraint Removed
straight line (routing)	The path has to follow existing roads in the map.
$h_1(n)$: (8-puzzle)	A tile can be moved to an adjacent space in a step.
$h_2(n)$: (8-puzzle)	A tile can only be moved to an empty space.



Can you design a heuristic function for this problem?

Dominance Between Heuristics

If $h_2(n) \geq h_1(n)$ for all n (both admissible), then h_2 **dominates** h_1 and is better for search.

Average search costs (8-puzzle with known solution depth):

d = 12	IDS:	3644035 nodes
	A*(h_1):	227 nodes
	A*(h_2):	73 nodes
d = 24	IDS:	54000000000 nodes
	A*(h_1):	39135 nodes
	A*(h_2):	1641 nodes

Given any admissible heuristics h_a and h_b, then

$$h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates h_a and h_b.