CS 5/7320
Artificial Intelligence

Solving problems by searching

AIMA Chapter 3

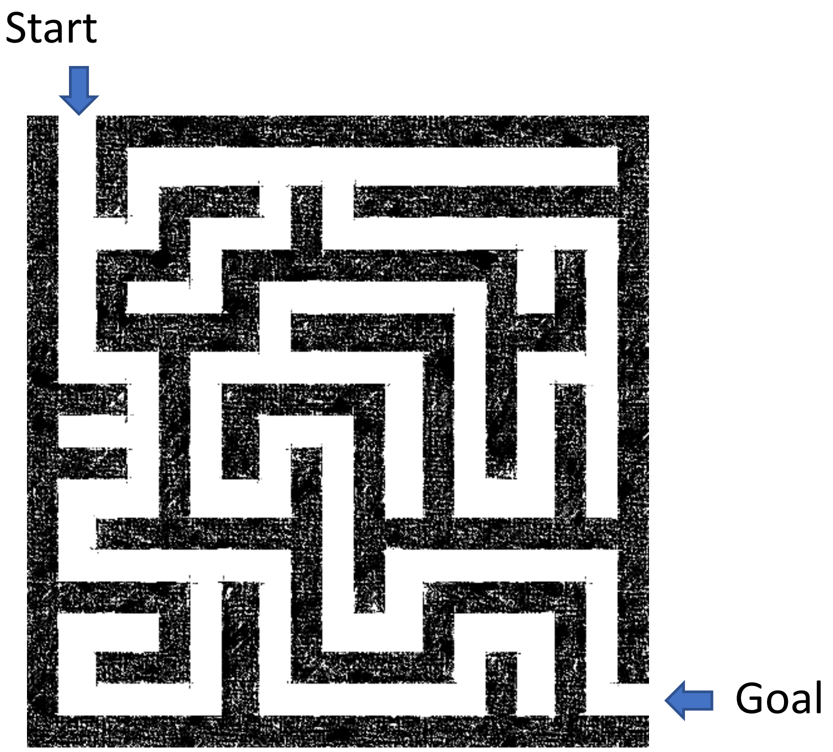
Slides by Michael Hahsler
based on slides by Svetlana Lazepnik
with figures from the AIMA textbook.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Contents

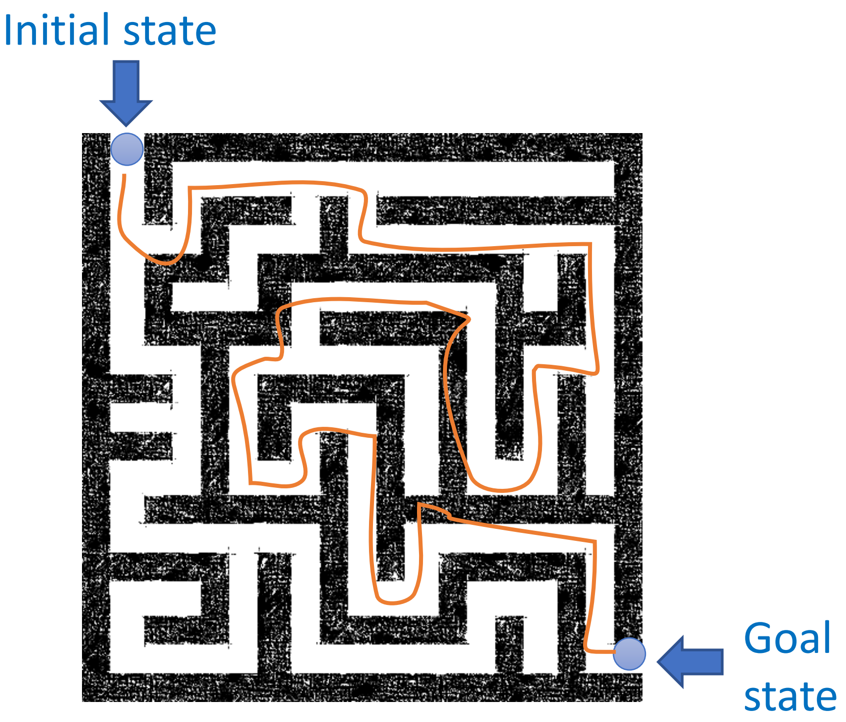
What are search problems?

- We will consider the problem of designing **goal-based agents** in **known, fully observable, and deterministic** environments.
- Example:



What are search problems?

- We will consider the problem of designing **goal-based agents** in **, known, fully observable, deterministic** environments.
- For now, we consider only a discrete environment using an **atomic state representation** (states are just labeled 1, 2, 3, …).
- The **state space** is the set of all possible states of the environment and some states are marked as **goal states**.

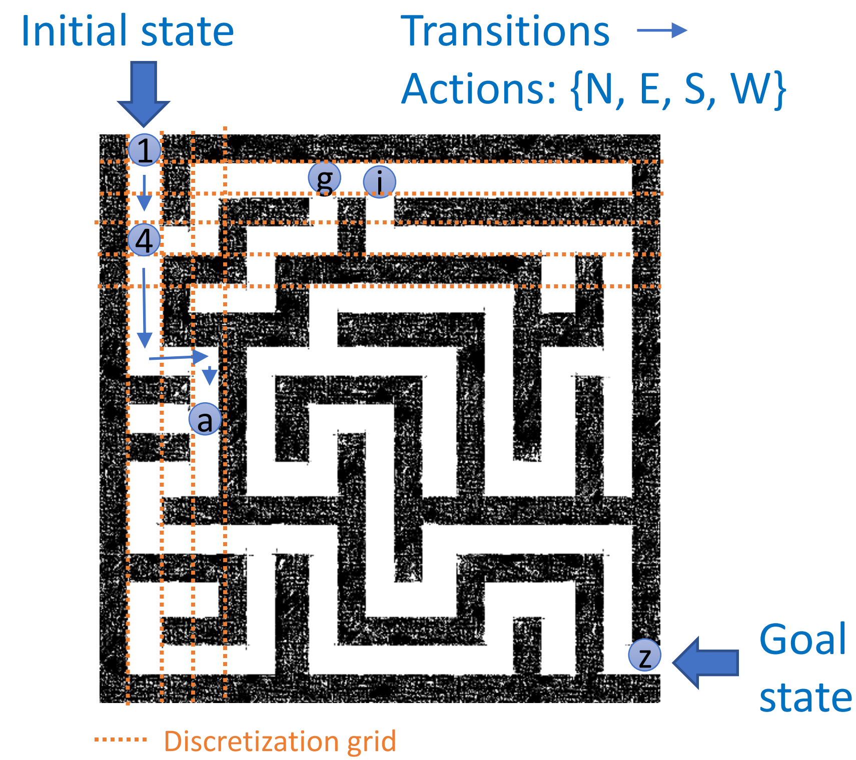


Phases:

1) **Search**: the process of looking for the **sequence of actions** that reaches a goal state.
2) **Execution**: Once the agent begins executing the search solution in a deterministic, known environment, it can ignore its percepts (**open-loop system**).

Search problem components

- **Initial state:** state description
- **Actions:** set of possible actions A
- **Transition model:** a function that defines the new state resulting from performing an action in the current state
  $f: S \times A \rightarrow S$ (S is the set of states)
- **Goal state:** state description
- **Path cost:** the sum of nonnegative *step costs*



Notes:

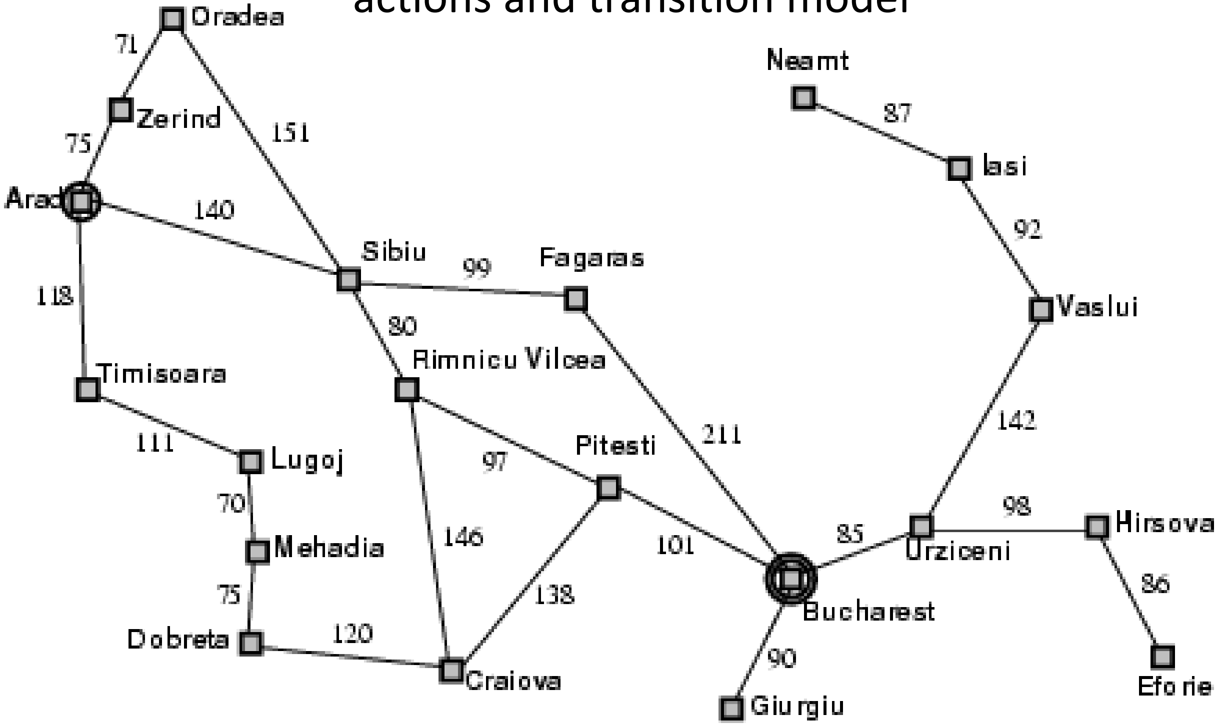
- The **state space** is typically too large to be enumerated or it is continuous. Therefore, the problem is defined by initial state, actions and the transition model and not the set of all possible states.
- The **optimal solution** is the sequence of actions (or equivalently a sequence of states) that gives the lowest path cost for reaching the goal.

Example: Romania Vacation

- On vacation in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest

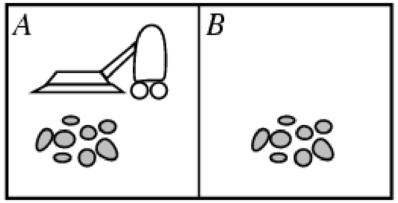
- **Initial state:** Arad
- **Actions:** Drive from one city to another.
- **Transition model and states:** If you go from city A to city B, you end up in city B.
- **Goal state:** Bucharest
- **Path cost:** Sum of edge costs.

State Space: Defined by initial state, actions and transition model

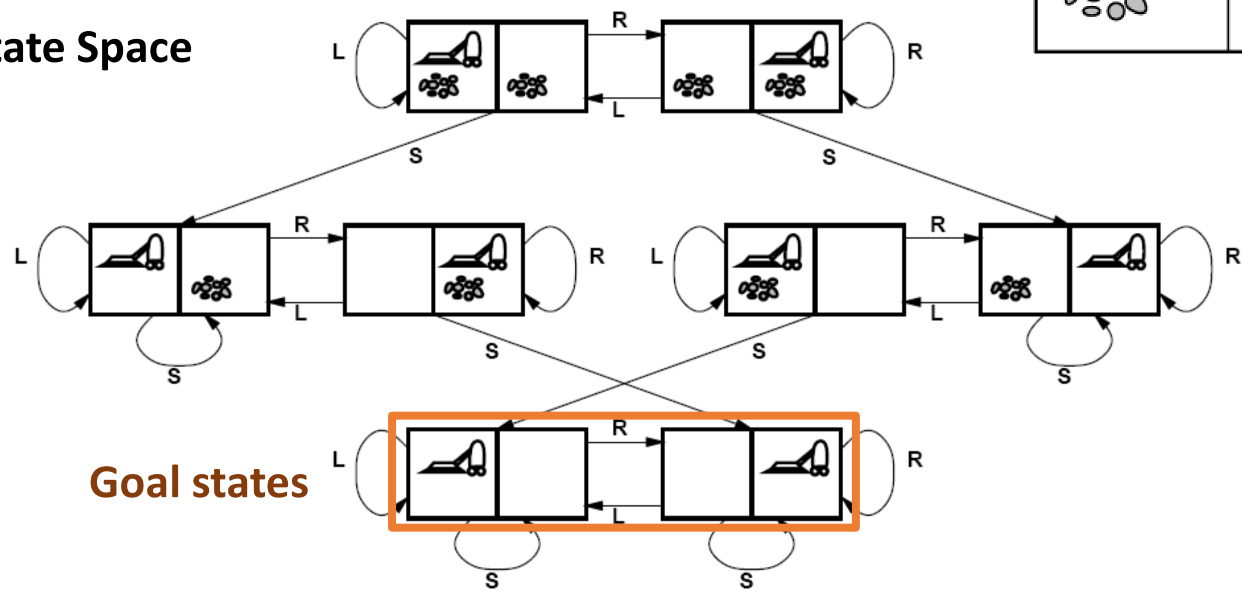


Distance in miles

Example: Vacuum world



State Space

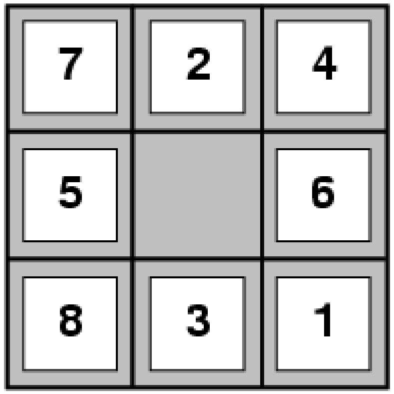


Goal states

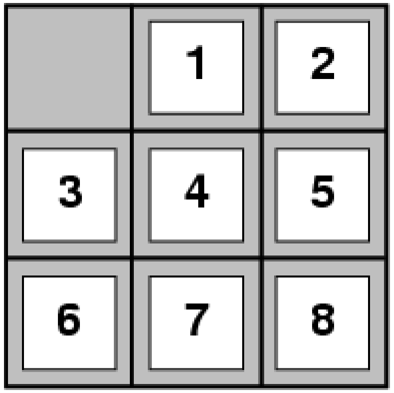
- **Initial State:** Defined by agent location and dirt location.
- **Actions:** Left, right, suck
- **Transition model and states**
  - There are 8 possible atomic states of the system.
  - Why is the number of states for *n* possible locations $n(2^n)$?
- **Goal state:** All locations are clean.
- **Path cost:** E.g., number if actions

Example: Sliding-tile puzzle

- **Initial State:** A given configuration.
- **Actions:** Move blank left, right, up, down
- **States as a result of the Initial state and the Transition model**
  - The location of each tile (including the empty one, ½ of the permutations are unreachable)
    - 8-puzzle: $9!/2 = 181{,}440$ states
    - 15-puzzle: $16!/2 \approx 10^{13}$ states
    - 24-puzzle: $25!/2 \approx 10^{25}$ states
- **Goal state:** Tiles are arranged empty and 1-8
- **Path cost:** 1 per tile move.

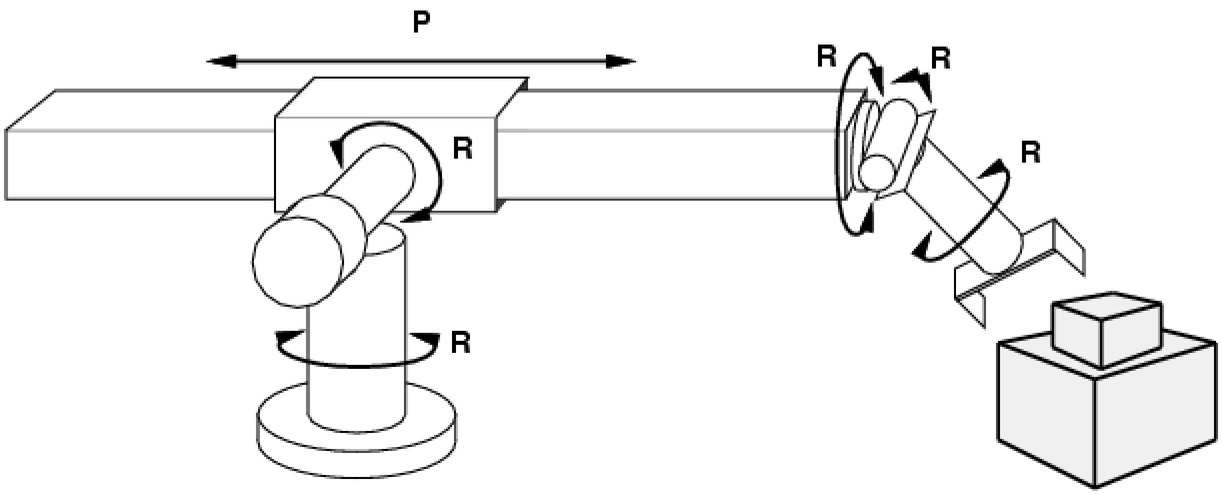


Start State



Goal State

Example: Robot motion planning



- **Initial State:** Current arm position.
- **States:** Real-valued coordinates of robot joint angles.
- **Actions:** Continuous motions of robot joints.
- **Goal state:** Desired final configuration (e.g., object is grasped).
- **Path cost:** Time to execute, smoothness of path, etc.

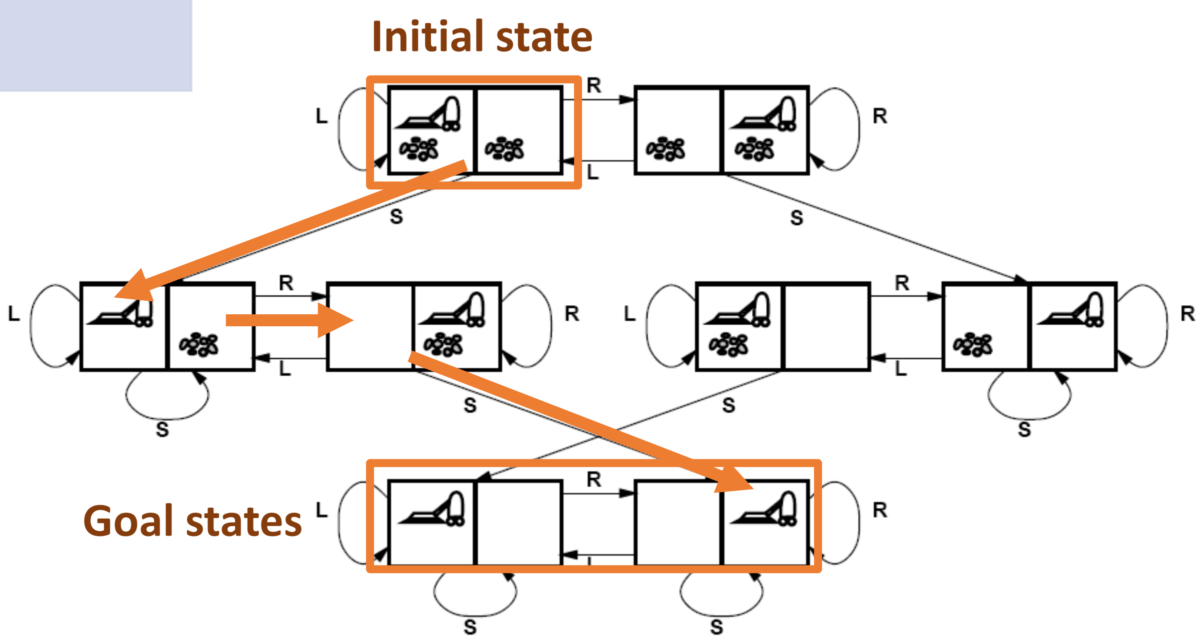
Solving search problems

Given a search problem definition

- Initial state
- Actions
- Transition model
- Goal state
- Path cost

How do we find the optimal solution (sequence of actions/states)?

State space



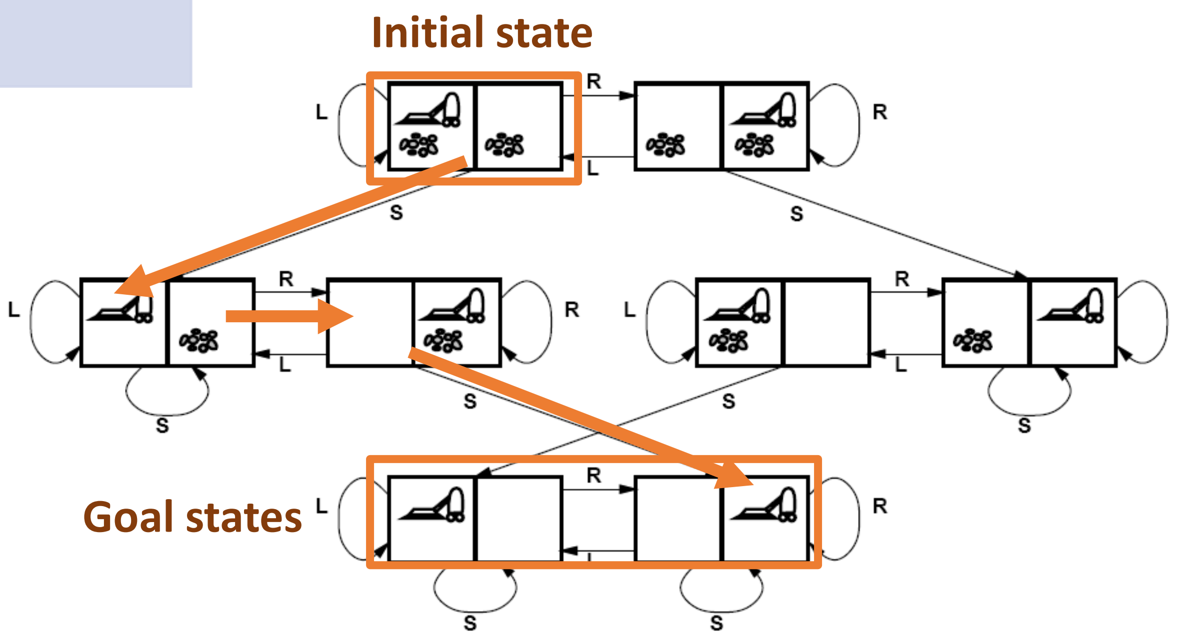
Solving search problems

Given a search problem definition

- Initial state
- Actions
- Transition model
- Goal state
- Path cost

How do we find the optimal solution (sequence of actions/states)?

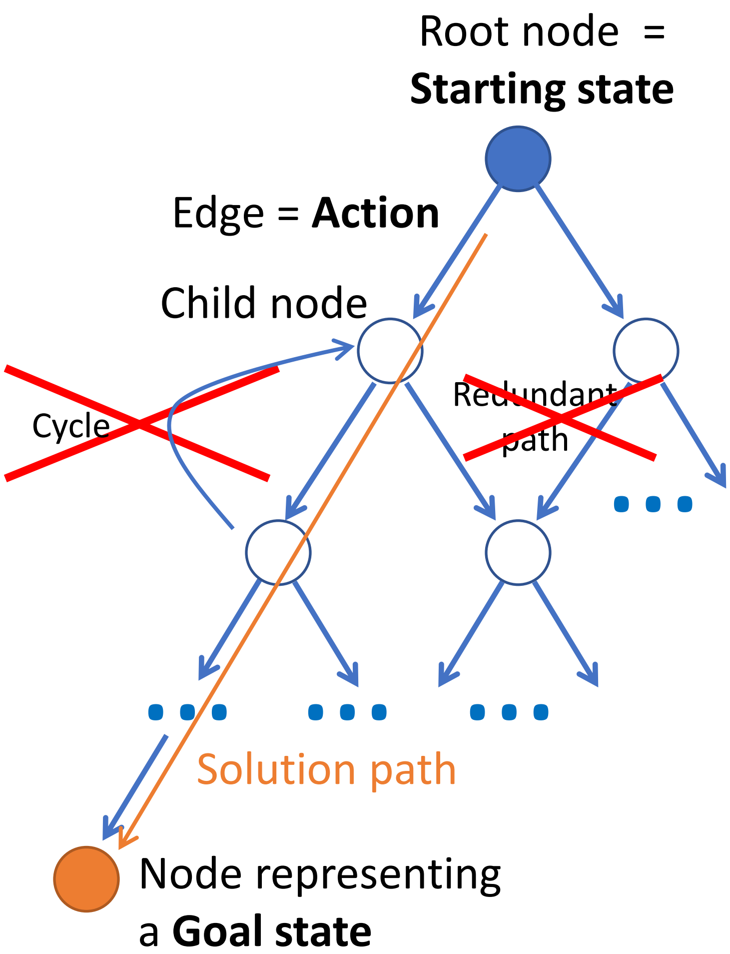
State space



Construct a search tree for the state space graph!

Search tree

- Superimpose a "what if" tree of possible actions and outcomes (states) on the state space graph.
- The **Root node** represents the starting state.
- An action child node is reached by an **edge** representing an action. The corresponding state is defined by the transition model.
- Trees have no **cycles** or **redundant paths**. Cycles in the search space need to be broken. Removing redundant paths improves search efficiency.
- A **path** through the tree corresponds to a sequence of actions (states).
- A **solution** is a path ending in a node representing a goal state.
- **Nodes vs. states**: Each node represents a state of the environment. It contains the data structure that creates the search tree.



Differences between typical Tree search and AI search

Typical tree search

- Assumes a given tree that fits in memory.
- Trees have no cycles or redundant paths.

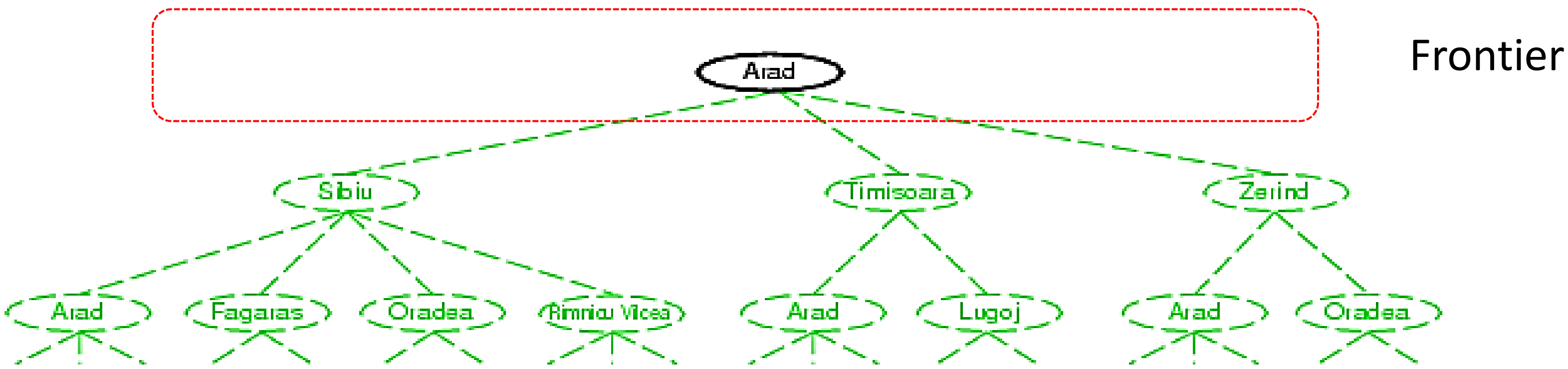
AI tree/graph search

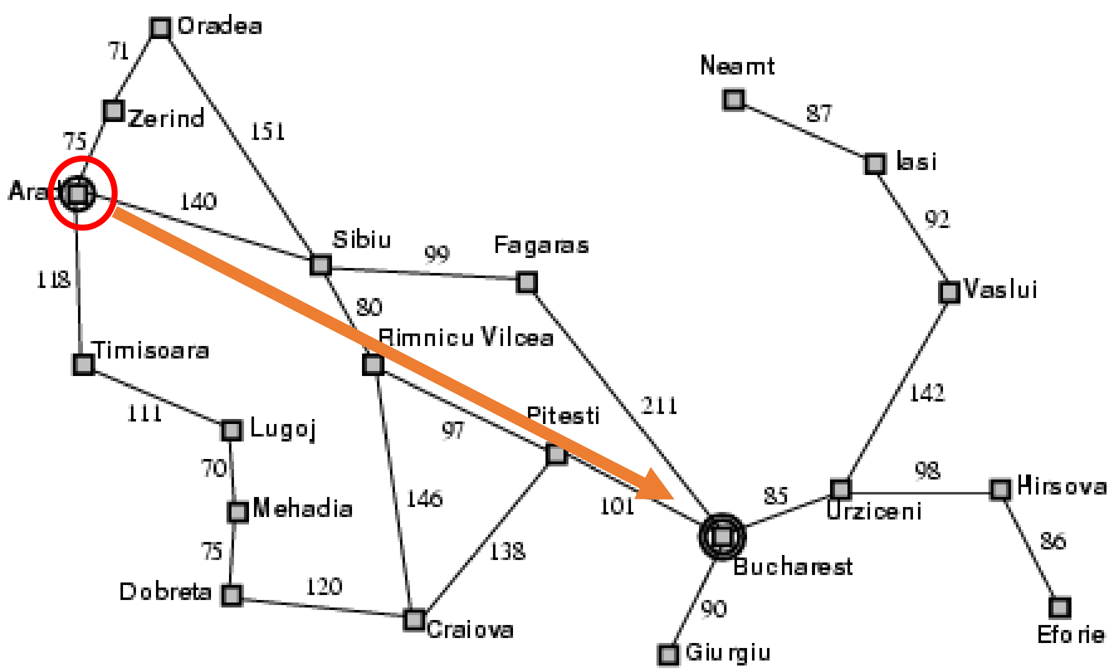
- The search space is too large to fit into **memory**.
  a. **Builds parts of the tree** from initial state and transition function.
  b. **Memory management** is very important.
- The search space is typically a complicated graph. Memory-efficient **cycle checking** is very important to avoid infinite loops or minimize searching parts of the search space multiple times.
- Redundant paths are often too memory-expensive to check.

Tree Search Algorithm Outline

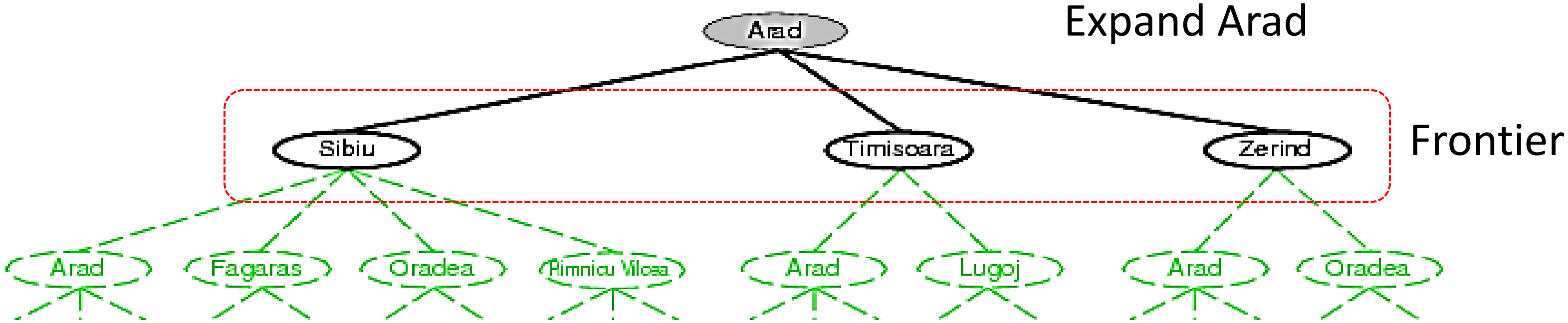
1. Initialize the **frontier** (set of unexplored know nodes) using the **starting state/root node.**
2. While the frontier is not empty:
   a) Choose a frontier node to expand according to **search strategy.**
   b) If the node represents a **goal state,** return it as the solution.
   c) Else **expand** the node (i.e., apply all possible actions to the transition model) and add its children nodes representing the newly reached states to the frontier.

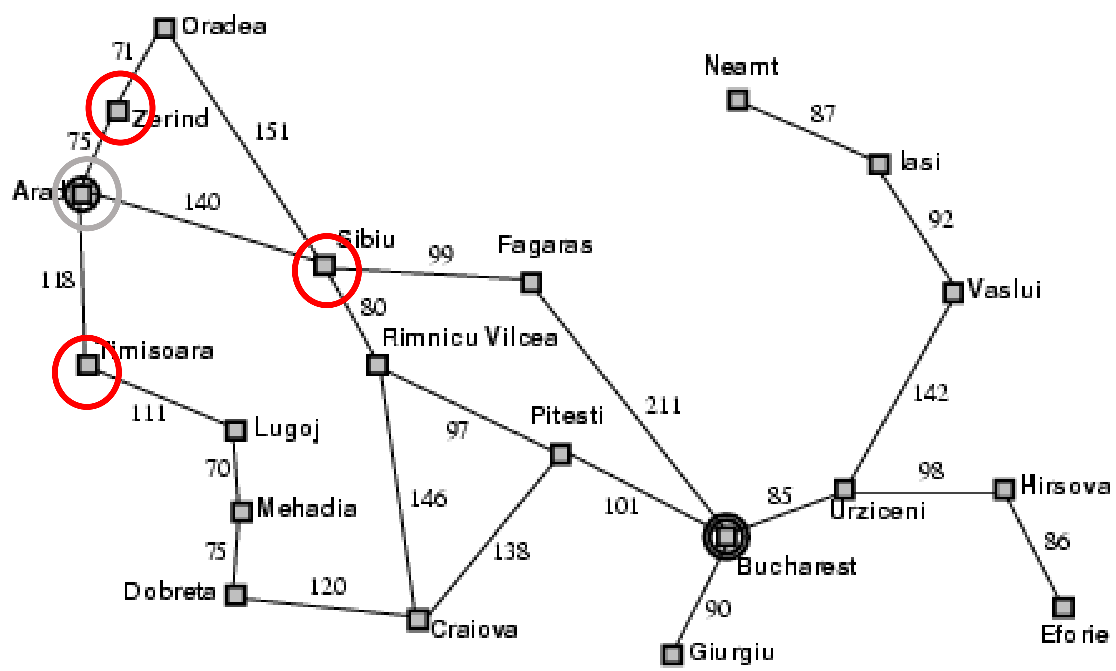
Tree search example



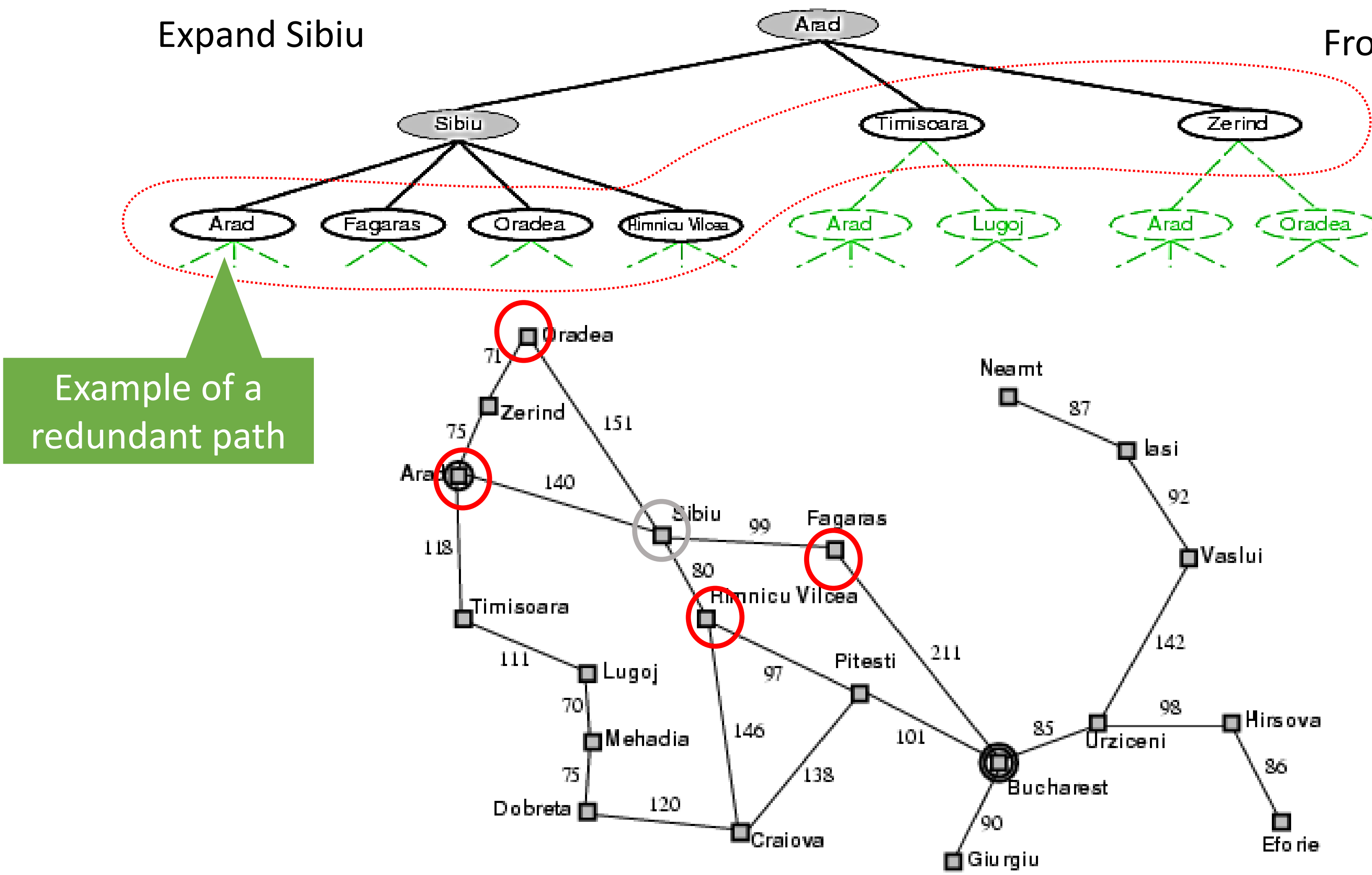


Tree search example





Tree search example



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?
  - **Optimality:** does it always find a least-cost solution?
  - **Time complexity:** how long does it take?
  - **Space complexity:** how much memory does it need?

- Worst case time and space complexity are measured in terms of the **size of the state space *n***. Metrics used if the state space is only implicitly defined by initial state, actions and a transition function are:
  - *d:* depth of the optimal solution (= number of actions needed)
  - *m:* the number of actions in any path (may be infinite with loops)
  - *b:* maximum branching factor of the search tree (number of successor nodes for a parent)

State Space for Search

State Space

- Number of different states the agent and environment can be in.
- **Reachable states** are defined by the initial state and the transition model.
- **Search tree** spans the state space. Note that a single state can be represented by several search tree nodes.
- State space size is an indication of problem size.

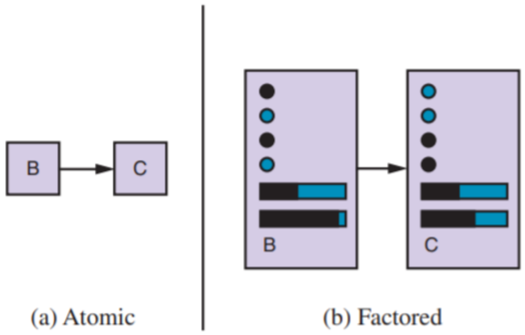
- Even if the used algorithm represents the state space using atomic states, we may know that internally they are factored.

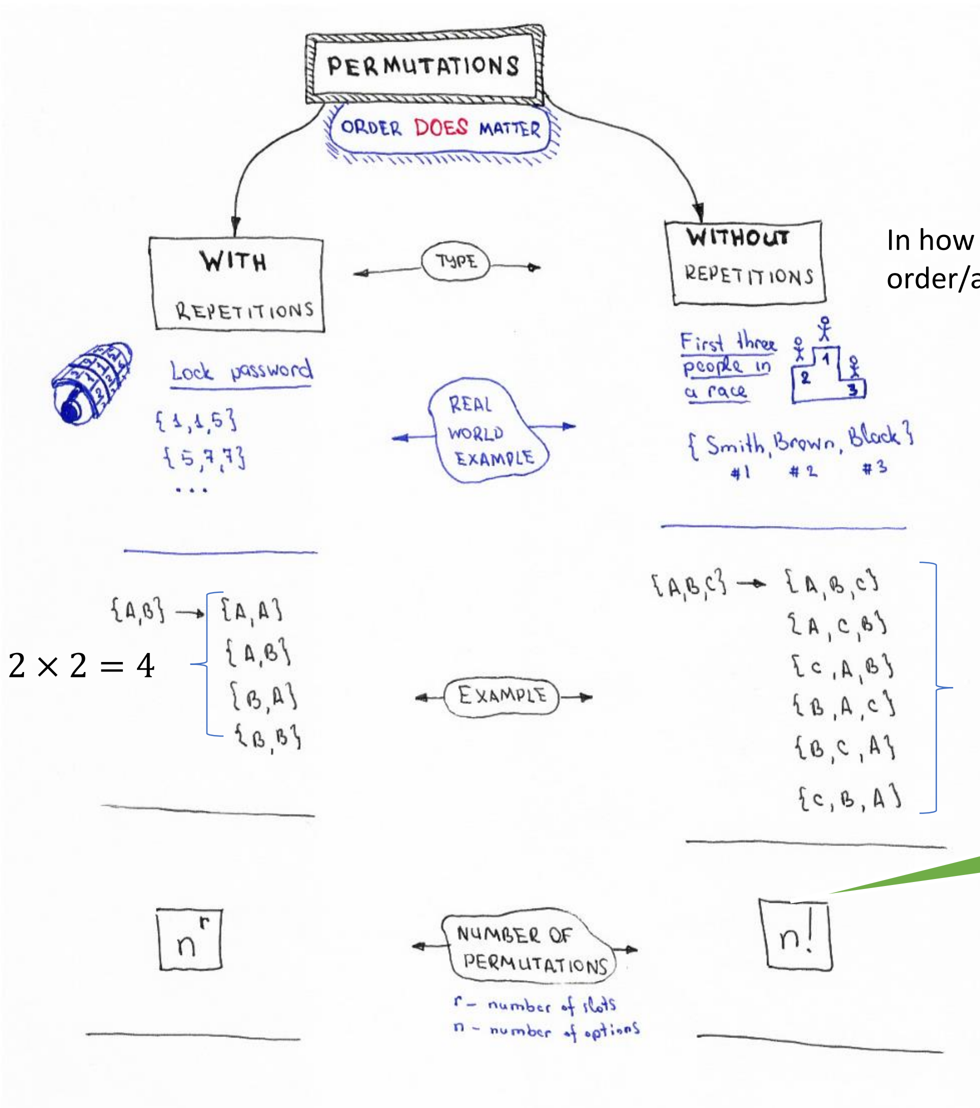
- Basic rule to calculate (estimate) the state space size for factored state representation with n variables is:

$$|x_1| \times |x_2| \times \cdots \times \ |x_n|$$

  where $|\cdot|$ is the number of possible values.

State representation





$2 \times 2 = 4$

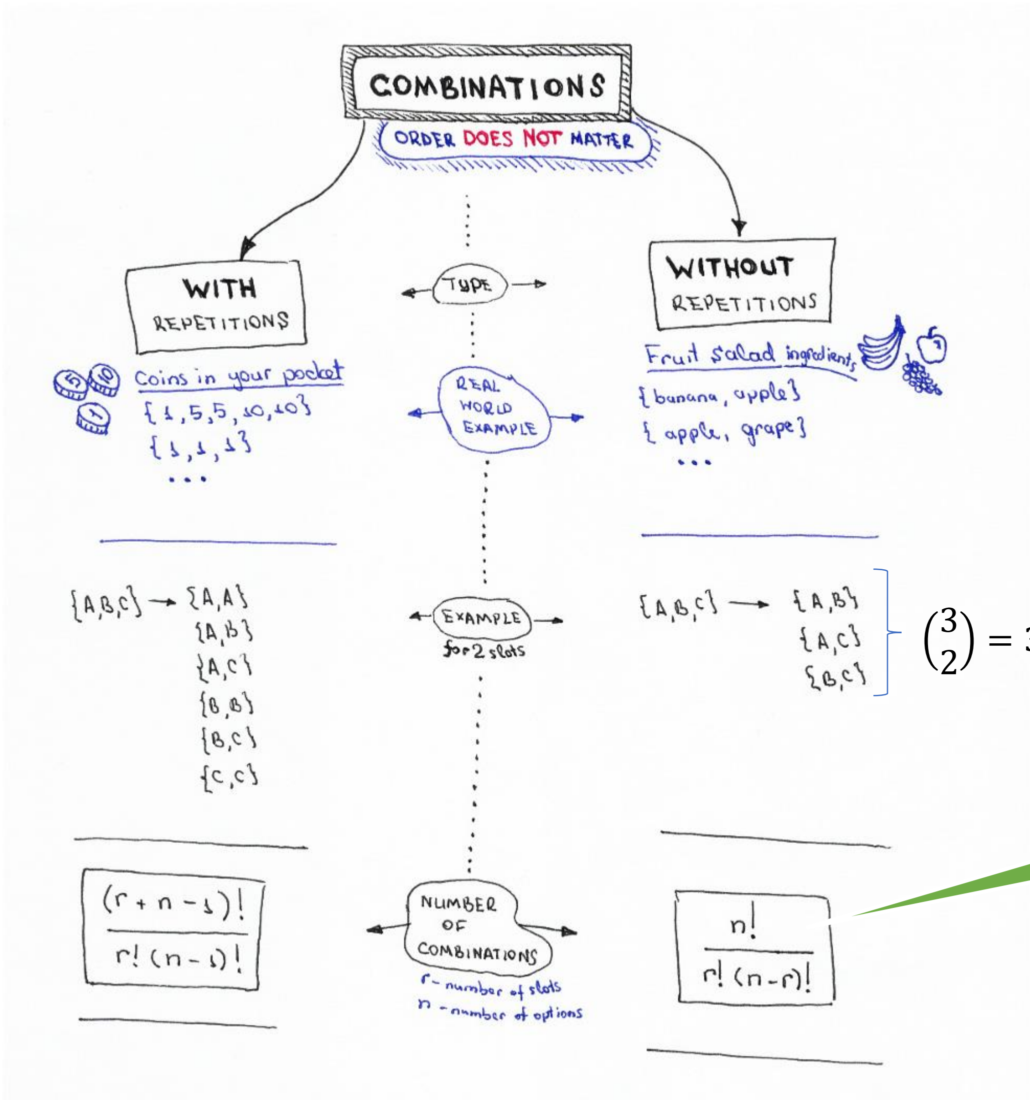
In how many ways can we order/arrange n objects?

$3 \times 2 \times 1 = 6$

Factorial: $n! = n \times (n-1) \times \cdots \times 2 \times 1$

```
#Python
import math

print (math.factorial(23))
```

Source: Permutations/Combinations Cheat Sheets by Oleksii Trekhleb
https://itnext.io/permutations-combinations-algorithms-cheat-sheet-68c14879aba5



$$\binom{3}{2} = 3$$

Binomial Coefficient: $\binom{n}{r} = C(n,r) = {}_nC_r$

In how many ways can we choose r out of n objects?

Special case for $r = 2$: $\binom{n}{2} = \frac{n(n-1)}{2}$

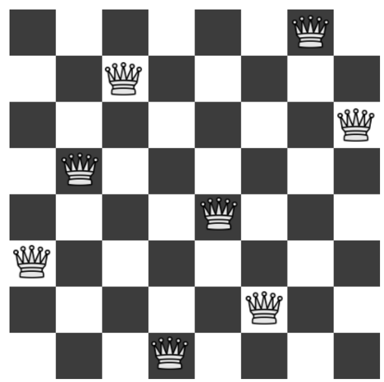
```
#Python
import scipy.special

# the two give the same results
scipy.special.binom(10, 5)
scipy.special.comb(10, 5)
```

Source: Permutations/Combinations Cheat Sheets by Oleksii Trekhleb
https://itnext.io/permutations-combinations-algorithms-cheat-sheet-68c14879aba5

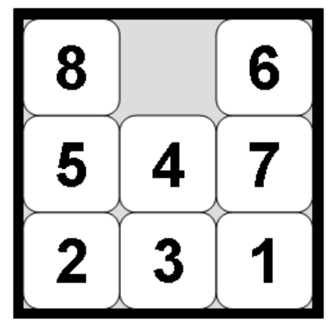
Examples: What is the state space size?

Often a rough upper limit is sufficient to determine how hard the search problem is.

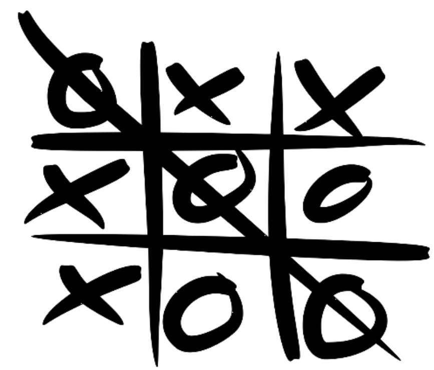
Maze



8-quees problem



8-puzzle problem



Tic-tac-toe

Uninformed Search

Uninformed search strategies

The search algorithm/agent is **not** provided information about how close a state is to the goal state.

It blindly searches until it finds the goal state by chance.

Algorithms:

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Iterative deepening search

Breadth-first search (BFS)

- **Expansion rule:** Expand shallowest unexpanded node in the frontier (first added).

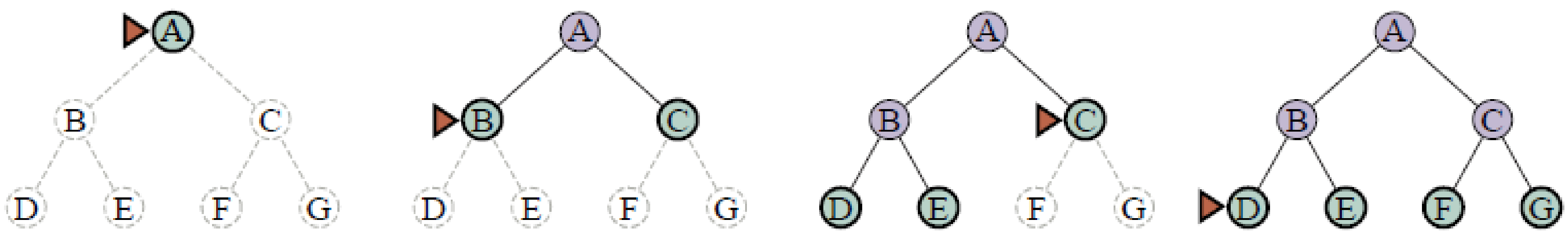


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

- **Frontier** data structure: holds references to the green nodes (green) and is implemented as a FIFO queue.
- **Reached** data structure: holds references to all visited nodes (gray and green) and is used to prevent visiting nodes more than once (cycle checking).

Implementation: 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
```

reached makes sure we do not visit nodes twice (e.g., in a loop). Fast lookup is important.

```
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)
```

Transition function

Node structure: Yield can also be implemented by returning a list of Nodes.

Properties of breadth-first search

d: depth of the optimal solution
m: max. depth of tree
b: maximum branching factor

- **Complete?**
  Yes
- **Optimal?**
  Yes – if cost is the same per step (action). Otherwise: Use uniform-cost search.
- **Time?**
  Number of nodes created in a *b*-ary tree of depth *d* (depth of optimal solution):
  $$1 + b + b^2 + \cdots + b^d = O(bd)$$
- **Space?**
  Stored nodes: $O(b^d)$

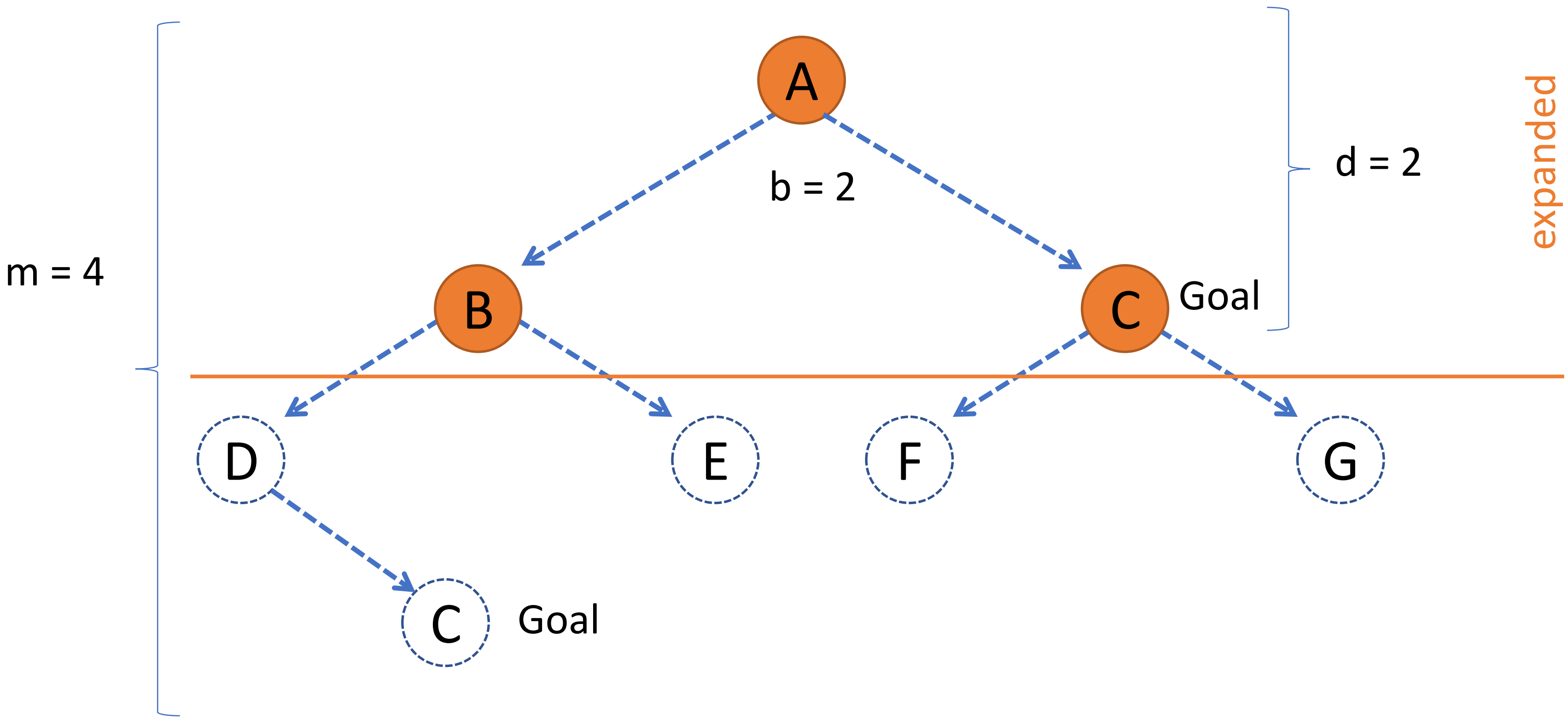
Notes:

- The state space is typically very large: $O(b^d) << O(n)$
- Space is usually a bigger problem than time!

Breadth-first search

d: depth of the optimal solution
m: max. depth of tree
b: maximum branching factor

- Time and Space: $O(b^d)$ - all paths to the depth of the goal are expanded



Uniform-cost search (= Dijkstra's shortest path algorithm)

- A **best-first search strategy**: Expand node with the least path cost from the initial state in frontier
- Implementation: the frontier is a priority queue ordered by $f(n)$ = **path cost**
- Breadth-first search is a special case when all step costs all equal, i.e., each action costs the same!

- **Complete?**
  Yes, if all step cost is greater than some small positive constant $\varepsilon > 0$

- **Optimal?**
  Yes – nodes expanded in increasing order of path cost

- **Time?**
  Number of nodes with path cost ≤ cost of optimal solution (C^*) is $O(b^{1+C^*/\varepsilon})$.
  This can be greater than $O(b^d)$: the search can explore long paths consisting of small steps before exploring shorter paths consisting of larger steps

- **Space?**
  $O(b^{1+C^*/\varepsilon})$

See Dijkstra's algorithm on Wikipedia

d: depth of the optimal solution
m: max. depth of tree
b: maximum branching factor

Implementation: Best-First 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)
```

The order for expanding the frontier is determined by *f(n)* = path cost to node *n*.

This check is the different to BFS! It visits a node again if it can be reached by a better (cheaper) path.

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

Depth-first search (DFS)

- **Expansion rule**: Expand deepest unexpanded node in the frontier (last added).
- **Frontier**: stack (LIFO)
- **Reached**: No reached data structure! Cycle checking can only check the current path and the frontier. This may lead to infinite loops!

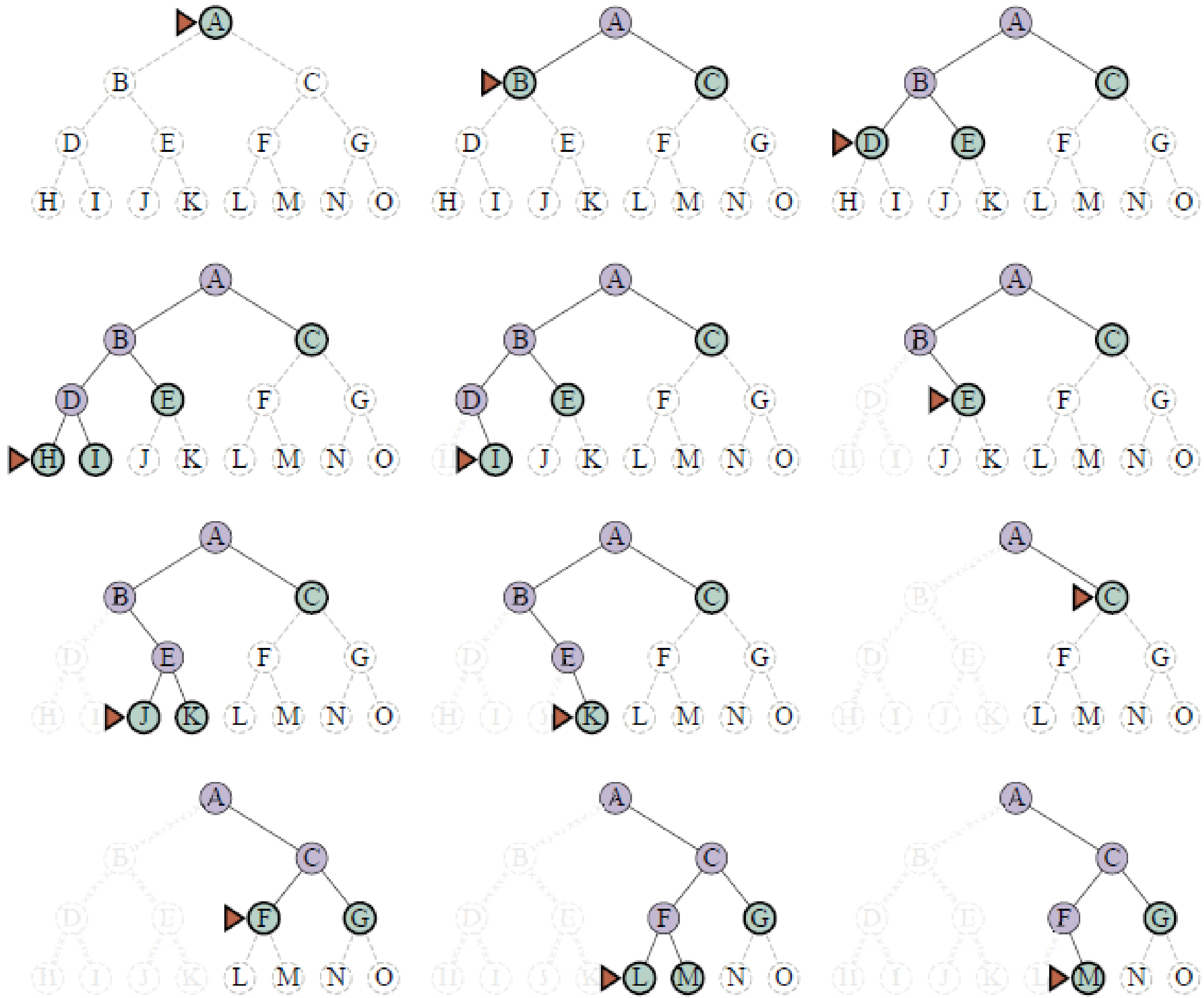


Figure 3.11 A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint dashed lines. Expanded nodes with no descendants in the frontier (very faint lines) can be discarded.

Implementation: DFS

- DFS could be implemented like BFS/Best-first search and just taking the last element from the frontier (LIFO).
- However, to reduce the space complexity to *O(bm),* the reached data structure needs to be removed! Options:
  - Recursive implementation (cycle checking is a problem).
  - We can use a search tree where the abandoned branches are removed from memory (cycle checking is only done against the current path). This is similar to Backtracking search.

DFS-SEARCH

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*

If we only keep the current branch in memory, then we can only check against the path from the root to the current node to prevent cycles.

Properties of depth-first search

- **Complete?**
  - Only in finite search spaces. Some cycles can be avoided by checking for repeated states along the path.
  - **Incomplete in infinite search spaces** (e.g., with cycles).
- **Optimal?**
  No – returns the first solution it finds.

d: depth of the optimal solution
m: max. depth of tree
b: maximum branching factor

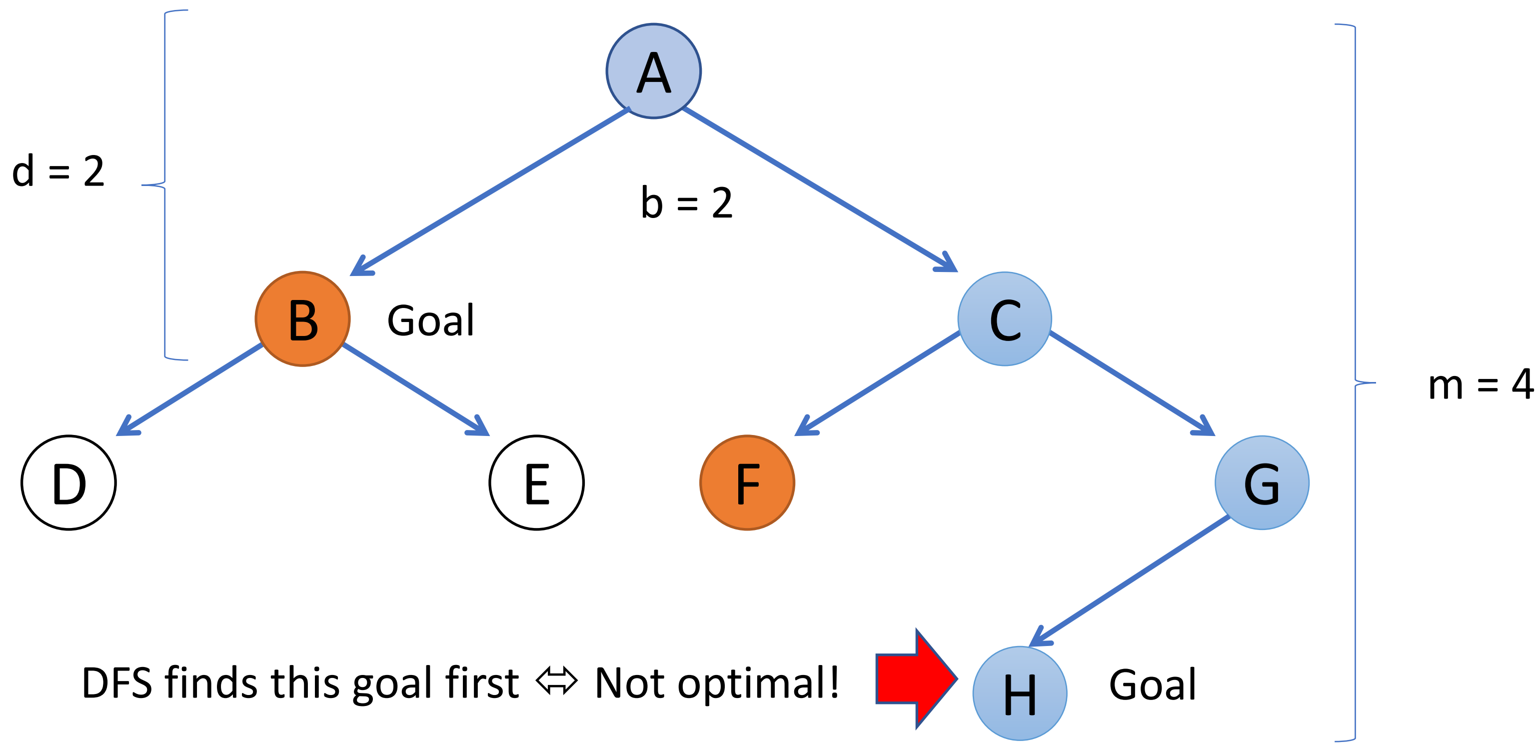
- **Time?**
  Could be the time to reach a solution at maximum depth m in the last path: $O(b^m)$
  Terrible if $m \gg d$, but if there are many shallow solutions, it can be much faster than BFS.
- **Space?**
  $O(bm)$ ⇔ **linear space! (if the reached data structure is not stored)**

This makes DFS into the workhorse of AI.

Depth-first search

d: depth of the optimal solution
m: max. depth of tree
b: maximum branching factor

- Time: $O(b^m)$ – worst case is expanding all paths.
- Space: $O(bm)$ - if it only stores the frontier nodes and the current path.



Note: The order in which we add new nodes to the frontier can change what goal we find!

Iterative deepening search (IDS)

Can we

a) get DFS's good memory footprint,

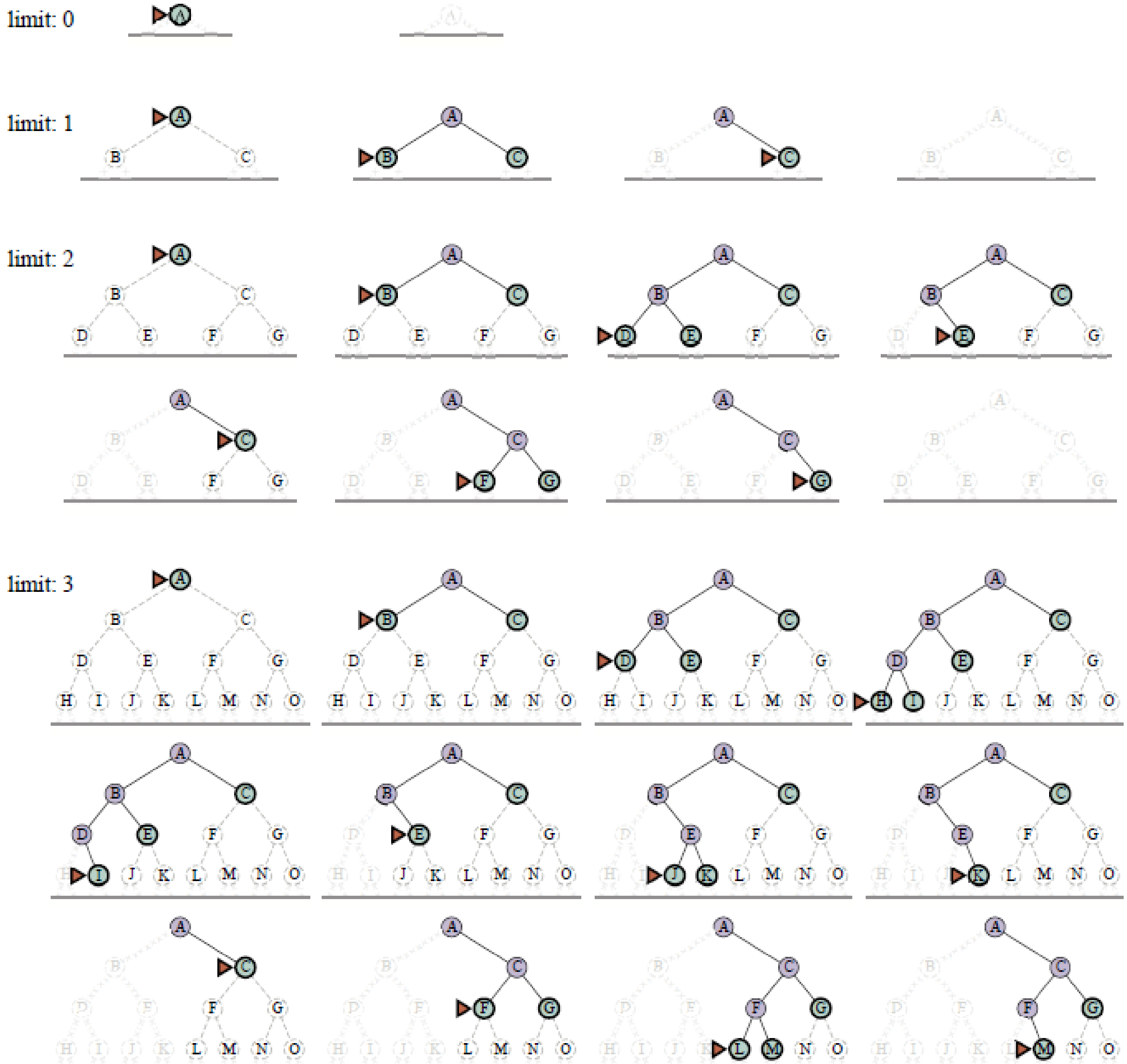
b) avoid infinite cycles, and

c) preserve BFS's optimality guaranty?

Use depth-restricted DFS and gradually increase the depth.

1. Check if the root node is the goal.
2. Do a DFS searching for a path of length 1
3. If there is no path to the goal of length 1, do a DFS searching for a path of length 2
4. If there is no path of length 2, do a DFS searching for a path of length 3
5. …

Iterative deepening search (IDS)



Implementation: IDS

```
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
```

```
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
```

Properties of iterative deepening search

- **Complete?**
  Yes

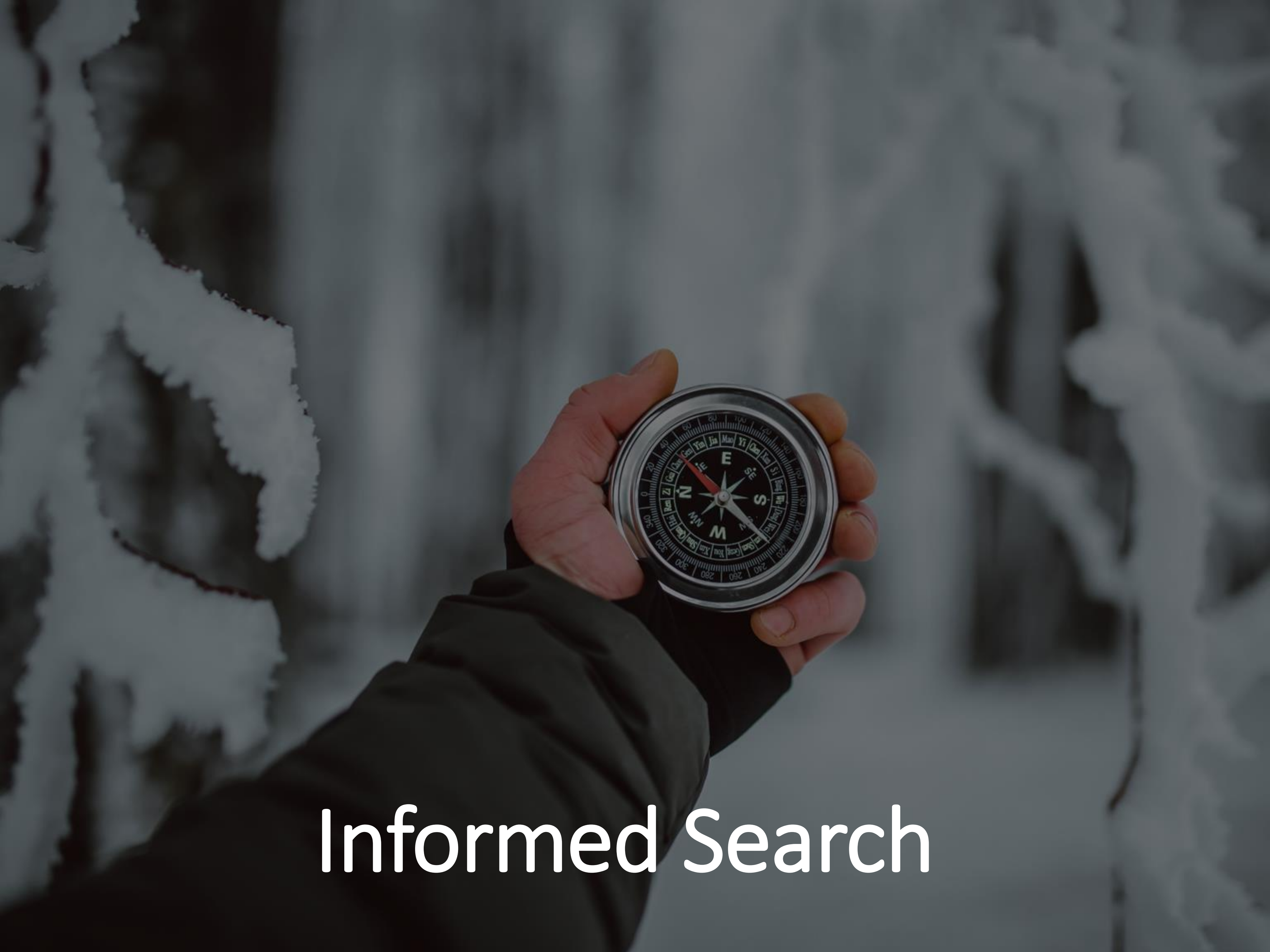
- **Optimal?**
  Yes, if step cost = 1

- **Time?**
  Consists of rebuilding trees up to d times
  $d\ b^1 + (d-1)b^2 + \ldots + bd = O(bd)$ ⇔ Slower than BFS, but the same complexity!

- **Space?**
  ***O*(*bd*) ⇔ linear space. Even less than DFS since m<=d.** Cycles need to be handled by the depth-limited DFS implementation.

d: depth of the optimal solution
m: max. depth of tree
b: maximum branching factor

Note: IDS produces the same result as BFS but trades better space complexity for worse run time.



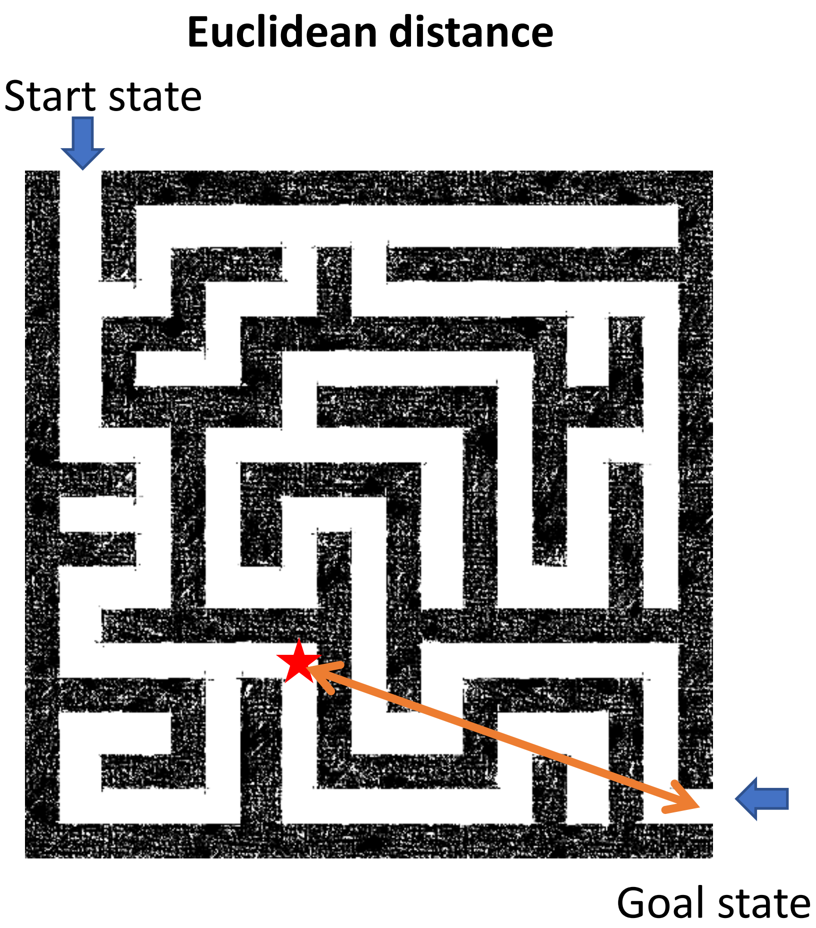
Informed Search

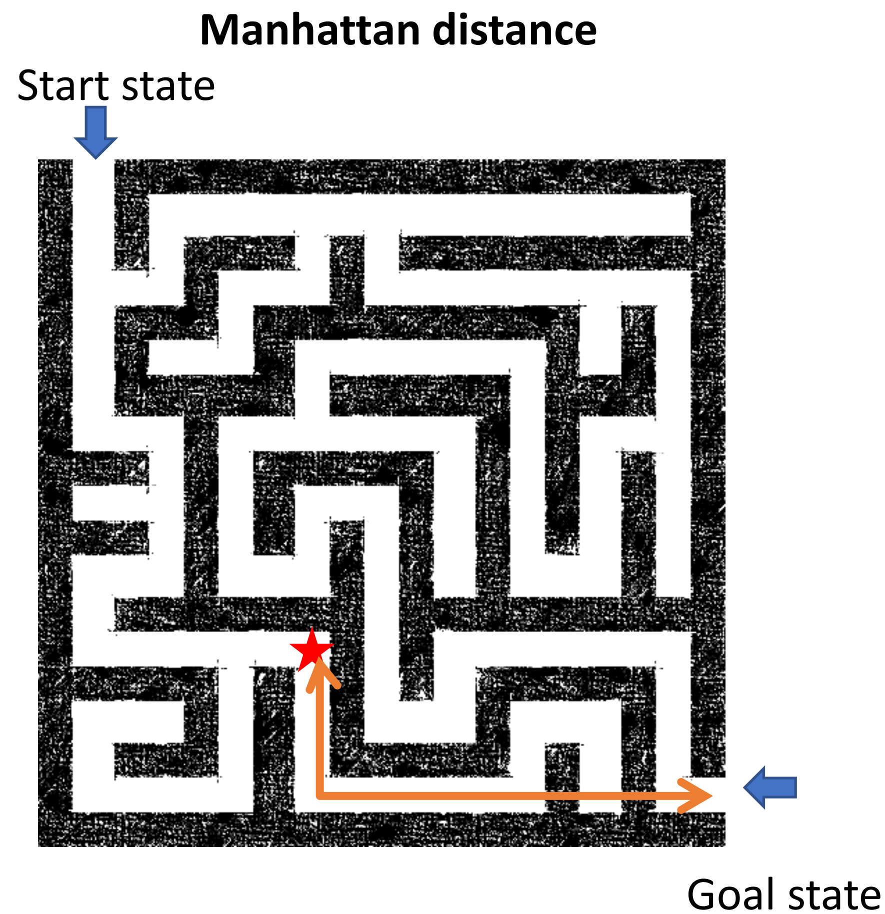
Informed search

- AI search problems are typically very large. We would like to improve efficiency by **expanding as few nodes as possible.**
- The agent can use **additional information** in the form of “hints” about how promising different states/nodes are to lead to the goal. These hints are derived from
  - information the agent has (e.g., a map) or
  - percepts coming from a sensor.
- The agent uses a **heuristic function $f(n)$** to rank nodes in the frontier and select the most promising state in the frontier for expansion using a **best-first search** strategy.
- Algorithms:
  - Greedy best-first search
  - A* search

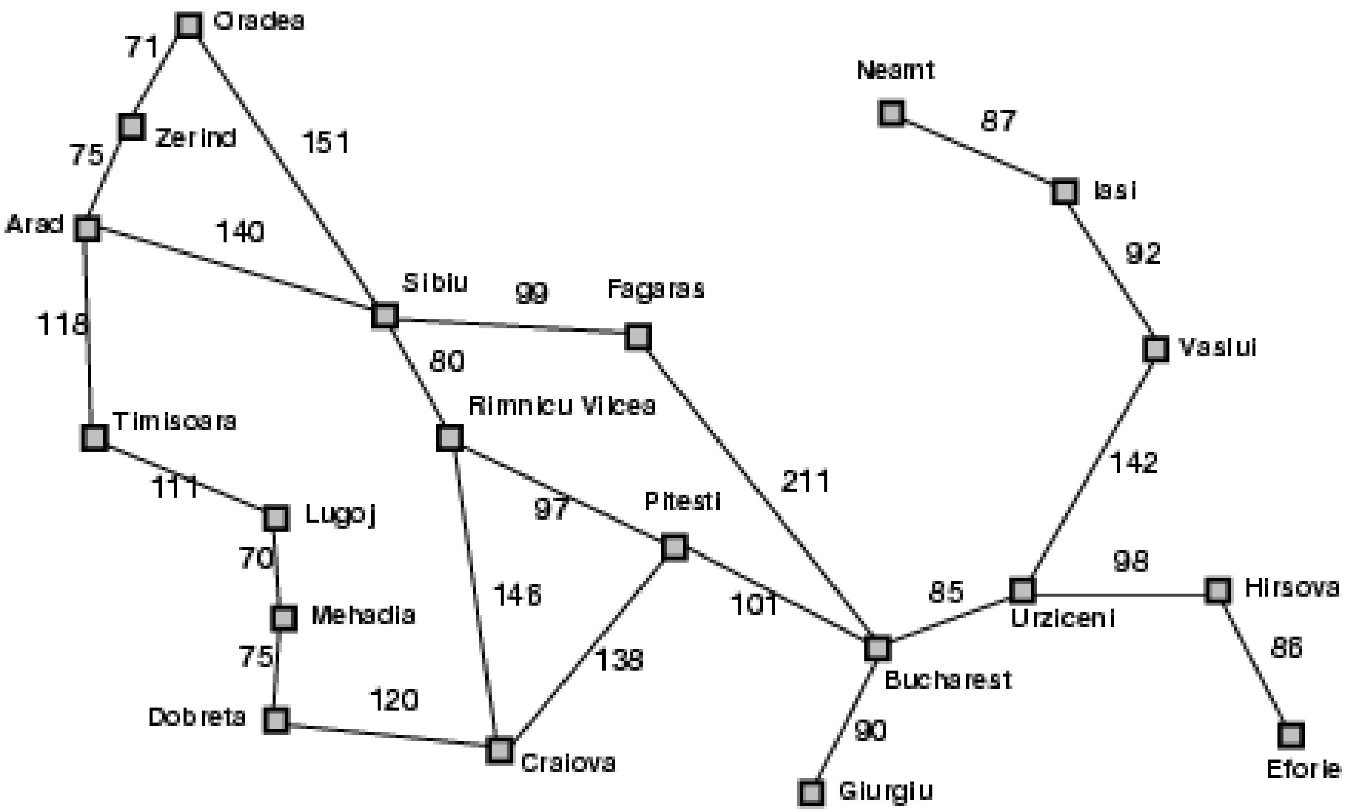
Heuristic function

- **Heuristic function** $h(n)$ estimates the cost of reaching a node representing the goal state from the current node n.
- Examples:





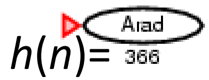
Heuristic for the Romania problem

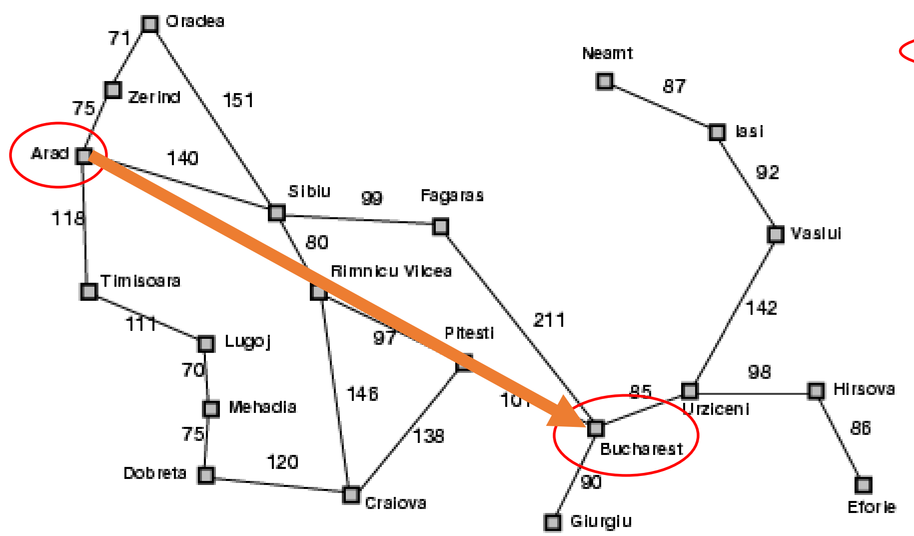


Straight–line distance to Bucharest	$h(n)$
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example

Expansion rule: Expand the node that has the lowest value of the heuristic function *h*(*n*)

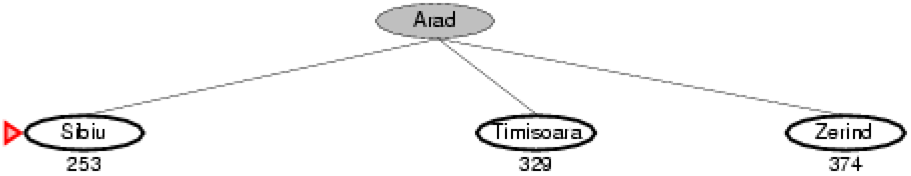


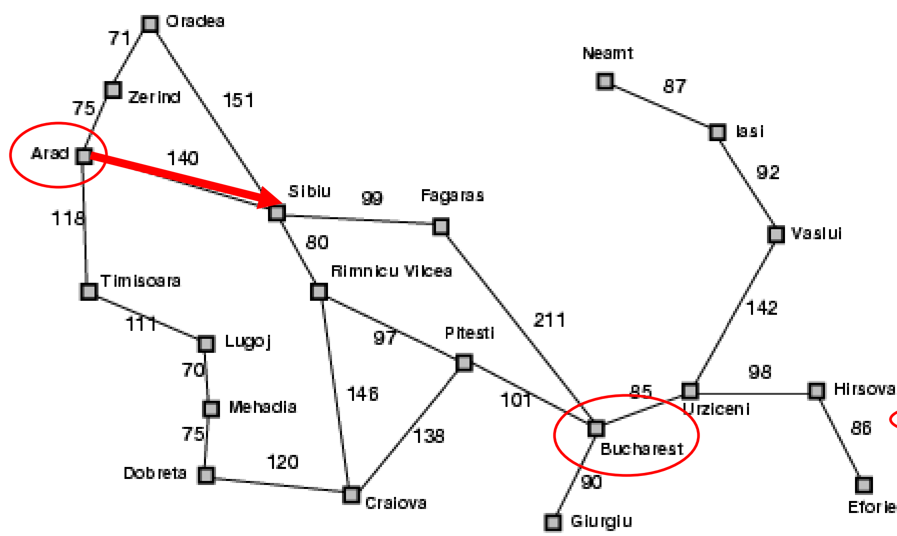


Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

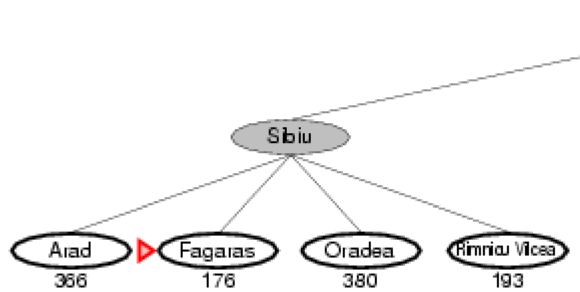
Greedy best-first search example

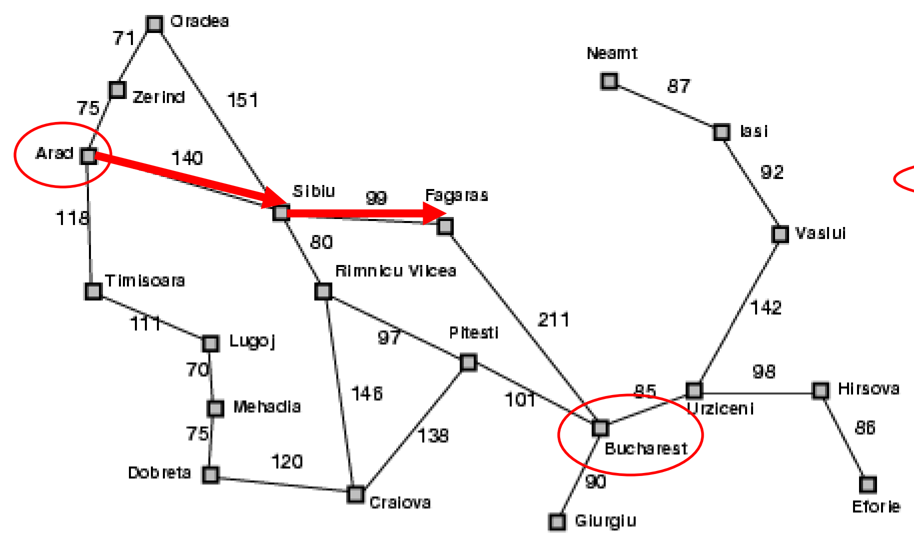




Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example

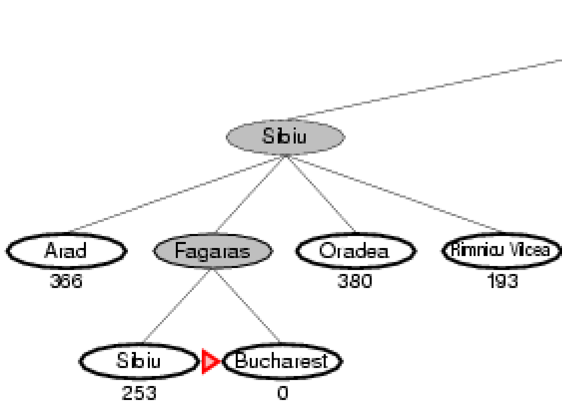




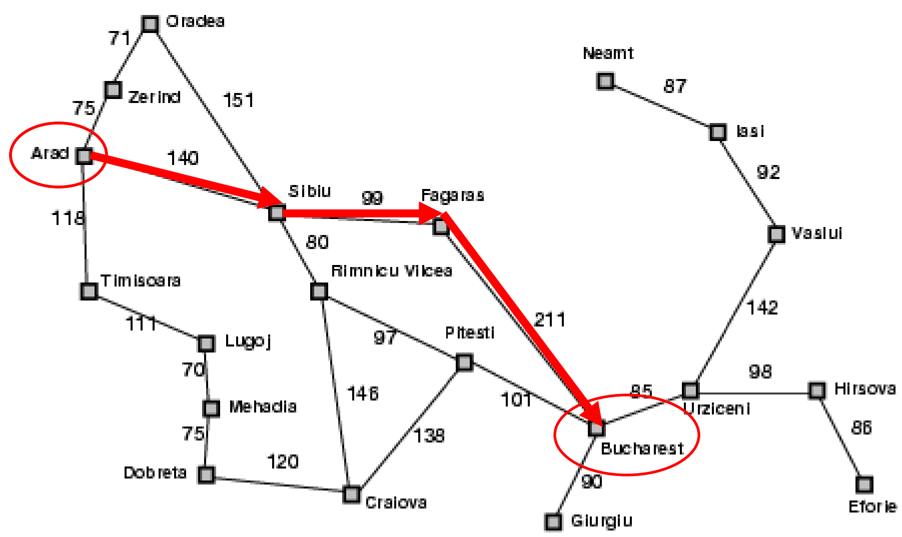
Straight–line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example



Total:
140 + 99 + 211 = 450 miles



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

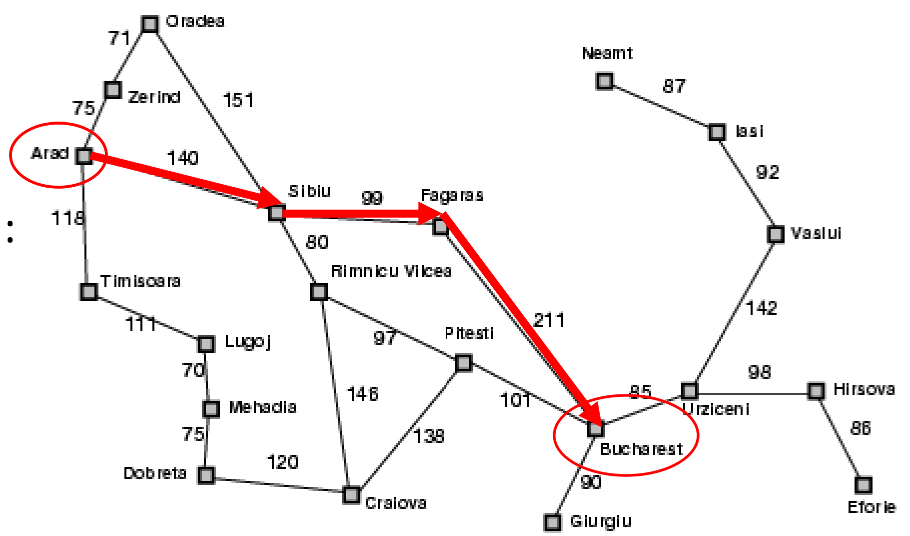
Properties of greedy best-first search

- **Complete?**
  Yes – Best-first search if complete in finite spaces.

- **Optimal?**
  No

Total:
140 + 99 + 211 = 450 miles

Alternative through Rimnicu Vilcea:
140 + 80 + 97 + 101 = 418 miles



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Implementation of greedy best-first search

Best-First Search*

+

Expand the frontier using
$f(n) = h(n)$

* See Uniform-cost search for the pseudo code.

Properties of greedy best-first search

- **Complete?**
  Yes – Best-first search if complete in finite spaces.

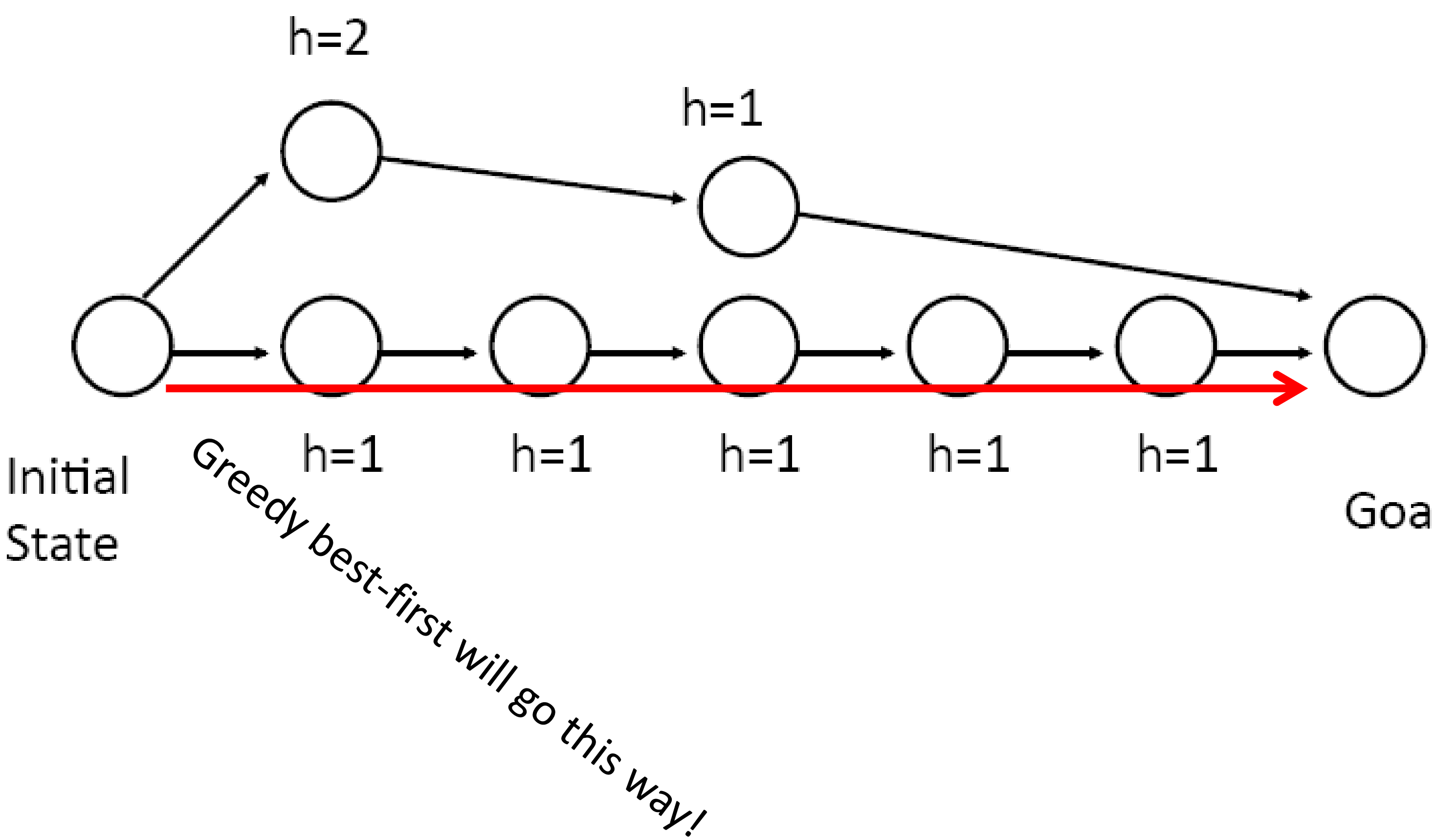
- **Optimal?**
  No

d: depth of the optimal solution
m: max. depth of tree
b: maximum branching factor

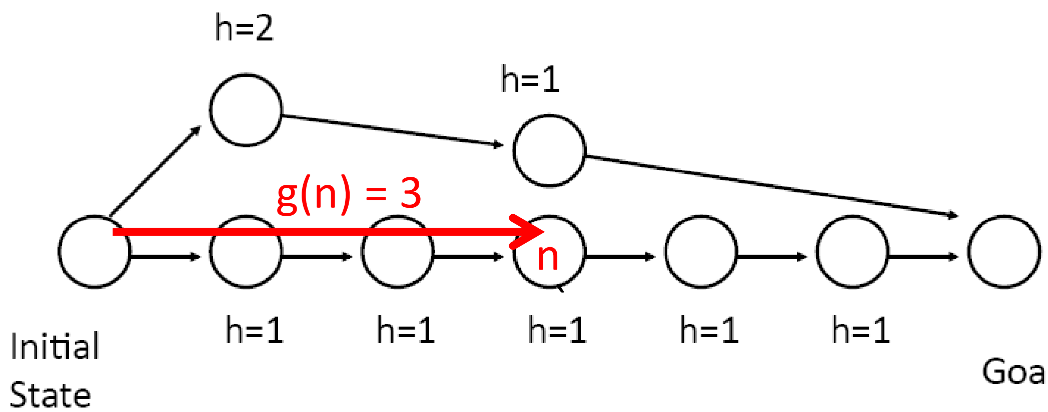
- **Time?**
  Worst case: $O(b^m)$ ⇔ like DFS
  Best case: $O(bm)$ – If $h(n)$ is 100% accurate

- **Space?**
  Same as time complexity.

How can we fix the optimality problem with greedy best-first search?



A* search



- **Idea**: Take current path cost into account and avoid further expanding paths that are already very expensive.
- The evaluation function $f(n)$ is the estimated total cost of the path through node n to the goal:

$$f(n) = g(n) + h(n)$$

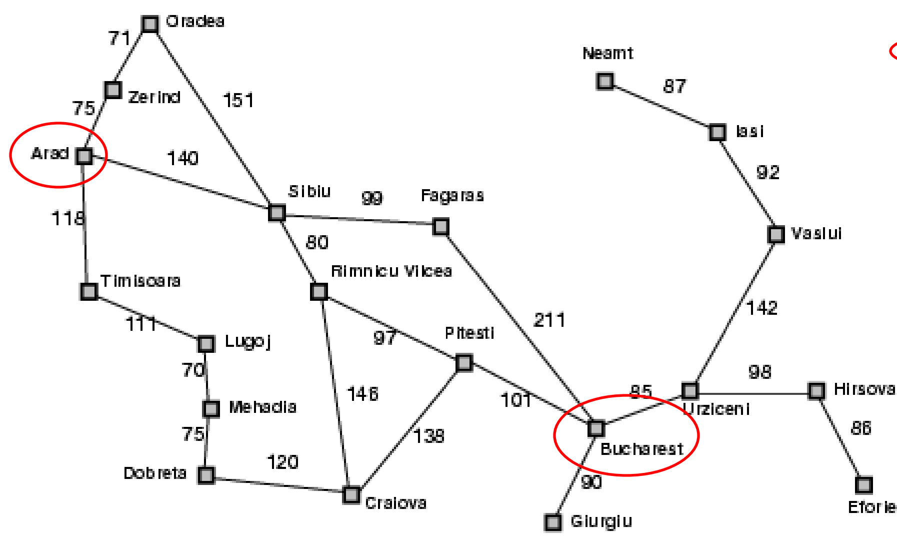
$g(n)$: cost so far to reach n (path cost)
$h(n)$: estimated cost from n to goal (heuristic)

Note: For greedy best-first search we just use $f(n) = h(n)$.

A* search example

Expansion rule:
Expand the node with the smallest f(n)

$f(n) = g(n) + h(n) =$ 

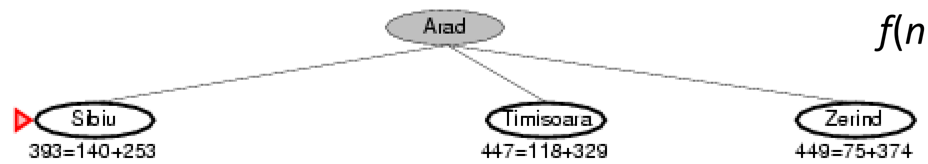


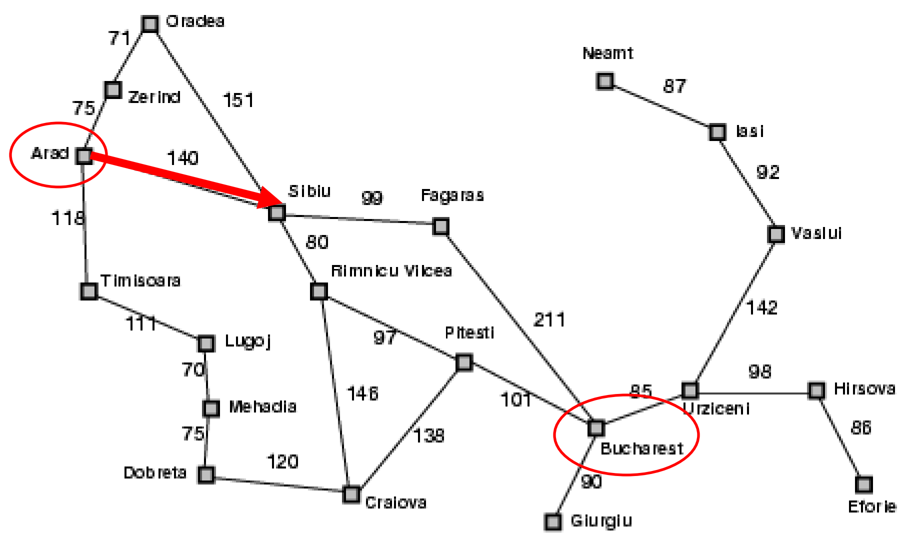
Straight-line distance to Bucharest

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

A* search example

$f(n) = g(n) + h(n)$

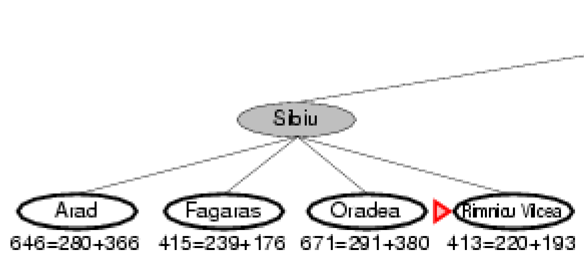




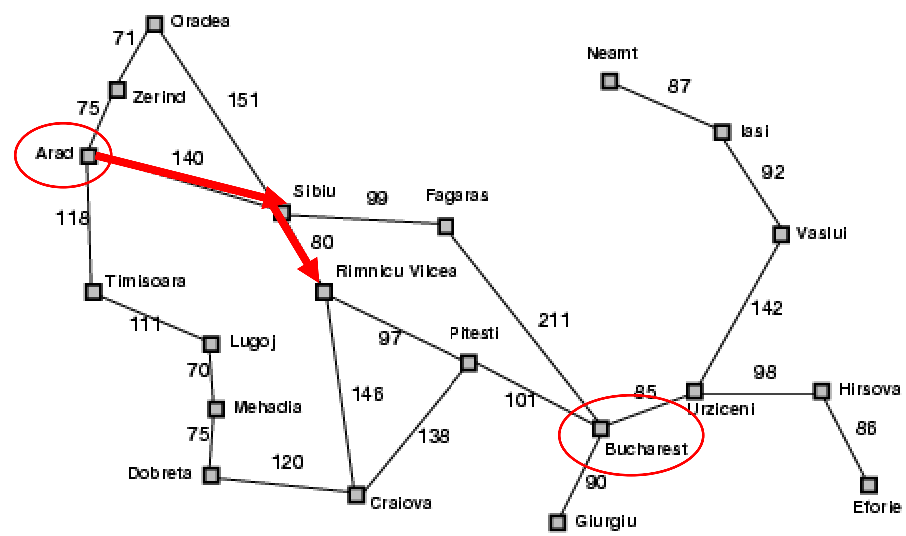
Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example



$f(n) = g(n) + h(n)$

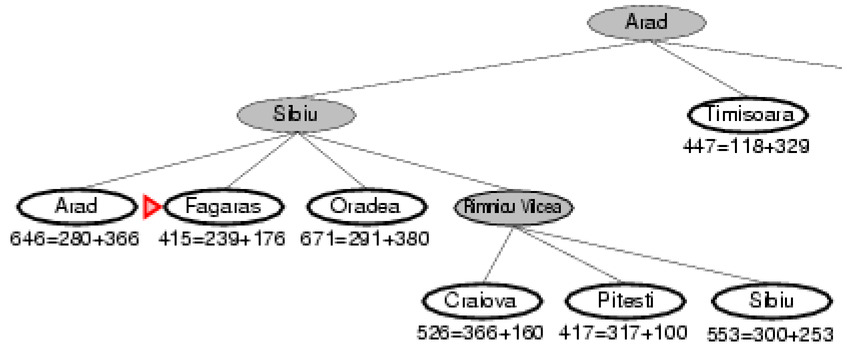


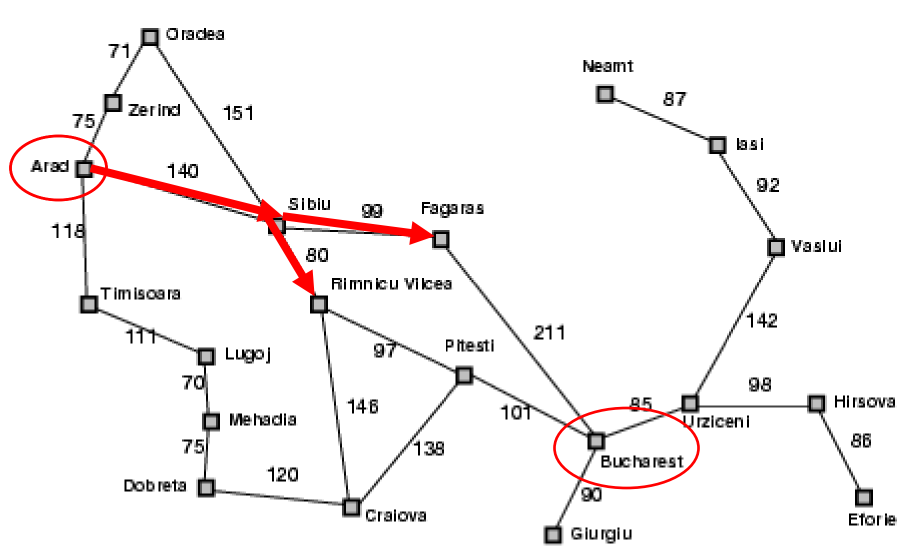
Straight–line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example

$f(n) = g(n) + h(n)$

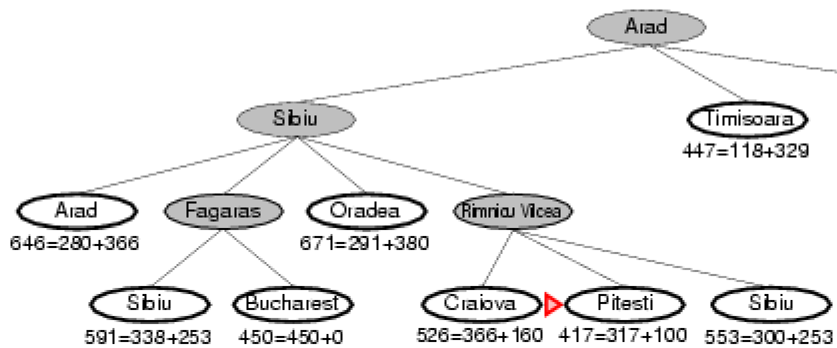


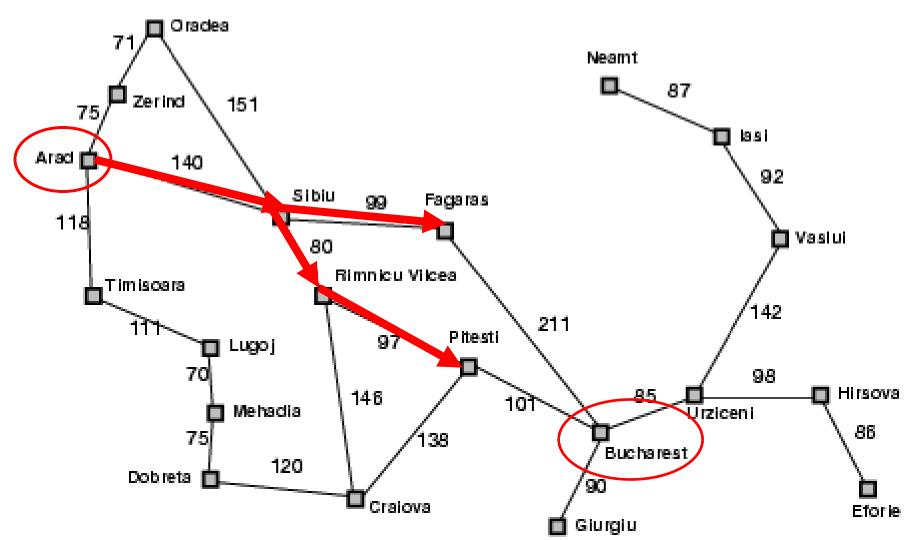


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example

$f(n) = g(n) + h(n)$

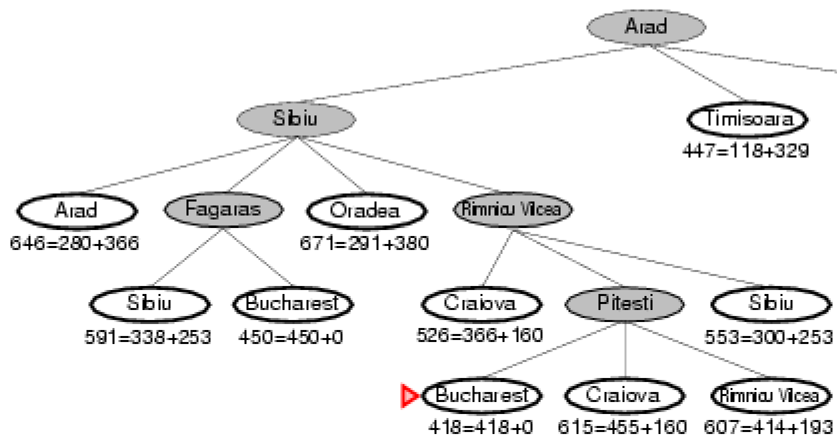


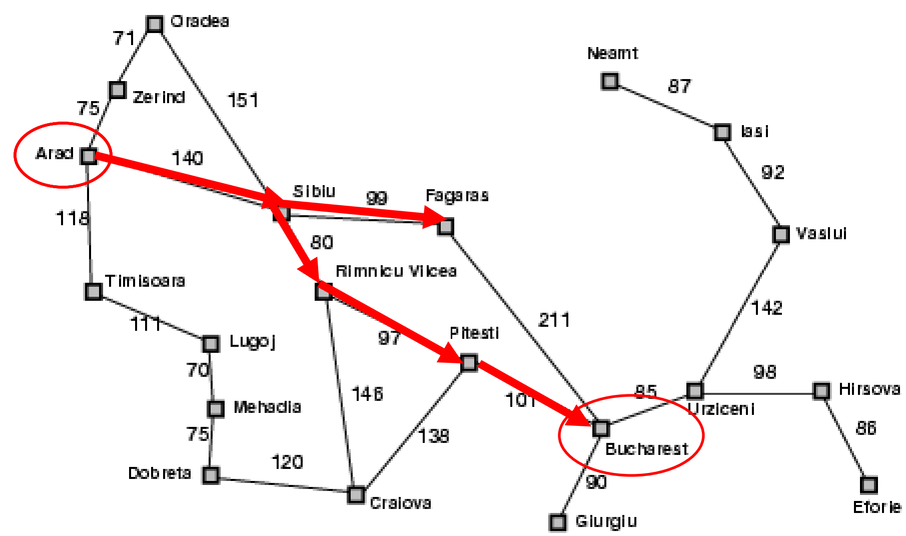


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example

$f(n) = g(n) + h(n)$

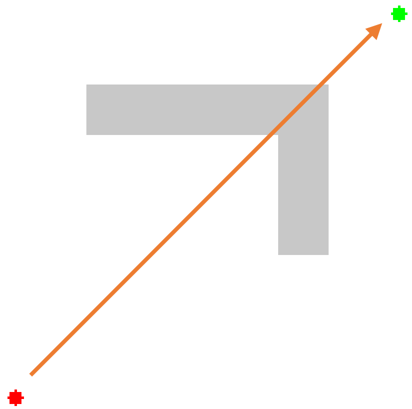




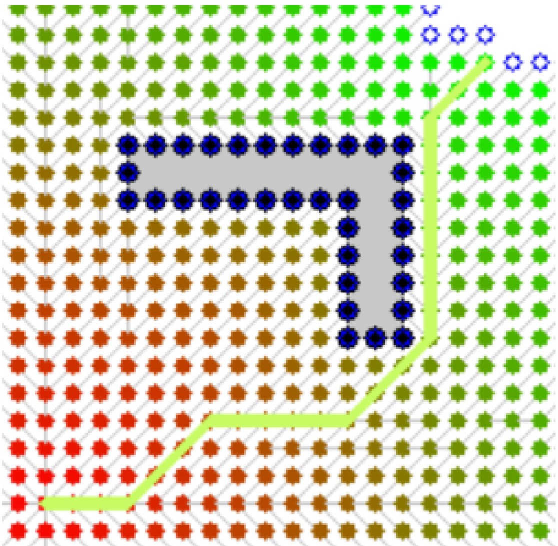
Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

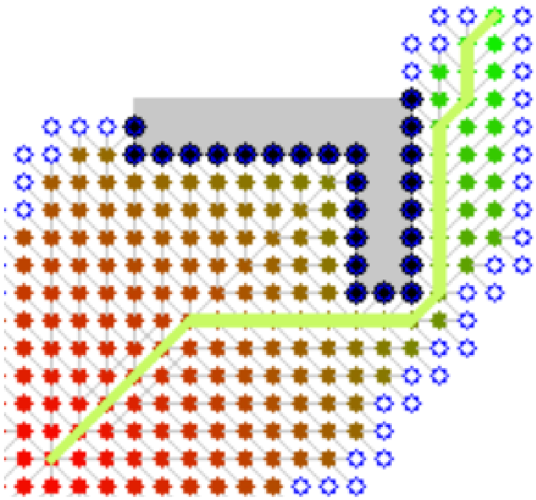
BFS vs. A* search



BFS



A*



Source: [Wikipedia]

Implementation of A* Search

Best-First Search

+

Expand the frontier using $f(n) = g(n) + h(n)$

Admissible heuristics

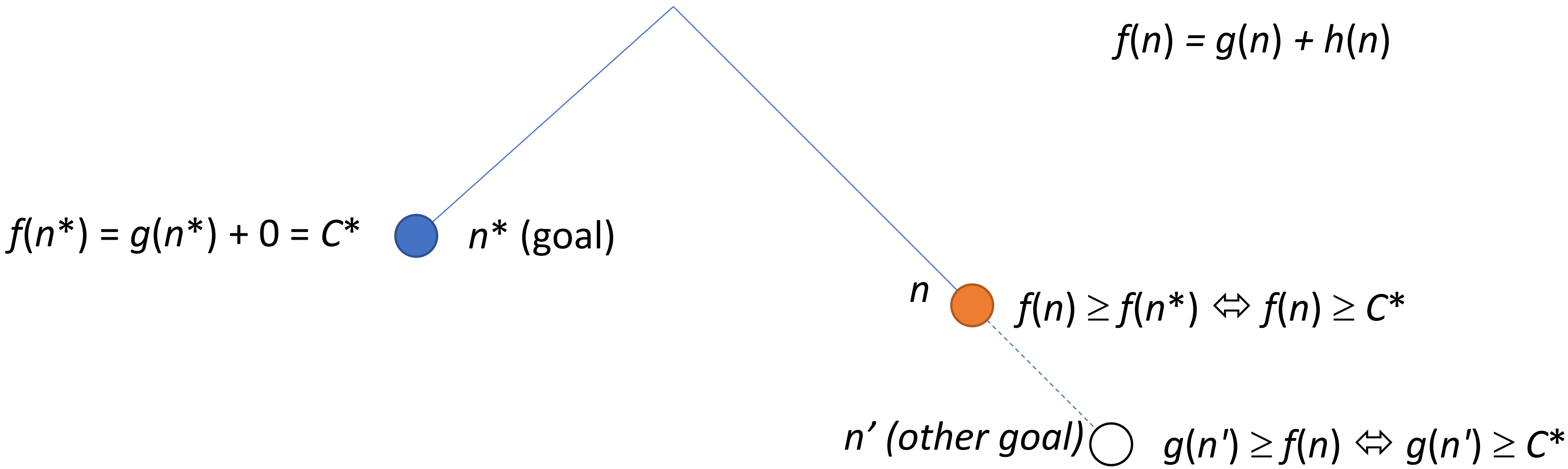
Definition: A heuristic $h(n)$ is **admissible** if for every node n, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.

I.e., an admissible heuristic is a **lower bound** and never overestimates the true cost to reach the goal.

Example: straight line distance never overestimates the actual road distance.

Theorem: If $h(n)$ is admissible, A^* is optimal.

Proof of Optimality of A*



- Suppose A* terminates its search at *n**.
- It has found a path whose *actual cost* $f(n^*) = g(n^*) + 0$ is lower than the *estimated cost* $f(n)$ of any path going through any frontier node.
- Since $f(n)$ is an *optimistic* estimate, it is impossible for *n* to have a successor goal state *n'* with $g(n') < C^*$.

Guarantees of A*

- A* is **optimally efficient**
  - No other tree-based search algorithm that uses the same heuristic can expand fewer nodes and still be guaranteed to find the optimal solution.
  - Any algorithm that does not expand all nodes with $f(n) < C^*$ (the lowest cost of going to a goal node) risks missing the optimal solution.

Properties of A*

- **Complete?**
  Yes

- **Optimal?**
  Yes

- **Time?**
  Number of nodes for which $f(n) \leq C^*$ (exponential)

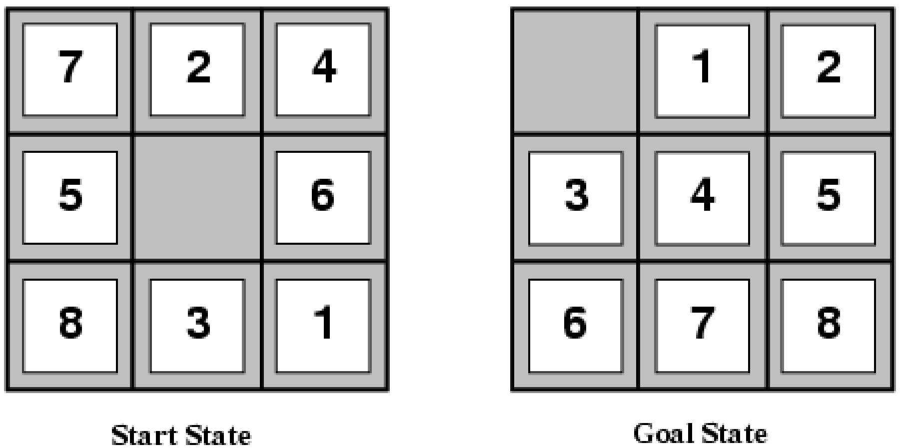
- **Space?**
  Same as time complexity.

Designing heuristic functions

- Heuristics for the 8-puzzle

  $h_1(n)$ = number of misplaced tiles

  $h_2(n)$ = total Manhattan distance (number of squares from desired location of each tile)



$$h_1(\text{start}) = 8$$

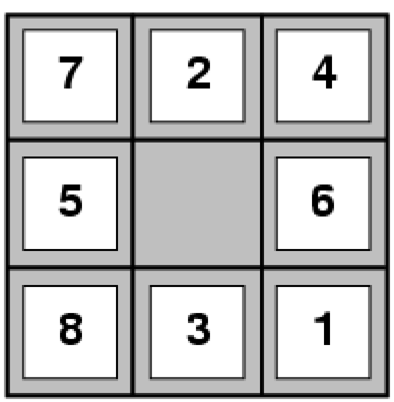
$$h_2(\text{start}) = 3+1+2+2+2+3+3+2 = 18$$

1 needs to move 3 positions

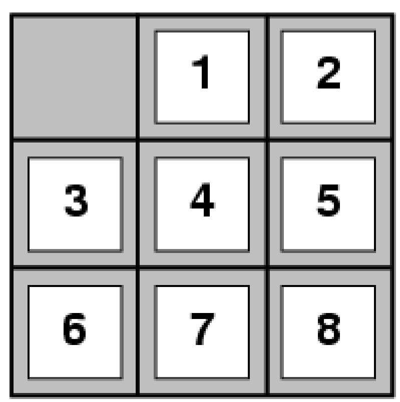
- Are h_1 and h_2 admissible?

Heuristics from relaxed problems

- A problem with fewer restrictions on the actions is called a relaxed problem.
- **The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.**
- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution.
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution.



Start State



Goal State

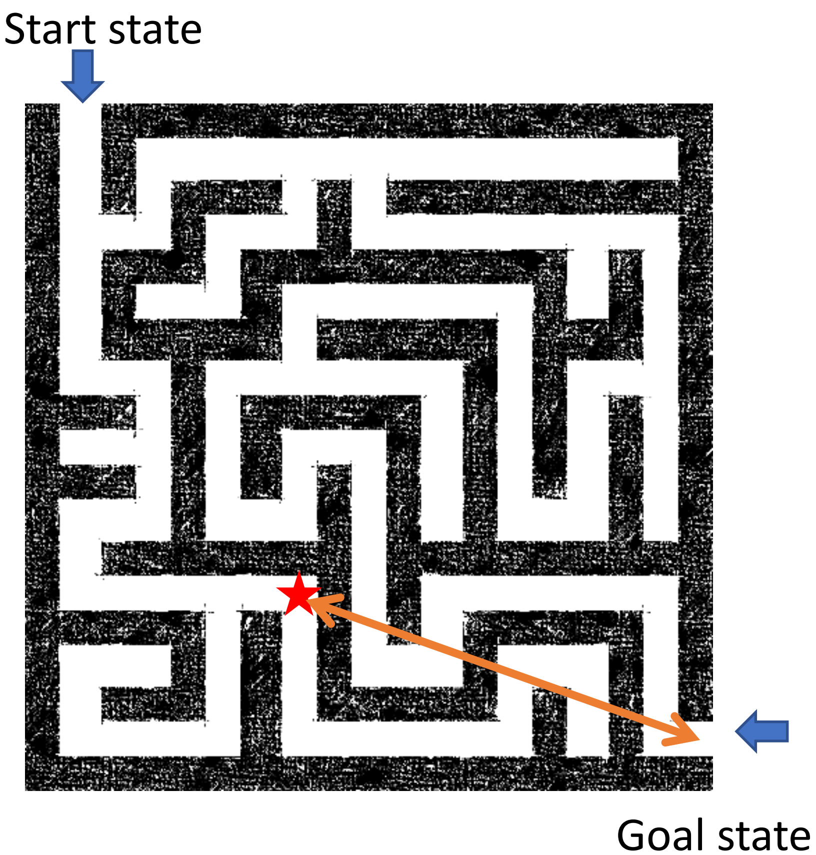
$h_1(\text{start}) = 8$

$h_2(\text{start}) = 3+1+2+2+2+3+3+2 = 18$

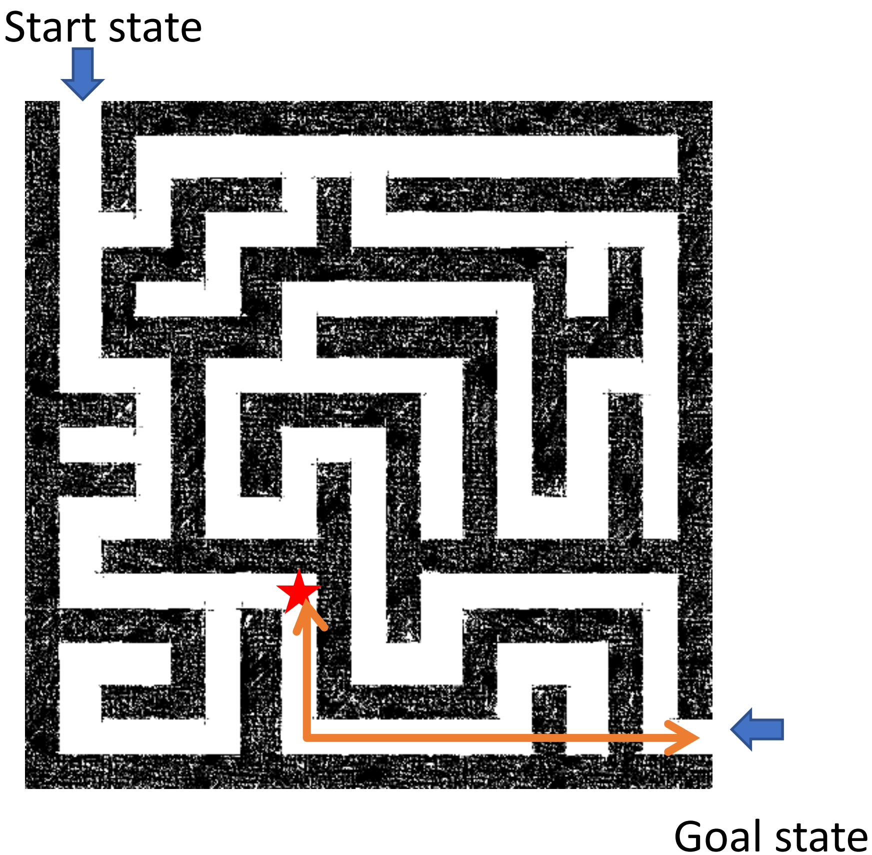
Heuristics from relaxed problems

What relaxations are used in these two cases?

Euclidean distance

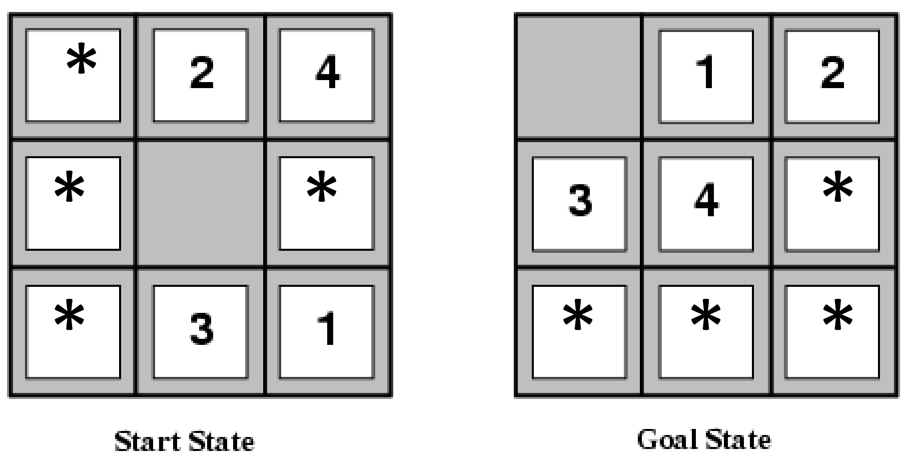


Manhattan distance



Heuristics from subproblems

- Let $h_3(n)$ be the cost of getting a subset of tiles (say, 1,2,3,4) into their correct positions. The final order of the * tiles does not matter.
- Small subproblems are often easy to solve.
- Can precompute and save the exact solution cost for every or many possible subproblem instances – *pattern database.*



Dominance

Definition: If h_1 and h_2 are both admissible heuristics and $h_2(n) \geq h_1(n)$ for all *n,* then h_2 dominates h_1

Which one is better for search?

- A* search expands every node with $f(n) < C^* \Leftrightarrow h(n) < C^* - g(n)$
- A* search with h_2 will expand less nodes and is therefore better.

Dominance

- Typical search costs for the 8-puzzle (average number of nodes expanded for different solution depths d):

- *d*=12 IDS = 3,644,035 nodes
  $A^*(h_1)$ = 227 nodes
  $A^*(h_2)$ = 73 nodes

- *d*=24 IDS ≈ 54,000,000,000 nodes
  $A^*(h_1)$ = 39,135 nodes
  $A^*(h_2)$ = 1,641 nodes

Combining heuristics

- Suppose we have a collection of admissible heuristics $h_1(n), h_2(n), \dots, h_m(n)$, but none of them dominates the others
- How can we combine them?

$$h(n) = \max\{h_1(n), h_2(n), \dots, h_m(n)\}$$

- That is, always pick for each node the heuristic that is closest to the real cost to the goal $h^*(n)$.

Satisficing Search: Weighted A* search

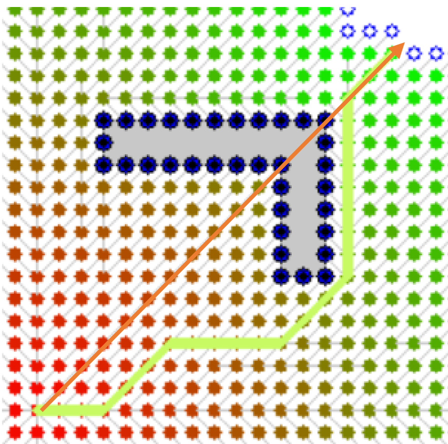
- Often it is sufficient to find a **"good enough" solution** if it can be found very quickly or with way less computational resources.

- We could use inadmissible heuristics in A* search that sometimes overestimate the optimal cost to the goal slightly.
  1. It potentially reduces the number of expanded nodes significantly.
  2. This will break the algorithm's optimality guaranty!

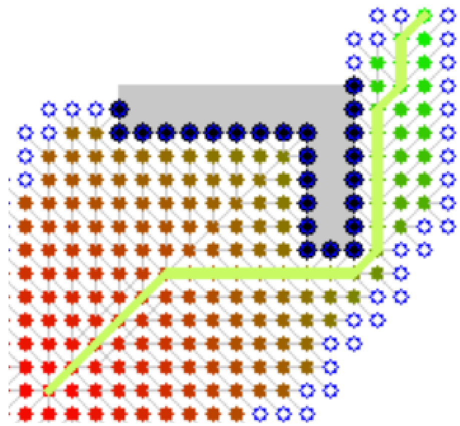
$$\mathrm{f}(n) = g(n) + W \times h(n)$$

A* search:	$g(n) + h(n)$	$(W = 1)$
Uniform cost search:	$g(n)$	$(W = 0)$
Greedy best-first search:	$h(n)$	$(W = \infty)$
Weighted A* search:	$g(n) + W \times h(n)$	$(1 < W < \infty)$

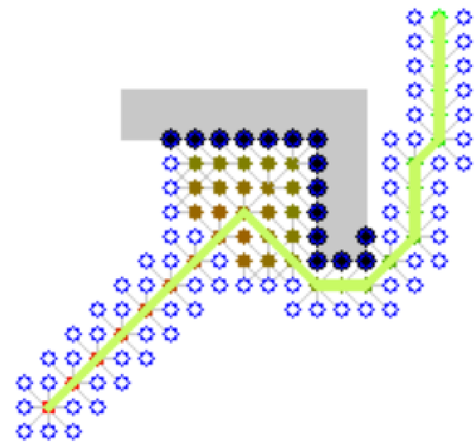
Example of weighted A* search



BFS
$f(n) =$ # actions to reach n



Exact A* Search
$f(n) = g(n) + h_{Eucl}(n)$



Weighted A* Search
$f(n) = g(n) + 5\, h_{Eucl}(n)$

Source: [Wikipedia]

Memory-bounded search

- The memory usage of A* (search tree and frontier) can still be exorbitant.
- How can we make A* more memory-efficient while maintaining completeness and optimality?
  - Iterative deepening A* search.
  - Recursive best-first search, SMA*: Forget some subtrees but remember the best f-value in these subtrees and regenerate them later if necessary.

- **Problems**: memory-bounded strategies can be complicated to implement, and suffer from “thrashing” (regenerating forgotten nodes).

Uninformed search strategies

Algorithm	Complete?	Optimal?	Time complexity	Space complexity
BFS	Yes	If all step costs are equal	$O(b^d)$	$O(b^d)$
Uniform-cost Search	Yes	Yes	Number of nodes with $g(n) \le C^*$	
DFS	In finite spaces (cycle checking)	No	$O(b^m)$	$O(bm)$ More with cycles
IDS	Yes	If all step costs are equal	$O(b^d)$	$O(bd)$ More with cycles

b: maximum branching factor of the search tree
d: depth of the optimal solution
m: maximum length of any path in the state space
C*: cost of optimal solution

All search strategies

Algorithm	Complete?	Optimal?	Time complexity	Space complexity
BFS	Yes	If all step costs are equal	$O(b^d)$	$O(b^d)$
Uniform-cost Search	Yes	Yes	Number of nodes with $g(n) \leq C^*$	
DFS	In finite spaces (cycles checking)	No	$O(b^m)$	$O(bm)$ more with cycles
IDS	Yes	If all step costs are equal	$O(b^d)$	$O(bd)$ More with cycles
Greedy best-first Search	In finite spaces (cycles checking)	No	Depends on heuristic	Worst case: $O(b^m)$ Best case: $O(bd)$
A* Search	Yes	Yes	Number of nodes with $g(n)+h(n) \leq C^*$	

Implementation as Best-first Search

- All discussed search strategies can be implemented using Best-first search.
- Best-first search expands always the **node with the minimum value** of an evaluation function.

Search Strategy	Evaluation function
BFS	$g(n)$ = path uniform cost
Uniform-cost Search	$g(n)$ = path cost
DFS/IDS (see below)	$-g(n)$
Greedy best-first Search	$h(n)$
(weighted) A* Search	$g(n) + w \times h(n)$

- **Note:** DFS/IDS is typically implemented differently to achieve the lower space complexity.