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

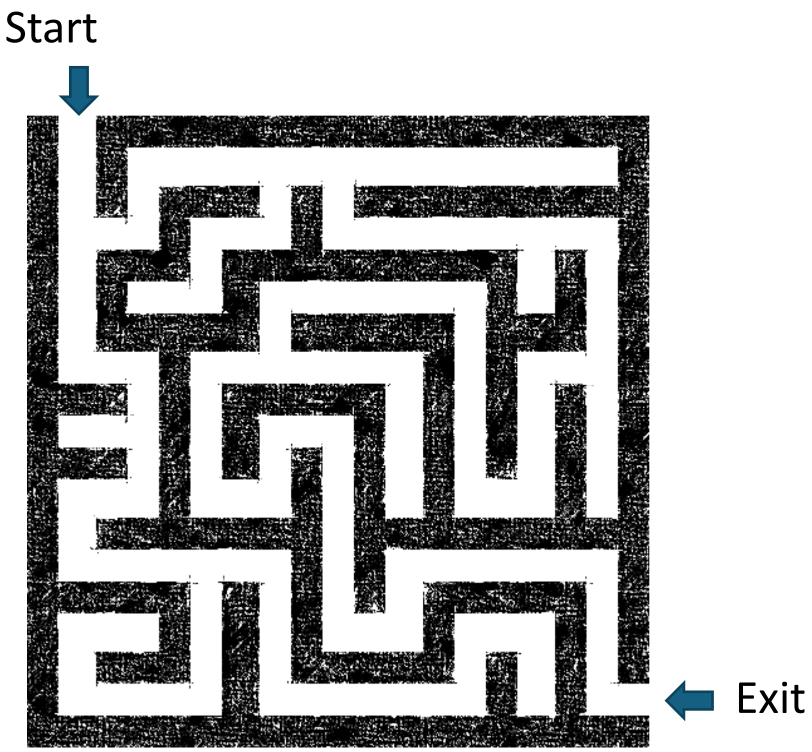
Contents

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

Contents

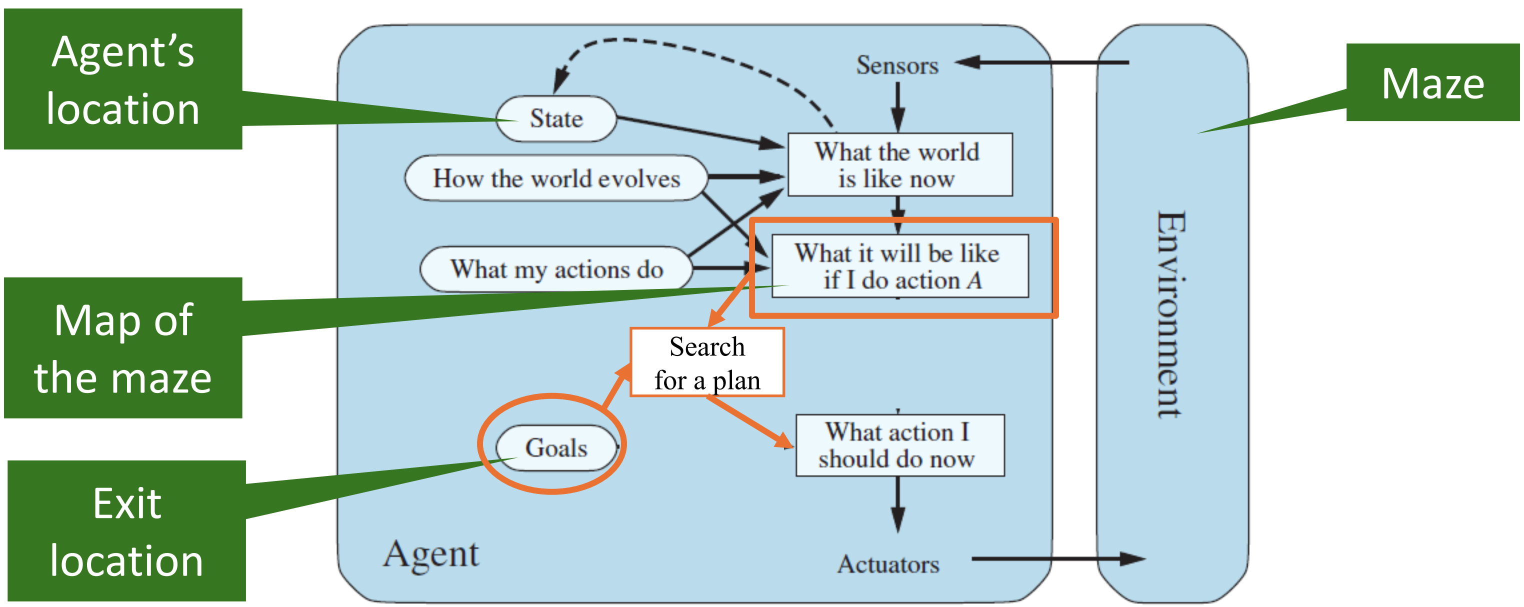
What are Search Problems?

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



Remember: Goal-based Agent

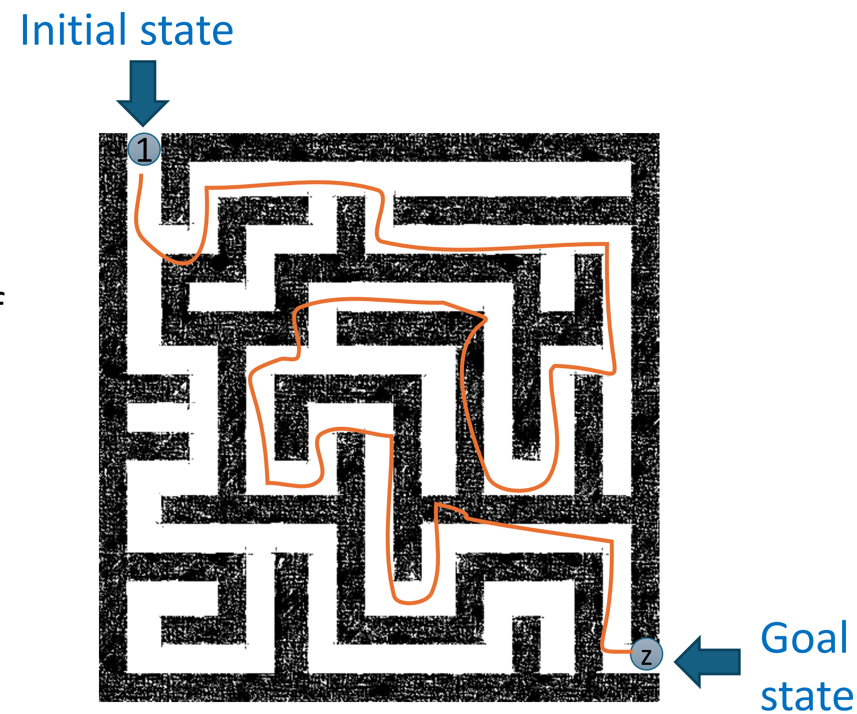
- The agent has the task to reach a defined **goal state**.
- The performance measure is typically the cost to reach the goal.
- We will discuss a special type of goal-based agents called **planning agents** which use **search algorithms** to plan a sequence of actions that lead to the goal.



$$a_s = \text{argmin}_{a \in A}[cost(s, s_1, s_2, \ldots, s_n | s_n \in S^{goal})]$$

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.

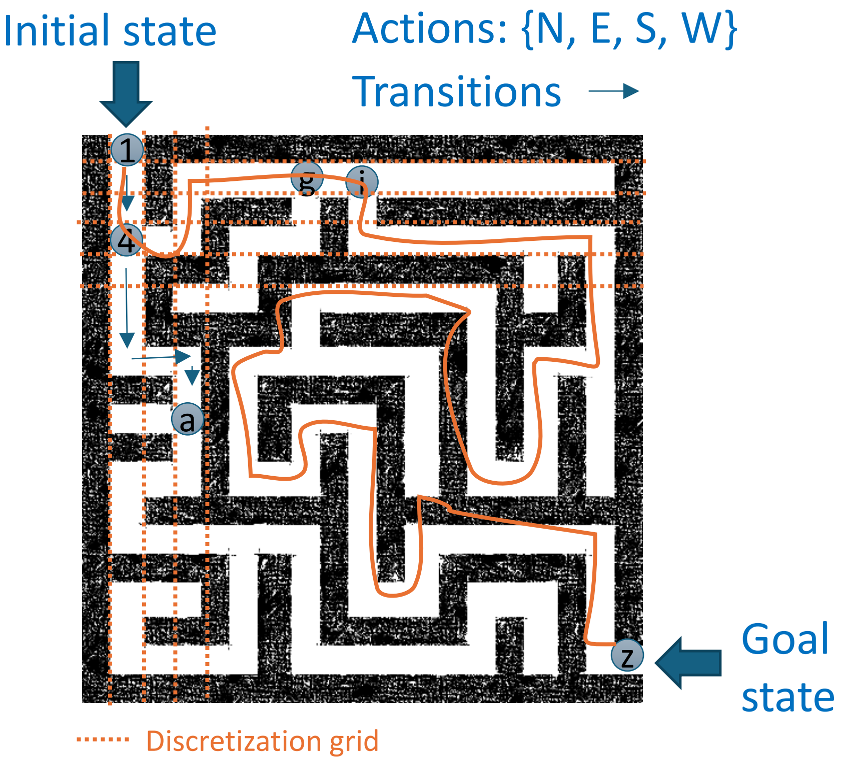


Phases:

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

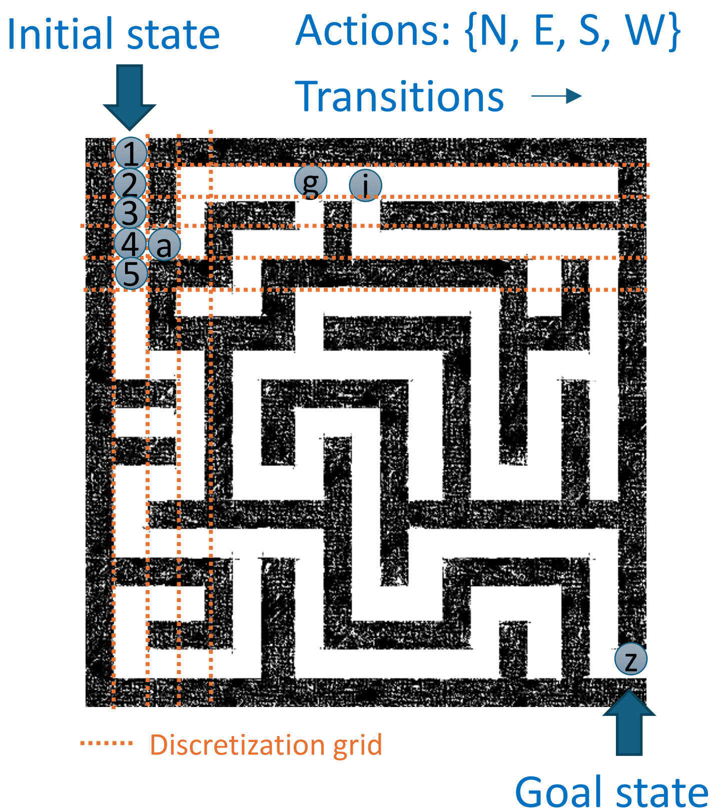
- **Initial state:** state description
- **Actions:** set of possible actions A
- **Transition model:** a function that defines the new state resulting from performing an action in the current state
- **Goal state:** state description
- **Path cost:** the sum of *step costs*



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

Original Description



- As an action schema:
  $Action(go(dir))$
  PRECOND: no wall in direction dir
  EFFECT: change the agent's location according to dir
- As a function:
  $f: S \times A \rightarrow S$ or $s' = result(a, s)$
- 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

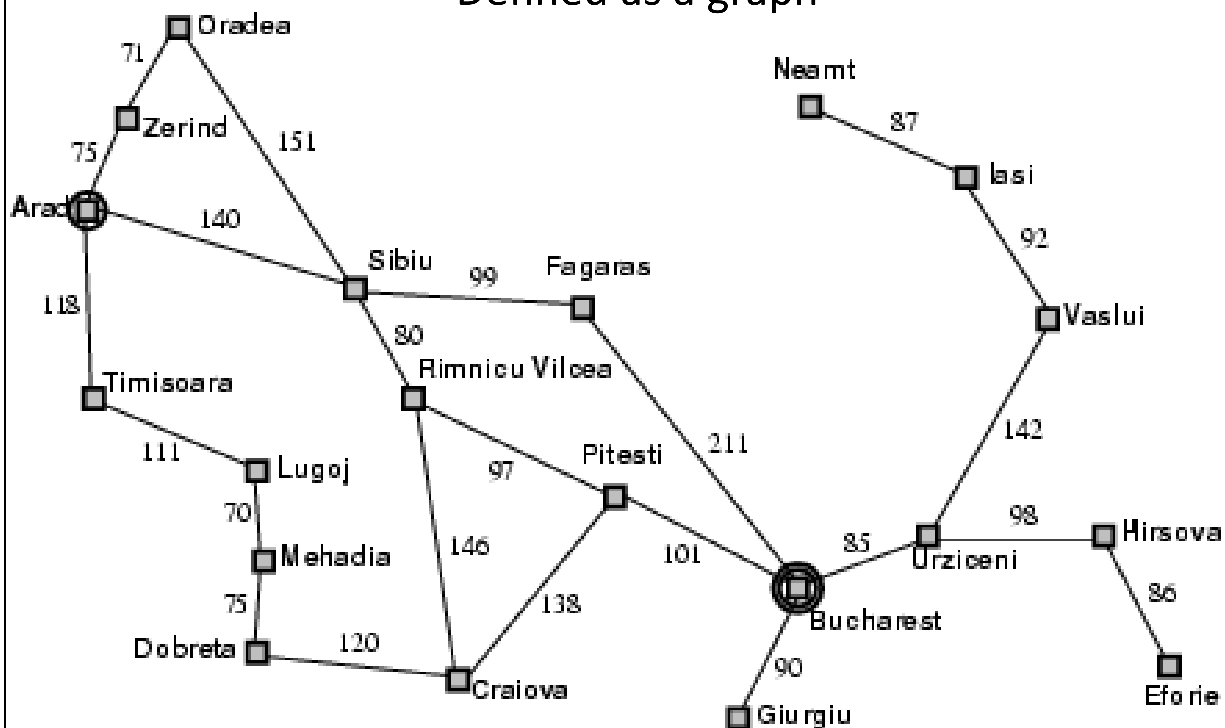
- On vacation in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest

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.

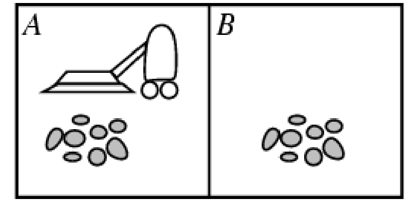
State Space/Transition model

Defined as a graph

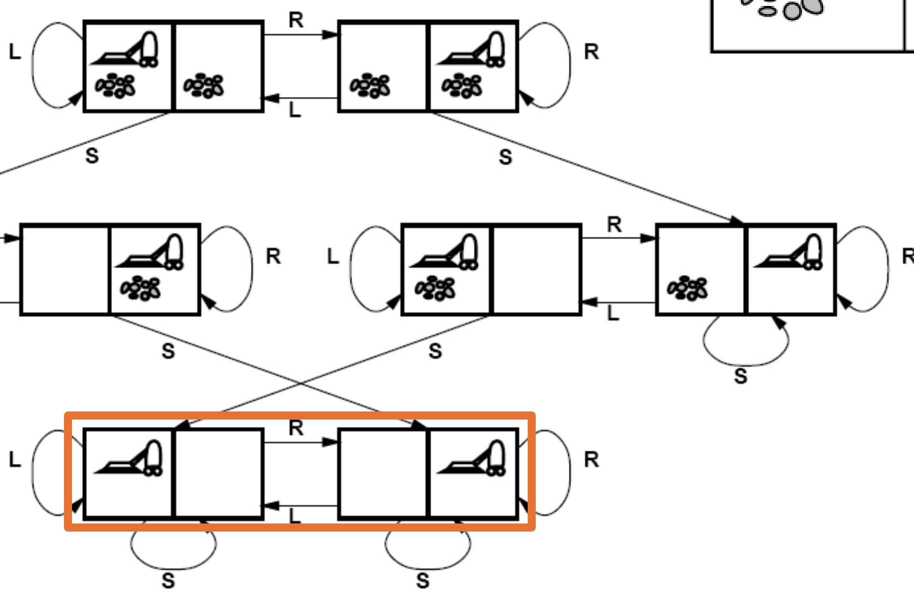


Distance in miles

Example: Vacuum world



State Space



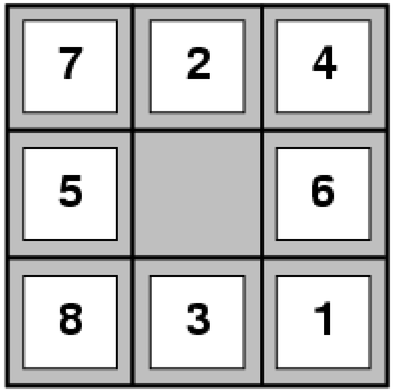
Goal states

- **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

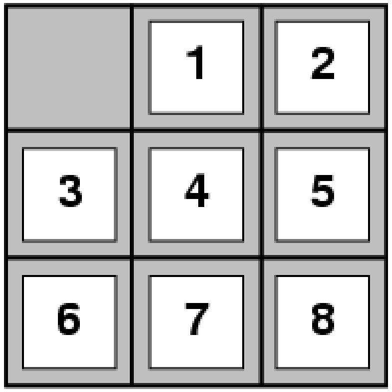
There are 8 possible atomic states of the system.
Why is the number of states for *n* possible locations $n(2^n)$?

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



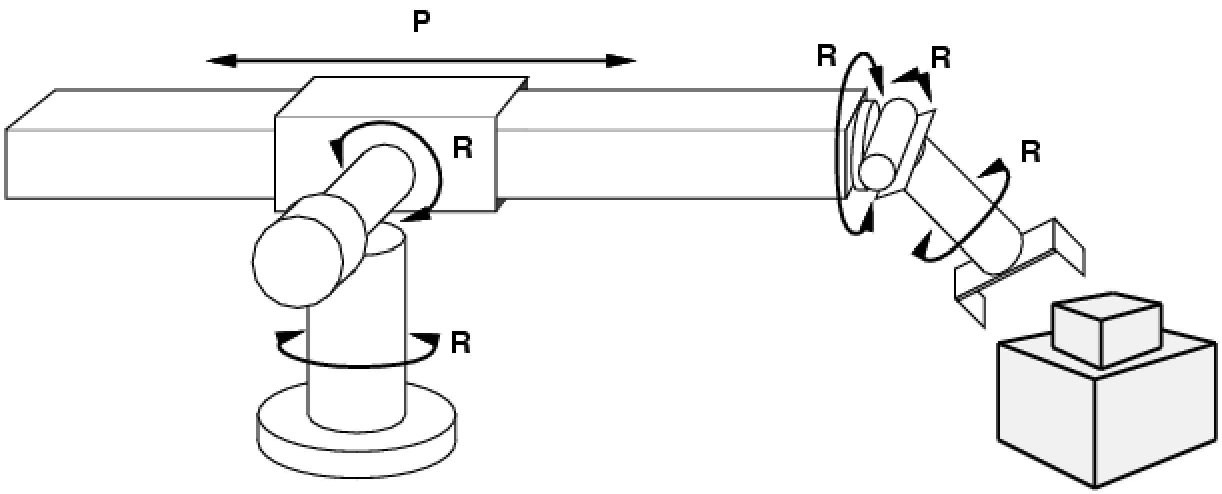
Goal State

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.
- **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.

Contents

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

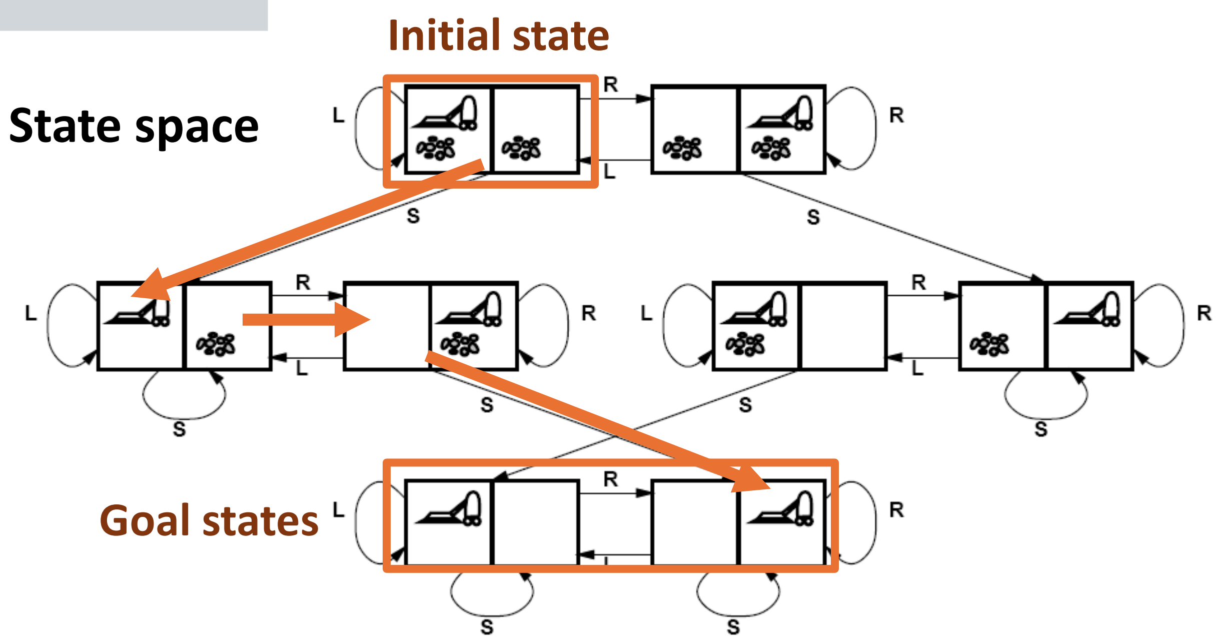
Solving Search Problems

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)?

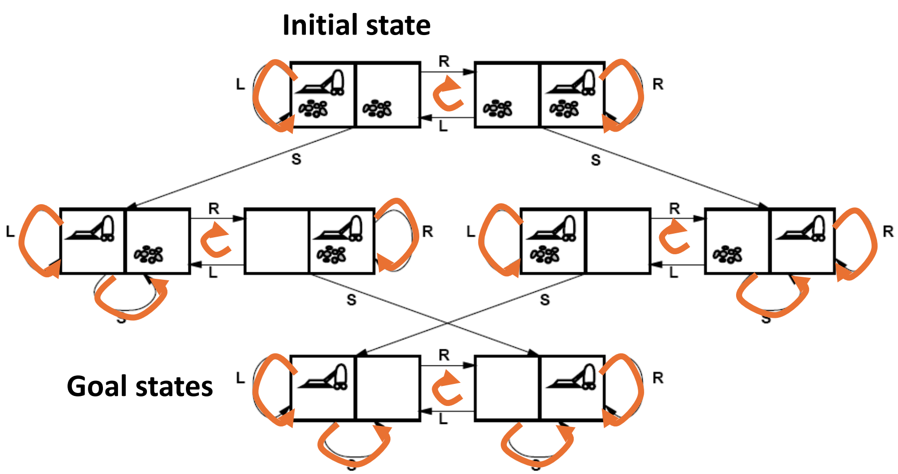
Construct a search tree for the state space graph!



Issue: Transition Model is Not a Tree! It can have Redundant Paths

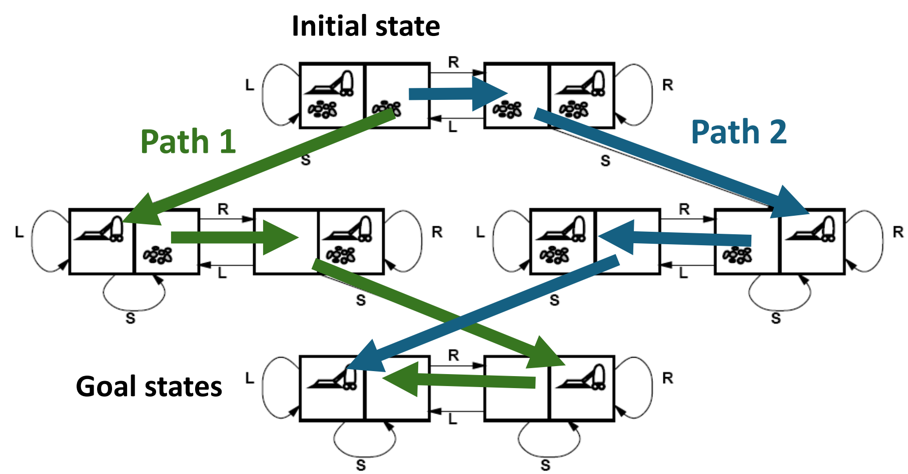
Cycles

Return to the same state. The search tree will create a new node!



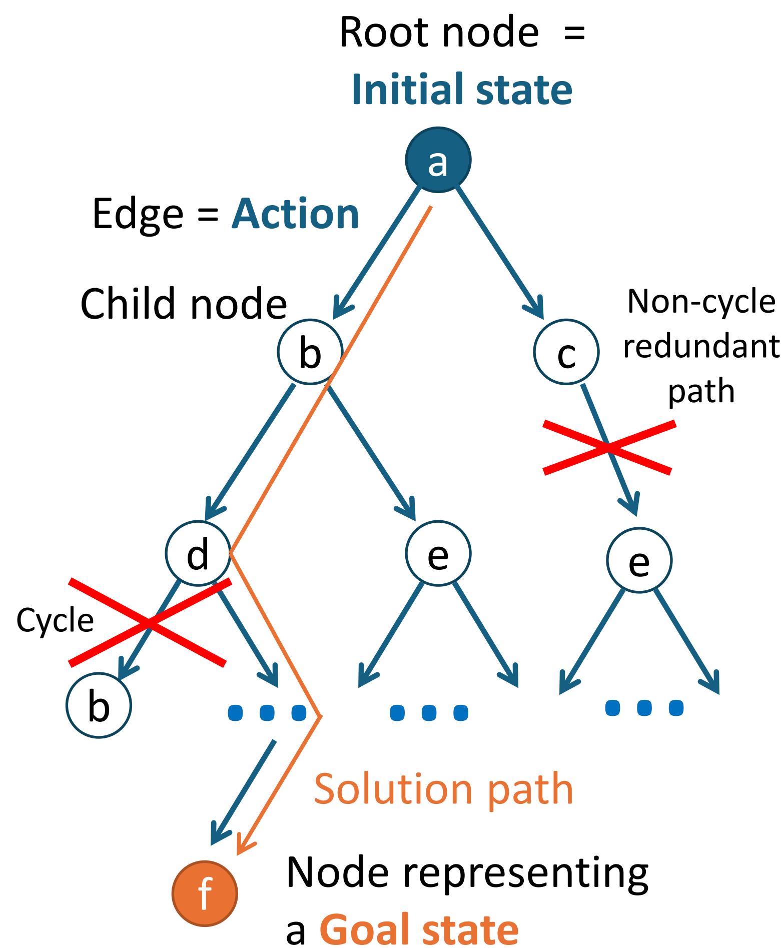
Non-cycle redundant paths

Multiple paths to get to the same state



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)** or **multiple paths to the same state.** These are called redundant paths. Cycles in the search space must be broken to prevent infinite loops. Removing other 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.



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

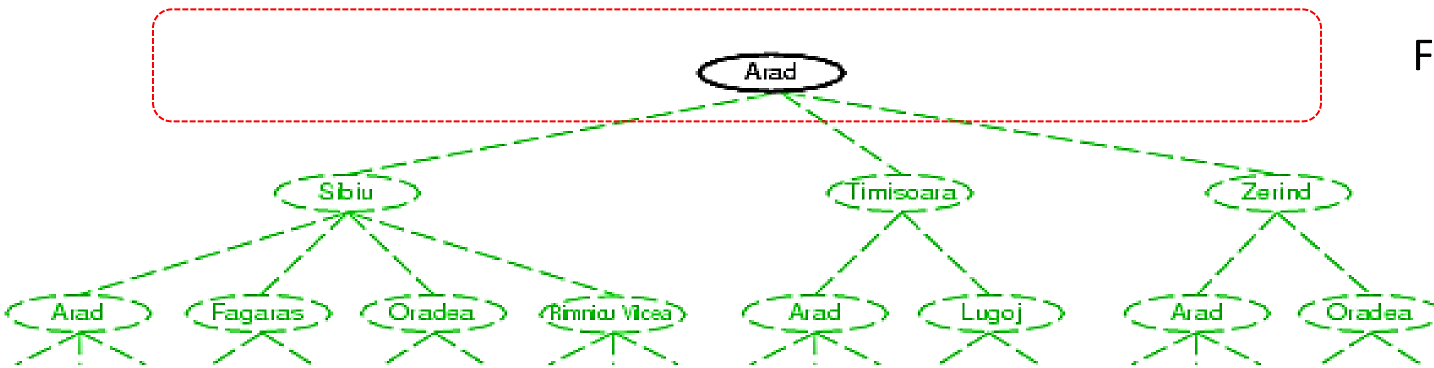
- The search tree is too large to fit into **memory**.
  a. **Builds parts of the tree** from the initial state using the transition function representing the graph.
  b. **Memory management** is very important.

- 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.

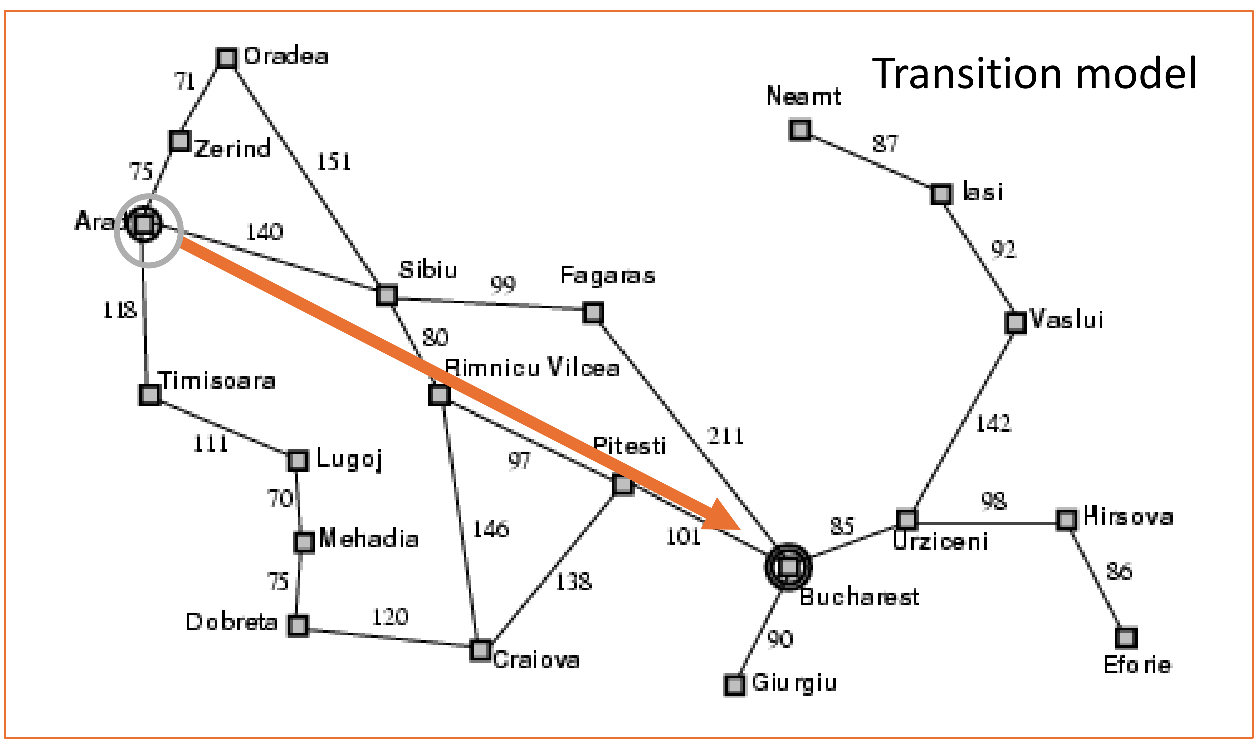
Tree Search Algorithm Outline

1. Initialize the **frontier** (set of unexplored know nodes) using the **starting state/root node.**
2. While the frontier is not empty:
   a) Choose next 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.

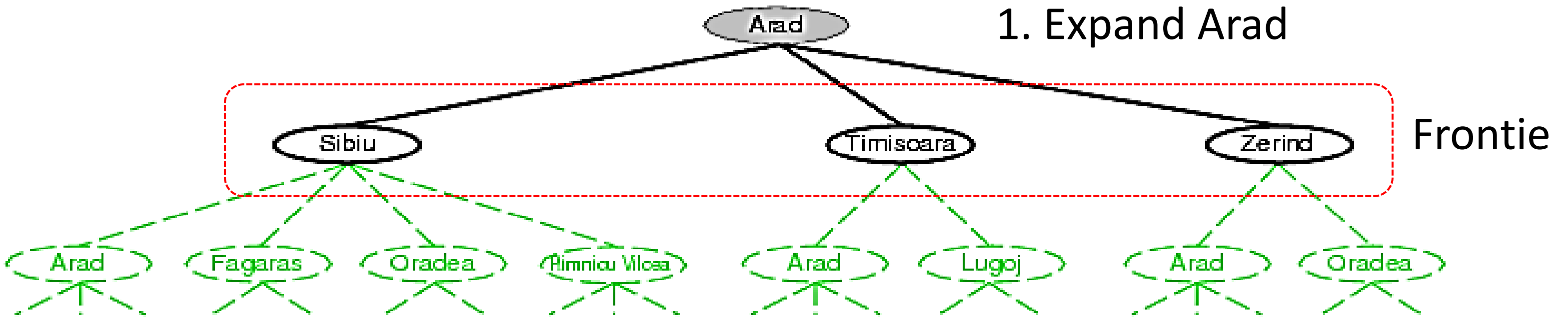
Tree Search Example

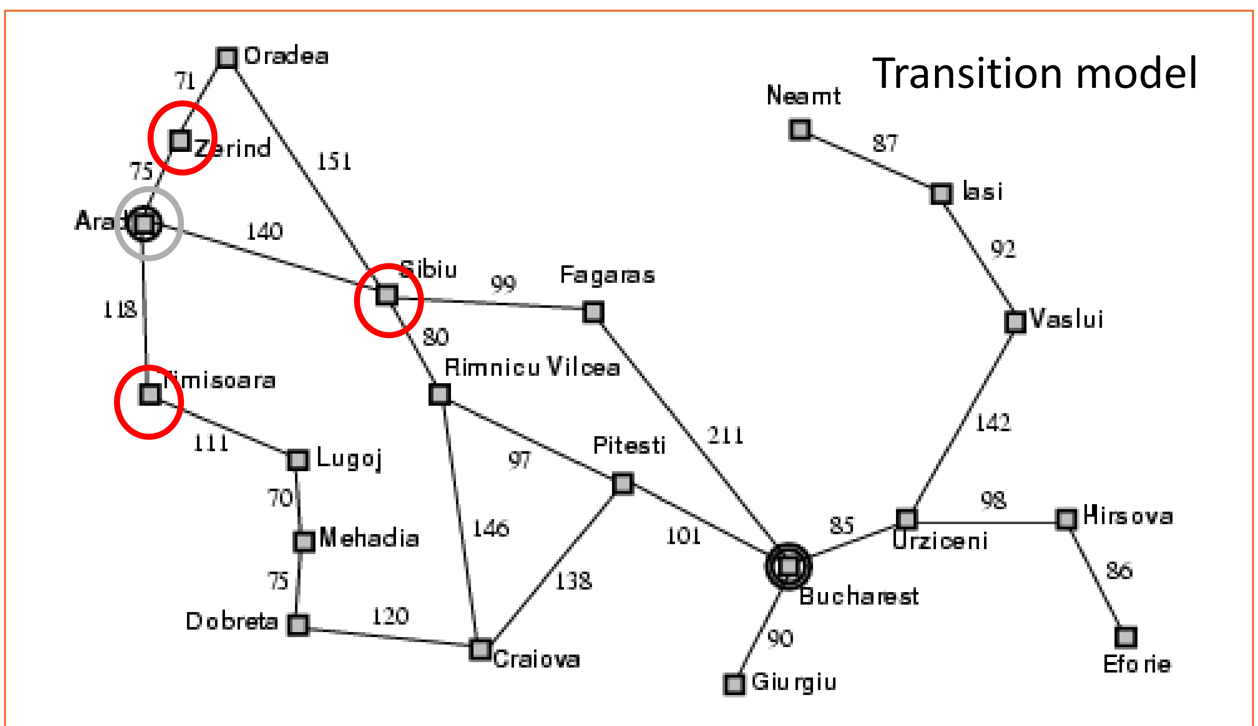


Frontier



Tree Search Example

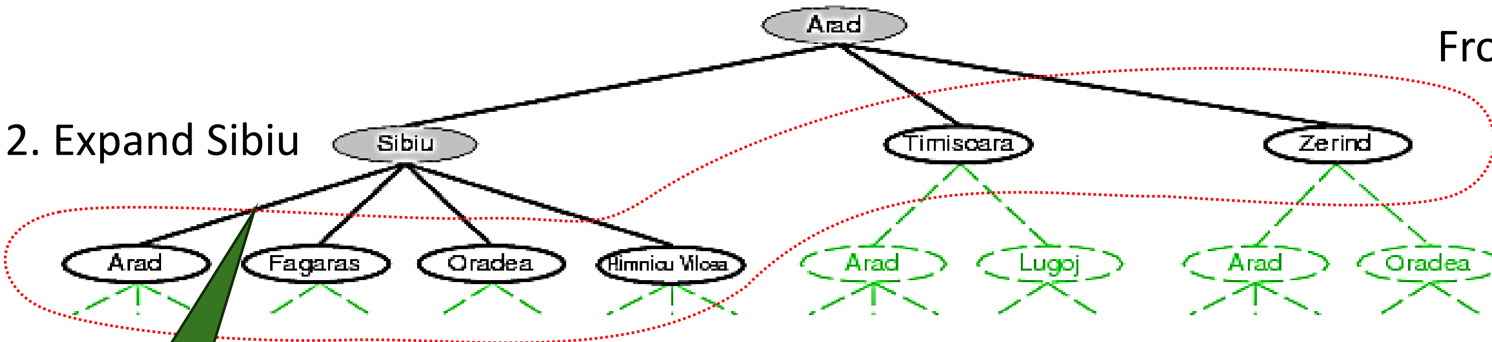




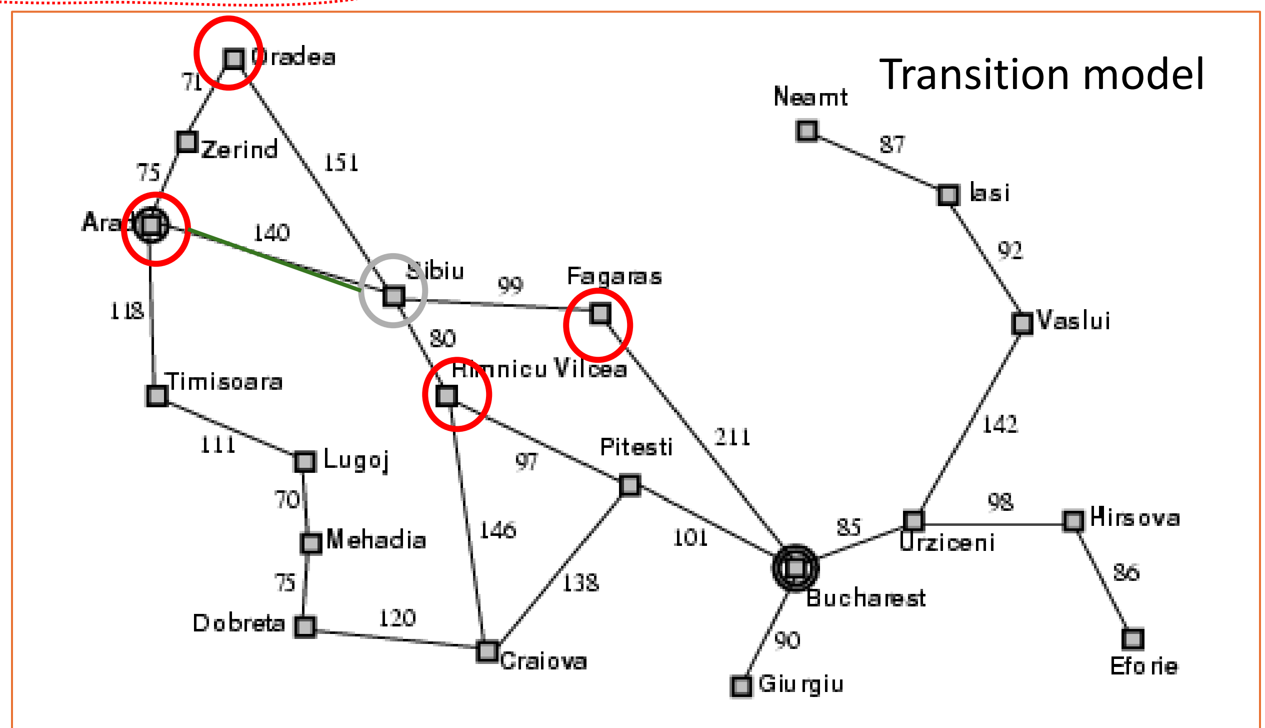
Tree Search Example

2. Expand Sibiu

Frontier



Example of a cycle



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?

Search Strategies: Time and Space Complexity

- A **search strategy** is defined by picking the **order of node expansion**.

- Worst case time and space complexity are measured in terms of the **size of the state space *n*** (= number of nodes in the search tree).

$$O(n)$$

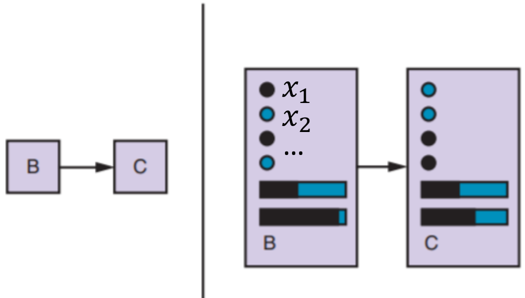
- Often used metrics if the state space is only implicitly defined by initial state, actions and a transition function are:
  - b: maximum branching factor of the search tree (number of available actions).
  - m: length of the longest path (loops need to be removed).
  - d: depth of the optimal solution.

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

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. Not all states may be reachable from the initial state.
- **Search tree** spans the state space. Note that a single state can be represented by several search tree nodes if we have redundant paths.
- State space size is an indication of problem size.

State representation



(a) Atomic (b) Factored

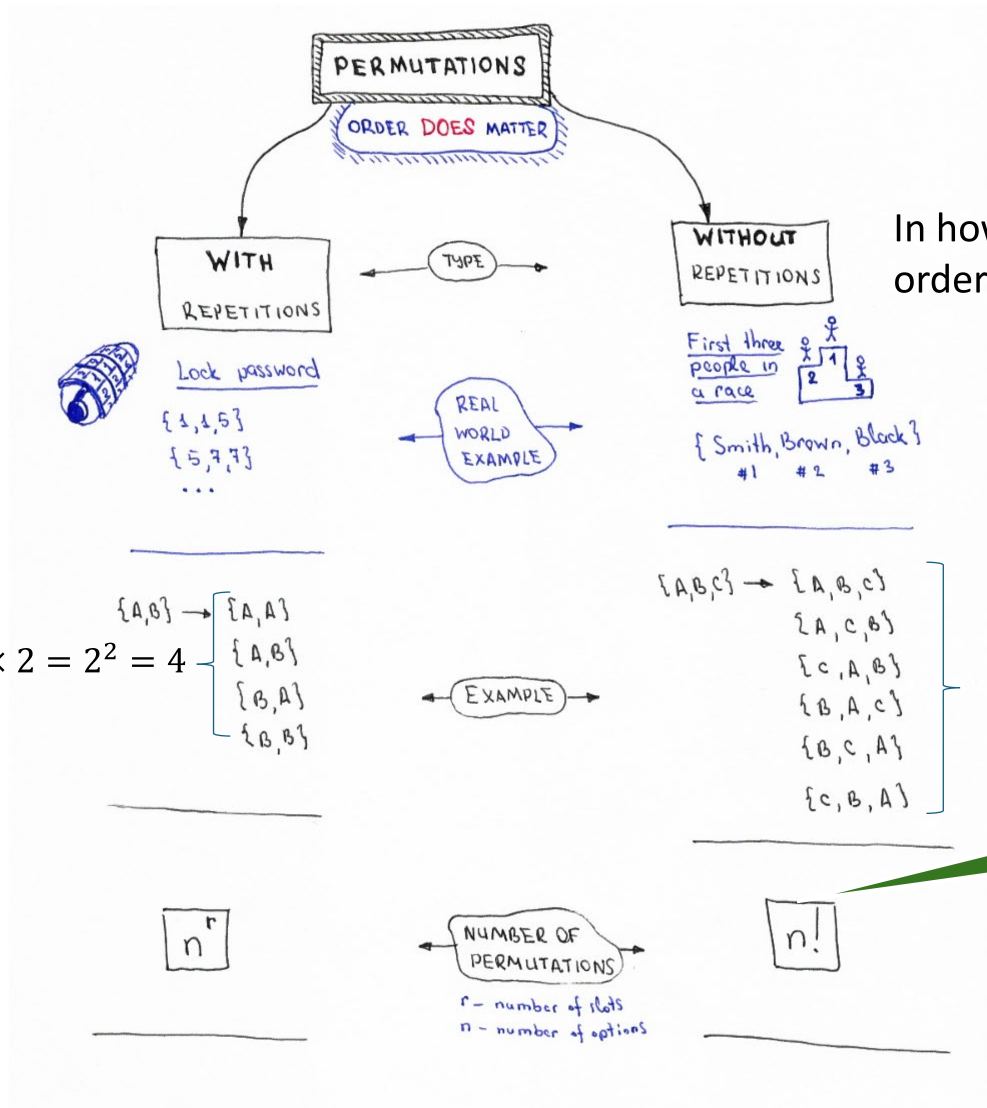
The state consists of variables called fluents that represent conditions that can change over time.

State Space Size Estimation

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

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

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



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

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

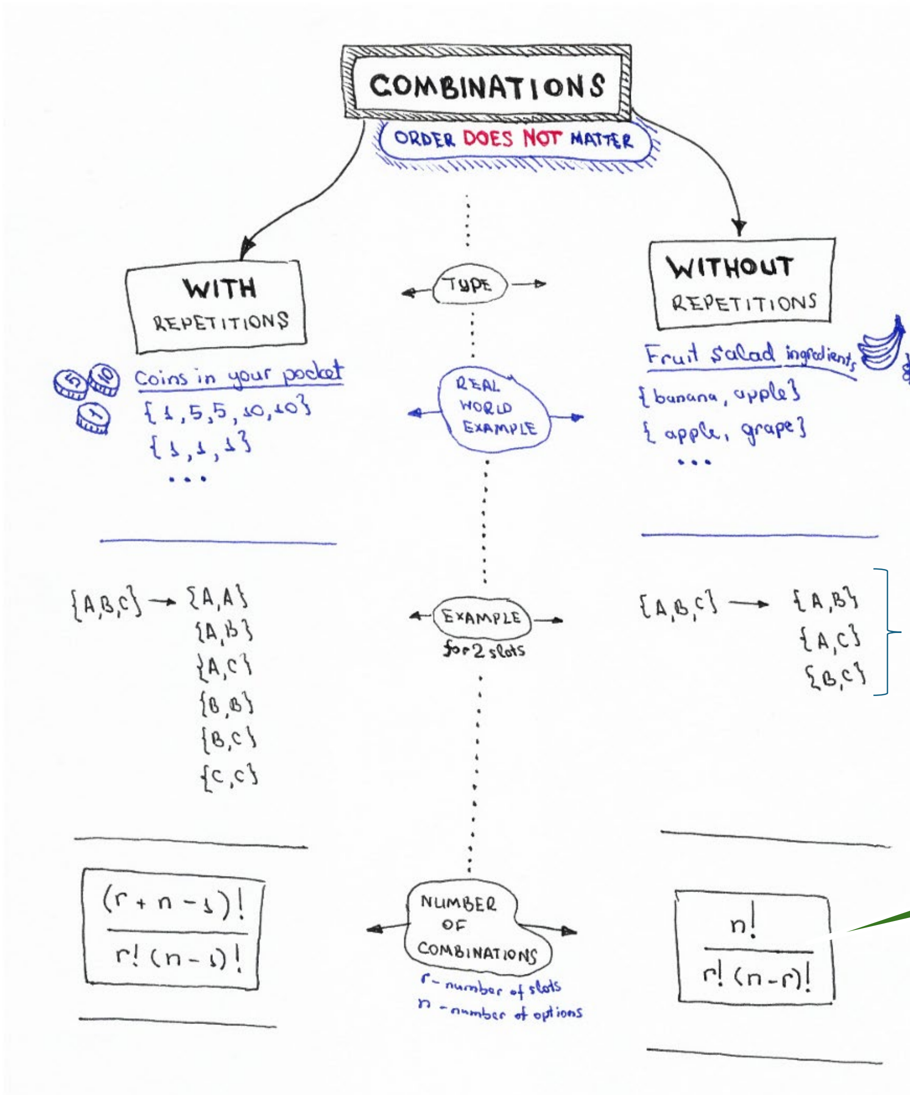
$3 \times 2 \times 1 = 6$

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

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

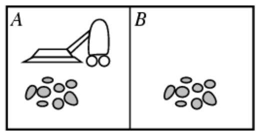
Special case for $r = 2$: $\binom{n}{2} = \frac{n(n-1)}{2}$

```
#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)

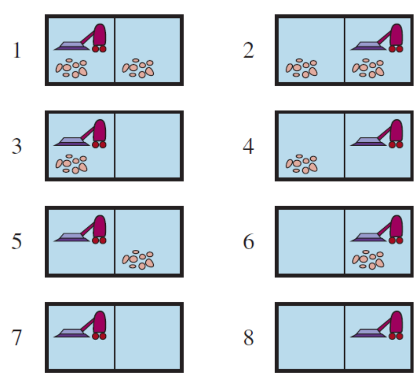
$\rightarrow 2^2$

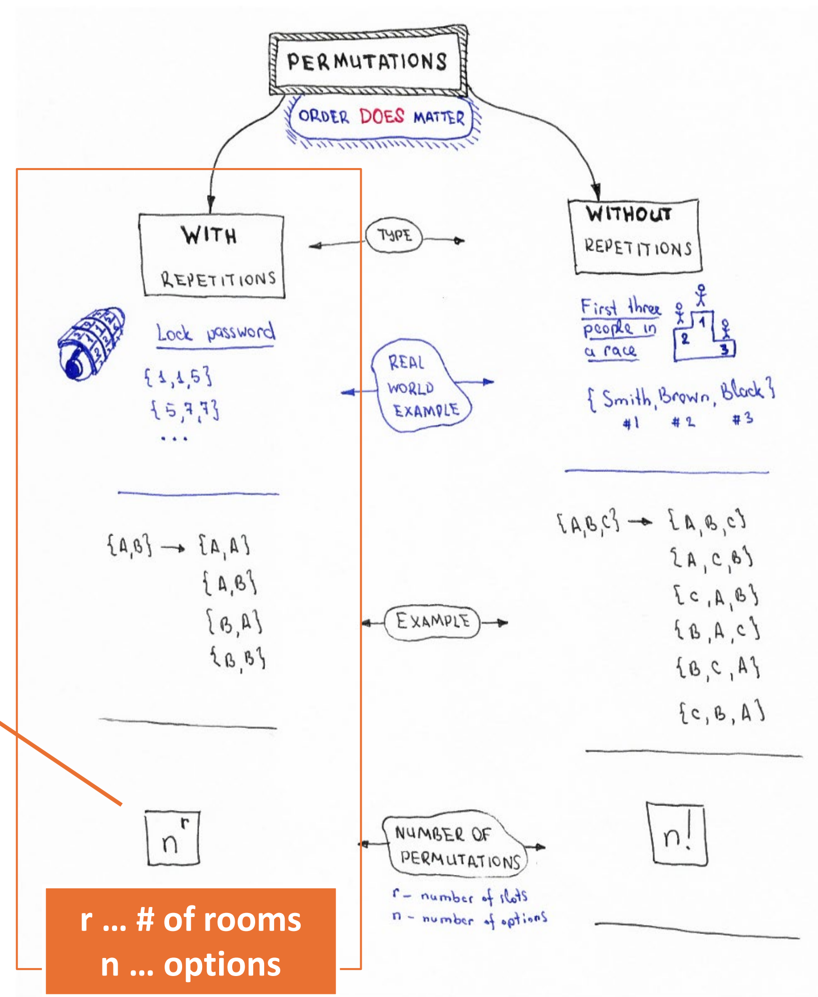
Robot location

- Can be in 1 out of 2 rooms.

$\rightarrow 2$

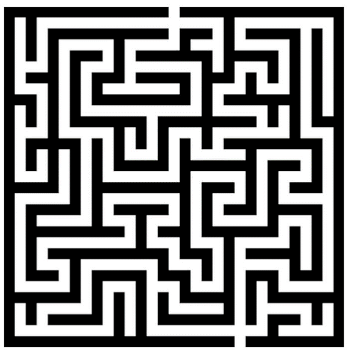
Total: $n = 2 \times 2^2 = 2^3 = 8$



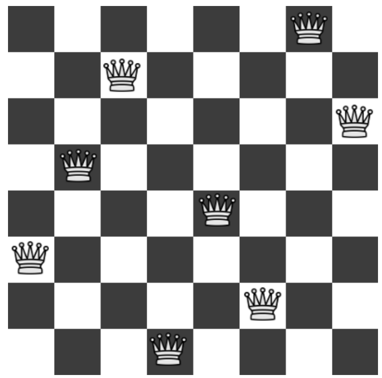


Examples: What is the State Space Size?

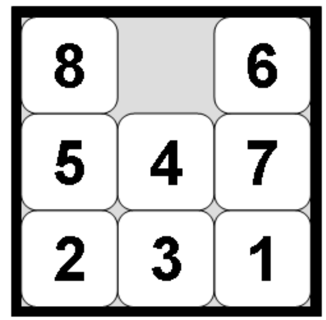
Often a rough upper limit is sufficient to determine how hard the search problem is.



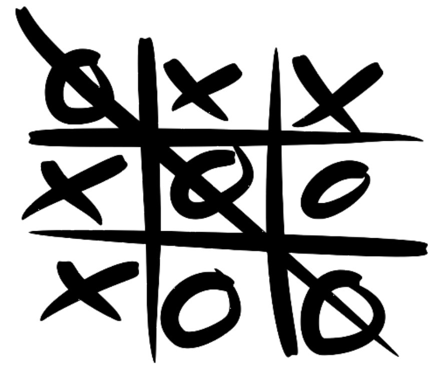
Maze



8-queens problem



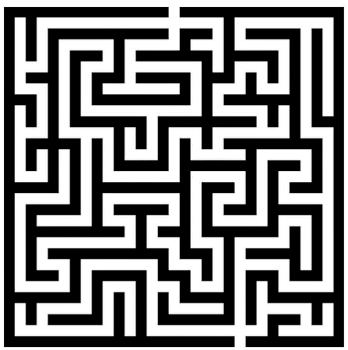
8-puzzle problem



Tic-tac-toe

Examples: What is the State Space Size?

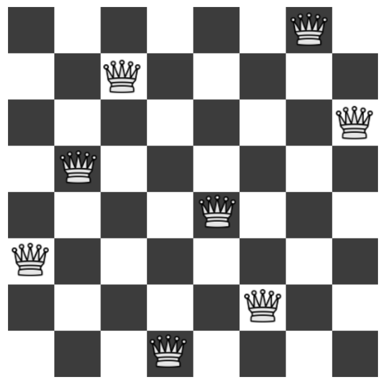
Often a rough upper limit is sufficient to determine how hard the search problem is.



Maze

Positions the agent can be in.

n = Number of white squares.

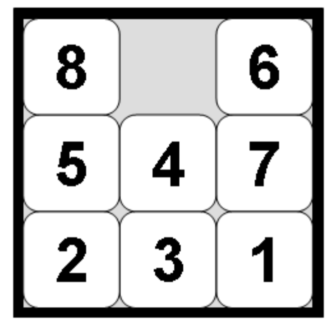


8-queens problem

All arrangements with 8 queens on the board.

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

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

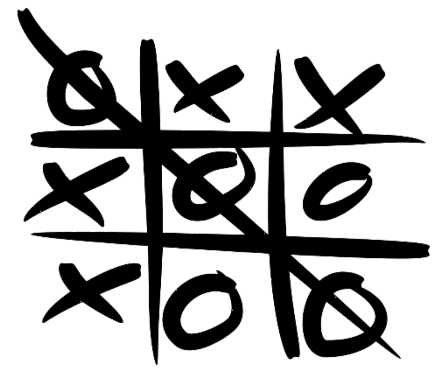


8-puzzle problem

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.

Example: What is the Search Complexity?

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

  3

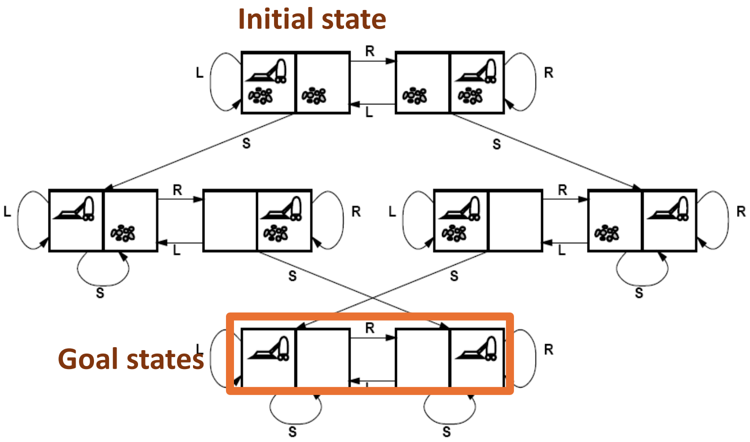
- m: the number of actions in any path? Without loops!

  4

- d: depth of the optimal solution?

  3

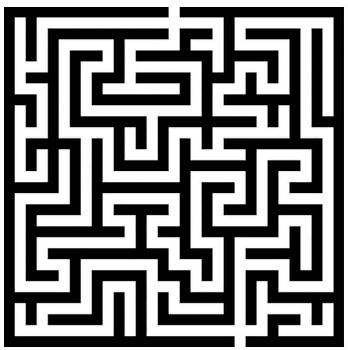
State Space with Transition Model

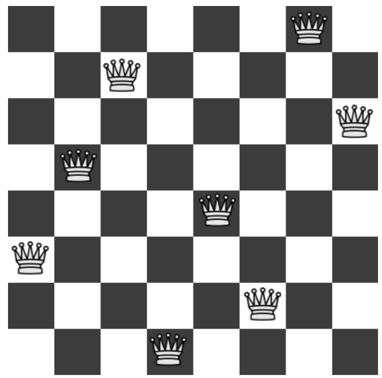


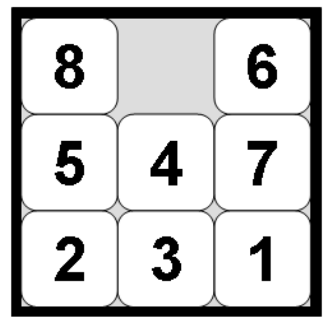
Examples: What is the Search Complexity?

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

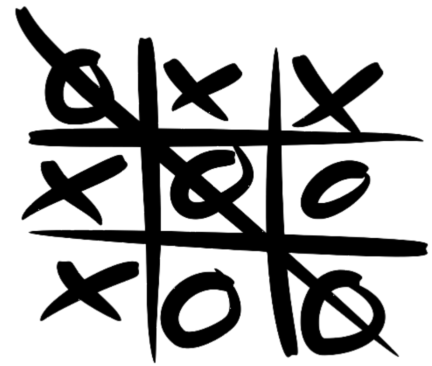
Often a rough upper limit is sufficient to determine how hard the search problem is.


Maze


8-queens problem



8-puzzle problem

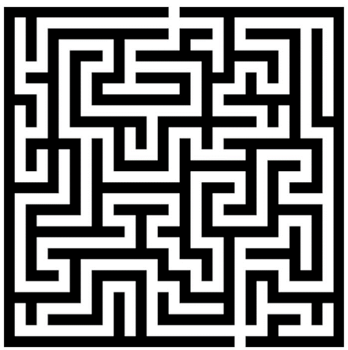

Tic-tac-toe

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

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.

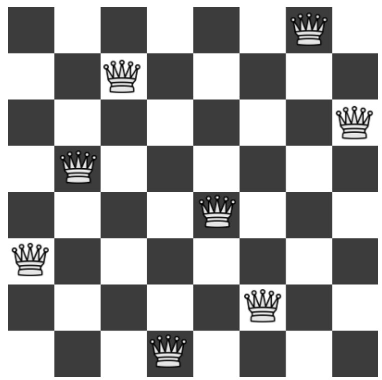


Maze

$b = 4$ actions

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

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

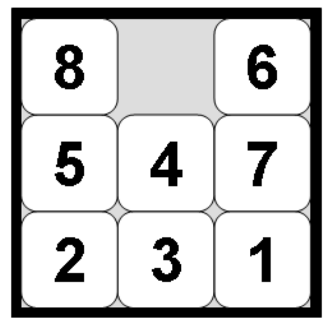


8-queens problem

$b = ?$ What are the actions? Moving one Queen: $64 - 7 = 57$

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

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

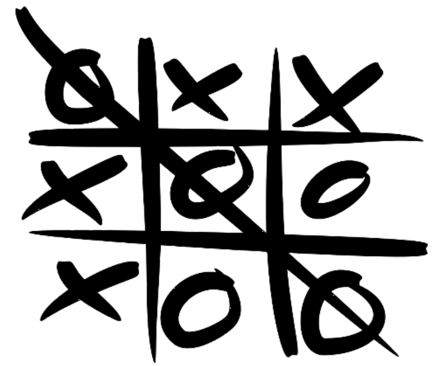


8-puzzle problem

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

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

$d = ???$



Tic-tac-toe

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

$d = 9$ (if both play optimal)

Uninformed Search

Uninformed Search Strategies

The search algorithm/agent is **not** provided information about how close a state is to the goal state. It just has the labels of the atomic states and the transition function.

It blindly searches following a simple strategy until it finds the goal state by chance.

Search strategies we will discuss:

- 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 (=**FIFO**).

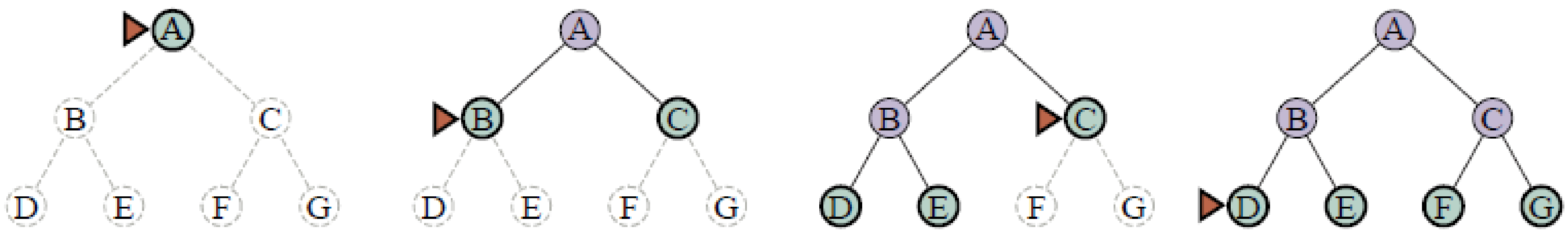


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

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

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
```

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 new nodes for the frontier.

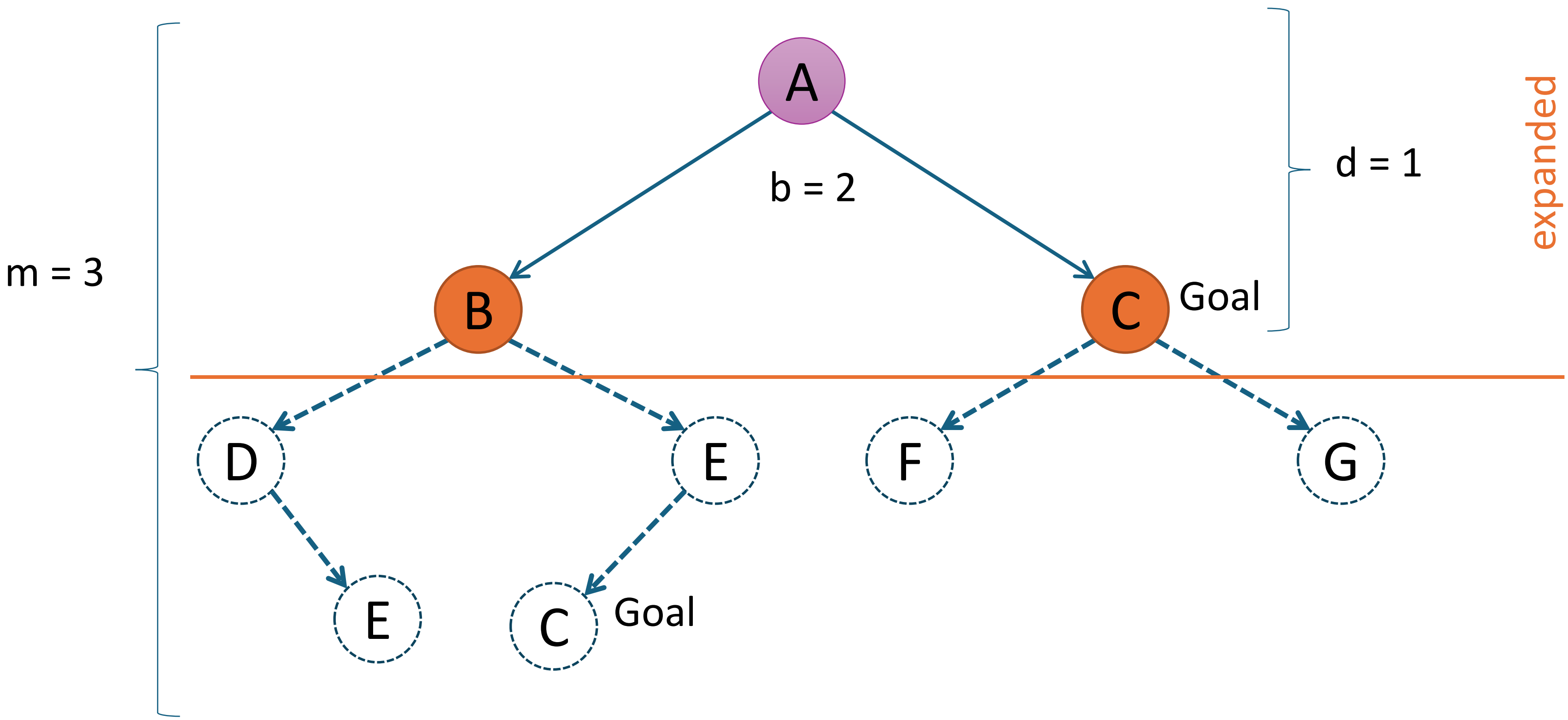
```
function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

Transition function

Node structure for the search tree. Yield 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:

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

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:

- 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!

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

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

- **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]

Implementation: Best-First Search Strategy

```
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 the difference to BFS! It visits a node again if it can be reached by a better (cheaper) path.

See BFS for function EXPAND.

Depth-First Search (DFS)

- **Expansion rule**: Expand deepest unexpanded node in the frontier (last added).
- **Frontier**: **stack** (LIFO)
- **No reached data structure!**

Cycle checking checks only the current path.

Redundant paths can not be identified and lead to replicated work.

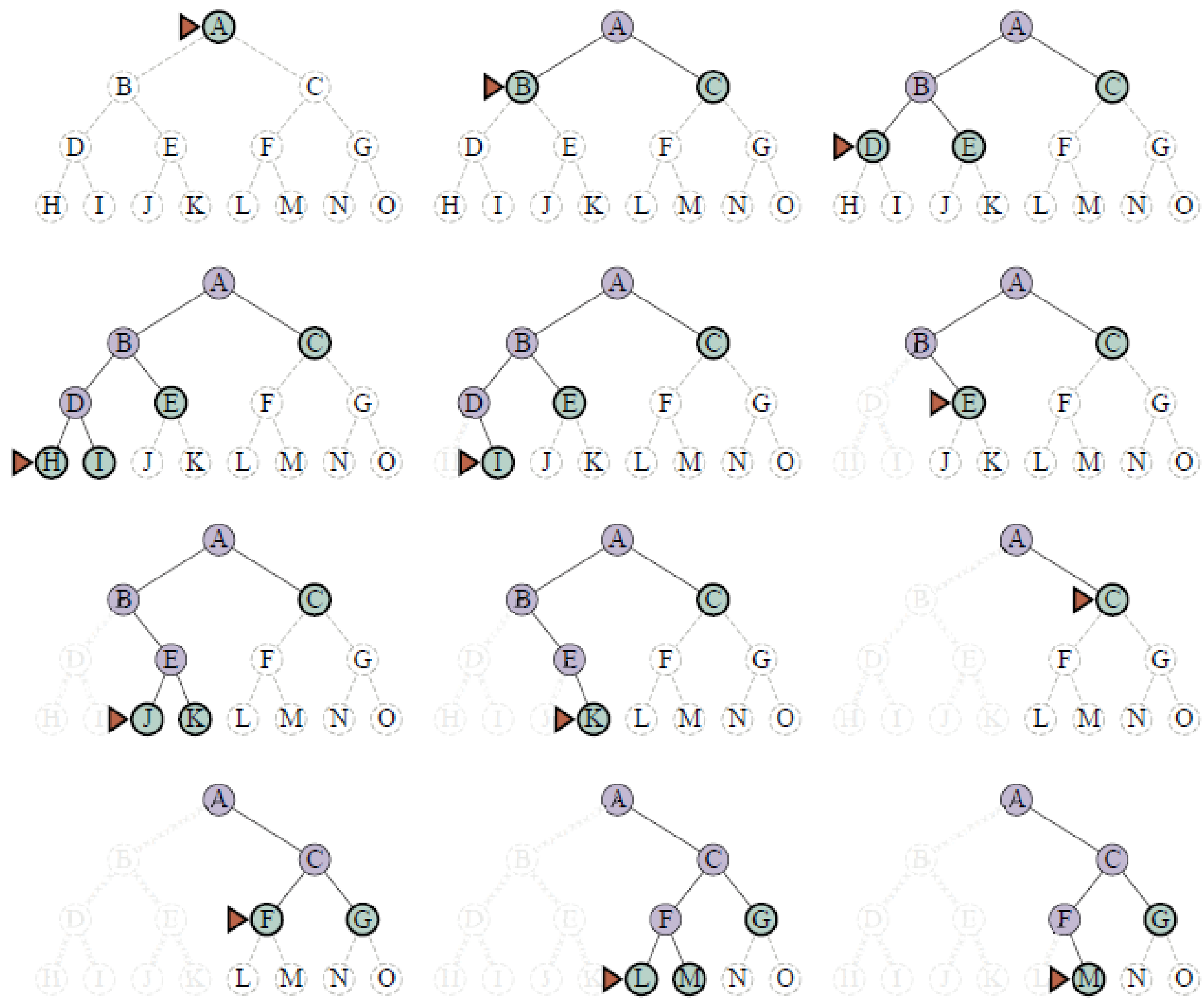


Figure 3.11 A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint 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 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 leading to infinite loops)
  - Iterative implementation: Build tree and abandoned branches are removed from memory. Cycle checking is only done against the current path. This is similar to Backtracking search.

DFS uses $\ell = \infty$

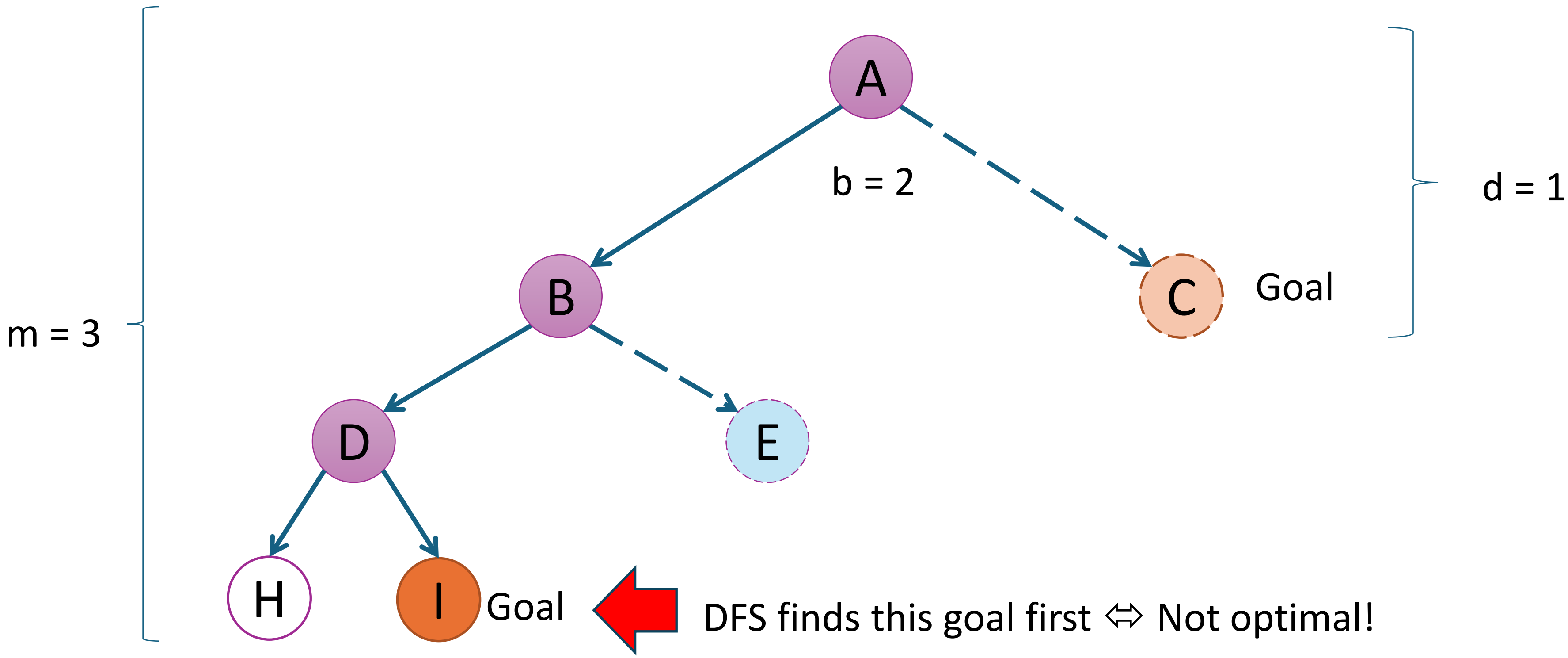
```
function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node)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

If we only keep the current path from the root to the current node in memory, then we can only check against that path to prevent cycles, but we cannot prevent other redundant paths. We also need to make sure the frontier does not contain the same state more then once!

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



- 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?**
  - Only in finite search spaces. 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?**
  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** (forget old branches) is used! Cycles can be broken but redundant paths cannot be checked.

Iterative Deepening Search (IDS)

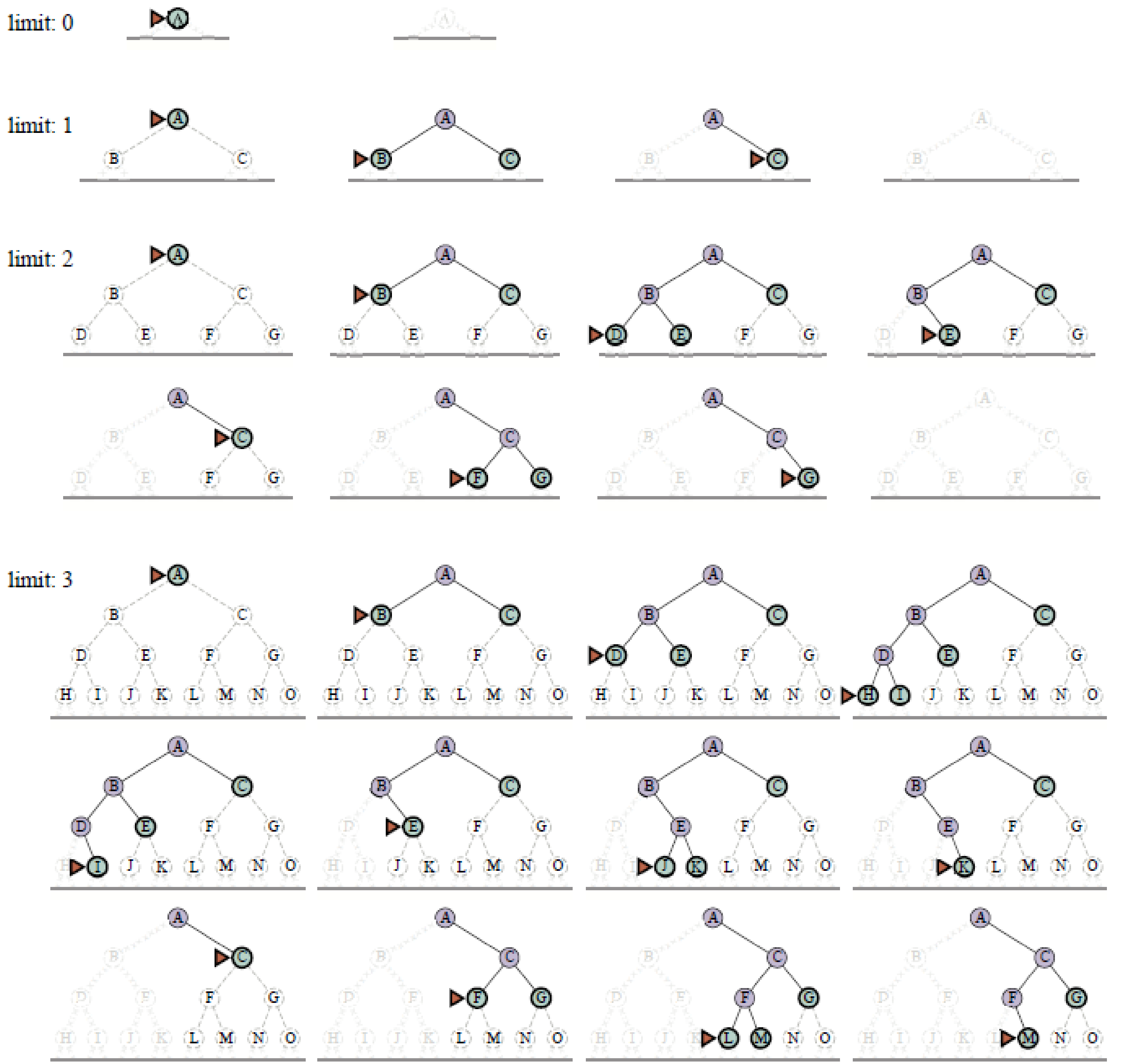
Can we

- **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 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

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

- **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.

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

This makes IDS/DFS the workhorse of AI.

Informed Search

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.**

- 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).

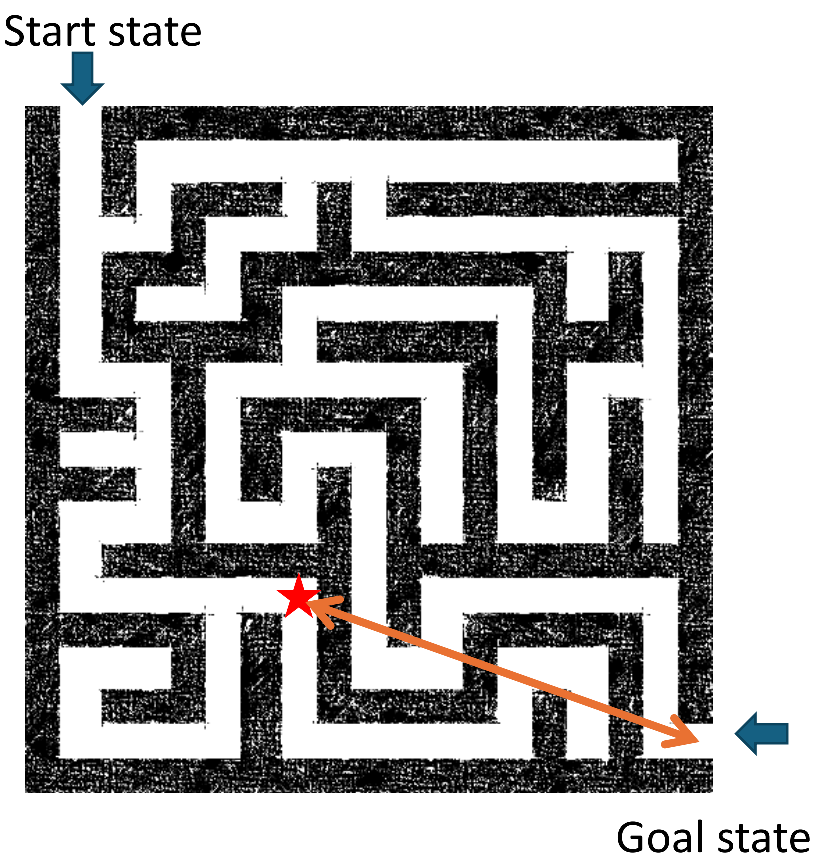
- The agent uses a **heuristic function $h(n)$** to rank nodes in the frontier and always select the most promising node in the frontier for expansion using the **best-first search** strategy.

- Algorithms:
  - Greedy best-first search
  - A* search

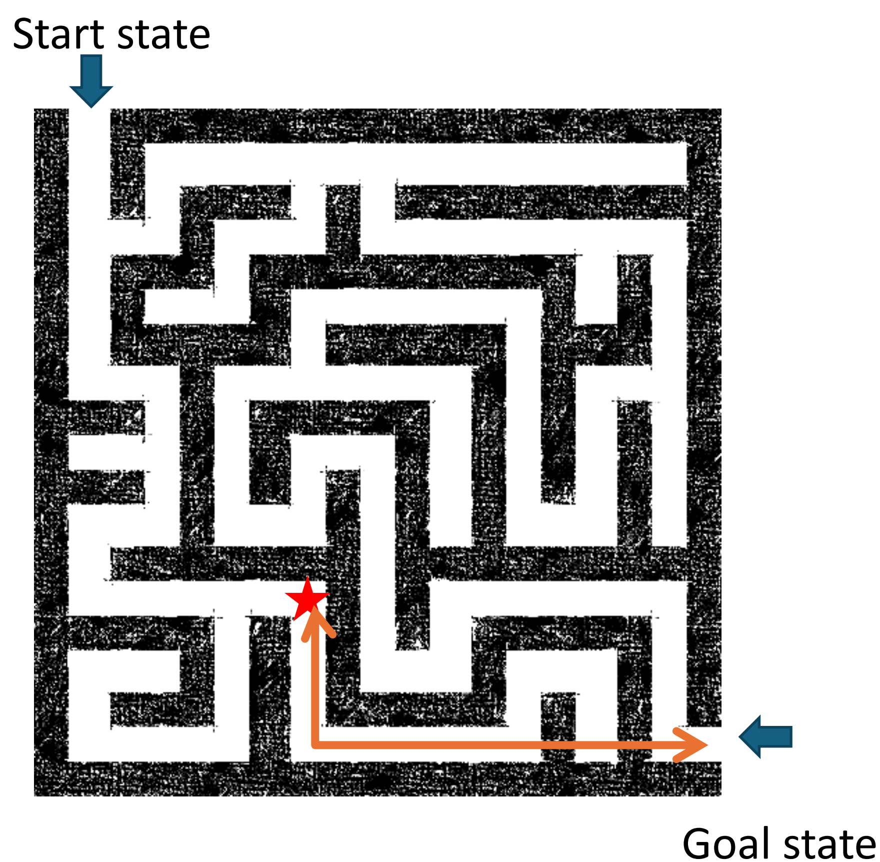
Heuristic Function

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

Euclidean distance

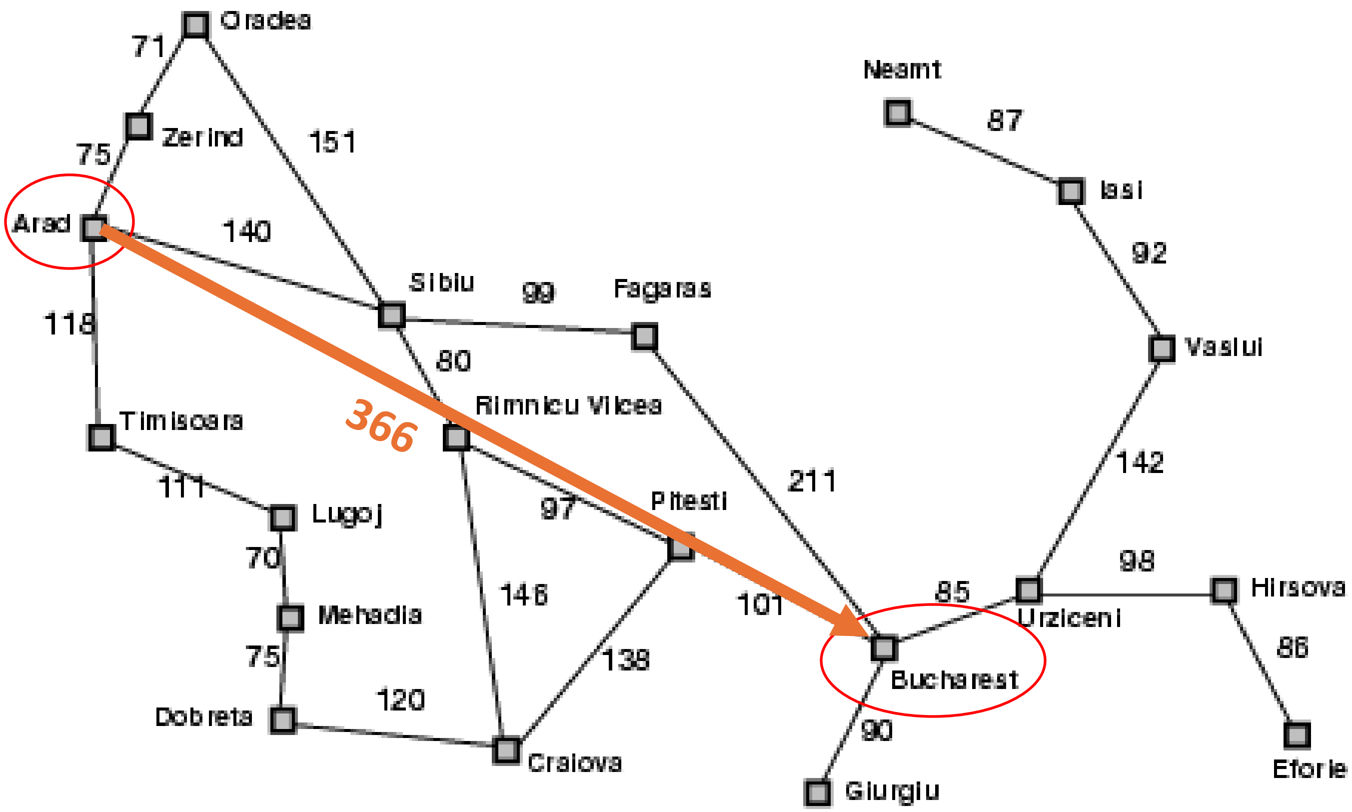


Manhattan distance



Heuristic for the Romania Problem

Estimate the driving distance from Arad to Bucharest using a straight-line distance.



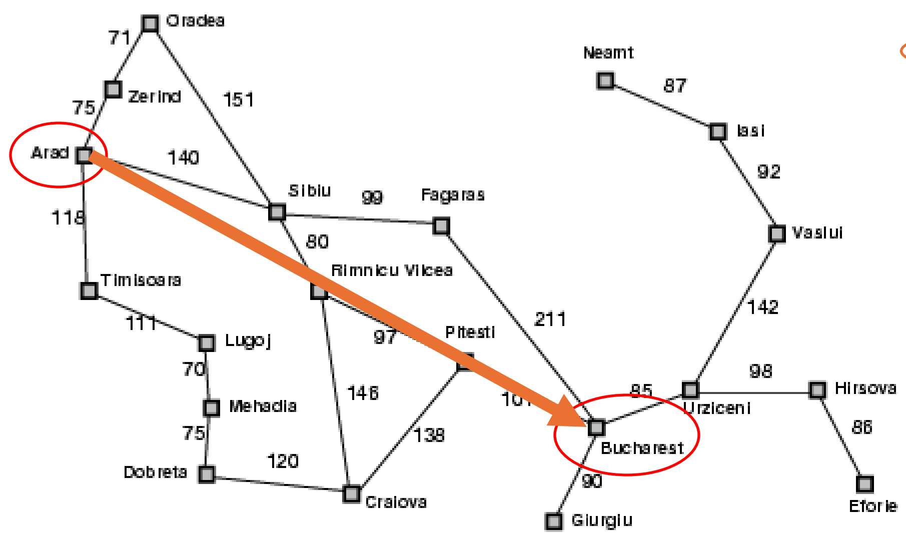
Straight–line distance to Bucharest $h(n)$

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*)

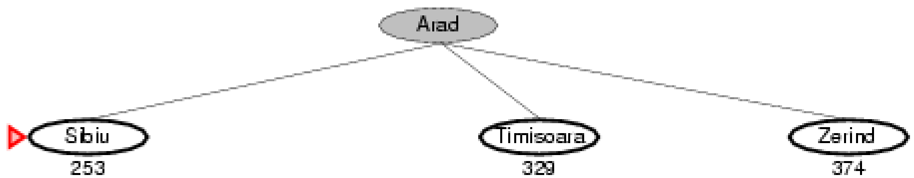
$h(n)=$

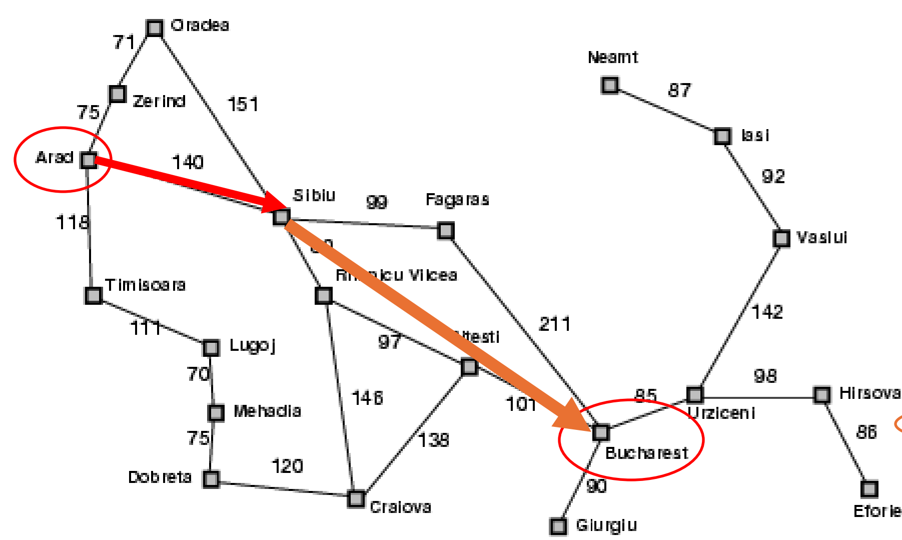


Straight–line distance to Bucharest

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

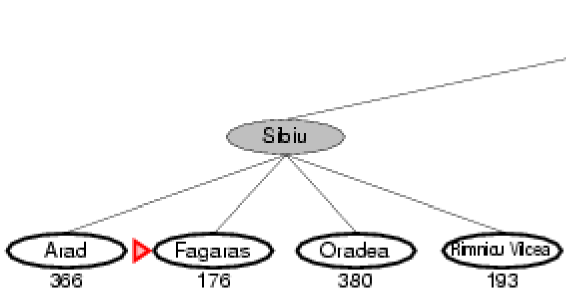


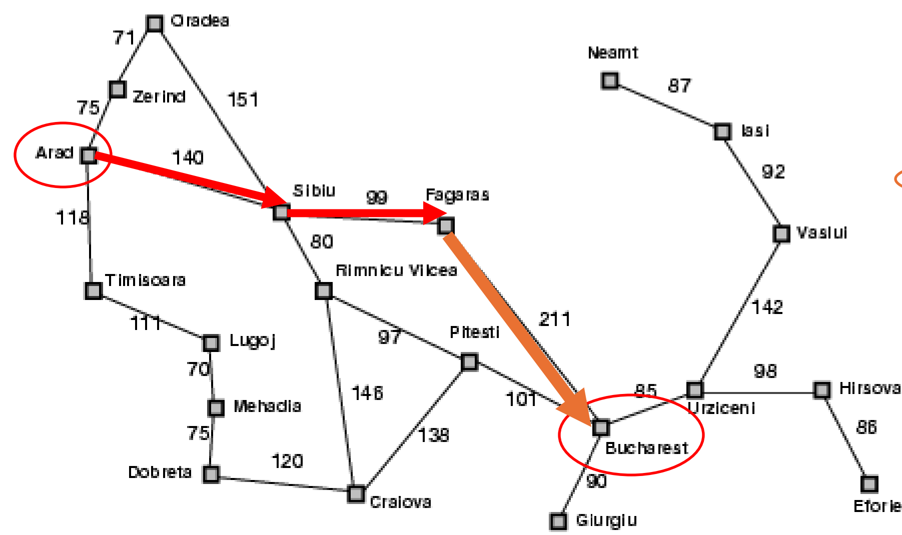


Straight–line distance
to Bucharest

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	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Best-First Search Example

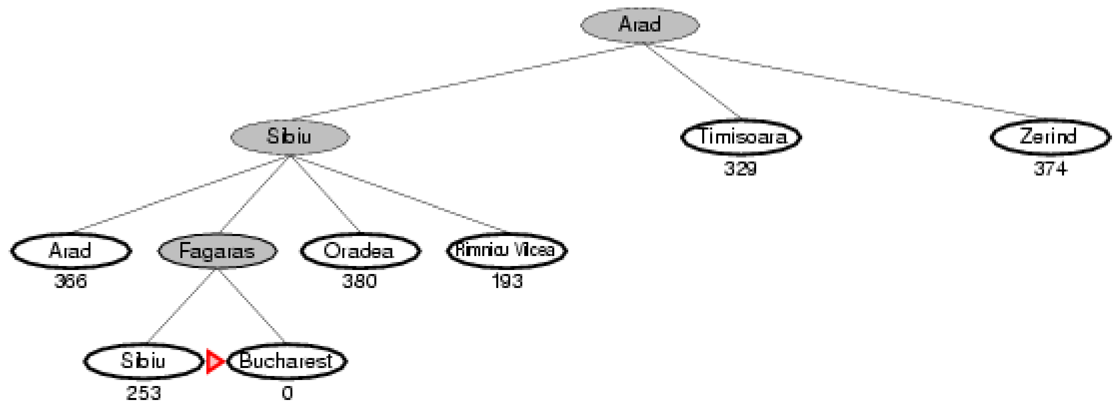




Straight–line distance to Bucharest

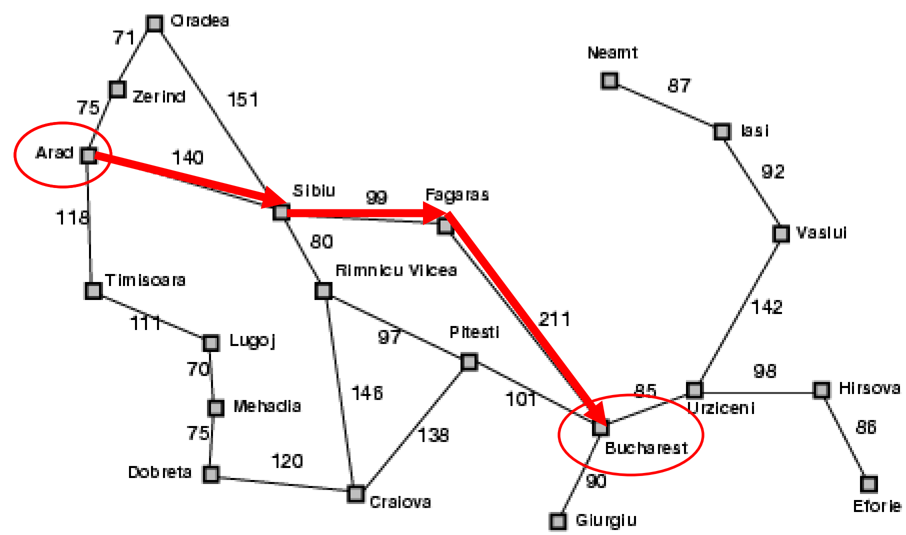
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

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

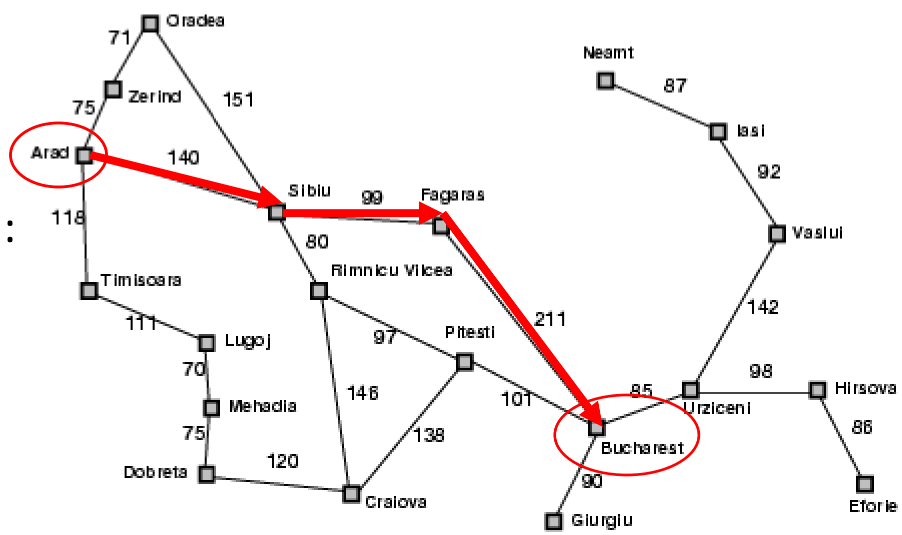
Properties of Greedy Best-First Search

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

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)*

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

See BFS for function EXPAND.

Properties of Greedy Best-First Search

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

- **Optimal?**
  No

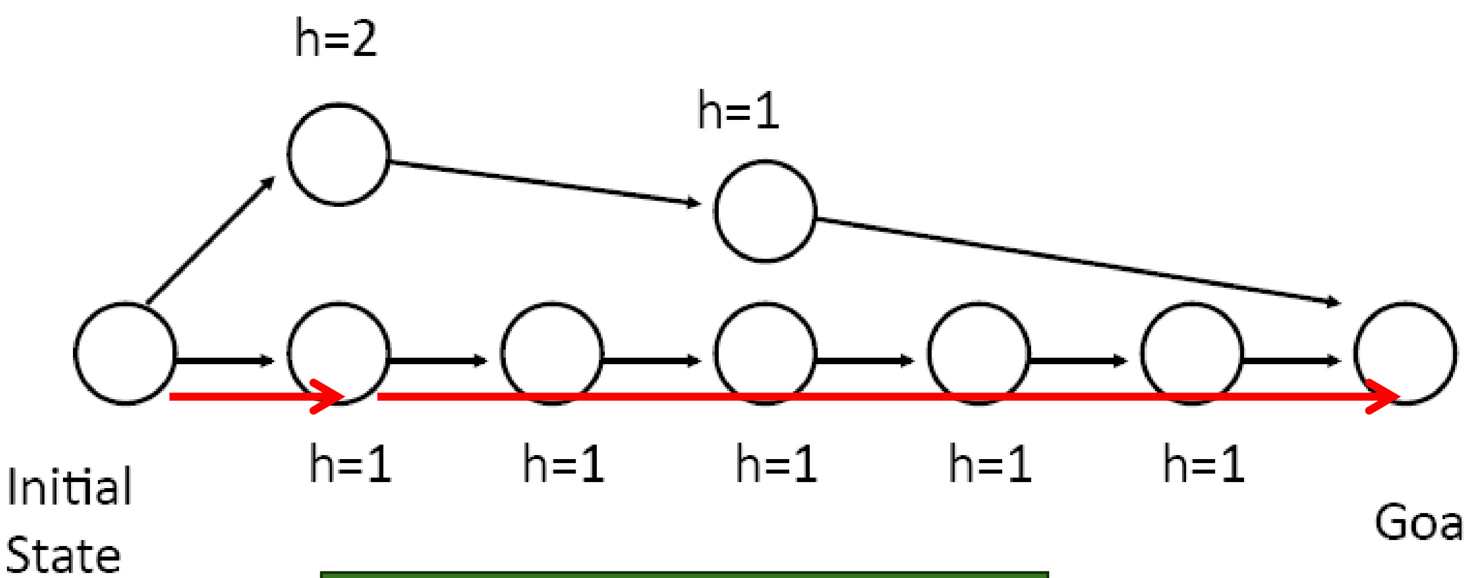
d: depth of the optimal solution
m: max. depth of tree
b: maximum branching factor

- **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.

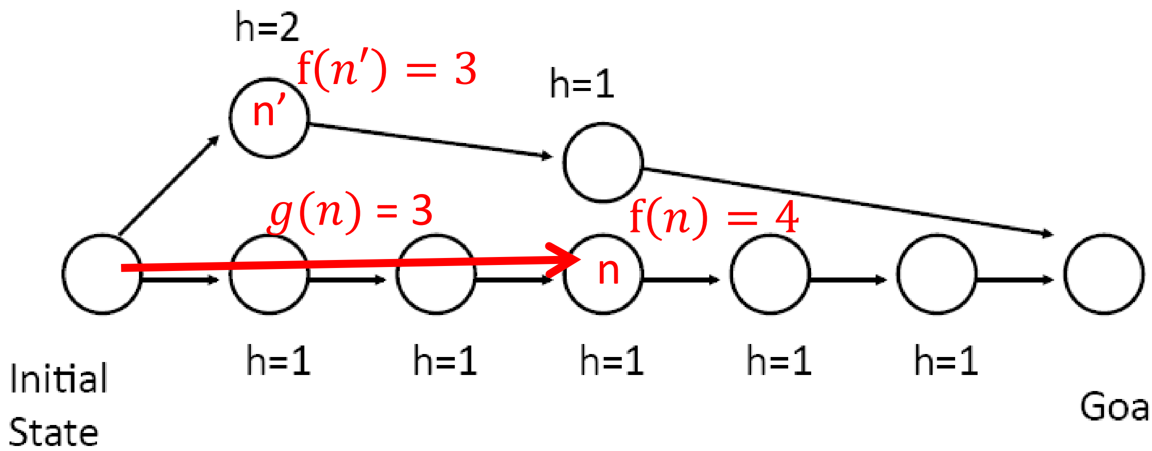
The Optimality Problem of Greedy Best-First search

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



$h = 1$ is always 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)

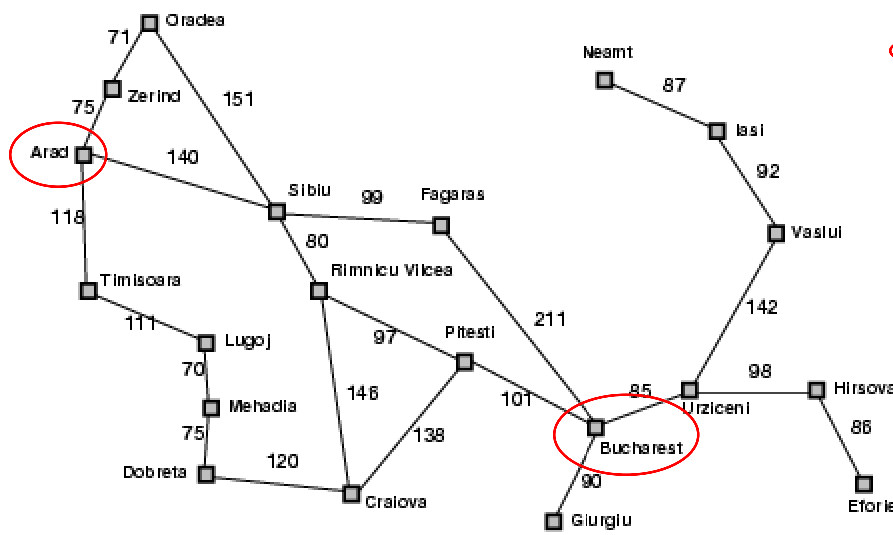
- The agent in the example above will stop at n with $f(n) = 3 + 1 = 4$ and chose the path up with a better $f(n') = 1 + 2 = 3$.

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)

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



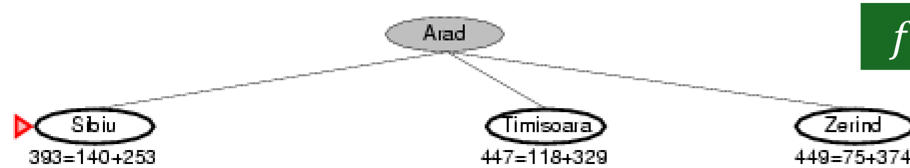
$h(n)$

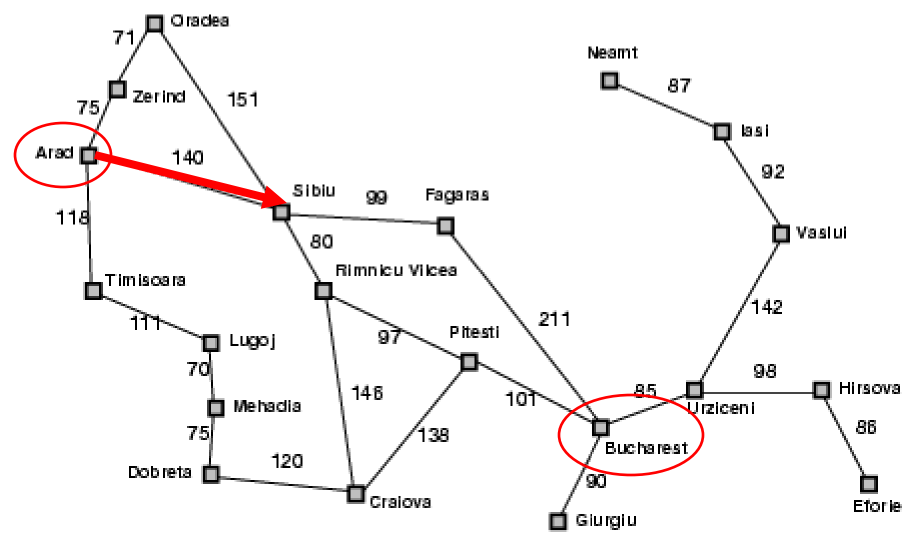
Straight–line distance to Bucharest

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

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





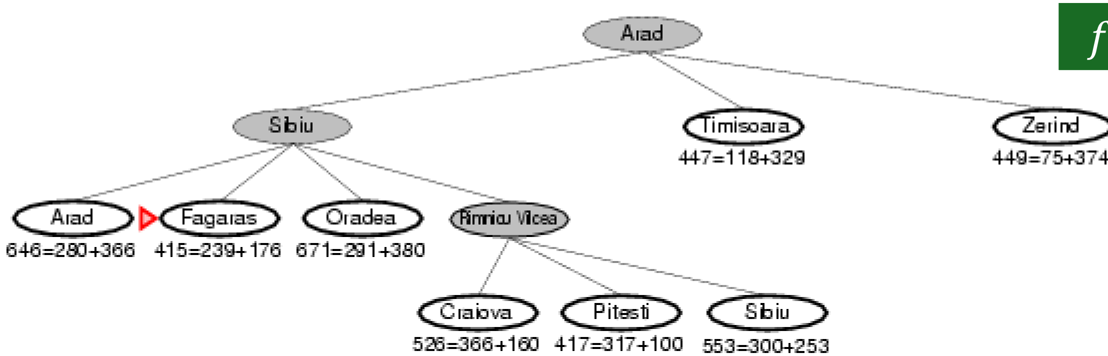
$h(n)$

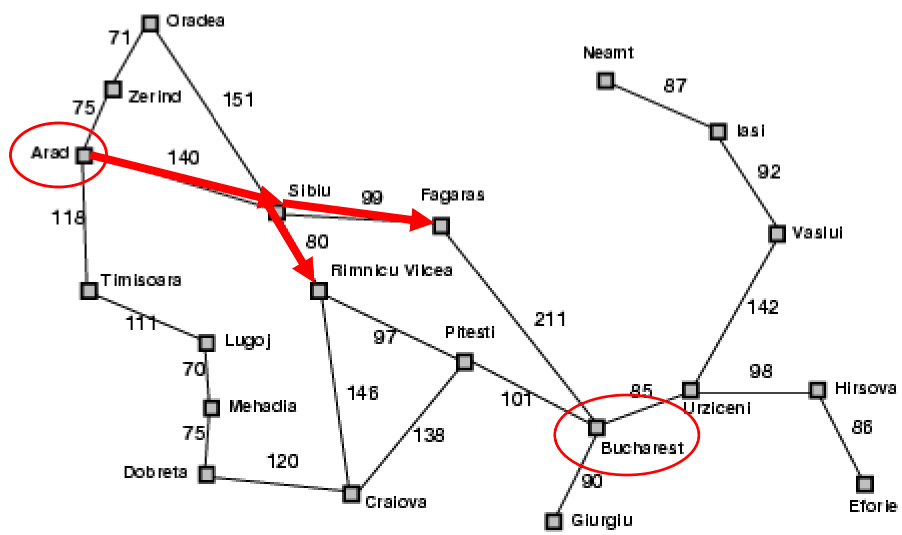
Straight–line distance to Bucharest

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

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

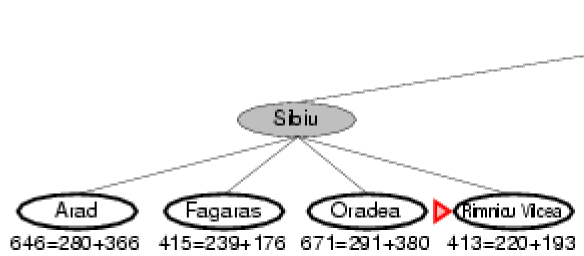




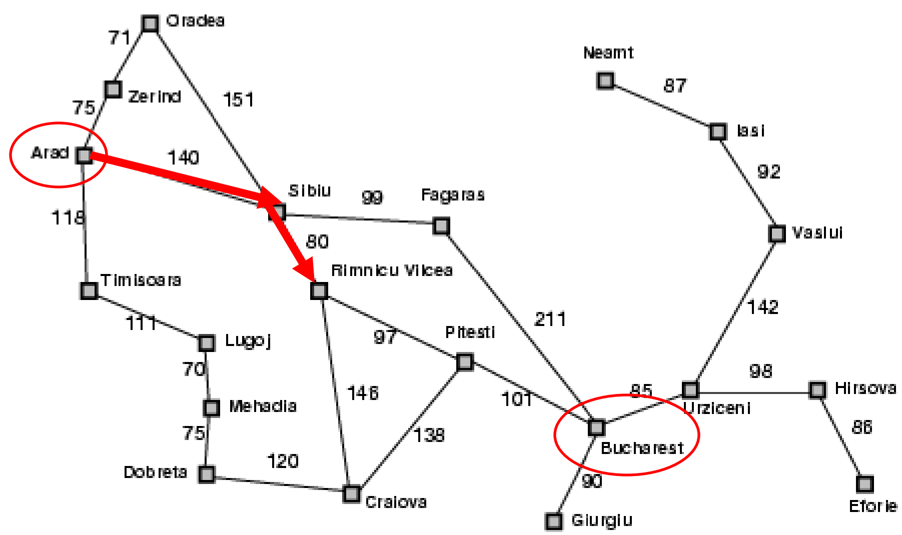
$h(n)$

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



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



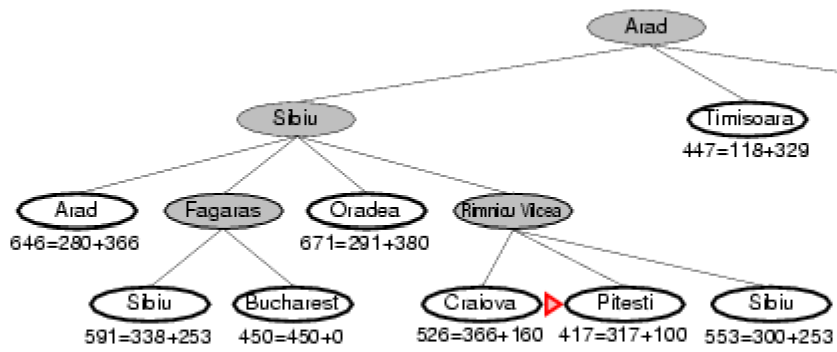
$h(n)$

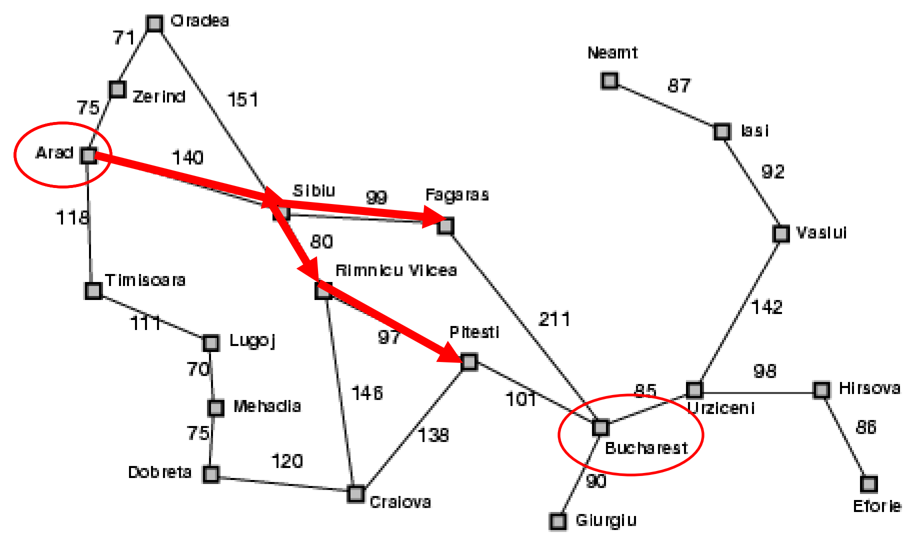
Straight–line distance to Bucharest

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

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





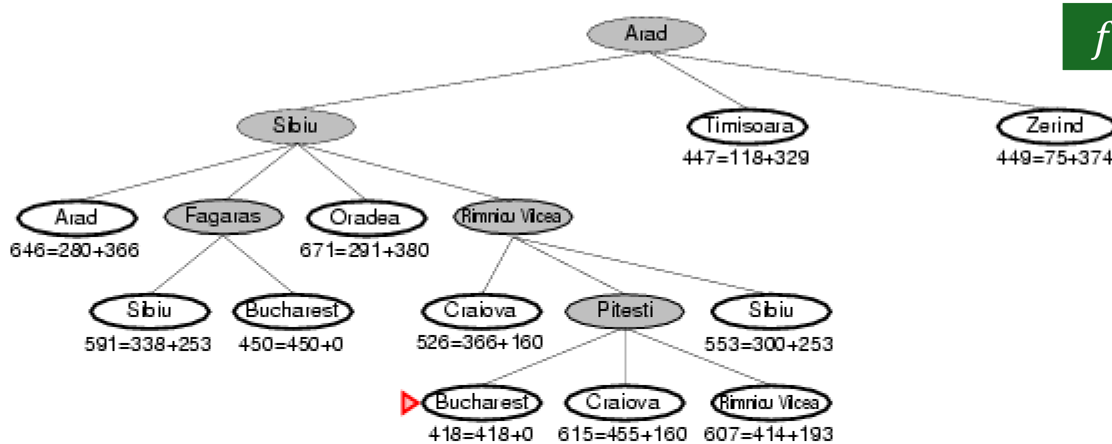
$h(n)$

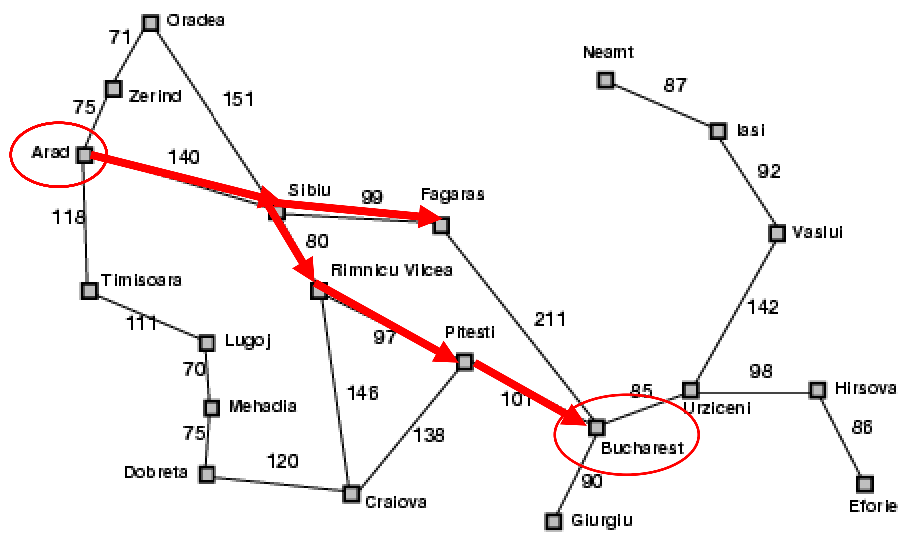
Straight–line distance to Bucharest

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

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





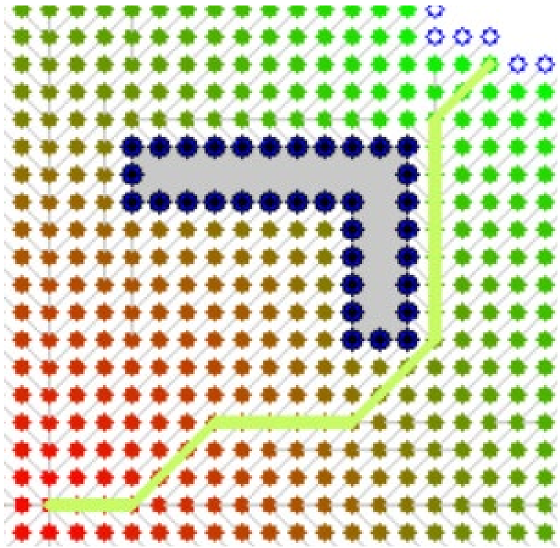
$h(n)$

Straight–line distance to Bucharest

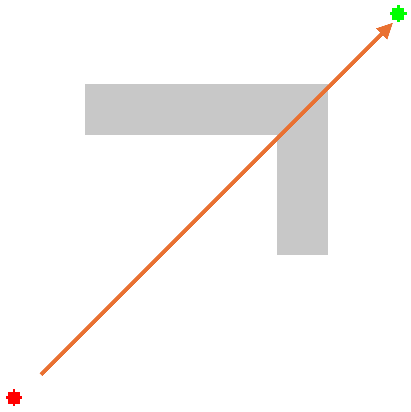
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

BFS vs. A* Search

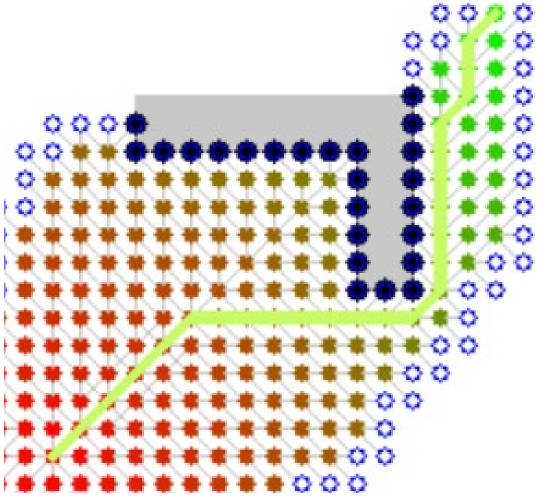
BFS



A*



A*



Source: [Wikipedia]

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 $f(n)$

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

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*

A* is **optimally efficient**

a. No other tree-based search algorithm that uses the same heuristic can expand fewer nodes and still be guaranteed to find the optimal solution.

b. 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*

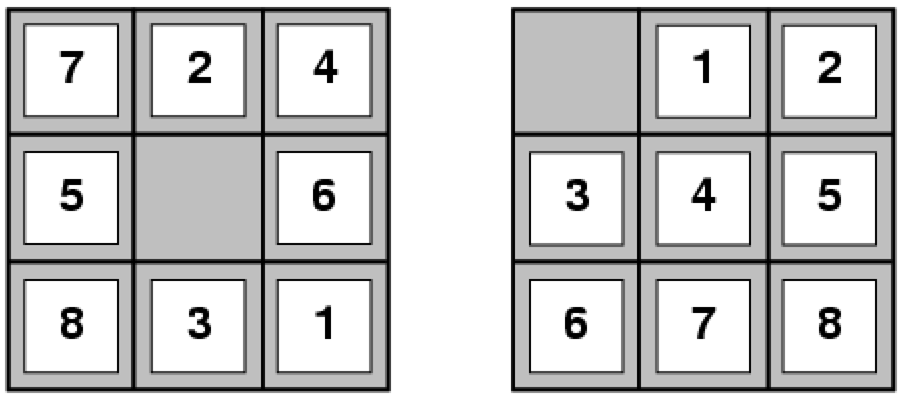
- **Complete?**
  Yes
- **Optimal?**
  Yes
- **Time?**
  Number of nodes for which $f(n) \le C^*$ (exponential)
- **Space?**
  Same as time complexity. This is often too much unless a very good heuristic is know.

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)



Start State　　Goal State

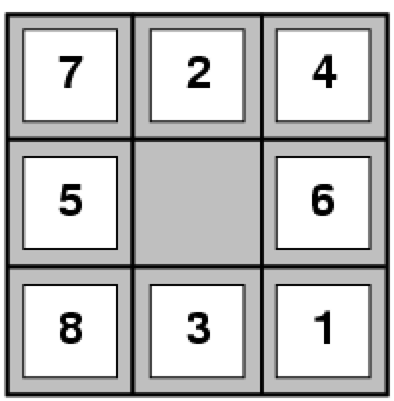
$$h_1(start) = 8$$
$$h_2(start) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$$

1 needs to move 3 positions

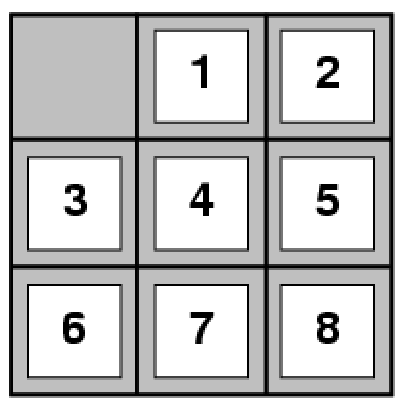
Are h_1 and h_2 admissible?

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

$$h_1(start) = 8$$

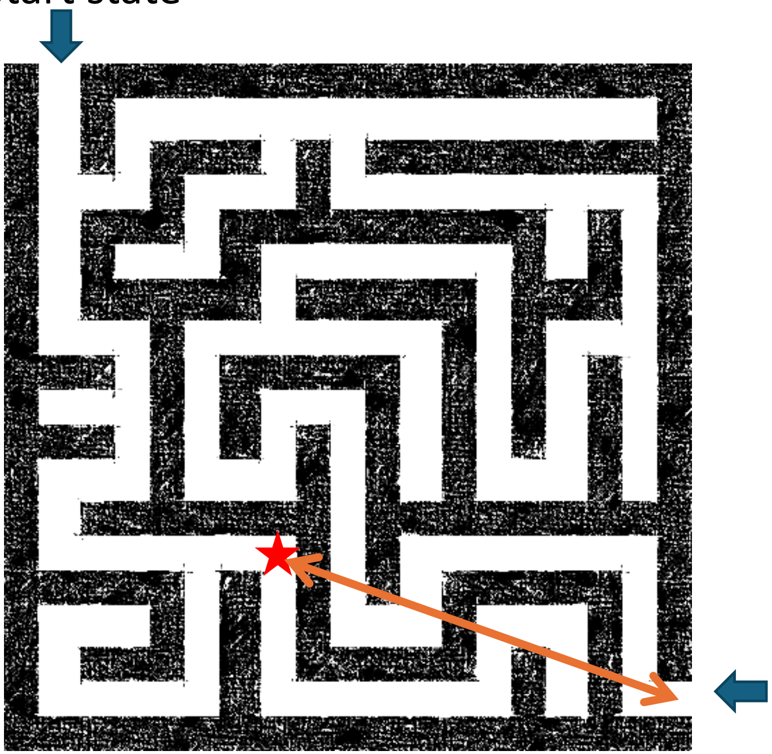
$$h_2(start) \\ = 3+1+2+2+2+3+3+2 \\ = 18$$

Heuristics from Relaxed Problems

What relaxations are used in these two cases?

Euclidean distance

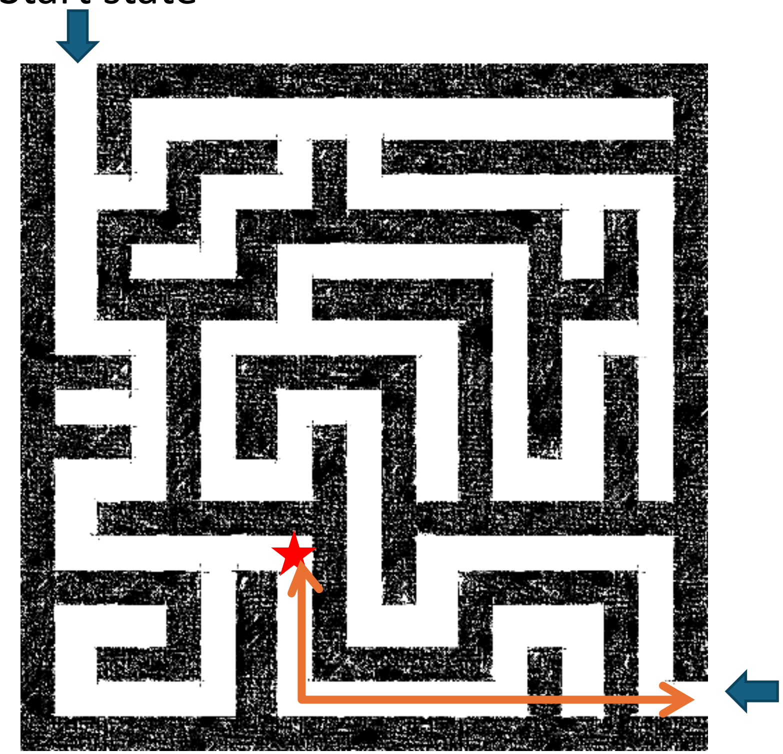
Start state



Goal state

Manhattan distance

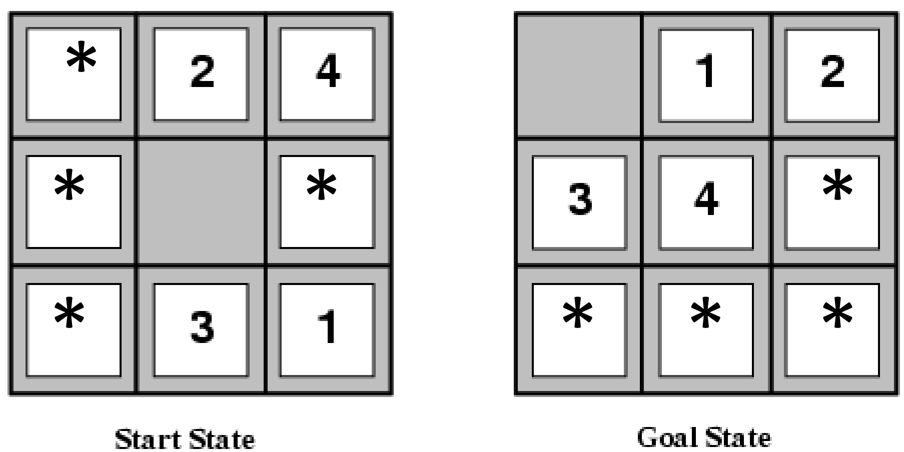
Start state



Goal state

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: What Heuristic is Better?

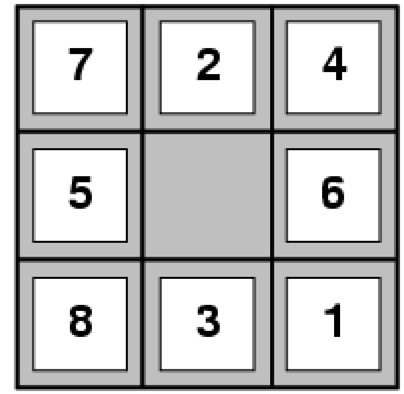
Definition: If h_1 and h_2 are both admissible heuristics and $h_2(n) \geq h_1(n)$ for all n, then
h_2 dominates h_1

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.

Example: Effect of Information in Search

Typical search costs for the 8-puzzle



- 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

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 $h^*(n)$.

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!**

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

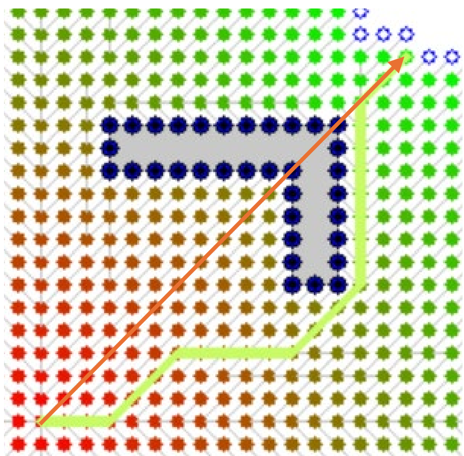
Weighted A* search:	$\boldsymbol{g(n) + W \times h(n)}$	$\boldsymbol{(1 < W < \infty)}$

The presented algorithms are special cases:

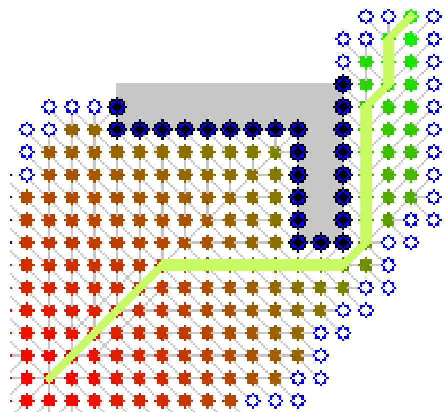
A* search:	$g(n) + h(n)$	$(W = 1)$
Uniform cost search/BFS:	$g(n)$	$(W = 0)$
Greedy best-first search:	$h(n)$	$(W = \infty)$

Example of Weighted A* Search

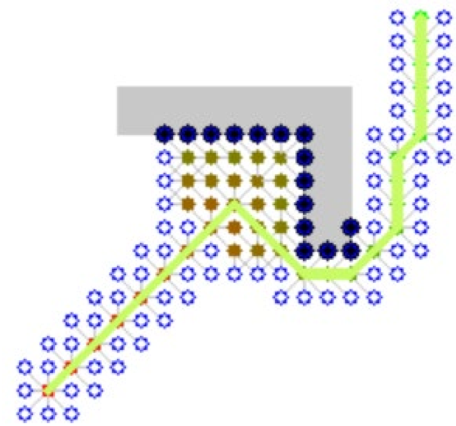
Reduction in the number of expanded nodes



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]

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 $\boldsymbol{f(n)}$.**

Search Strategy	Evaluation function $\boldsymbol{f(n)}$
BFS (Breadth-first search)	$g(n)$ (=uniform path cost)
Uniform-cost Search	$g(n)$ (=path cost)
DFS/IDS (see note below!)	$-g(n)$
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 the poor space complexity and the disadvantages of DFS (not optimal and worse time complexity)

Summary: Uninformed Search Strategies

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 $g(n) \leq C^*$	
DFS	In finite spaces (cycle checking)	No	$O(b^m)$	$O(bm)$
IDS	Yes	If all step costs are equal	$O(b^d)$	$O(bd)$

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

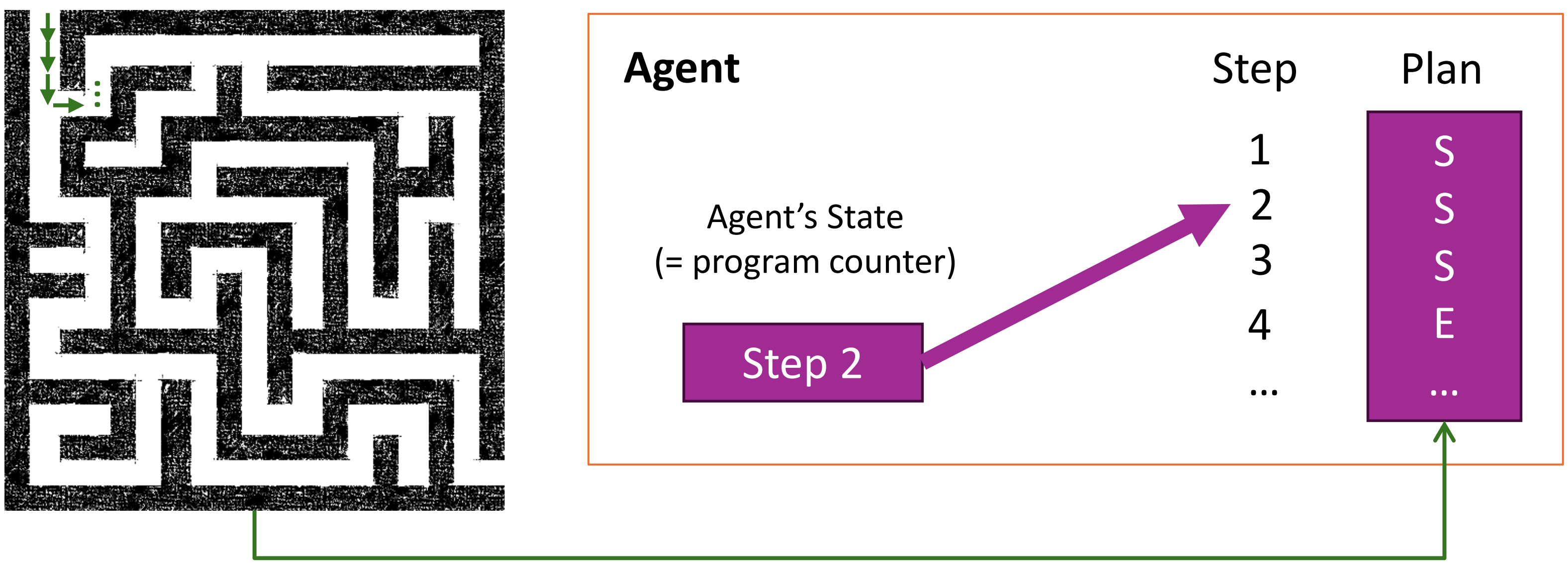
Summary: All Search Strategies

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 $g(n) \leq C^*$	
DFS	In finite spaces (cycles checking)	No	$O(b^m)$	$O(bm)$
IDS	Yes	If all step costs are equal	$O(b^d)$	$O(bd)$
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 $g(n) + h(n) \leq C^*$	

Planning vs. Execution Phase

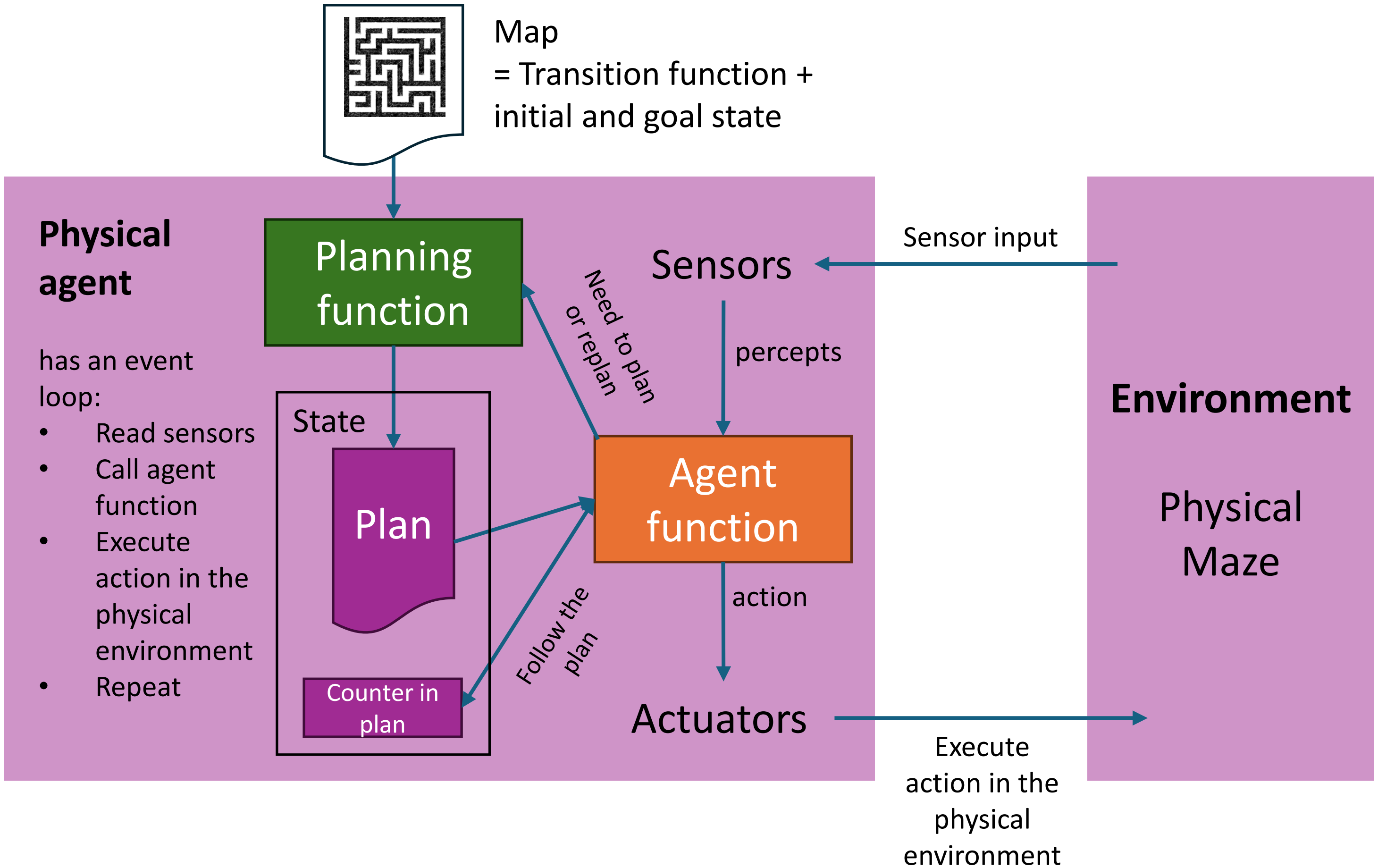
1. Planning is done by **a planning function** using search. The result is a **plan**.
2. The plan can be executed by a **model-based agent function.** The used model is the plan + a step counter so the agent function can follow the plan.

Planning function **Execution function** at step 2 in the plan



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

Complete Planning Agent for a Maze-Solving Agent



- 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).

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 description of the **transition model.**
  - The search tree is built on the fly, and we have to deal with **cycles, redundant paths, and memory management**.

- 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).