CS 5/7320
Artificial Intelligence

Solving problems by searching

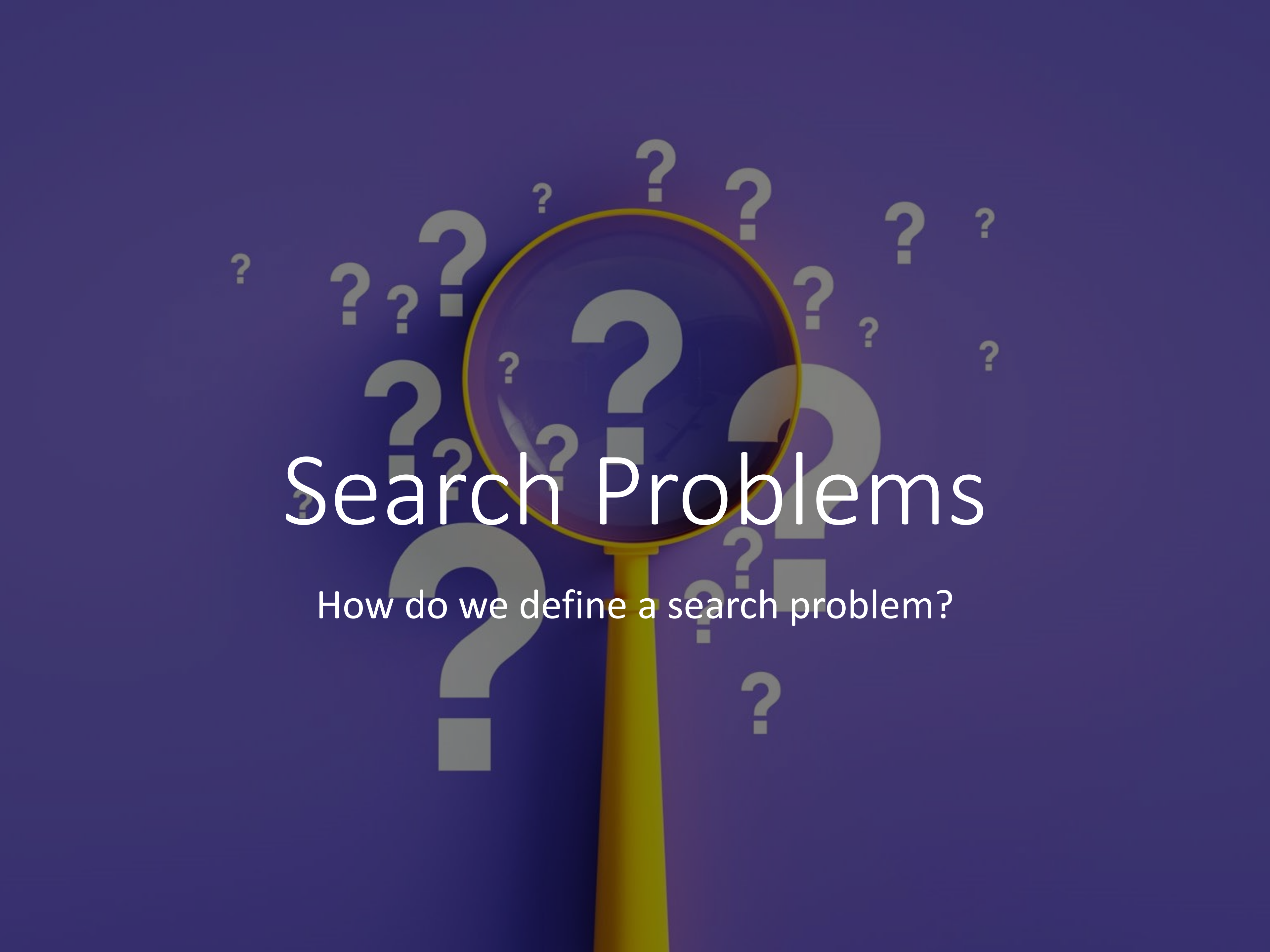
AIMA Chapter 3

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

Online Material

Contents

- What are search problems?
- Tree search
- Search space
- Uninformed search
- Informed search

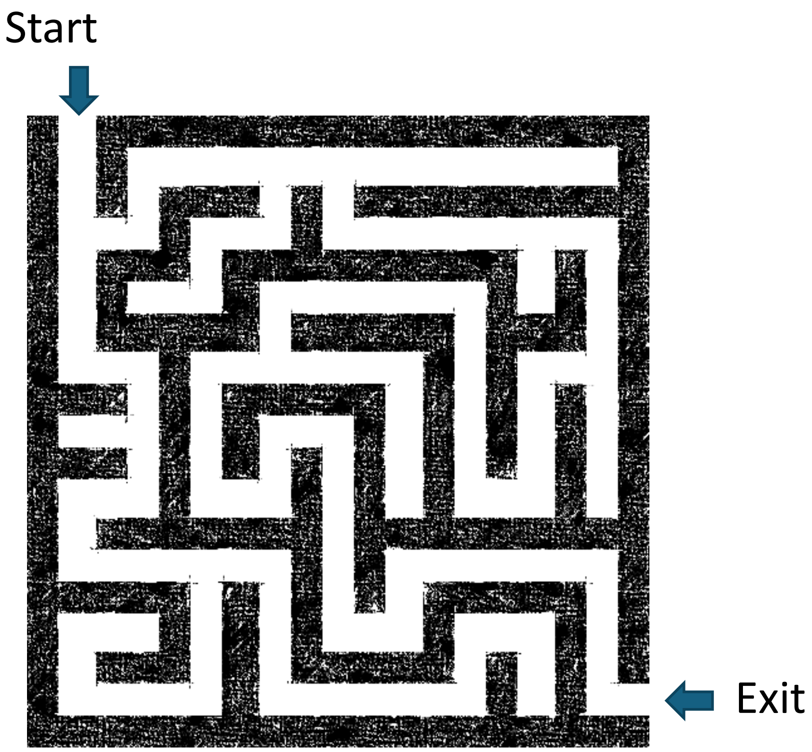


Search Problems

How do we define a search problem?

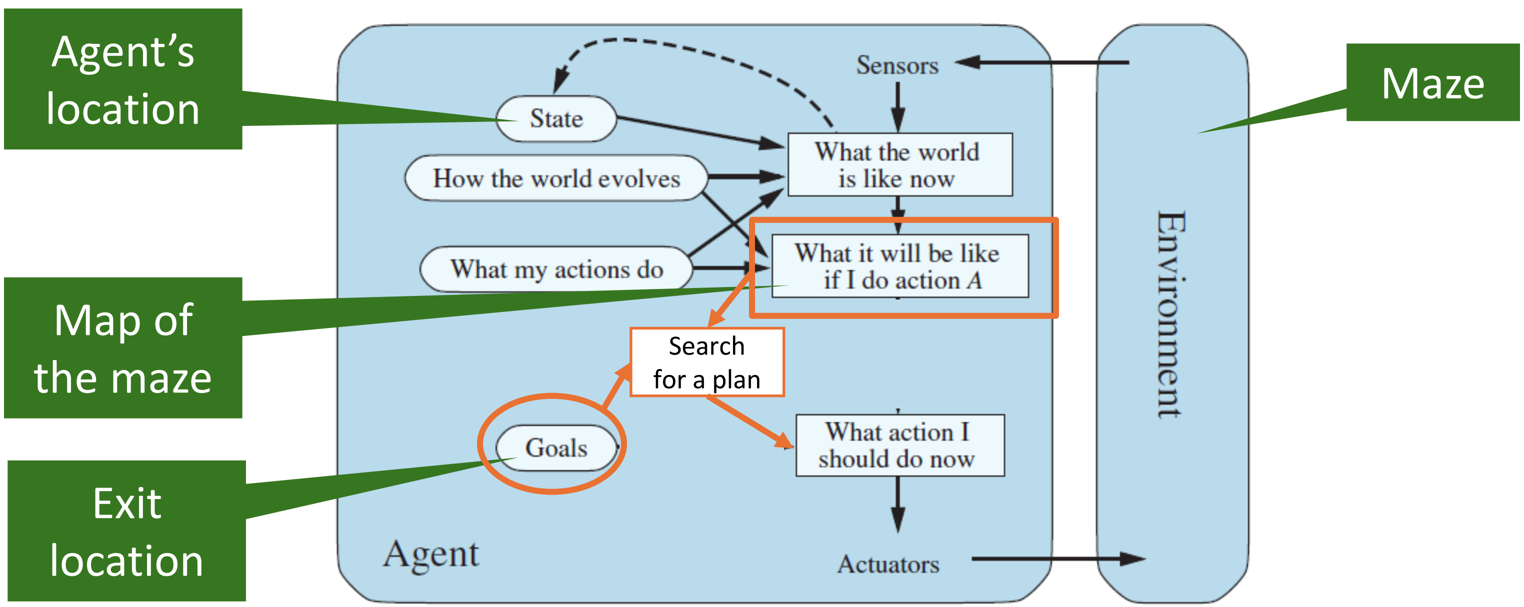
What are Search Problems?

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



Remember: Planning Agent (Goal-based)

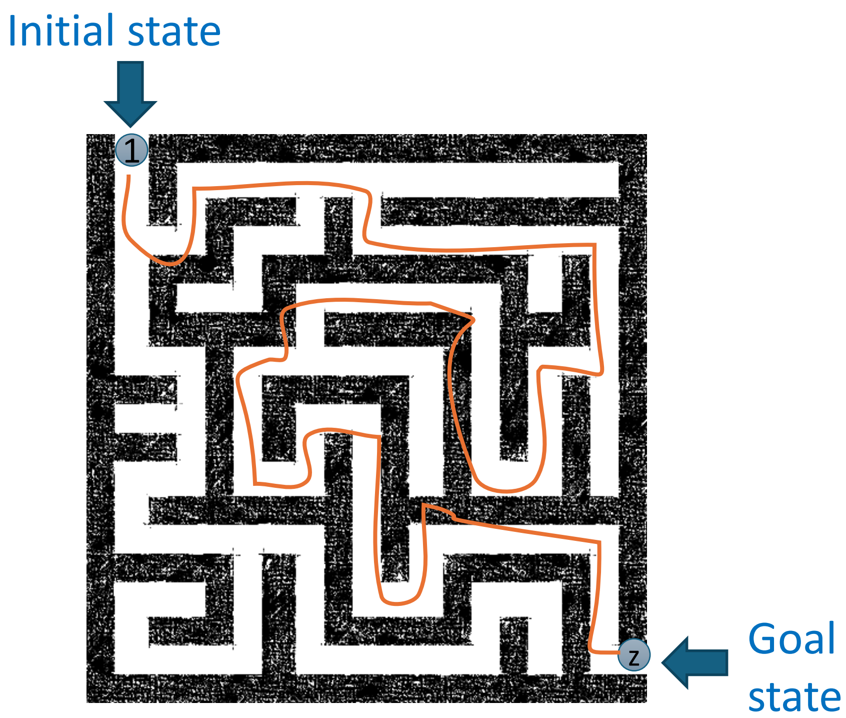
- The agent has the task of reaching a defined **goal state**.
- The performance measure is typically the cost to reach the goal.
- We will discuss a special type of **goal-based agents** called **planning agents,** which use **search algorithms** to plan a sequence of actions that lead to the goal.



$$a_i = \operatorname{argmin}_{a_i \in A} \left[\sum_{t=i}^{T} c_t \,\middle|\, s_T \in S^{goal} \right]$$

Planning for Search Problems

- 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**.
- The **optimal solution** is the sequence of actions (or equivalently a sequence of states) that gives the lowest path cost for reaching the goal.

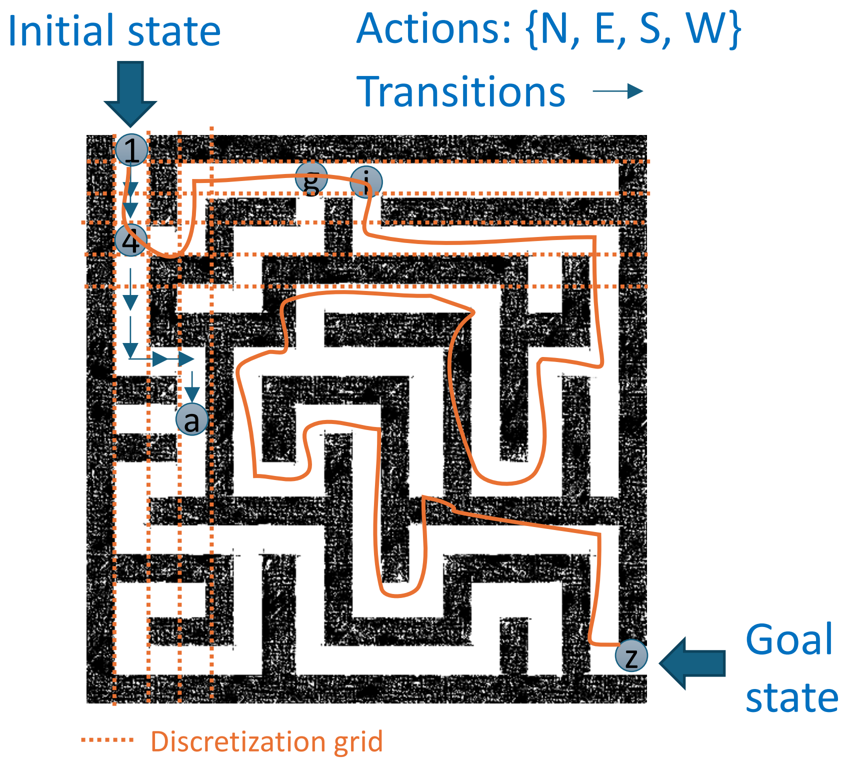


Phases:

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

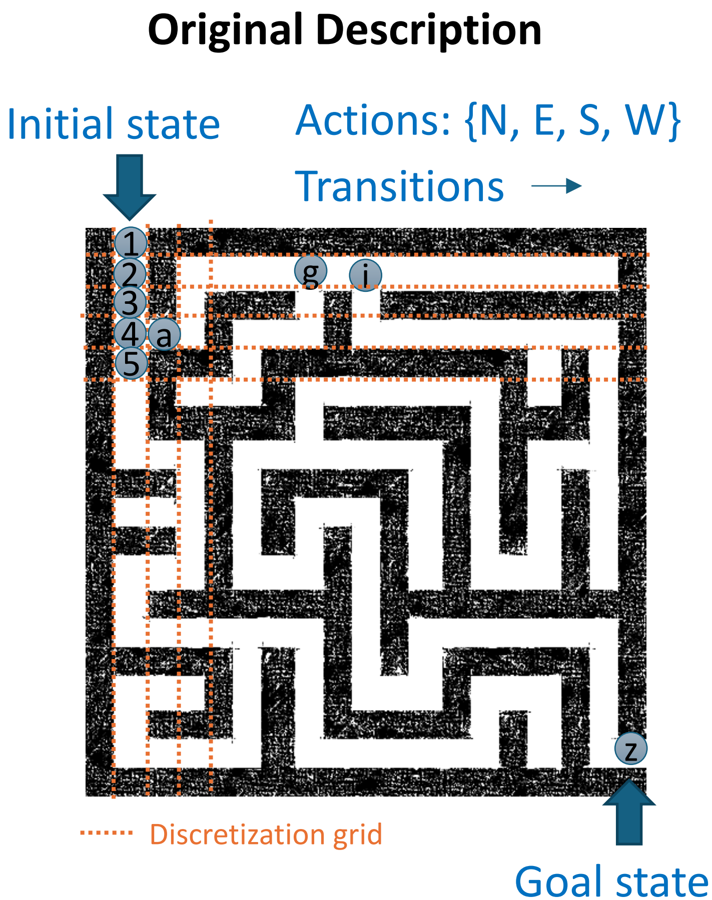
Definition of a Search Problem

- **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
- **Goal state:** state description
- **Path cost:** the sum of *step costs*



Important: 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.

Transition Function and Available Actions



- Definition as an **action schema**:
  $Action(go(dir))$
  PRECOND: no wall in direction dir
  EFFECT: change the agent's location according to dir
- Definition as a **function**:
  $f: S \times A \rightarrow S$ or $s' = result(s, a)$
- A **graph** with states as vertices and actions as edges.
- Function implemented as a **table** representing the state space as a graph.

s	a	s'
1	S	2
2	N	1
2	S	3
…	…	…
4	E	a
4	S	5
4	N	3
…	…	…

- **Available actions** in a state come from the transition function. E.g.,
  $actions(4) = \{E, S, N\}$

Note: Known and deterministic is a property of the transition function!

Example: Romania Vacation

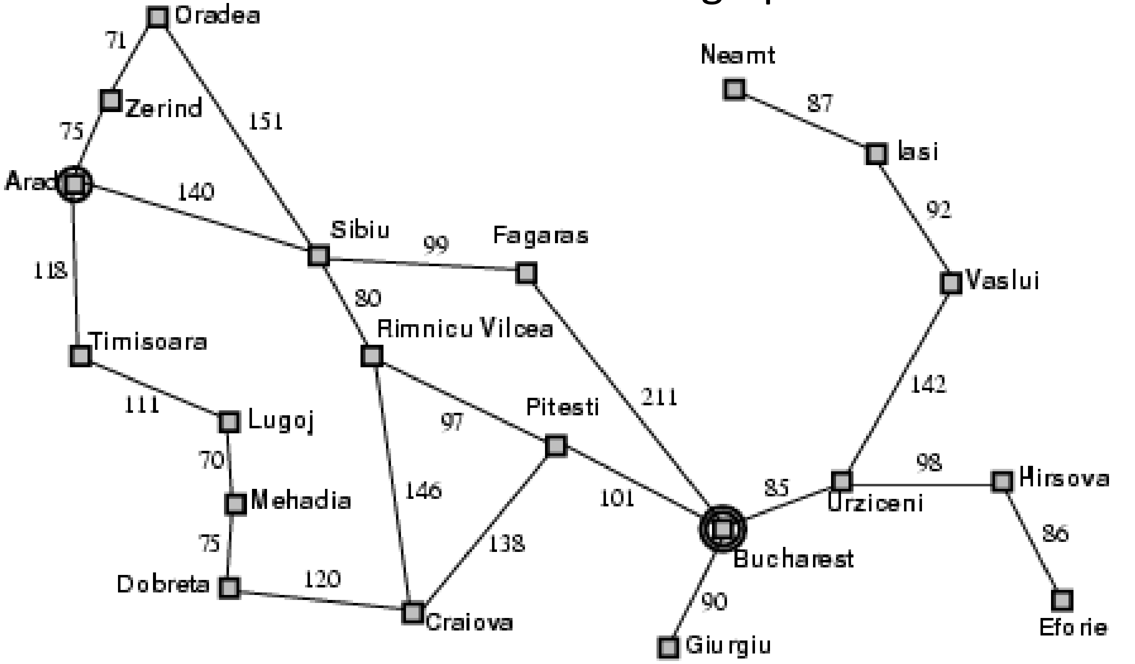
- On vacation in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest

Original Description

- **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.

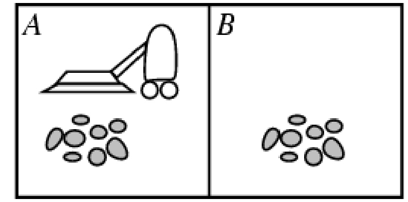
State Space/Transition model

Defined as a graph

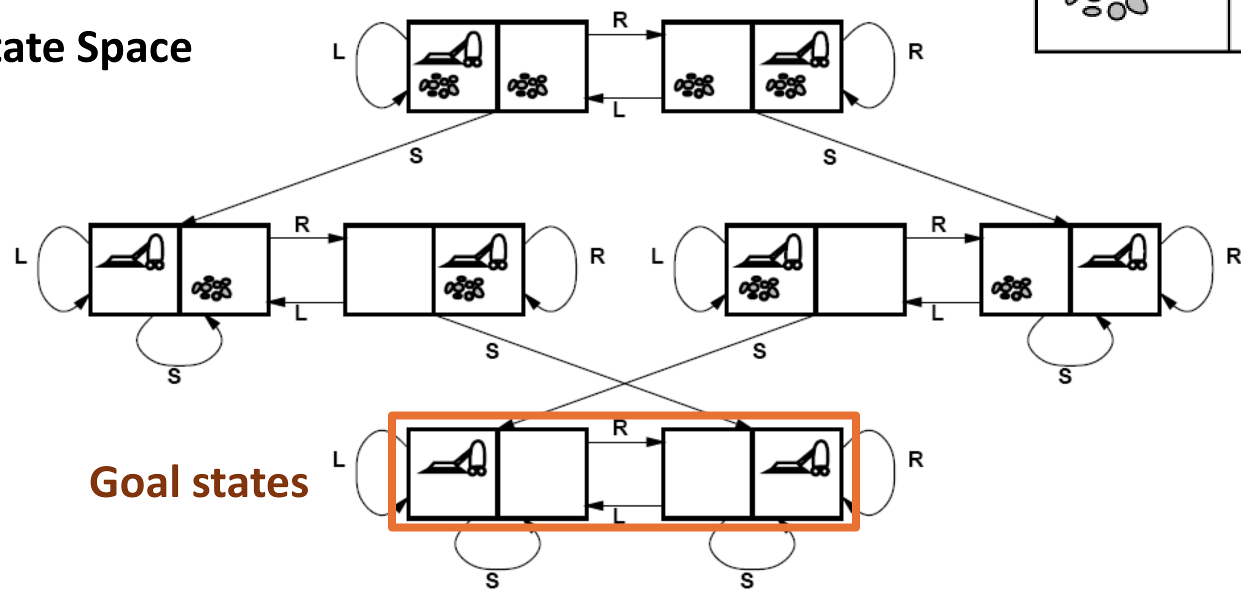


Distance in miles

Example: Vacuum world



State Space

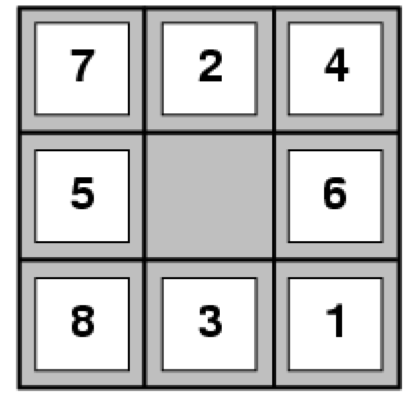


- **Initial State:** Defined by agent location and dirt location.
- **Actions:** Left, right, suck
- **Transition model:** Clean a location or move.
- **Goal state:** All locations are clean.
- **Path cost:** E.g., number if actions

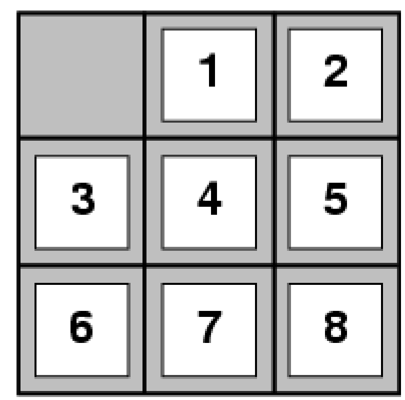
There are 8 possible atomic states of the system.
Why is the number of states for *n* possible locations $n(2^n)$?

Example: Sliding-tile Puzzle

- **Initial State:** A given configuration.
- **Actions:** Move blank left, right, up, down
- **Transition model:** Move a tile
- **Goal state:** Tiles are arranged empty and 1-8 in order
- **Path cost:** 1 per tile move.



Start State



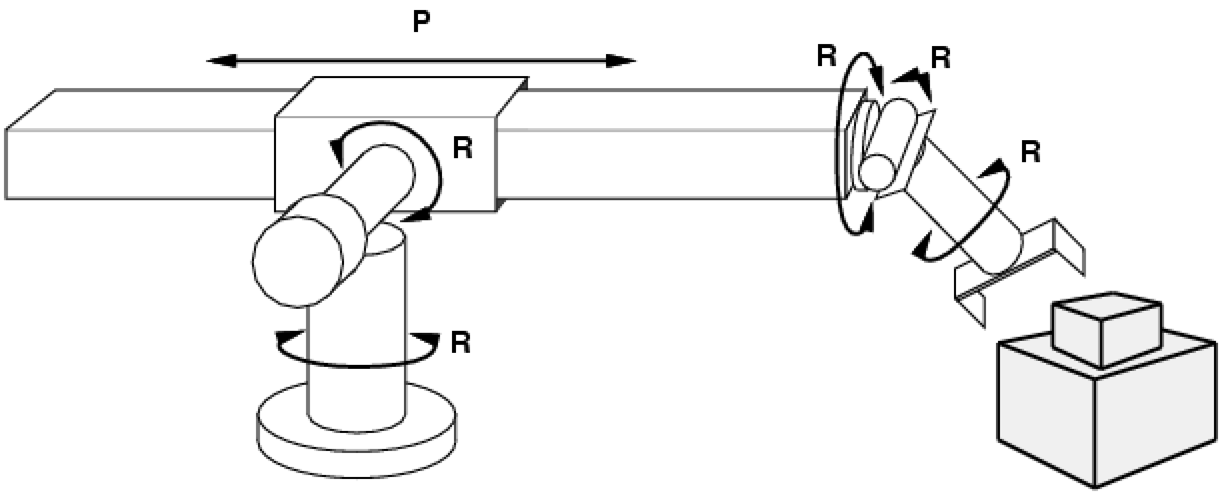
Goal State

State space size

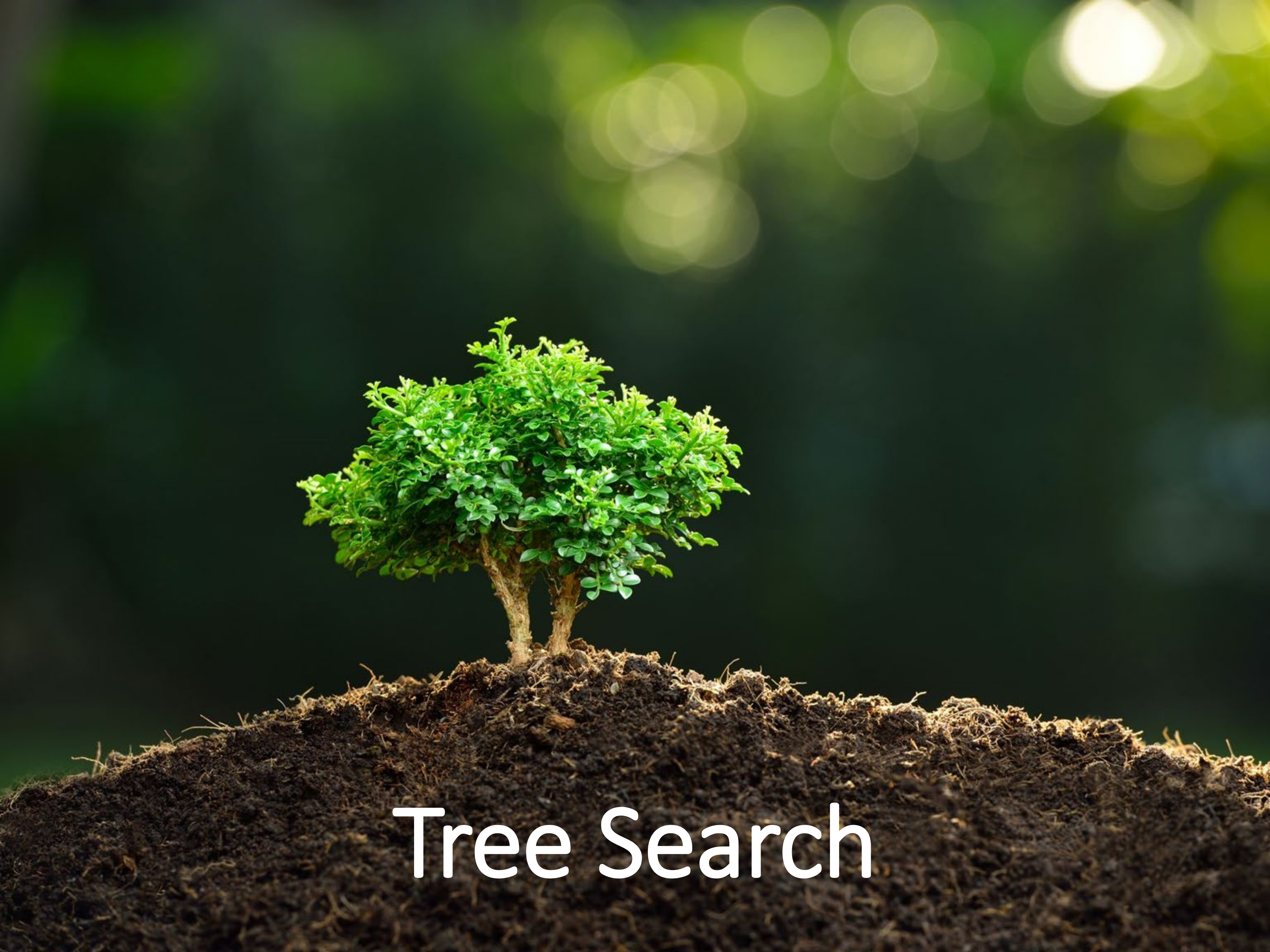
Each state describes the location of each tile (including the empty one). ½ of the permutations are unreachable.

- 8-puzzle: $9!/2 = 181{,}440$ states
- 15-puzzle: $16!/2 \approx 10^{13}$ states
- 24-puzzle: $25!/2 \approx 10^{25}$ states

Example: Robot Motion Planning



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



Tree Search

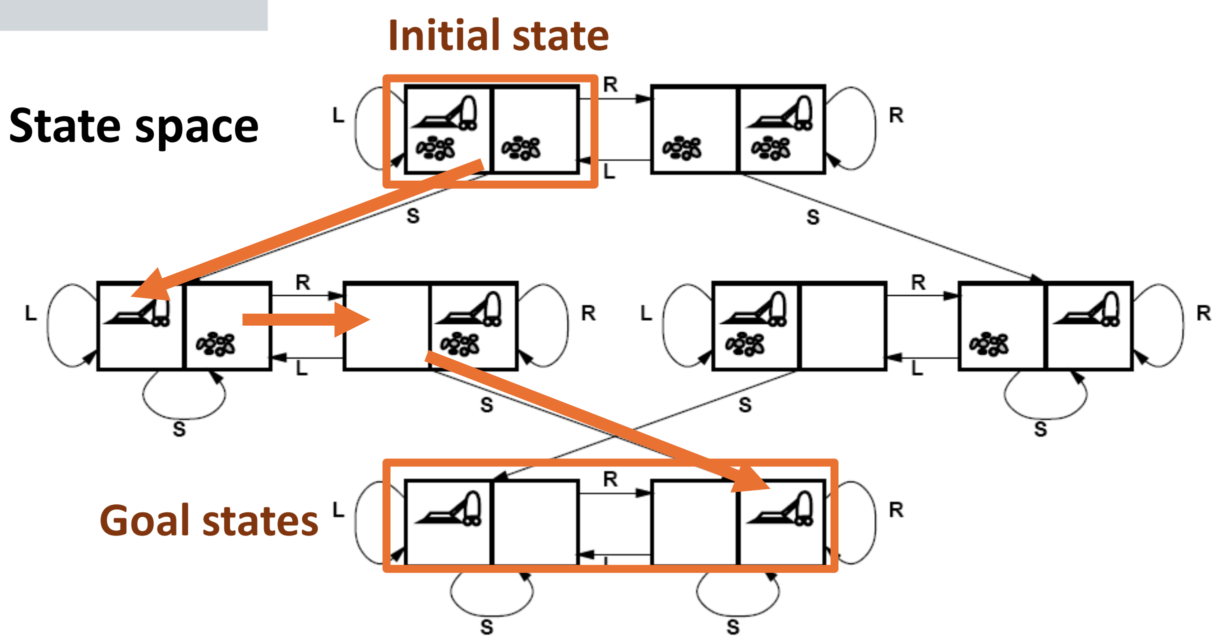
Solving Search Problems

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) when shortest path algorithms for graphs are too expensive?

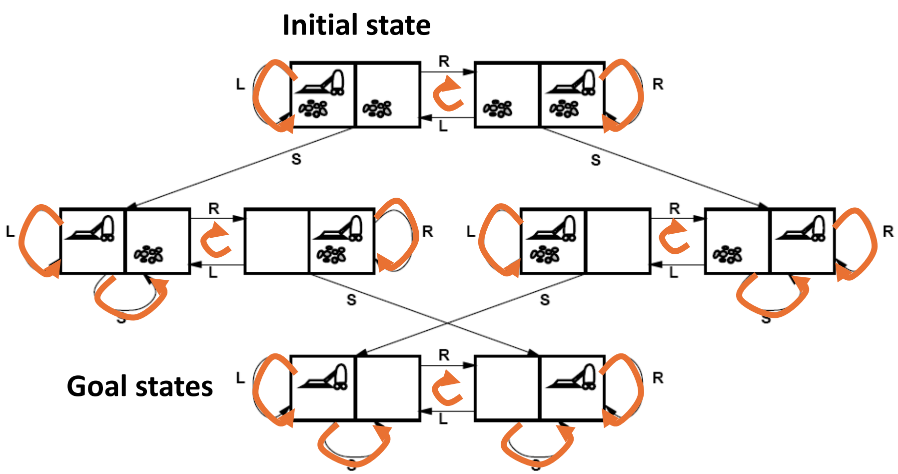
Construct a **search tree** for the state space graph so we can use much cheaper **tree search**!



Issue: Transition Model is a Graph and Not a Tree!

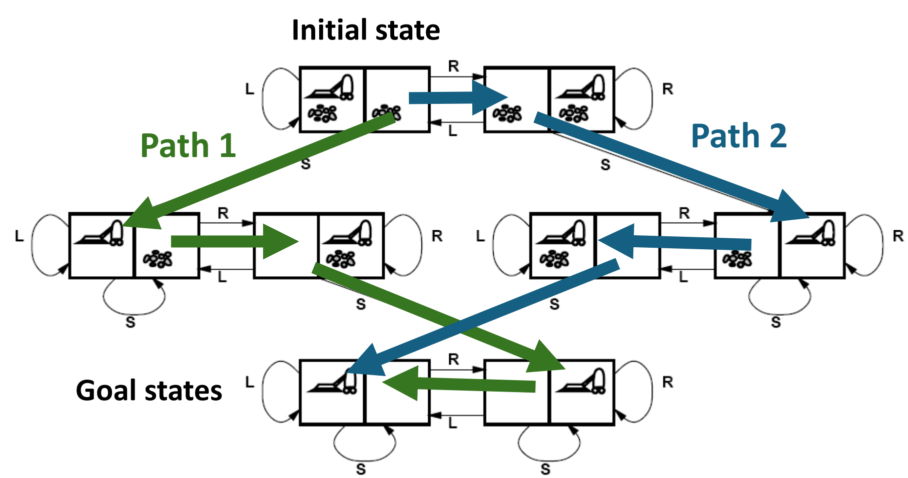
Cycles

Return to the same state



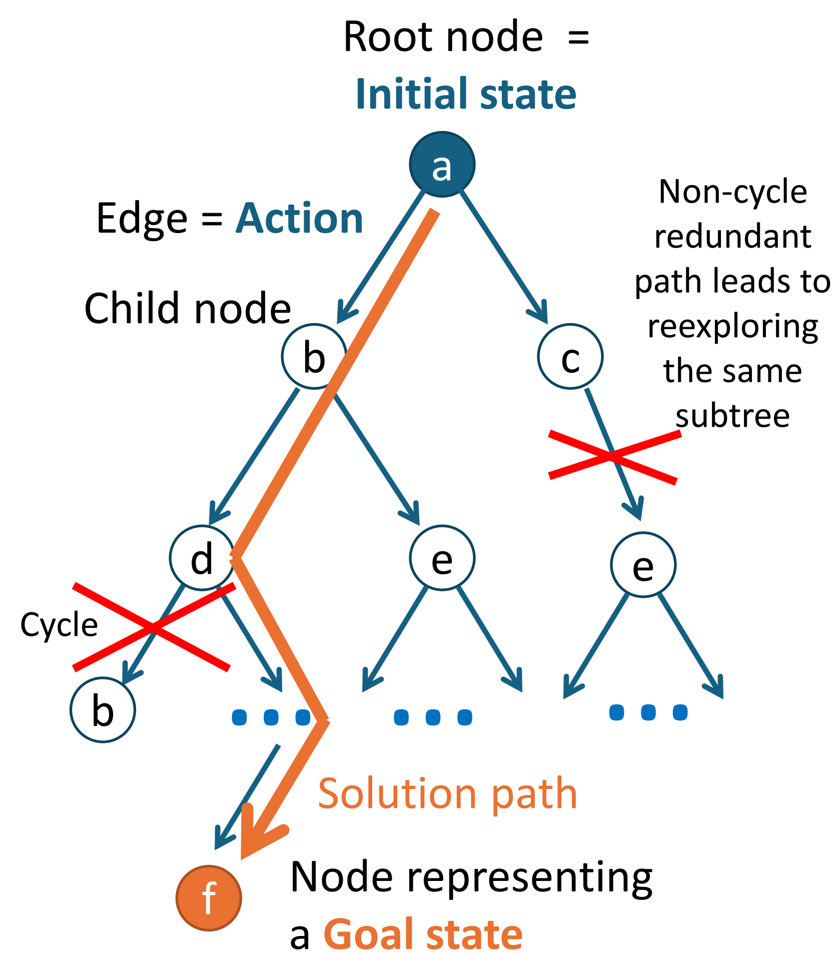
Non-cycle redundant paths

Multiple paths to get to the same state



Creating a Search Tree

- Superimpose a “what if” tree of possible actions and outcomes (states) on the state space graph.
- The **Root node** represents the initial stare.
- An action child node is reached by an **edge** representing an action. The corresponding state is defined by the transition model.
- Trees cannot have **cycles (loops).** Cycles in the search space must be broken to prevent infinite loops.
- Trees cannot have **multiple paths to the same state.** These are called redundant paths. Removing suboptimal 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 tree node represents a state of the system. If redundant path cannot be prevented then state can be represented by multiple nodes in the tree.



Differences Between Typical Tree Search and AI Search

Typical tree search

- Assumes a given tree that fits in memory.

- Trees have by construction no cycles or redundant paths.

AI tree/graph search

- The search tree is too large to fit into **memory**.
  a. **Builds parts of the tree** from the initial state using the transition function representing the graph.
  b. **Memory management** is very important.

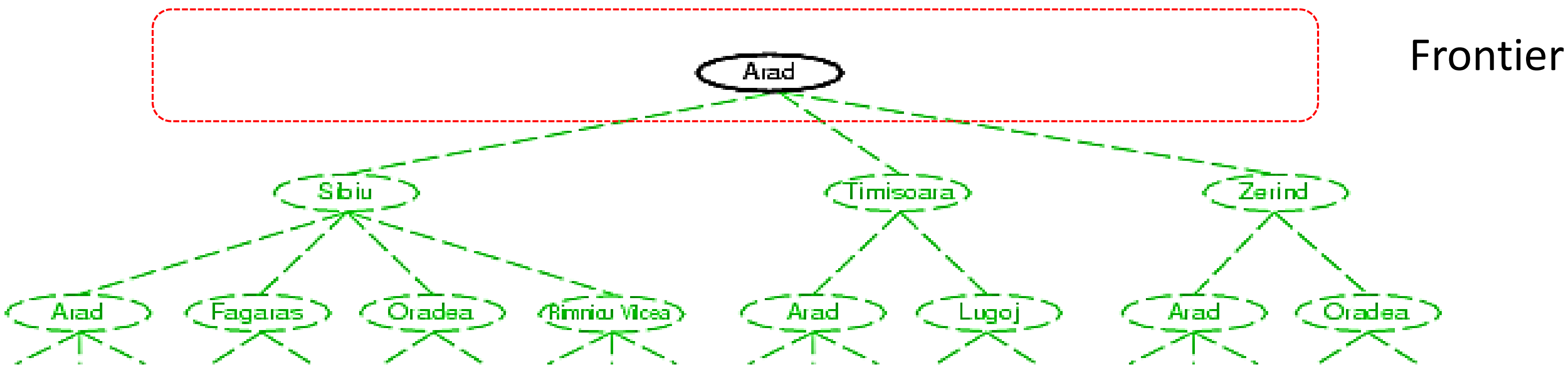
- The search space is typically a very large and complicated graph. Memory-efficient **cycle checking** is very important to avoid infinite loops or minimize searching parts of the search space multiple times.
- Checking redundant paths often requires too much memory and we accept searching the same part multiple times.

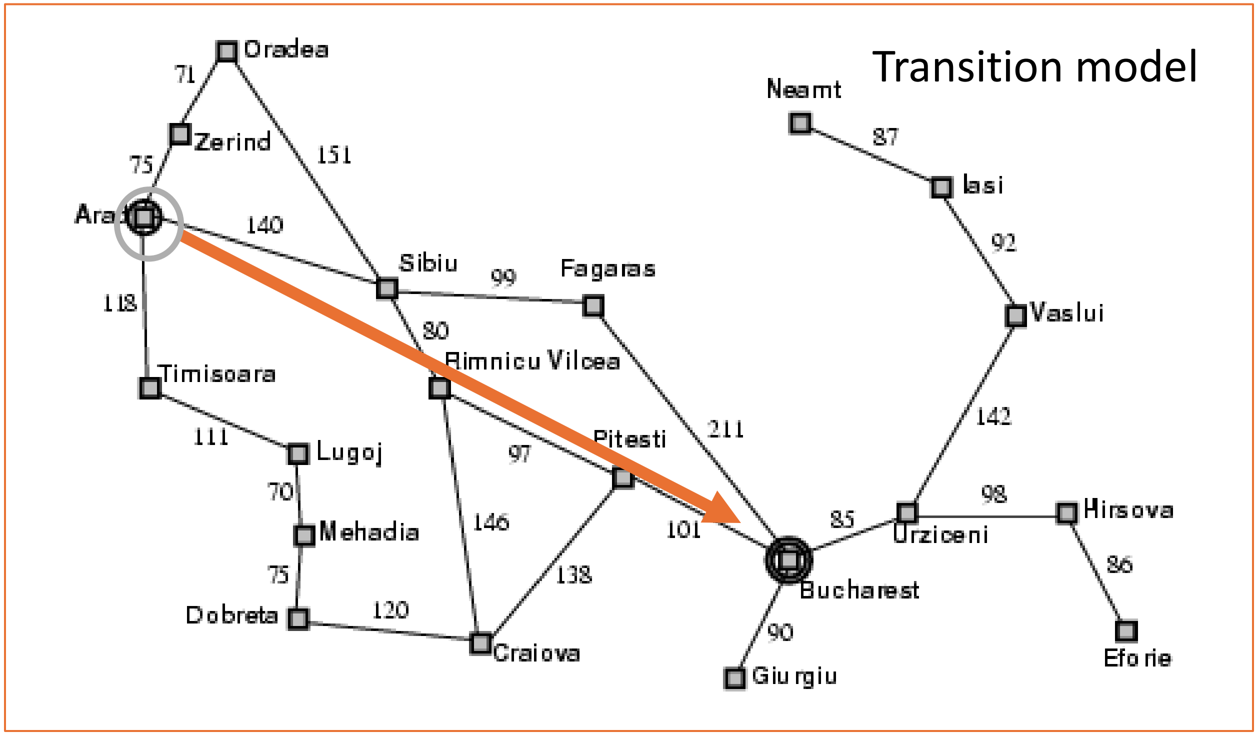
Tree Search Algorithm Outline

1. Initialize the **frontier** (set of unexplored known nodes) using the **starting state/root node.**

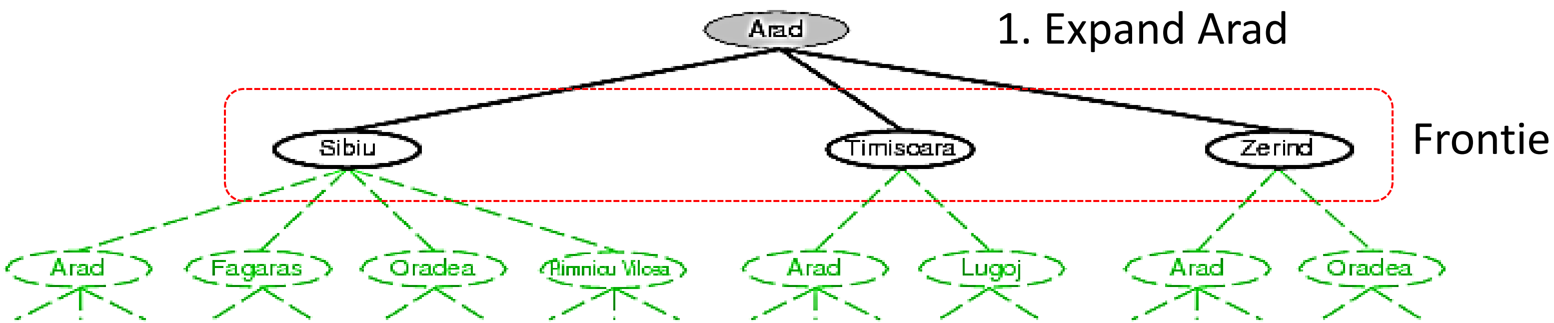
2. While the frontier is not empty:
   a) Choose the next frontier node to expand according to the **search strategy.**
   b) If the node represents a **goal state,** return it as the solution.
   c) Else **expand** the node (i.e., apply all possible actions to the transition model) and add its children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

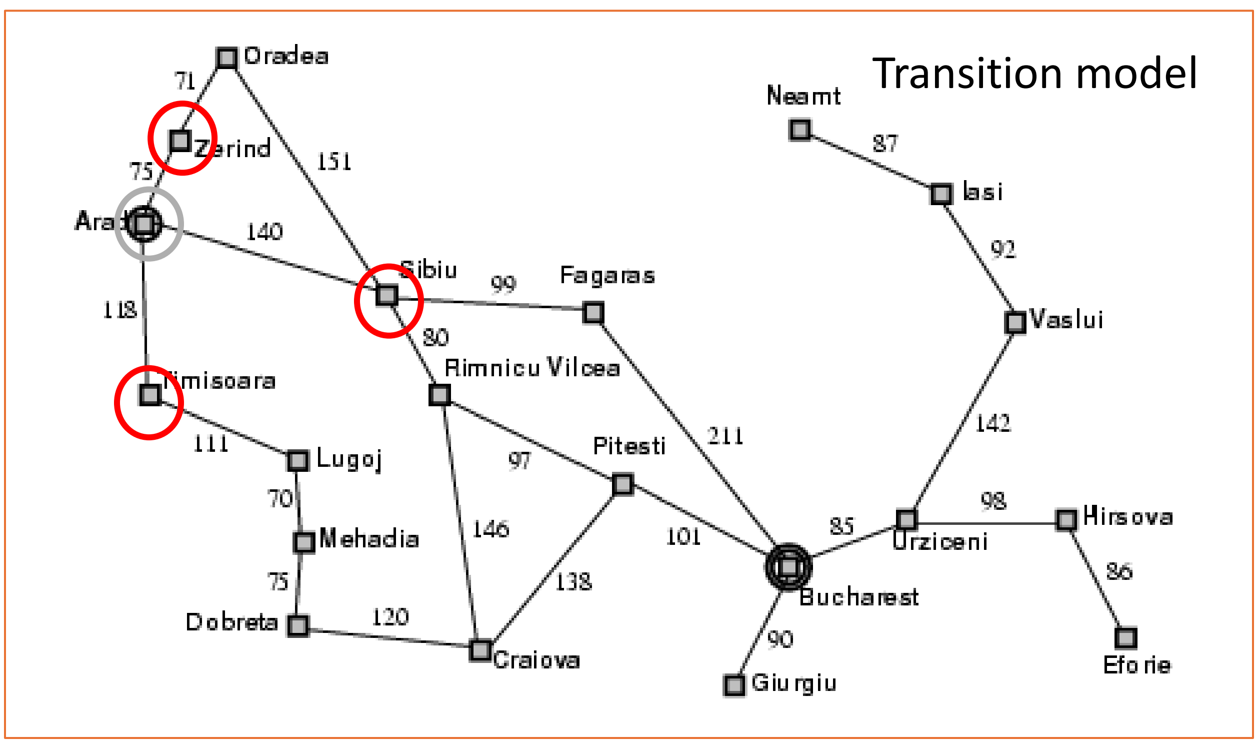
Tree Search Example



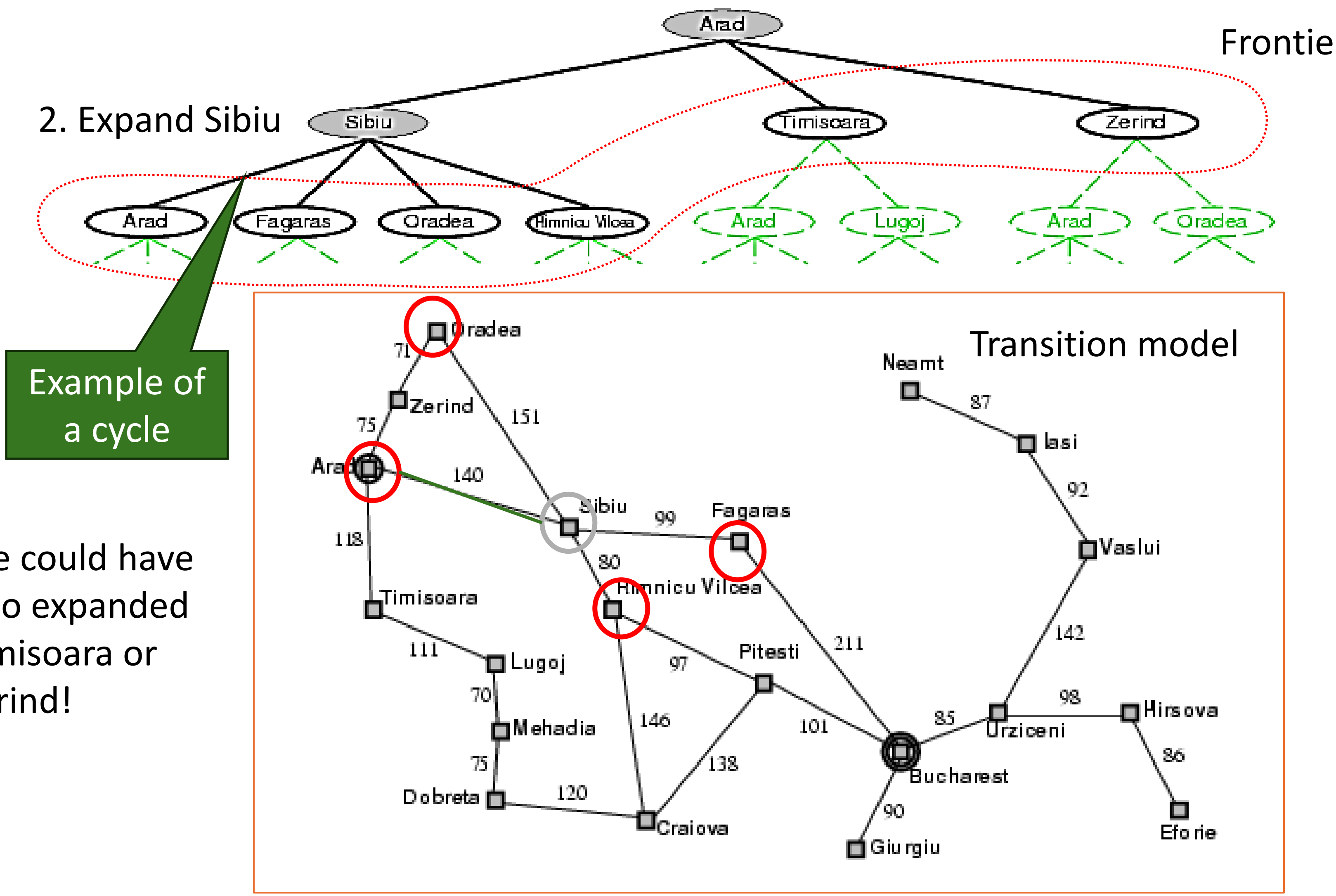


Tree Search Example





Tree Search Example



We could have also expanded Timisoara or Zerind!

Search Strategies: Properties

- 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?

- We will discuss different search strategies and use these properties to compare them.

Space and Time Complexity

State Space vs. Search Tree Size

State Space vs. Search Tree Size

- Space and time complexity depend on the **number of tree nodes** searched (created and visited) till a goal node is found. For a tree with n nodes we have:

$$O(n)$$

- **Remember**: For perfect cycle checking and redundant path elimination, we have a 1:1 mapping between nodes and states:

$$\text{Nodes in the search tree} = \text{states in the search space}$$

  Otherwise, we may have multiple nodes representing a state.

- We have the following options to estimate n for a search problem:
  a. **Estimate the reachable state space size.**
  b. **Estimate the number of searched tree nodes.**

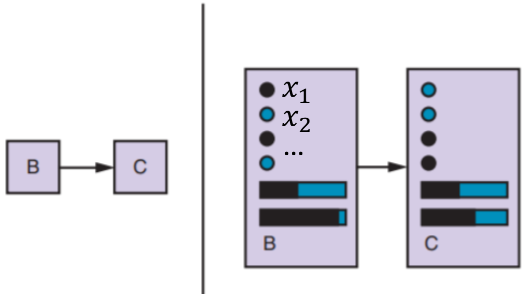
- Estimating the complexity is important to judge:
  - How difficult is the problem?
  - What algorithm will fit in memory?
  - Can we find a solution fast enough?
  - Can we find the optimal solution, or do we need to use a heuristic?

State Space Size Estimation

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. Only reachable states are important for search.

State representation



(a) Atomic (b) Factored

The factored state consists of variables called fluents that represent conditions that can change over time.

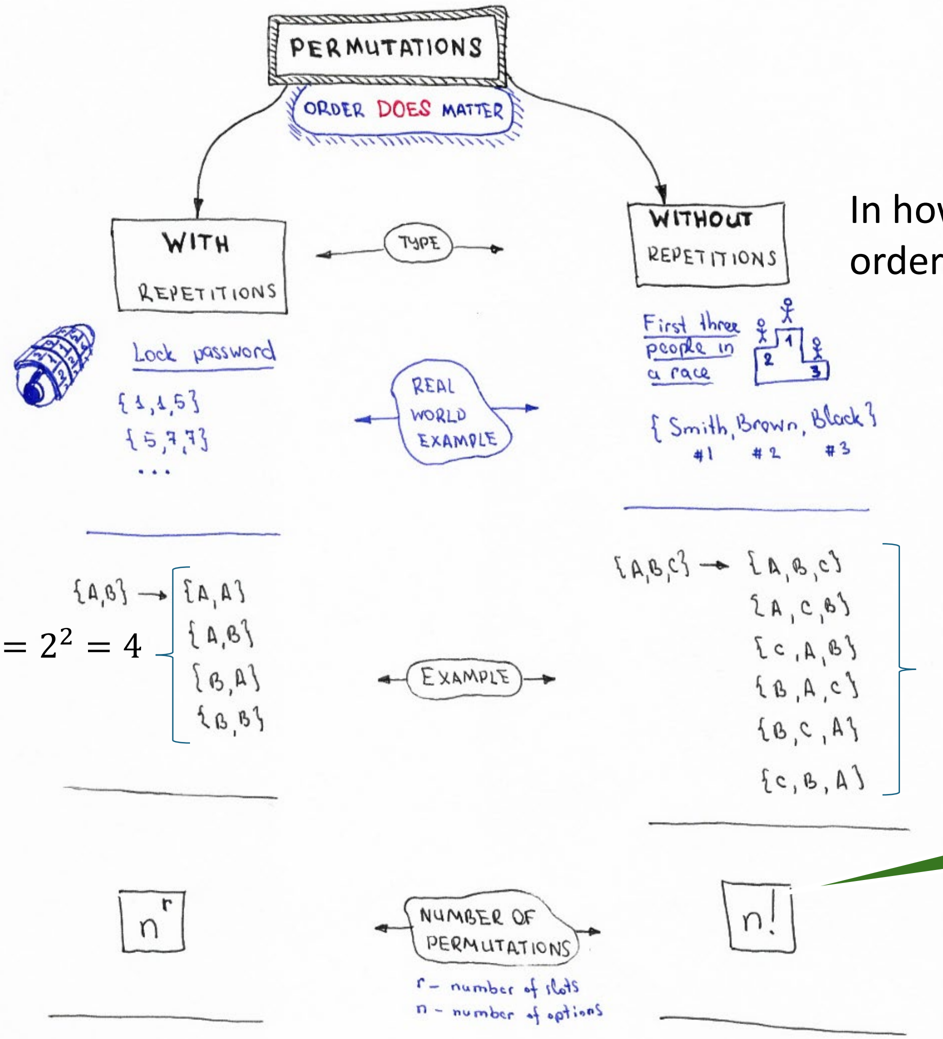
Estimation

- Even if the used algorithm represents the state space using atomic states, we may know the internal (factored) representation. It can be used to estimate the problem size.
- The basic rule is to estimate the state space size for factored state representation with l fluents (variables) as:

$$n = |X_1| \times |X_2| \times \cdots \times |X_l|$$

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

Reminder: Combinatorics - Permutations



$2 \times 2 = 2^2 = 4$

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

$3 \times 2 \times 1 = 6$

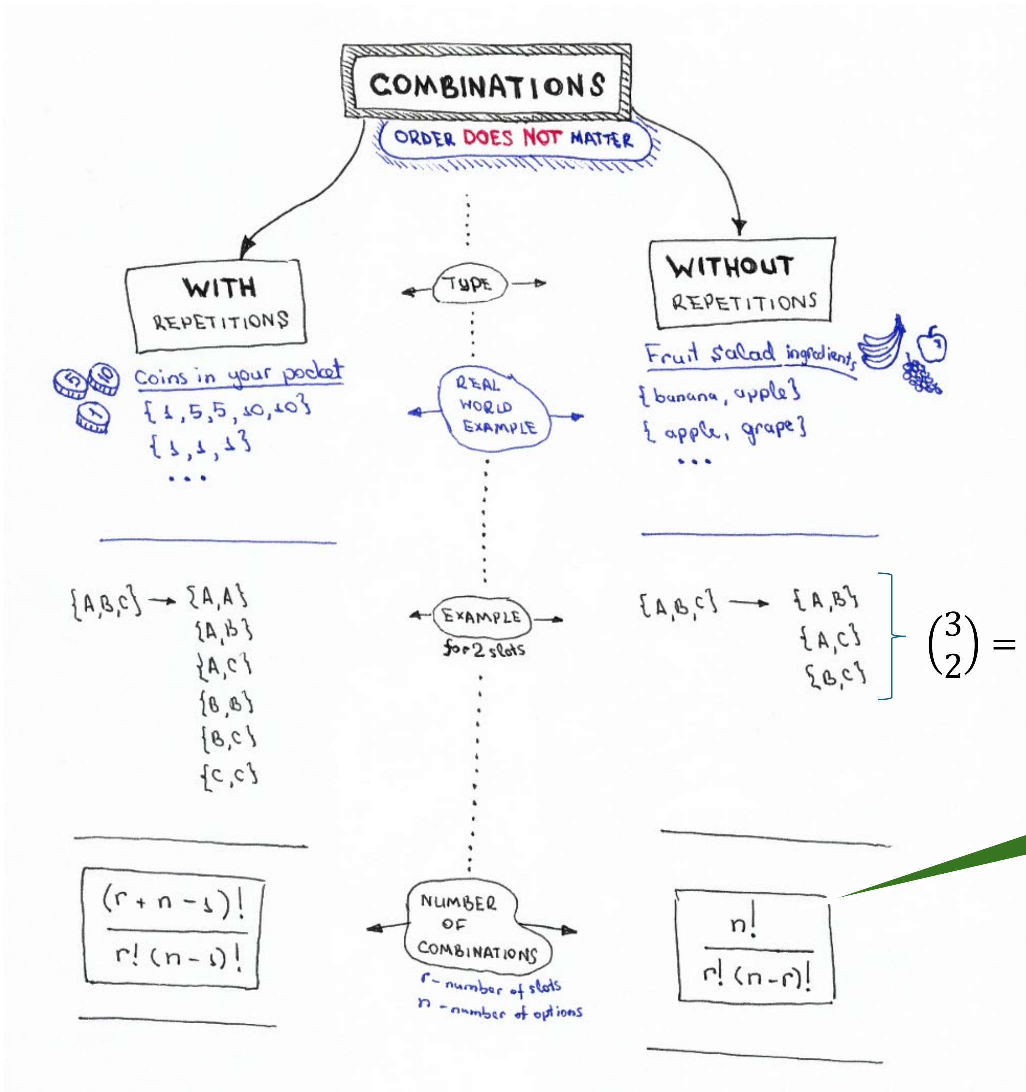
Factorial: $n! = n \times (n-1) \times \cdots \times 2 \times 1$

```
# Python
import math

math.factorial(23)
```

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

Reminder: Combinatorics - Combinations



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

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

Read as "n choose r" because it is the number of ways can we choose r out of n objects?

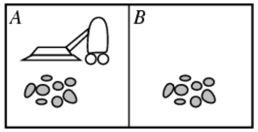
Special case for $r = 2$: $\binom{n}{2} = \frac{n(n-1)}{2}$

```
# 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

Example: What is the State Space Size?



Dirt

- **Permutation:** A and B are different rooms, order does matter!
- **With repetition:** Dirt can be in both rooms.
- There are 2 options (clean/dirty)

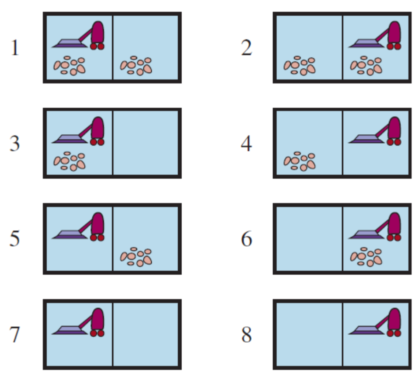
$\rightarrow 2^2$

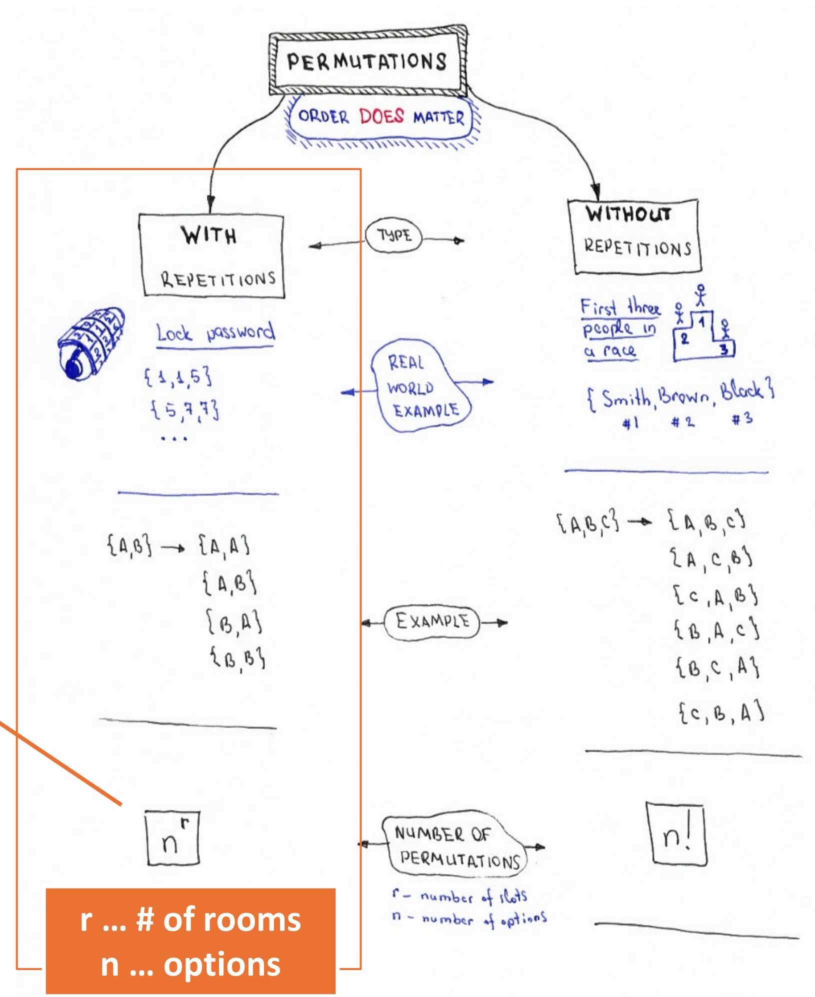
Robot location

- Can be in 1 out of 2 rooms.

$\rightarrow 2$

Total: $n = 2 \times 2^2 = 2^3 = 8$

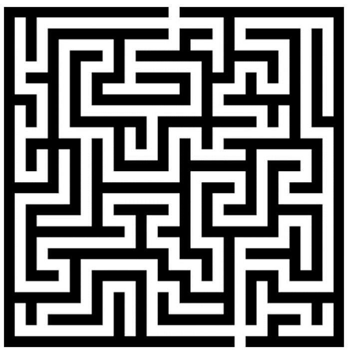




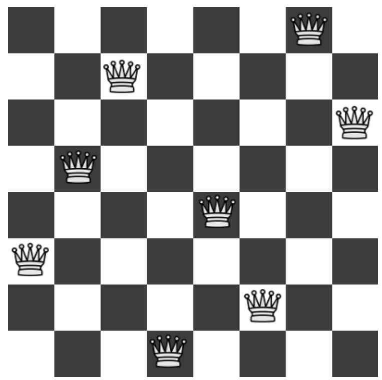
r … # of rooms
n … options

Examples: What is the State Space Size?

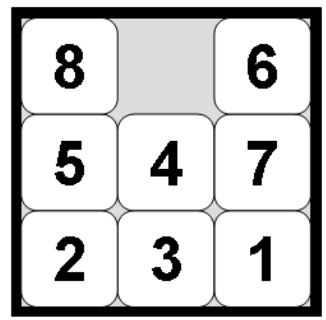
Often a rough upper limit is sufficient to determine how hard the search problem is.



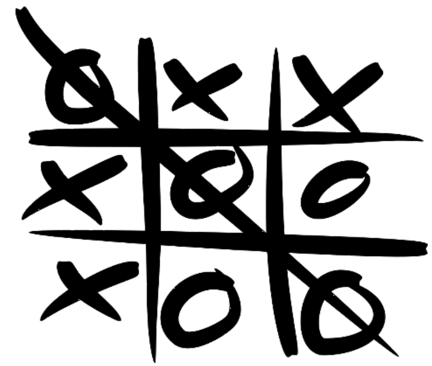
Maze



8-queens problem



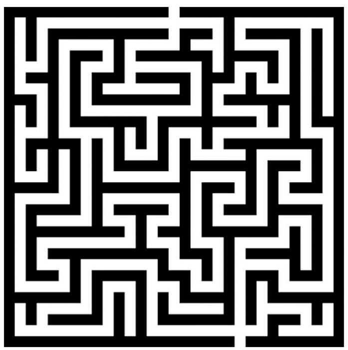
8-puzzle problem



Tic-tac-toe

Examples: What is the State Space Size?

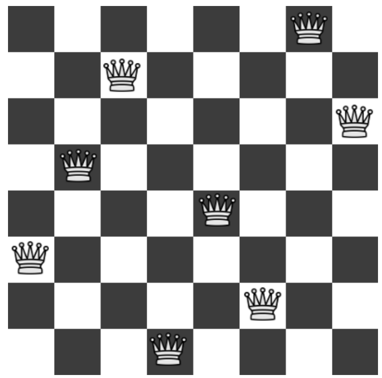
Often a rough upper limit is sufficient to determine how hard the search problem is.



Maze

Positions the agent can be in.

n = Number of white squares.



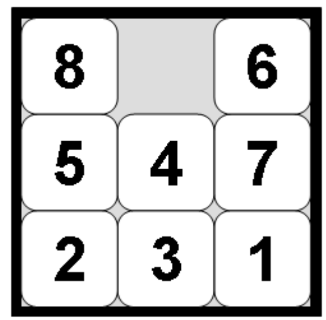
8-queens problem

Action: Move one queen at a time

All arrangements with 8 queens on the board.

$n < 2^{64} \approx 1.8 \times 10^{19}$

We only have 8 queens:
$n = \binom{64}{8} \approx 4.4 \times 10^9$



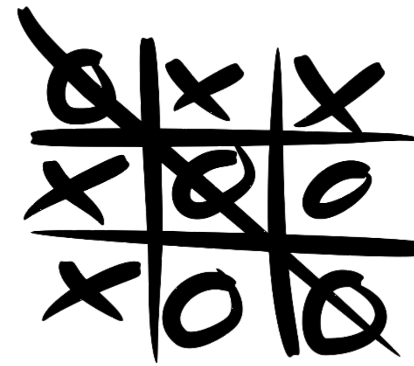
8-puzzle problem

All arrangements of 9 elements.

$$n \leq 9!$$

Half is unreachable:

$$n = \frac{9!}{2} = 181{,}440$$



Tic-tac-toe

All possible boards.

$$n < 3^9 = 19{,}683$$

Many boards are not legal (e.g., all x's)

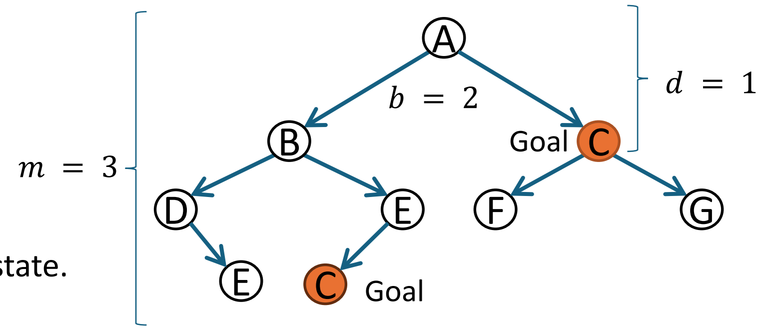
The actual number can be obtained by a depth-first traversal of the game tree.

Estimating the Search Tree Size

- Instead of estimating the state space size, it is often more useful to estimate the number of searched nodes in the search tree.
- This is especially important when **redundant paths are not eliminated,** where one state can be represented by multiple nodes.

- We can base the estimation on the search problem description:
  - initial state
  - Actions
  - transition function.

- Used metrics are:
  - b: maximum branching factor of the search tree
    max. number of available actions.
  - m: maximal tree depth
    length of the longest path with loops removed.
  - d: depth of the optimal solution
    min. length of the path from the initial state to a solution state.



- The number of searched nodes is then a function of b, m and d.

$$\boxed{n = f(b, m, d) \Rightarrow O(f(b, m, d))}$$

Example: What is the Search Complexity?

- b: maximum branching factor = max. number of available actions?

  3

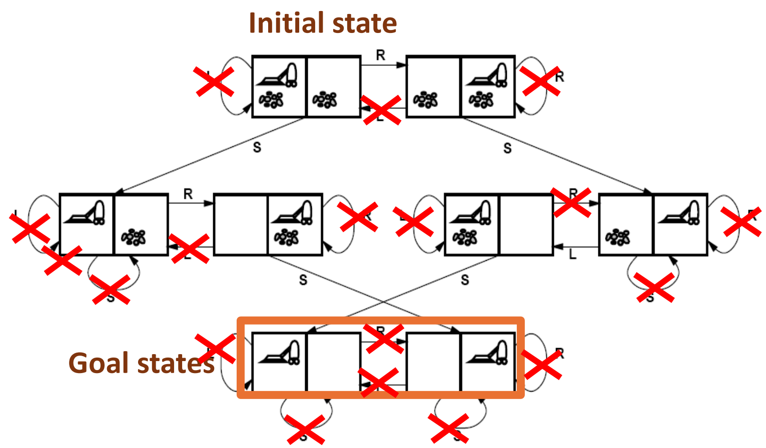
- m: the number of actions in longest path? Without loops!

  4

- d: min. depth of the optimal solution?

  3

State Space with Transition Model

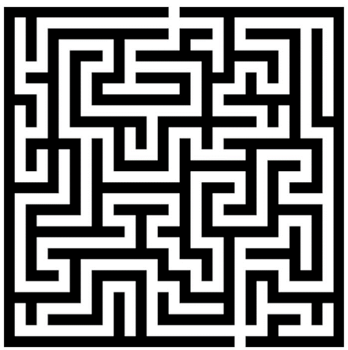


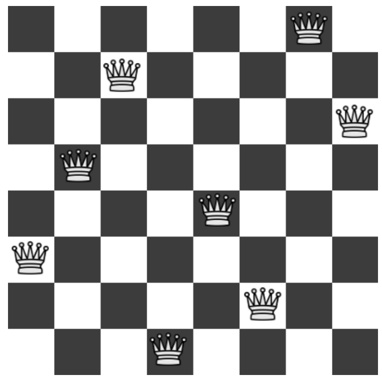
Make sure it is a tree!

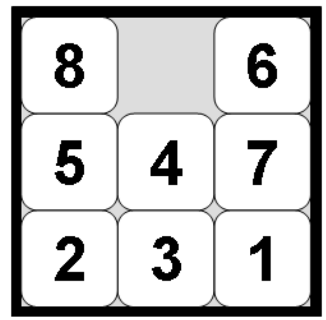
Examples: What is the Search Complexity?

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

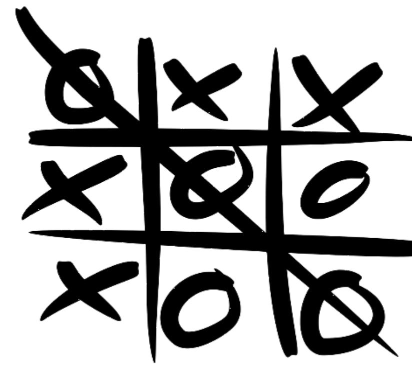
Often a rough upper limit is sufficient to determine how hard the search problem is.


Maze


8-queens problem



8-puzzle problem

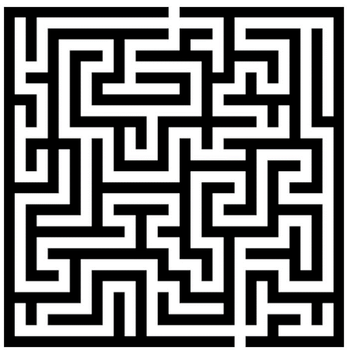

Tic-tac-toe

$b =$
$m =$
$d =$

Examples: What is the Search Complexity?

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

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

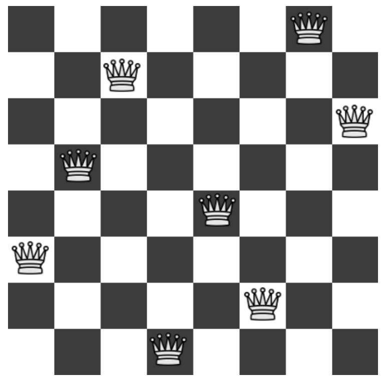


Maze

$b \ = 4$ actions

$m =$ longest path to the goal or a dead end (bounded by $x \times y$)

$d \ =$ shortest path to the goal (bounded by $x \times y$)



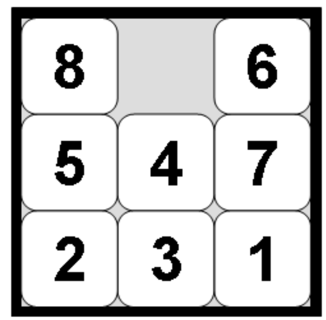
8-queens problem

Action: Move one queen at a time

$b \ = 8 \times (64 - 8) = 448$

$m =$ We may have to try all: $\binom{64}{8} \approx 4.4 \times 10^9$

$d \ =$ move each queen in the right spot = 8

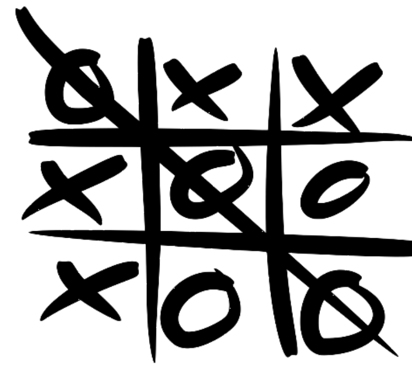


8-puzzle problem

$b \ = 4$ actions to move the empty tile.

$m =$ Try all $O(9!)$

$d \ =$??? We need to solve the problem to know.



Tic-tac-toe

$b \ = 9$ actions for the first move.

$m = 9$

$d \ = 9$ (if both play optimal)

Things to Remember

- Time and space complexity of search algorithms determine if we can implement a tree search solution!

- We can estimate the complexity by the following methods:
  1. Estimate the **state space size** using a factored state representation
  2. Estimate the **search tree size** using branching factor and tree depth.

- If each note represents exactly one state then both estimates will be equivalent. We will learn soon when this is or is not the case.

- We typically calculate an estimate of the actual size, or we use the Big-O notation if we are interested in how the problem scales with size.

Uninformed Search

Breadth-First Search

Uninformed Search Strategies

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

It can only use

- the labels of the **atomic states** and
- the transition function.

Idea: blindly search, following a simple strategy, until the goal state is reached.

Search strategies:

- **Breadth-first search strategy**: BFS and uniform-cost search
- **Depth-first search strategy**: DFS and Iterative deepening search

Breadth-First Search (BFS)

Expansion rule: Expand shallowest unexpanded node in the frontier (=**FIFO**).

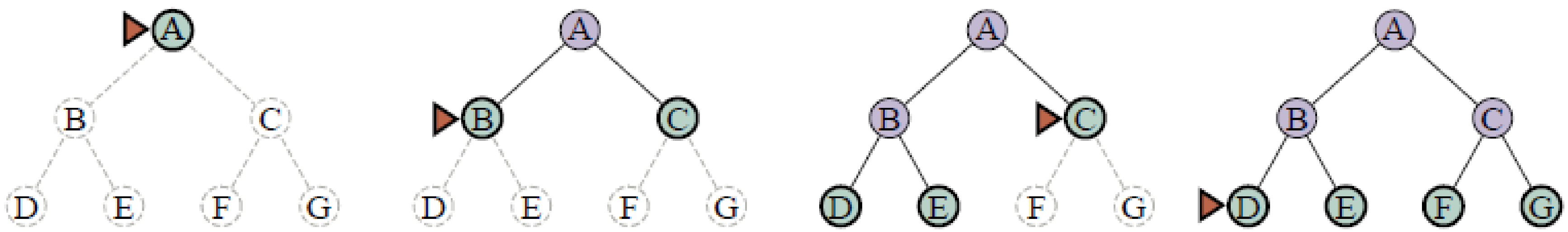


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

Data Structures

- **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 and redundant path checking).
- Builds a **complete tree** with links between parent and child.

Implementation: Breadth-First Search

```
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
```

Expand adds the next level below node to the frontier.

reached makes sure we do not visit nodes twice (e.g., in a cycle or other redundant path). Fast lookup is important.

Implementation: Expanding the Search Tree

- AI tree search creates the search tree while searching.
- The EXPAND function tries all available actions in the current node using the **transition function** (RESULTS).
- It returns a **list of child nodes** for the frontier.

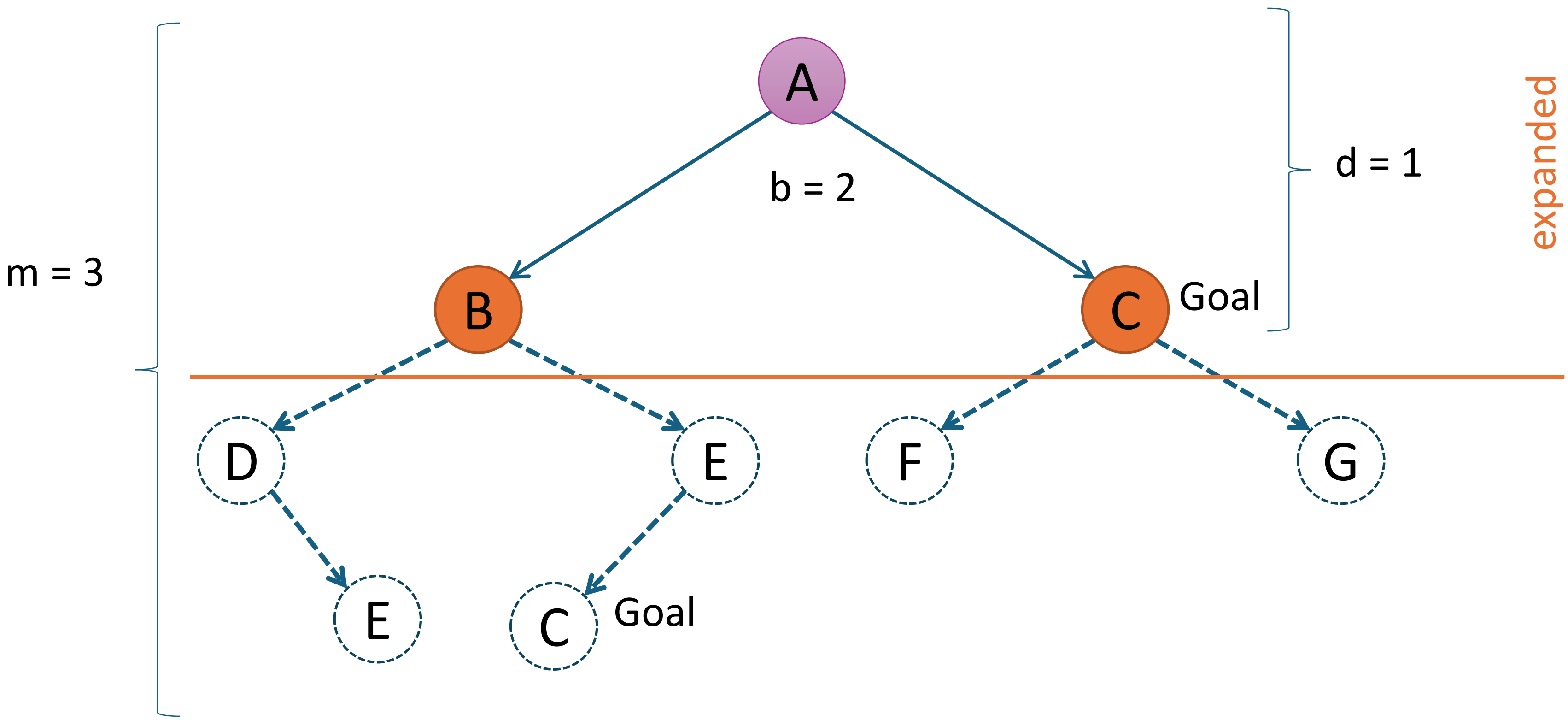
```
function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Transition function

Yield (generator function) can also be implemented by returning a list of nodes.

Time and Space Complexity Breadth-First Search

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



All paths to the depth of the goal are expanded. The search tree size is

$$1 + b + b^2 + \ldots + b^d \Rightarrow O(b^d)$$

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: $O(b^d)$
- **Space?**
  Stored nodes: $O(b^d)$

Note:

- In AI, the large space complexity is usually a bigger problem than time!

Uniform-cost Search (= Dijkstra's Shortest Path Algorithm)

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

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

- **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?**
  Expands all nodes with path cost $c \leq C^*$ (cost of optimal solution) leading to $O(b^{1+C^*/\varepsilon})$ for the number of nodes.
  Note: This can be greater than BFS's $O(b^d)$: 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]

Implementation: Best-First Search Strategy

Note: This generalizes Breadth-First-Search

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

```
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
```

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

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

See BFS for function EXPAND.

Uninformed Search

Depth-First Search

Depth-First Search (DFS)

- **Expansion rule**: Expand deepest unexpanded node in the frontier (last added).
- **Frontier**: **stack** (LIFO)
- **No reached data structure:** forgets completely explored subtrees.
- **Needs Cycle checking:** don't expand nodes that are already in the current path to the root node.
- **Cannot avoid redundant paths:** Leads to multiple nodes representing the same state and replicated work.

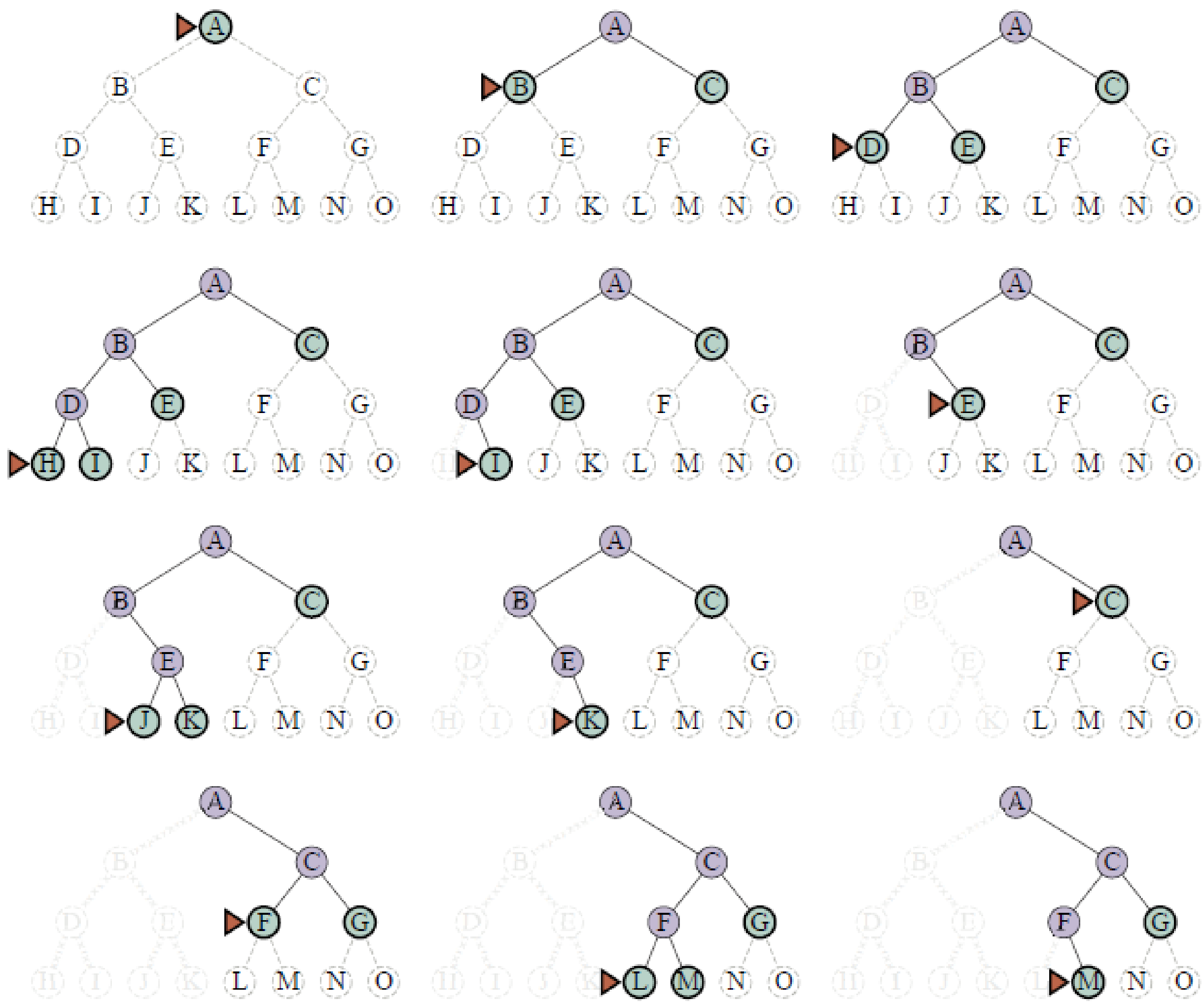


Figure 3.11 A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint 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, just taking the last element from the frontier (LIFO). However, to reduce the space complexity to $O(bm)$, **no reached data structure can be used!**
- Options:
  - **Iterative implementation**: Build the tree, and abandoned branches are removed from memory. Cycle checking is only done against the current path. This is similar to Backtracking search.
  - Recursive implementation: Cycle checking is an issue because the current path is stored in the function call stack, which is not accessible to the function. An additional data structure that contains the nodes in the current path can be used.

DFS uses $\ell = \infty$

```
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
```

Memory management: remove nodes for abandoned branches here!

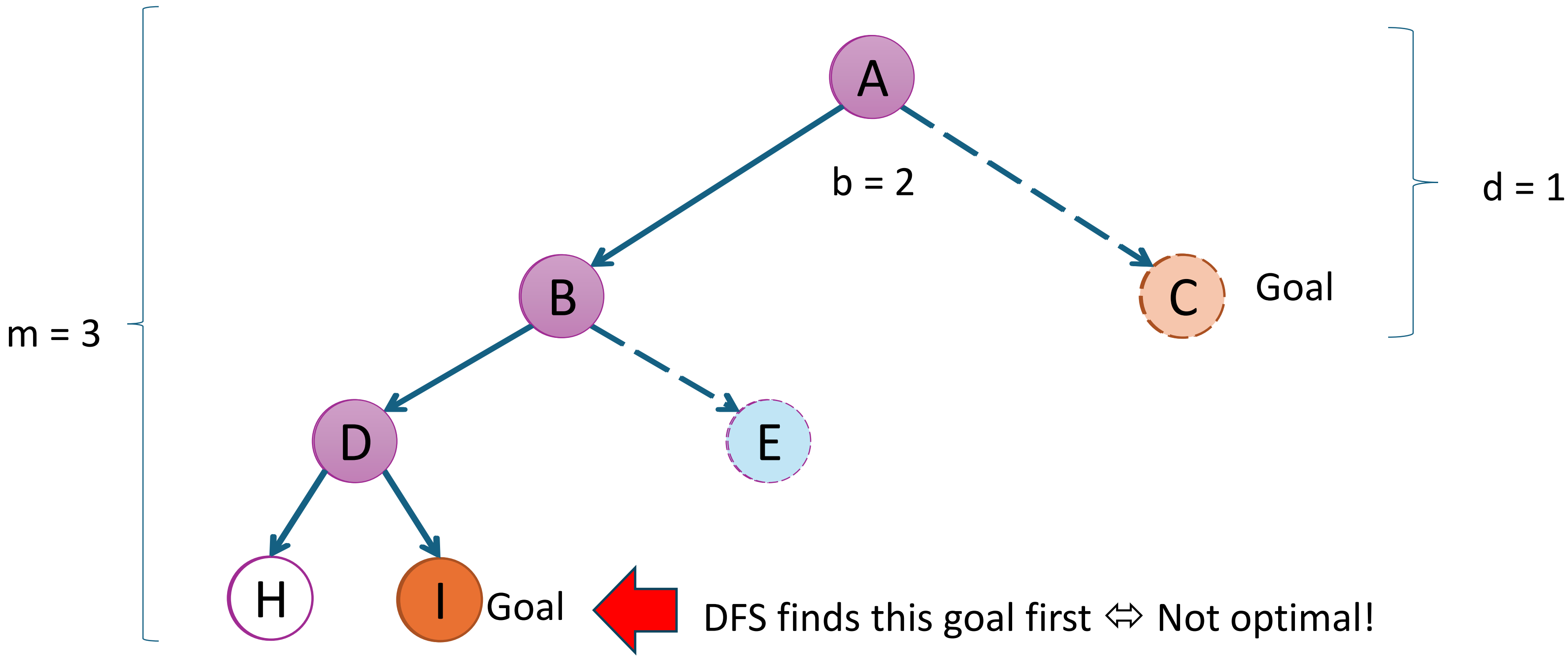
Cycles: Prevent cycles by checking against the current path. We also need to ensure that the frontier does not contain the same state more than once.

Redundant paths: We cannot prevent other redundant paths.

See BFS for function EXPAND.

Time and Space Complexity
Depth-First Search

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



DFS finds this goal first ⇔ Not optimal!

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

Properties of Depth-First Search

- **Complete?**
  - In finite search spaces, cycles are avoided by checking for repeated states along the path.
  - Incomplete in infinite search spaces.
- **Optimal?**

  No – returns the first solution it finds.

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

- **Time?**

  The worst case is to reach a solution at maximum depth *m* in the last path: $O(b^m)$
  Terrible compared to BFS if $m \gg d$.

- **Space?**

  $O(bm)$ is **linear in max. tree depth m** which is very good but only achieved if **no reached data structure and memory management** is used!
  Cycles can be broken but redundant paths cannot be checked.

Iterative Deepening Search (IDS)

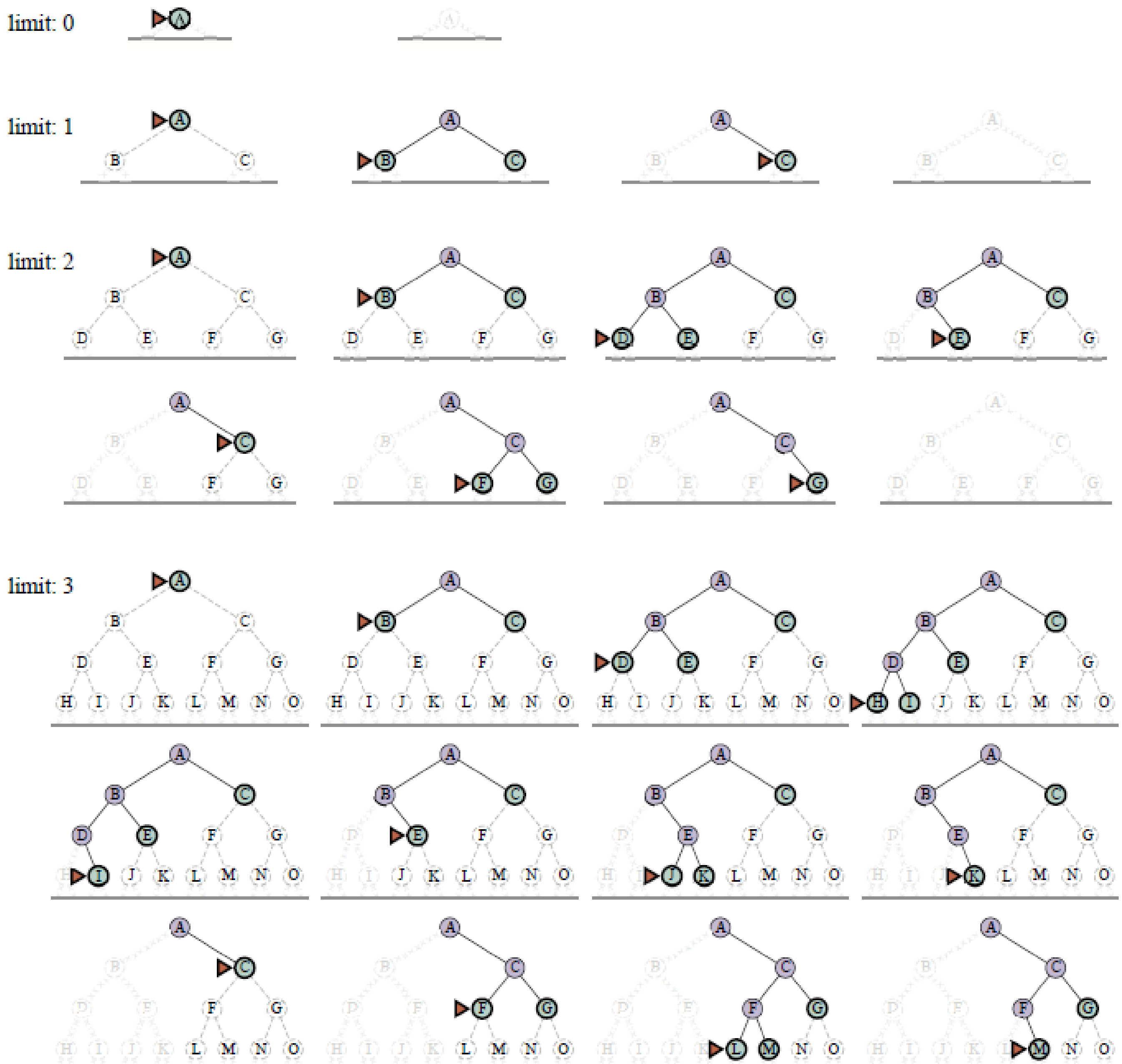
Can we

- **get DFS's good memory footprint,**
- **avoid infinite cycles, and**
- **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 goal not found, do a DFS searching for a path of length 2
4. If goal not found, 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
```

See BFS for function EXPAND.

Properties of Iterative Deepening Search

- **Complete?**
  Yes

- **Optimal?**
  Yes, if step cost = 1 (like BFS)

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

- **Space?**
  ***O*(*bd*) ⇔ linear space. Even less than DFS since $\boldsymbol{m \leq d}$.** 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 **much better space complexity** for worse run time.

This makes IDS/DFS the workhorse of AI.

Informed Search

Introduction

Informed Search

AI search problems typically have a very large search space. We would like to improve efficiency by **expanding as few nodes as possible.**

Idea: The agent can use **additional information** in the form of "hints" about what promising states are to explore first. These hints are derived from

- **information** the agent has (e.g., a map with the goal location marked) or
- **percepts** coming from a sensor (e.g., a GPS sensor and coordinates of the goal).

Method: The agent uses a **heuristic function** $\boldsymbol{h(n)}$

- to rank nodes in the frontier based on the additional information, and
- to select the most promising node in the frontier for expansion using the **best-first search** strategy.

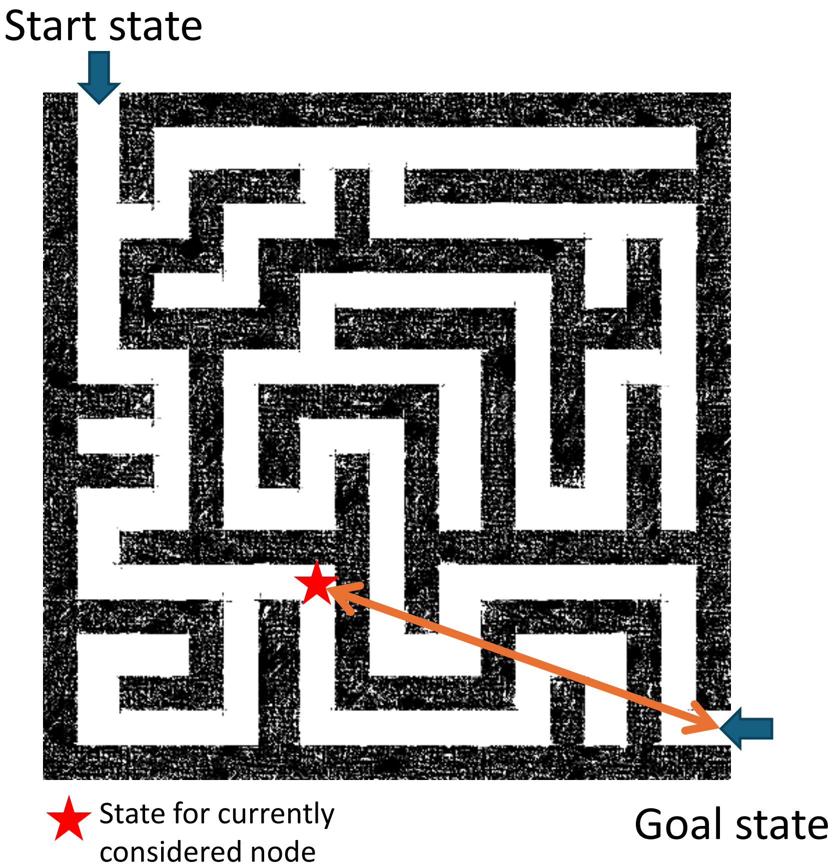
Discussed algorithms:

- Greedy best-first search
- A* search

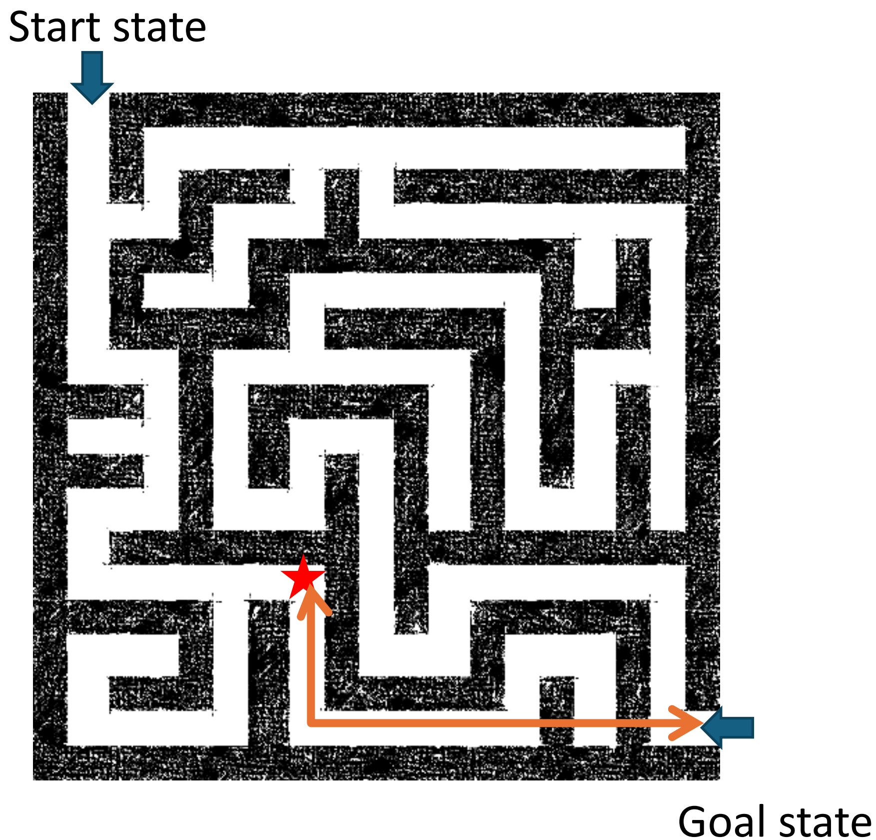
Heuristic Function

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

Euclidean distance



Manhattan distance



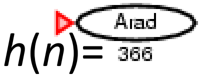
Heuristic for the Romania Problem

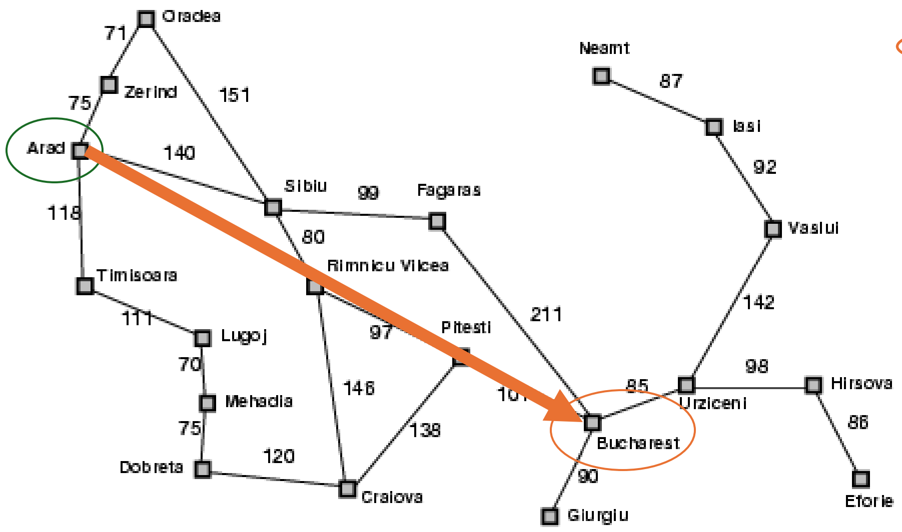
Use the map for hints: Estimate the driving distance from Arad to Bucharest using a straight-line distance on the map.

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*)

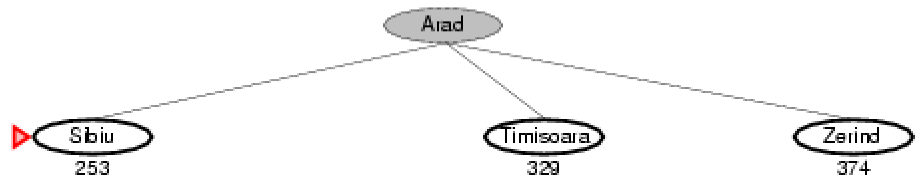


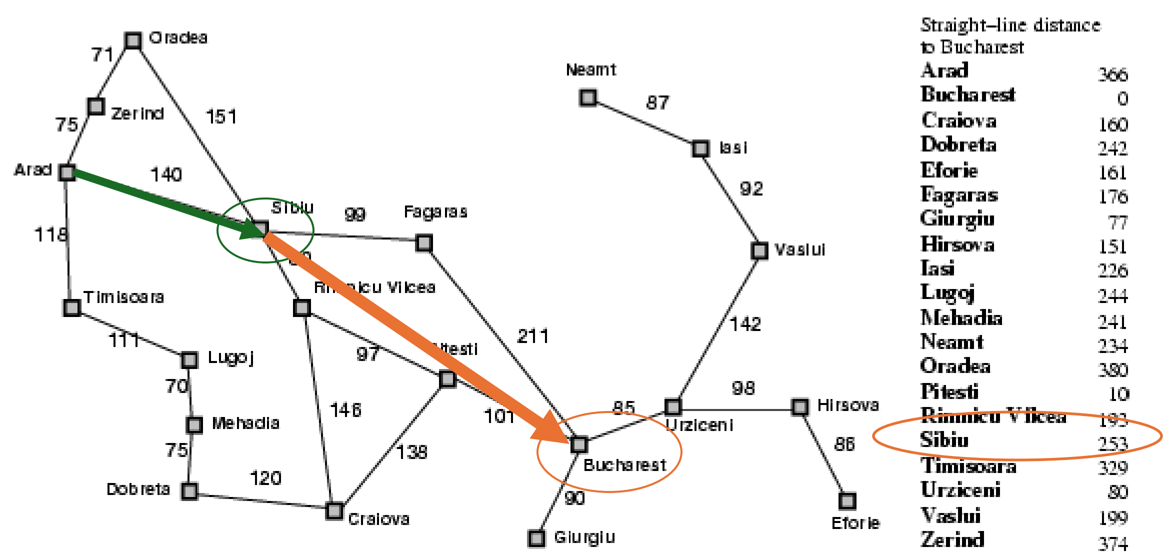


Straight–line distance to Bucharest

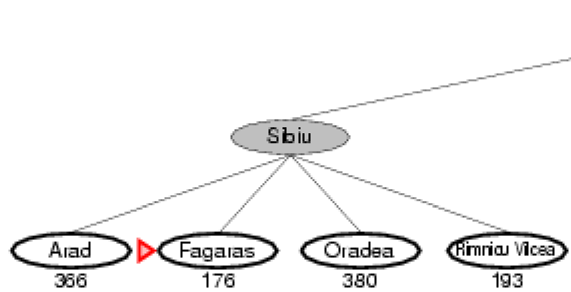
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

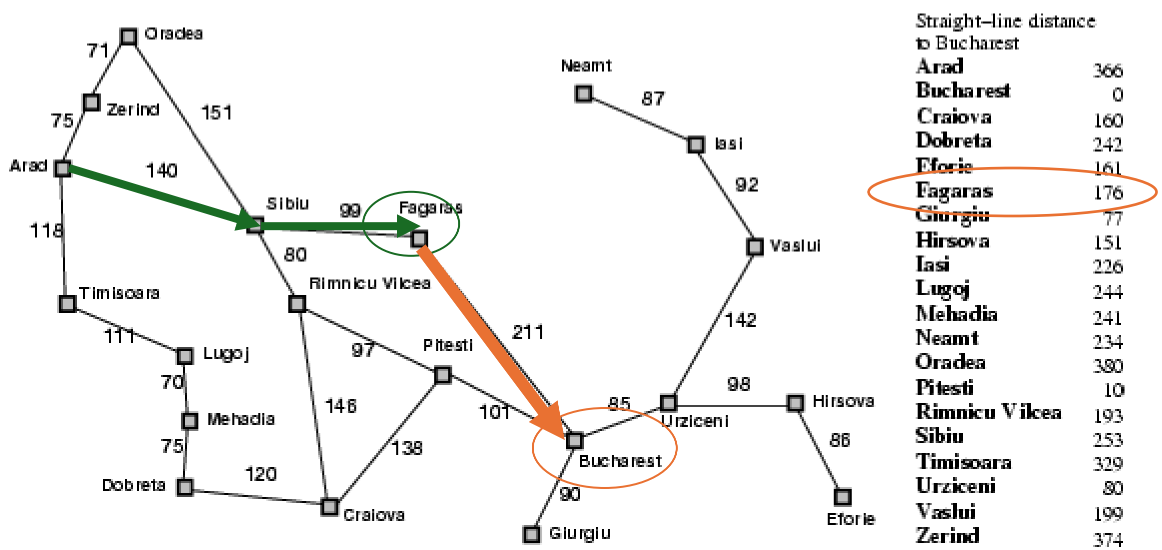
Greedy Best-First Search Example



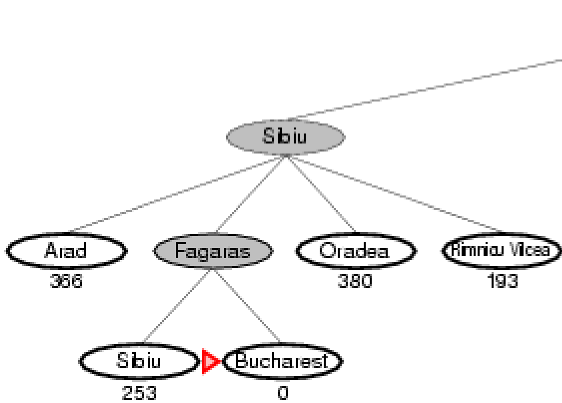


Greedy Best-First Search Example

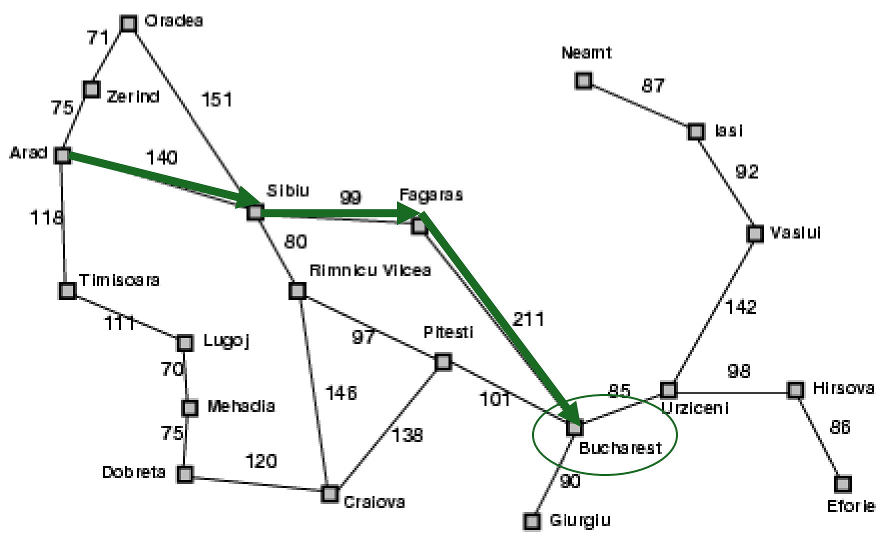




Greedy Best-First Search Example



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

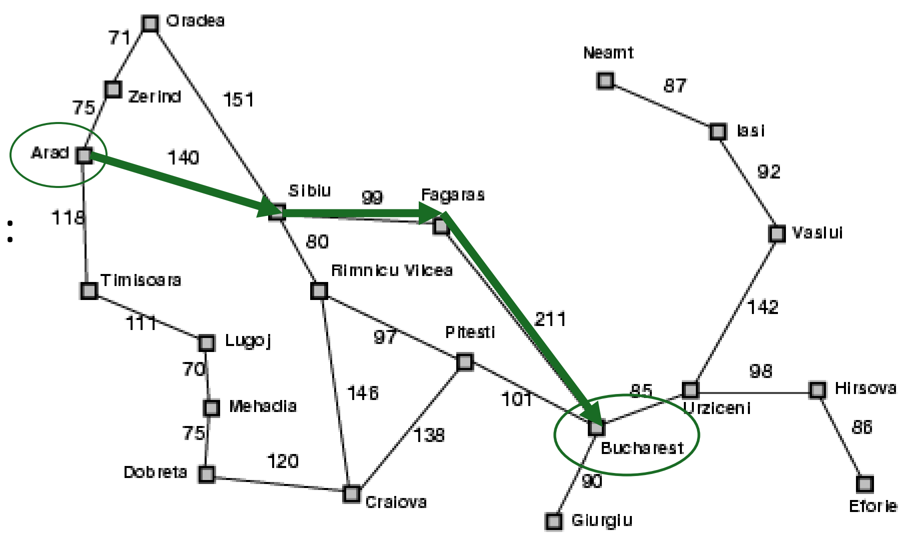
Properties of Greedy Best-First Search

- **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 $f(n) = h(n)$

Implementation of Greedy Best-First Search

Heuristic $h(n)$ so we expand the node with the lowest estimated cost

```
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
```

The order for expanding the frontier is determined by *f(n)*

See BFS for function EXPAND.

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 we only expand a single path.

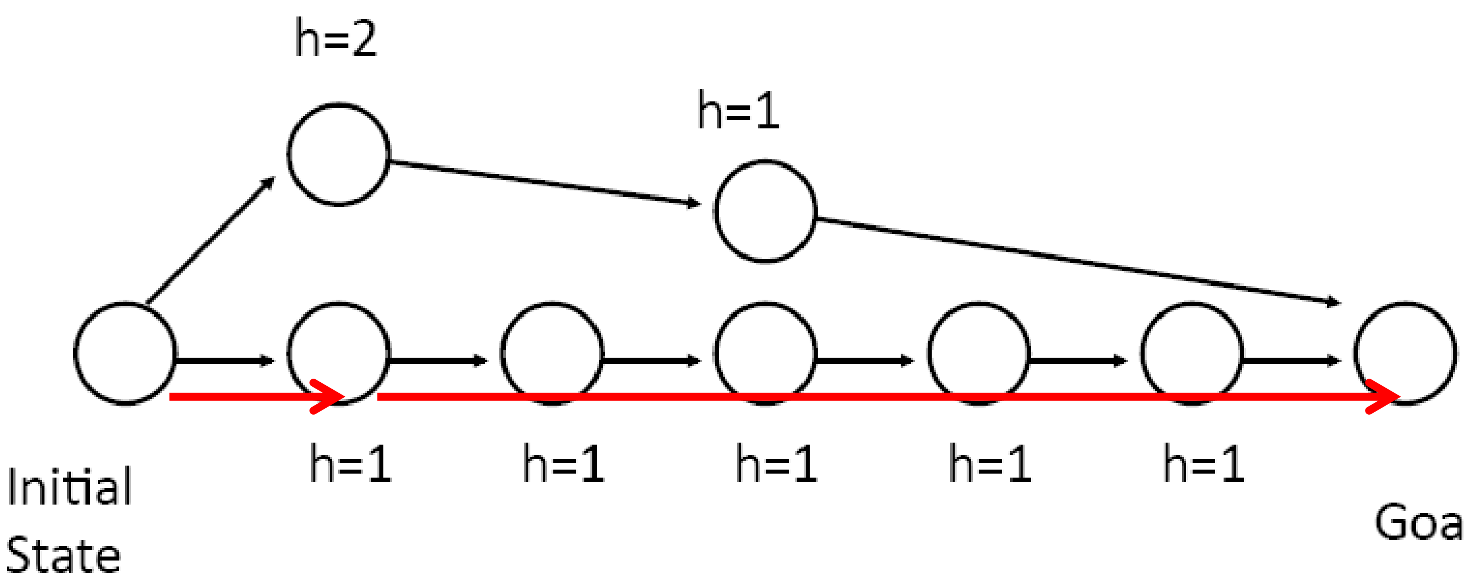
- **Space?**
  Same as time complexity.

Informed Search

A* Search

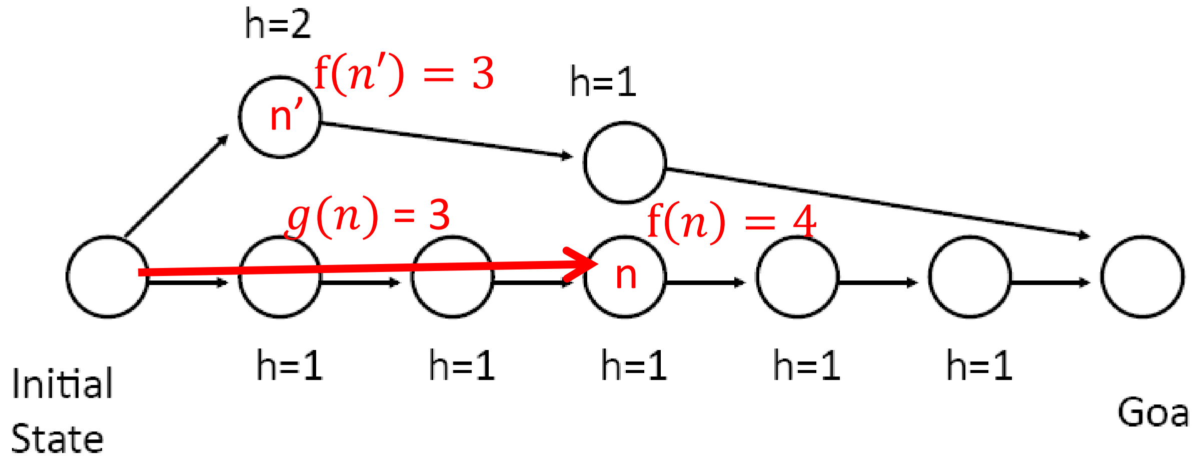
The Optimality Problem of Greedy Best-First search

Greedy best-first search only considers the estimated cost to the goal.



$h = 1$ is better than $h = 2$.
Greedy best-first will go this way and never reconsider!

A^* Search



- **Idea**: Take the cost of the path to n called $g(n)$ into account to avoid 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)

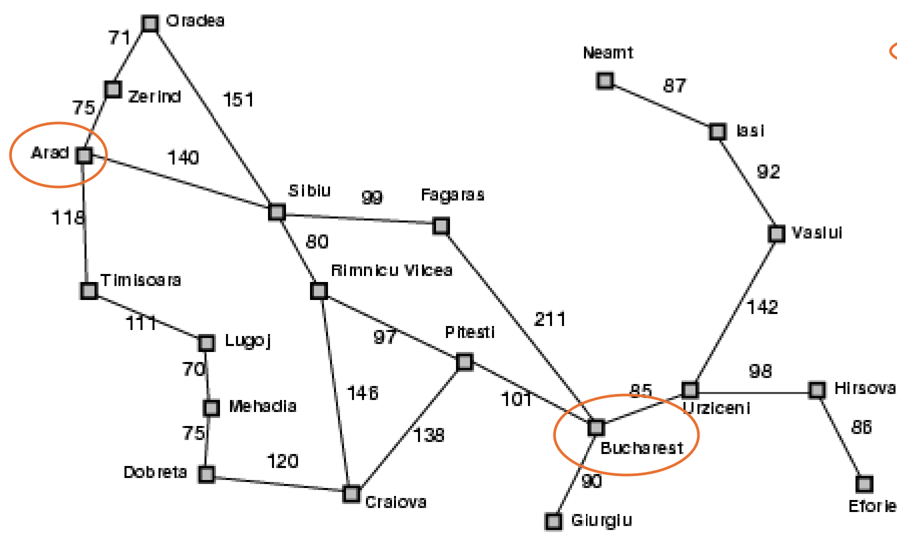
- The agent in the example above will stop at n with $f(n) = 3 + 1 = 4$ and chose the path up with a better $f(n') = 1 + 2 = 3$.

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

A* Search Example

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

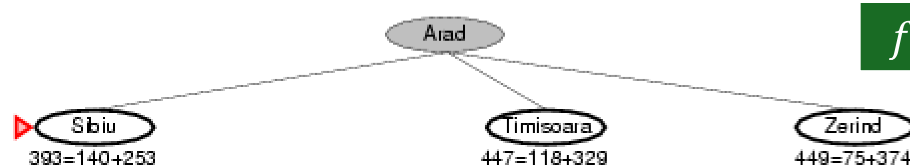
$f(n) = g(n) + h(n) =$ 



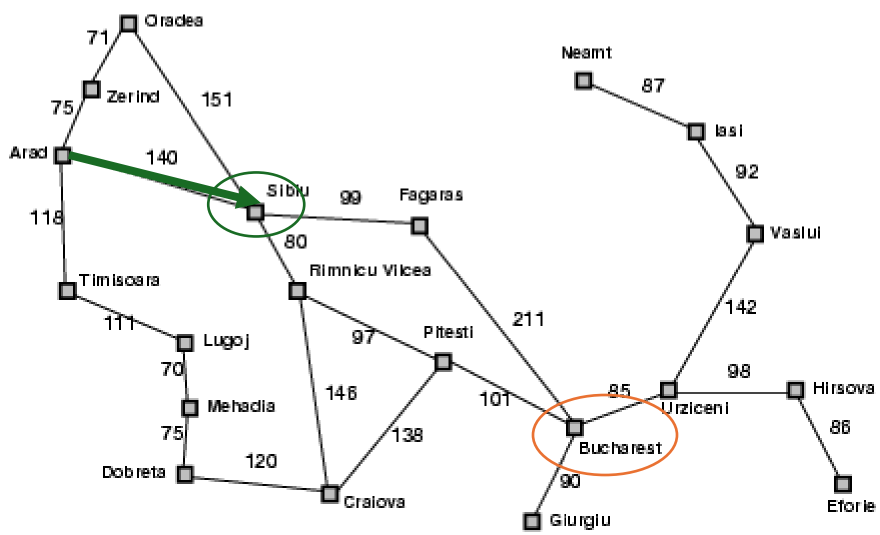
$h(n)$

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



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



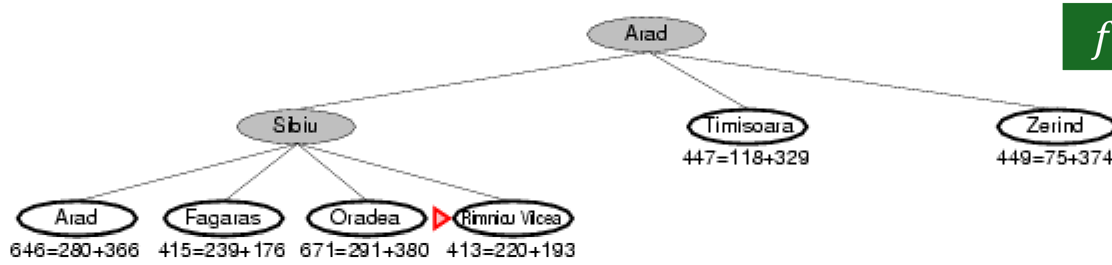
$h(n)$

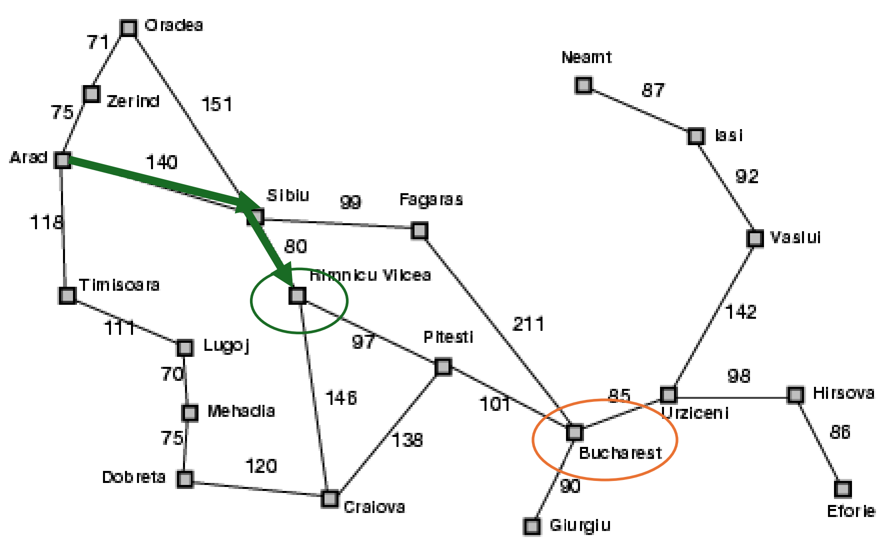
Straight–line distance to Bucharest

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

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





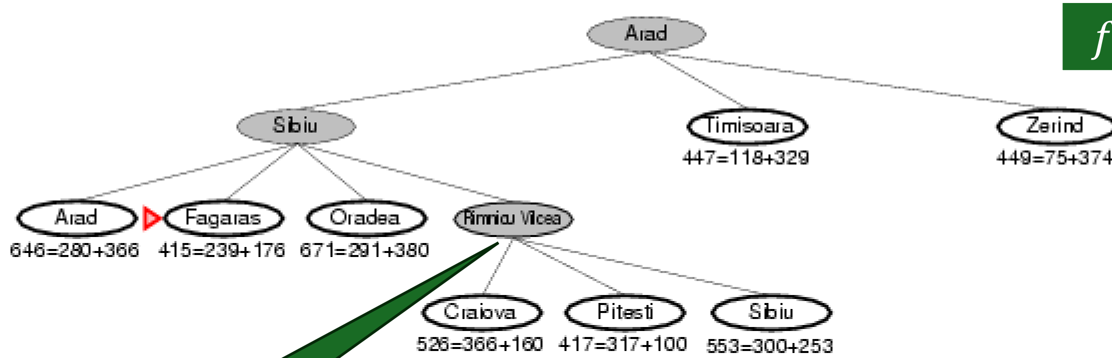
$h(n)$

Straight–line distance to Bucharest

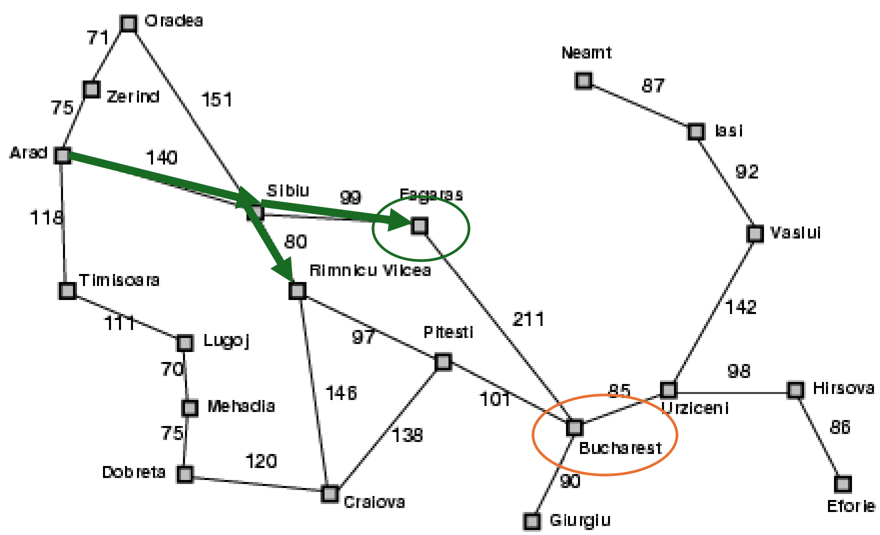
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Search Example

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



Reconsiders Rimnicu Vilcea because Fagaras may have a shorter total cost to Bucharest.



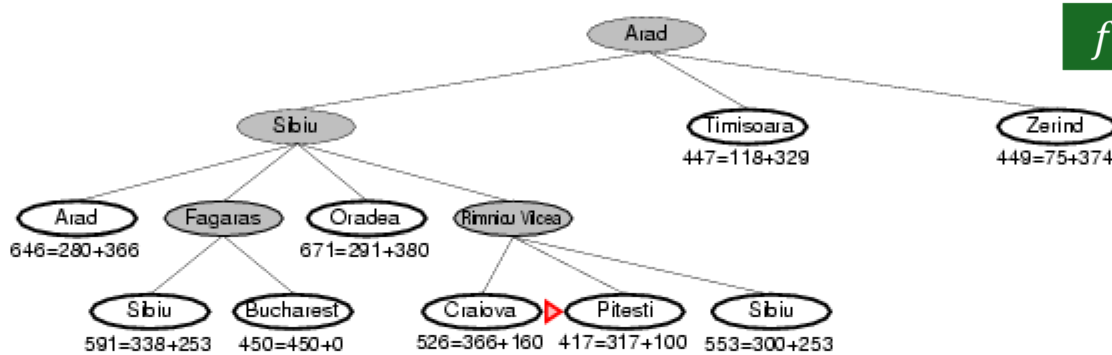
$h(n)$

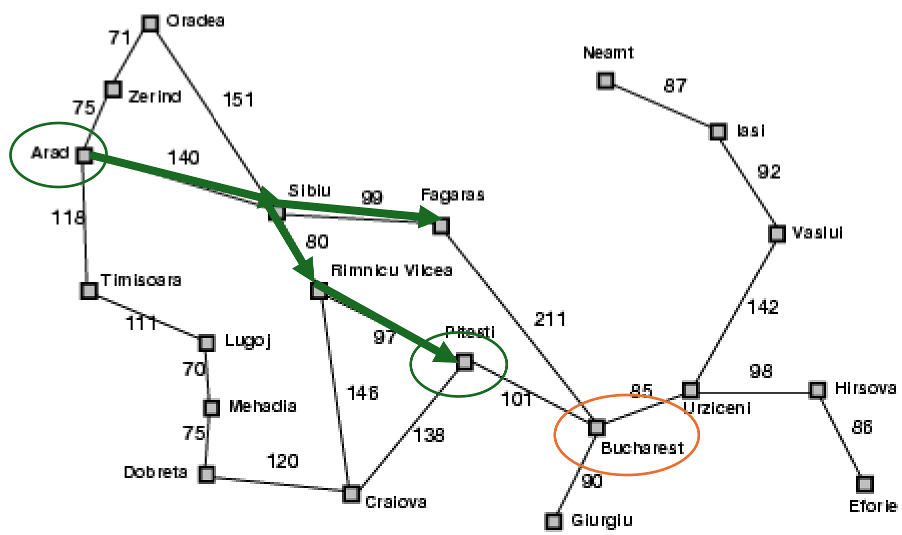
Straight–line distance to Bucharest

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

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





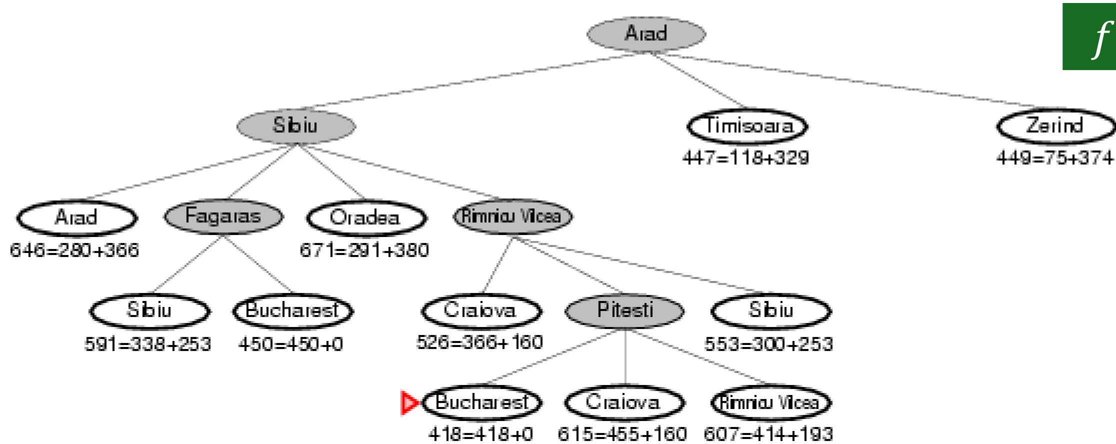
$h(n)$

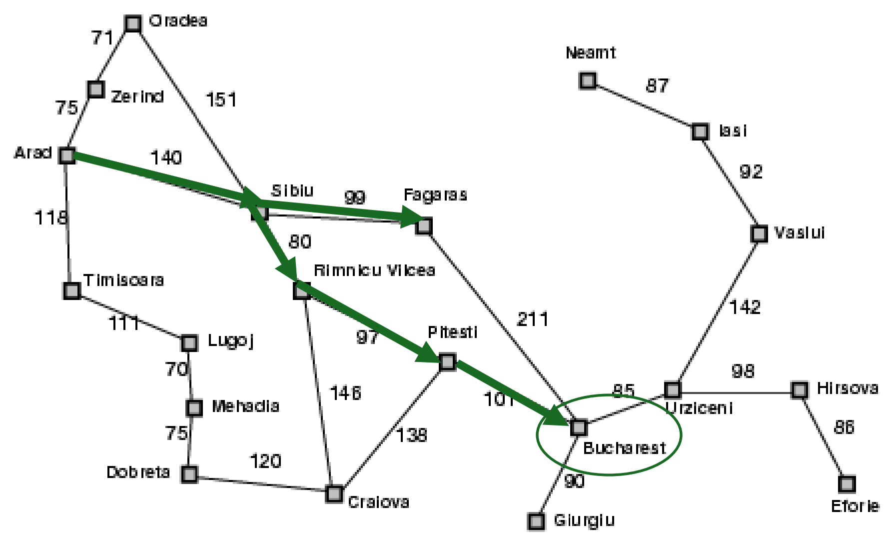
Straight–line distance to Bucharest

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

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



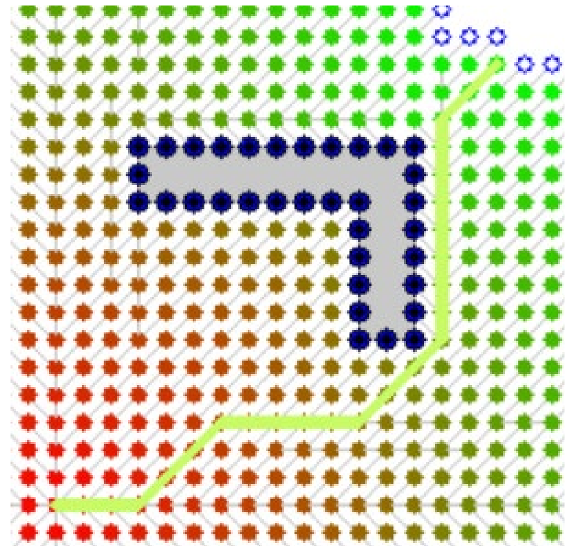


$h(n)$

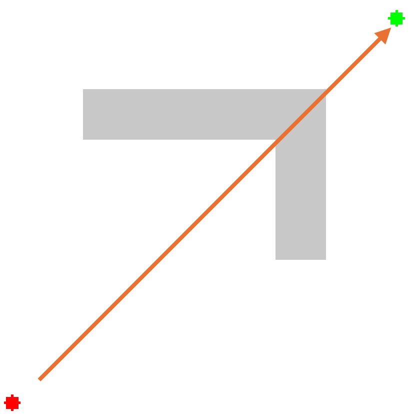
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

BFS vs. A* Search

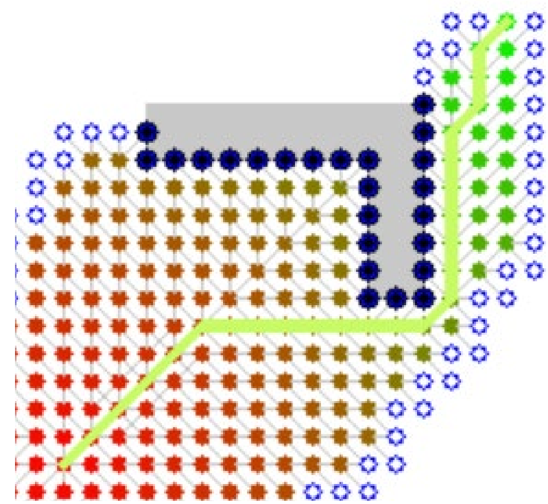
BFS



A*



A*



Source: [Wikipedia]()

A* Search expands
fewer nodes than BFS!

Implementation of A* Search

Path cost to n + heuristic from n to goal = estimate of the total cost

$$g(n) + h(n)$$

```
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
```

The order for expanding the frontier is determined by $f(n)$

See BFS for function EXPAND.

Optimality: Admissible Heuristics

Definition: A heuristic h 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 is admissible, A^* is optimal.

Guarantees of A* Search

A* is **optimally efficient**

No other tree-based search algorithm that employs the same heuristic can expand fewer nodes and still guarantee the optimal solution.

Proof: Any algorithm that does not expand all nodes with $f(n) < C^*$ (the lowest cost of going to a goal node) cannot be optimal. It risks missing the optimal solution.

Properties of A*Search

- **Complete?**
  Yes
- **Optimal?**
  Yes
- **Time?**
  Number of nodes for which $f(n) \leq C^*$ in the worst case $O(b^d)$ like BFS.
- **Space?**
  Same as time complexity. This is often too high unless a very good heuristic is know.

Iterative-Deepening A* Search – IDA*

- **Idea**: A* search without a reached data structure.
- **Remember**: Regular IDA is uninformed and increases the cutoff by one after each iteration.

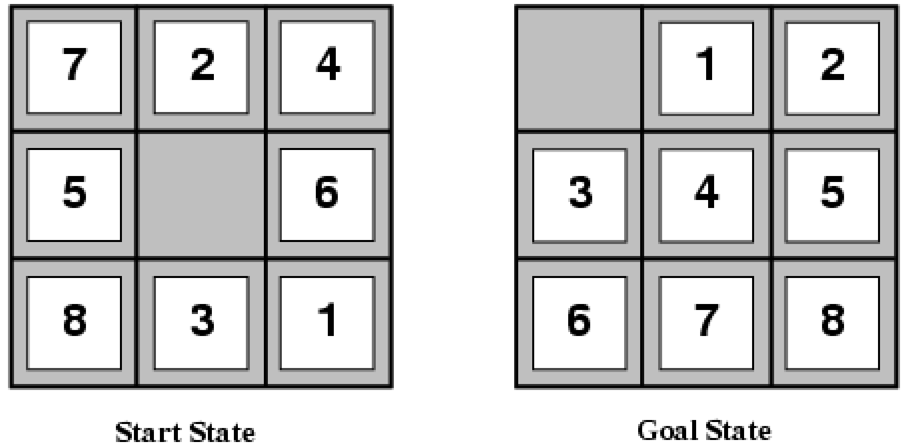
- IDA* uses the cost $f = g + h$ of a node as the cutoff. In each iteration, the cost cutoff increases slightly. It is optimal if steps are small and h is admissible.
- **Issues**:
  - By how much to increase the cutoff in each iteration.
  - Rebuilds the tree many times.

- **Other memory-bounded variants of A* search**:
  - Recursive best-first search (RBFS) adds a f-limit to the depth-first search behavior of best-first search.
  - Simplified memory-bounded A* (SMA*) performs A* till the memory is full and then drops the worst (highest f) leaf node from memory. It can rebuild the node later if needed.

Informed Search

Designing Heuristics

Designing Heuristic Functions



Example heuristics for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_1(start) \ = \ 8$

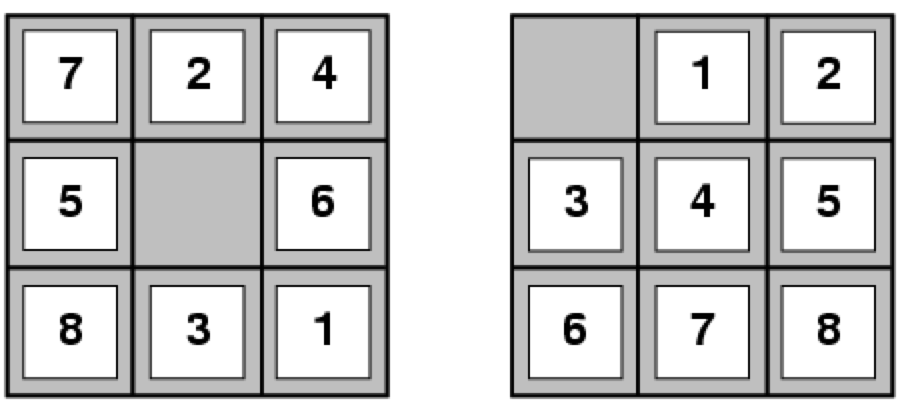
- $h_2(n)$ = total Manhattan distance (number of squares from the desired location of each tile)
- $h_2(start) \ = \ 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 \ = \ 18$

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. I.e., the true cost is never smaller.**
- What relaxation is used by h_1 and h_2?
  - h_1: If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution.
  - h_2: 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

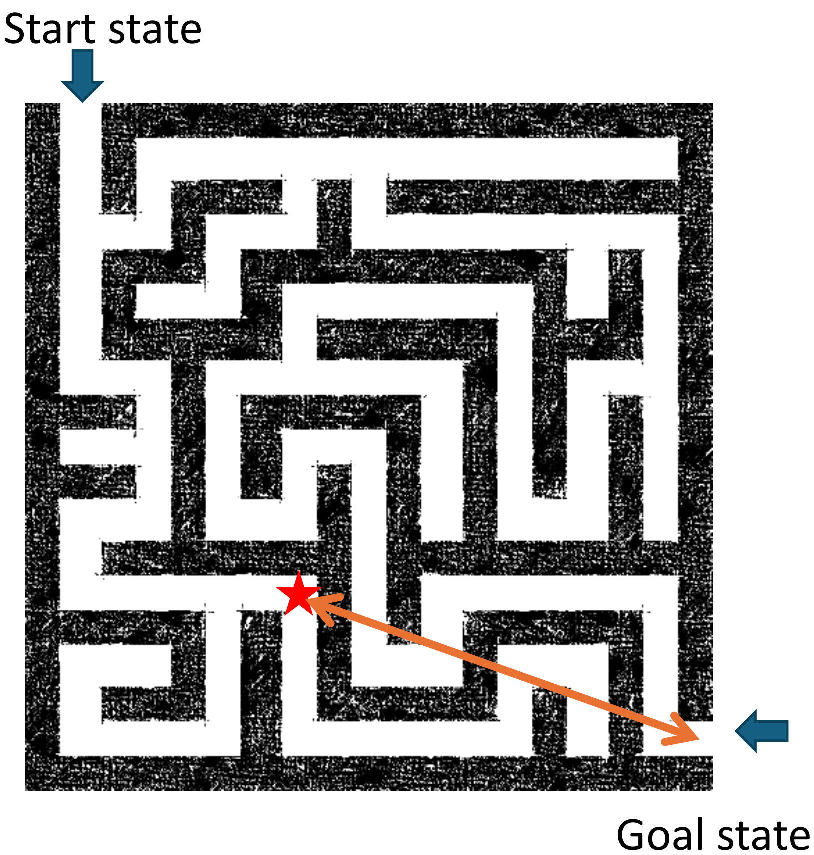
$$h_1(start) = 8$$

$$h_2(start) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$$

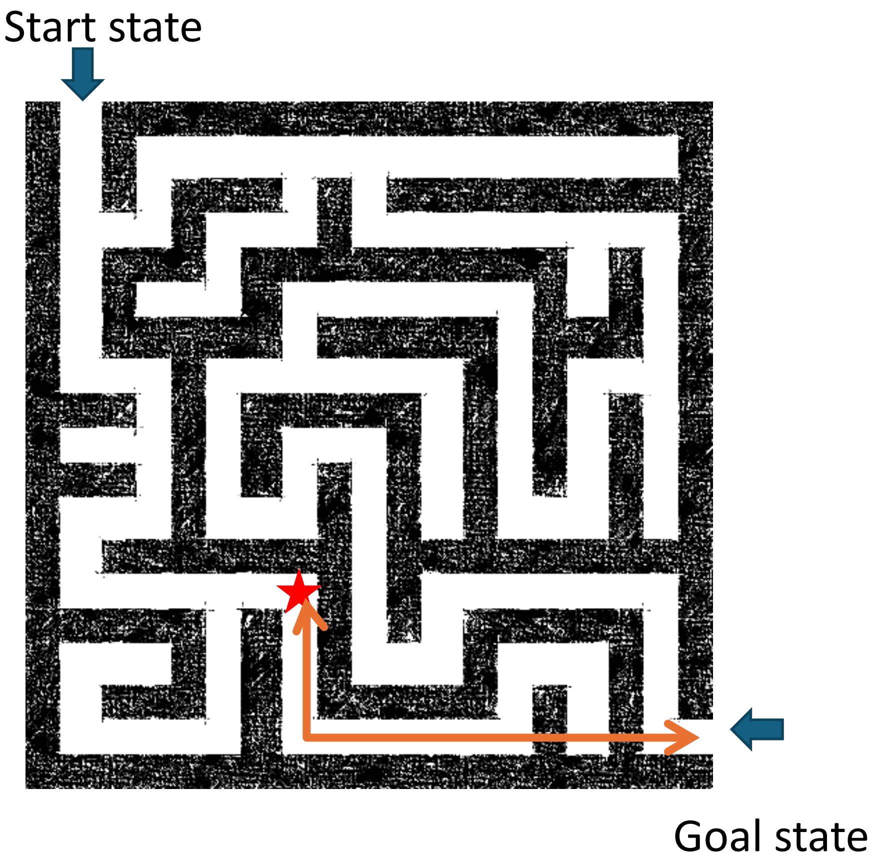
Heuristics from Relaxed Problems

What relaxations are used in these two cases?

Euclidean distance

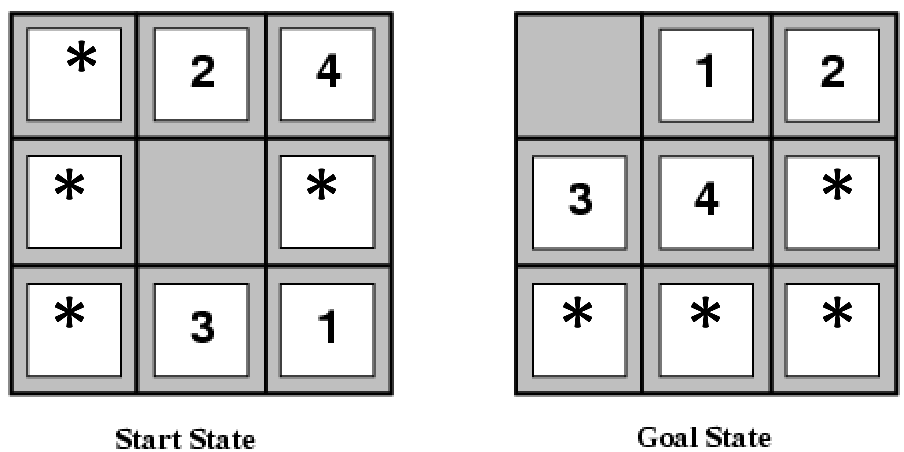


Manhattan distance



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.
- Solutions for subproblems are an admissible heuristic.
- Calculation:
  - Very small subproblems are often easy to solve.
  - We can precompute and save the exact solution cost for every or many possible subproblem instances – ***pattern database***.



Dominance: What Heuristic is Better?

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

Is h_1 or h_2 better for A* search?

- A* search expands every node with
  $f(n) < C^* \Leftrightarrow h(n) < C^* - g(n)$
- h_2 is never smaller than h_1. A* search with h_2 will expand less nodes and is therefore better.

Combining Heuristics

- Suppose we have a collection of admissible heuristics $h_1, h_2, \ldots, h_m$, but none of them dominates the others.
- Combining them is easy:

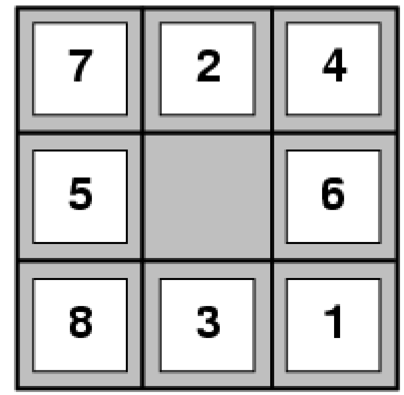
$$h(n) = \max\{h_1(n), h_2(n), \ldots, h_m(n)\}$$

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

Example: Effect of Information in Search

Typical search costs for the 8-puzzle

- State space: $\frac{9!}{2} = 1{,}811{,}440$ states
- Problem with solution at depth $d = 12$
  IDS = 3,644,035 nodes
  A*(h_1) = 227 nodes
  A*(h_2) = 73 nodes
- Solution at depth $d = 24$
  IDS ≈ 54,000,000,000 nodes
  A*(h_1) = 39,135 nodes
  A*(h_2) = 1,641 nodes



$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance

Contains many redundant paths which IDS cannot break!

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. I.e., expanding fewer nodes.

- We could use inadmissible heuristics in A* search (e.g., by multiplying $h(n)$ with a factor W) 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!**

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

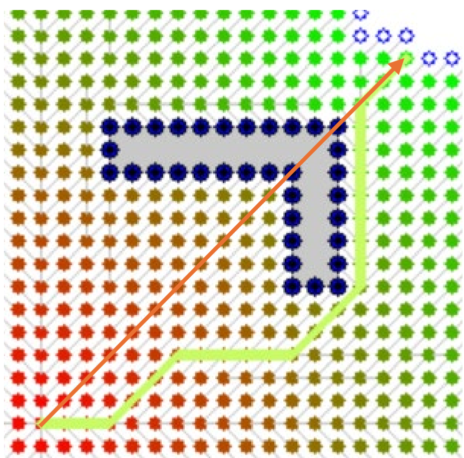
Weighted A* search: $\boldsymbol{g(n) + W \times h(n)}$ $\boldsymbol{(1 < W < \infty)}$

The presented algorithms are special cases:

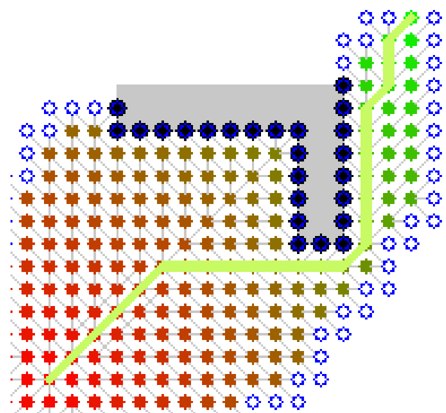
A* search:	$g(n) + h(n)$	$(W = 1)$
Uniform cost search/BFS:	$g(n)$	$(W = 0)$
Greedy best-first search:	$h(n)$	$(W = \infty)$

Example of Weighted A* Search

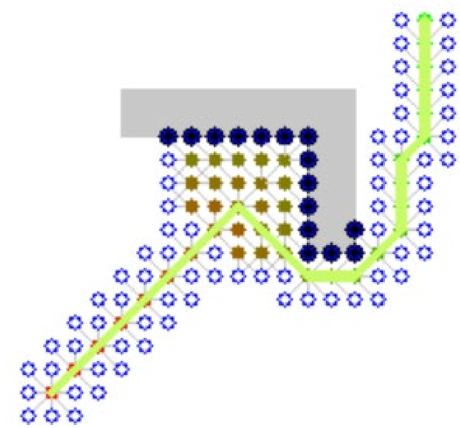
Reduction in the number of expanded nodes



Breadth-first Search (BFS)
$f(n) =$ # actions to reach n



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



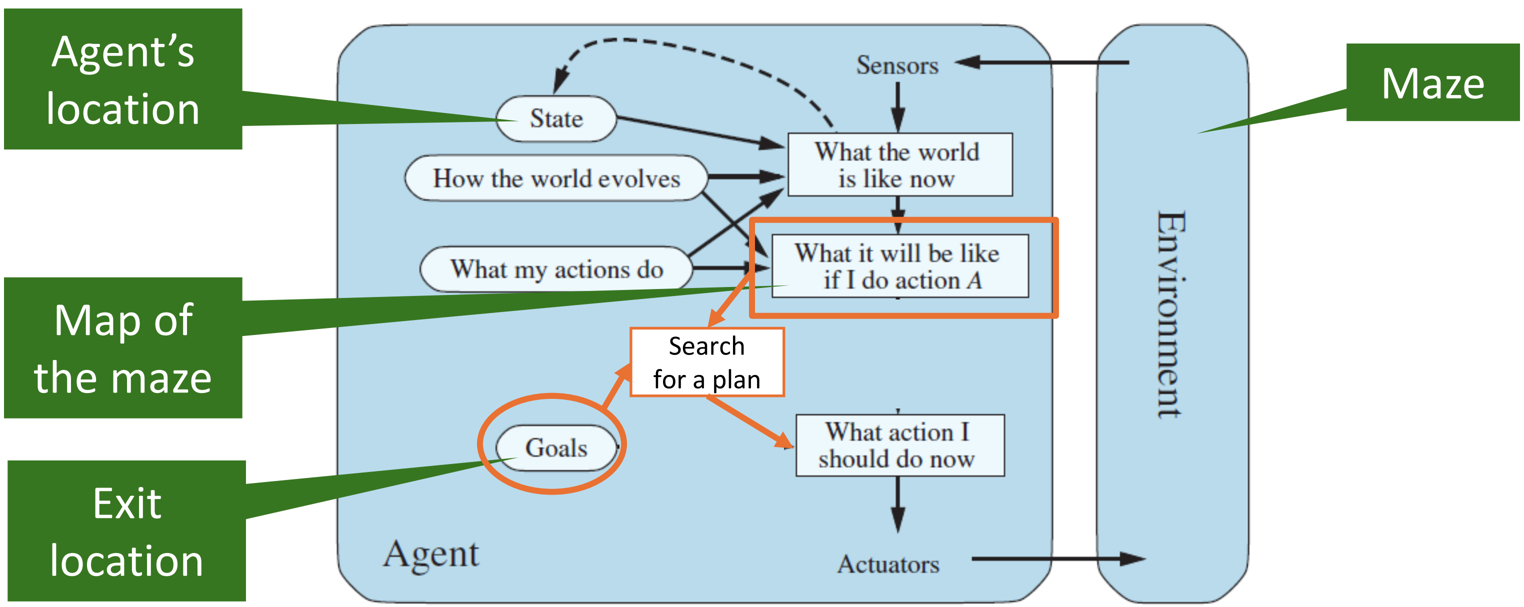
Weighted A* Search
$f(n) = g(n) + 5\, h_{Eucl}(n)$

Source and Animation: [Wikipedia]

Summary
Planning Agents

Remember: Planning Agent (Goal-based)

- The agent has the task of reaching a defined **goal state**.
- The performance measure is typically the cost to reach the goal.
- We will discuss a special type of **goal-based agents** called **planning agents,** which use **search algorithms** to plan a sequence of actions that lead to the goal.

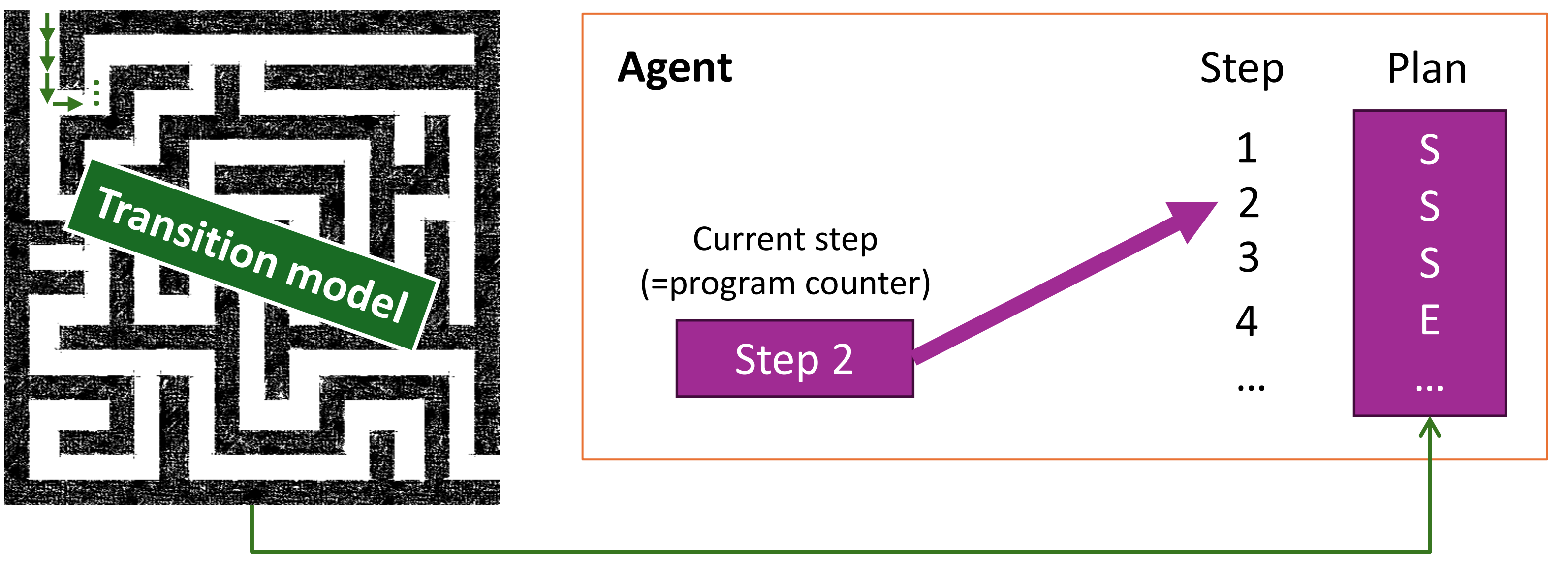


$$a_i = \text{argmin}_{a_i \in \text{A}} \left[\sum_{t=i}^{T} c_t \,\middle|\, s_T \in S^{goal} \right]$$

A Planning Agent: Planning vs. Execution Phase

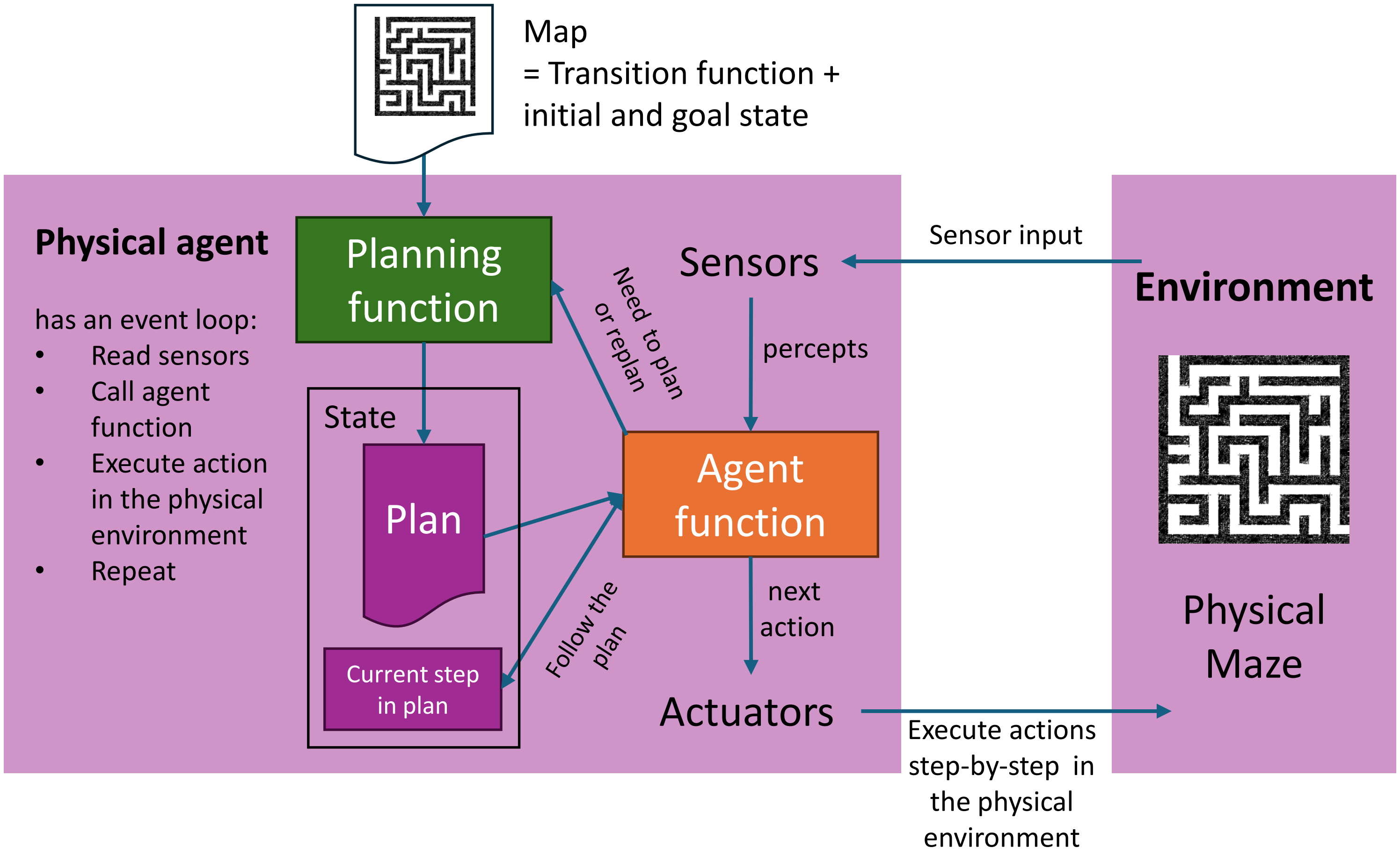
1. Planning is done by **a planning function** using search. The result is a **plan**.
2. The plan is executed by the agent function, which returns the planned actions from the plan step-by-step.

Planning function **Execution of the plan** at step 2 returns action S



Note: The execution agent does not use percepts or the transition function. It blindly follows the plan.
Caution: This only works in an environment with **deterministic transitions**.

Example: Complete Planning Agent to Solve a Maze



- The event loop calls the agent function for the next action.
- The agent function follows the plan or calls the planning function if there is no plan yet or it thinks the current plan does not work based on the percepts (replanning).

Summary:
All Search Strategies

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

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

Needs cycle checking
Cannot avoid redundant paths

With a good heuristic

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 $f(n)$.**

Search Strategy	Evaluation function $f(n)$
BFS (Breadth-first search)	$g(n)$ (=uniform path cost)
Uniform-cost Search	$g(n)$ (=path cost)
~~**DFS/IDS (see note below!)**~~	$\cancel{-g(n)}$
Greedy Best-first Search	$h(n)$
(weighted) A* Search	$g(n) + W \times h(n)$

- **Important note:** Do not implement DFS/IDS using Best-first search! You will get poor space complexity from BFS and the disadvantages of DFS (not optimal and worse time complexity).

Conclusion

- Tree search can be used for planning actions for **goal-based agents** in known, fully observable and deterministic environments.
- Issues are:
  - The large search space typically does not fit into memory. We use a transition function as a compact representation of the **transition model.**
  - The search tree is built on the fly, and we have to deal with **cycles, redundant paths, and memory management**.
- DFS/IDS is a memory efficient method used often in AI for **uninformed search**.
- **Informed search** uses heuristics based on knowledge or percepts to improve search performance (i.e., A* expand fewer nodes than BFS).