Machine Intelligence

2. Problem Solving as Search

Got a Problem? Gotta Solve It!

Álvaro Torralba

AALBORG UNIVERSITET

Fall 2023

A (Classical Search) Problem

→ Problem: Find a route to Madrid.

- **Starting from an initial state . . .** (Aalborg)
- **. . . apply actions . . .** (Using a road segment)
- **. . . to reach a goal state.** (Madrid)
- Performance measure: **Minimize summed-up action costs.** (Road segment lengths)

Another (Classical Search) Problem (The "15-Puzzle")

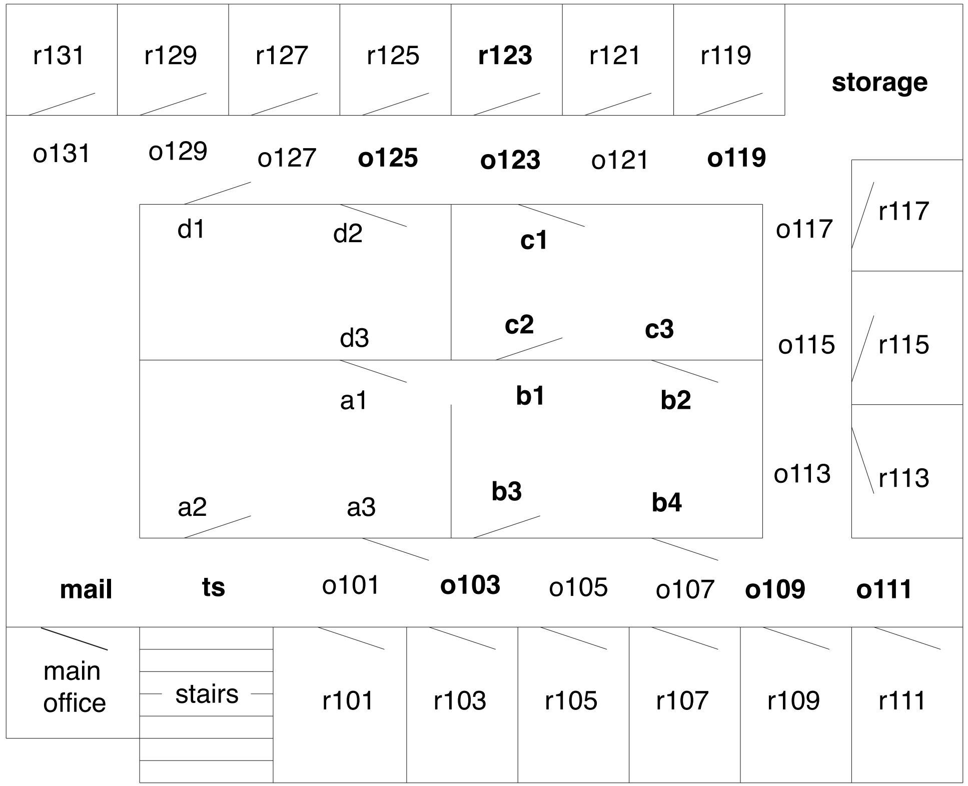
→ Problem: Move tiles to transform left state into right state.

9	2	12	6
5	7	14	13
3	4	1	11
15	10	8	

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

- **Starting from an initial state . . .** (Left)
- **. . . apply actions . . .** (Moving a tile)
- **. . . to reach a goal state.** (Right)
- Performance measure: **Minimize summed-up action costs.** (Each move has cost 1, so we minimize the number of moves)

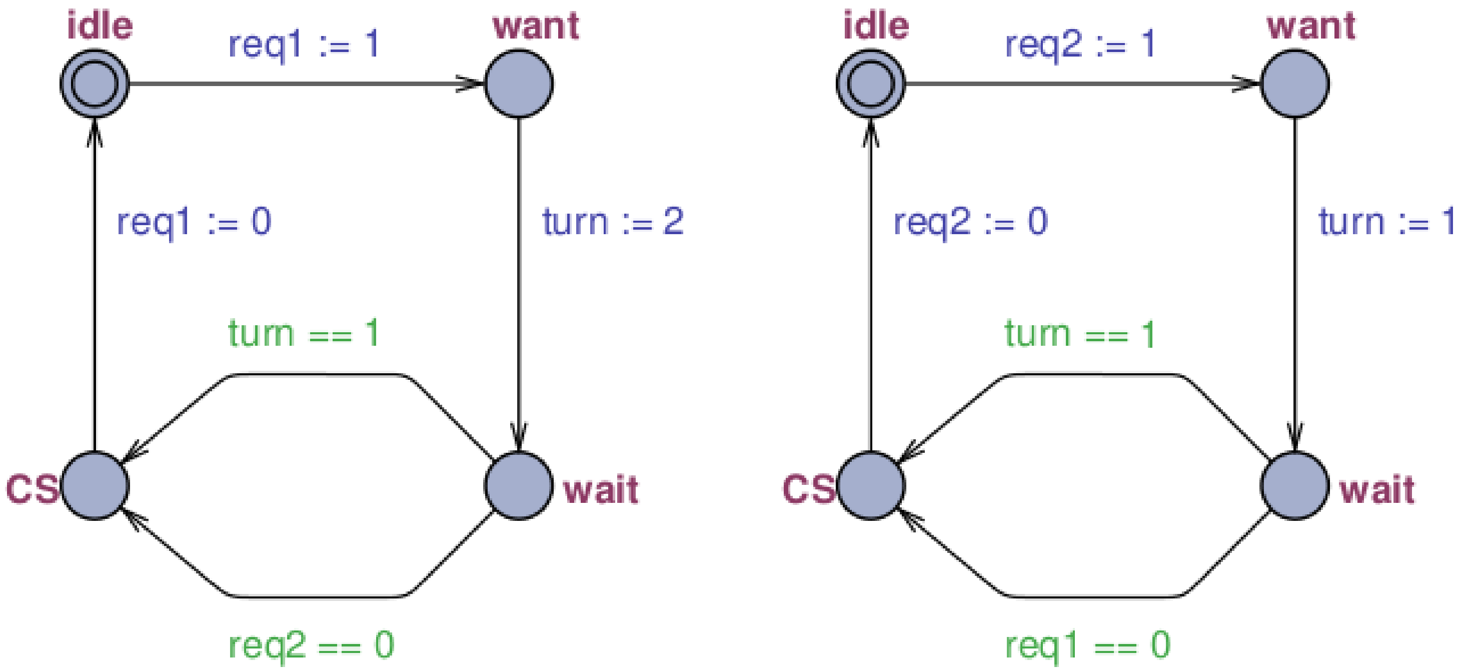
Another (Classical Search) Problem: Office Robot



- States: locations, e.g. r131, storage, o117, c3,. . .
- Actions: move to neighboring locations, e.g. *move_r131_o131*, *move_o119_storage*, *move_b2_c3*,. . .
- Performance measure: **Minimize summed-up action costs.** (Each move has cost proportional to time, so we minimize the time to reach a location)

Yet Another (Classical Search) Problem

→ Problem: Finding bugs in software artifacts.



- **Starting from an initial state . . .** (Both idle)
- **. . . apply actions . . .** (Automaton transitions)
- **. . . to reach a goal state.** (Goal=error: both in critical section CS)
- Performance measure: **Minimize summed-up action costs.** (Each transition has cost 1, so we minimize the length of the error path)

Classical Search Problems

. . . restrict the agent's environment to a very simple setting:

- **Finite numbers of states and actions** (in particular: discrete).
- **Single-agent** (nobody else around).
- **Fully observable** (agent knows everything).
- **Deterministic** (each action has only one outcome).
- **Static** (if the agent does nothing, the world doesn't change).

→ All of these restrictions can be removed, and a lot of work in AI considers such more general settings. We will talk about some of this in later chapters (but not in the present one).

The agent has a certain goal it wants to achieve:
→The agent needs to find a sequence of actions that lead it to a **goal state**: a state in which its goal is achieved.

→ Classical search problems are one of the simplest classes of action choice problems an agent can be facing. Despite that simplicity, classical search problems are very important in practice (see also next slide).

→ And despite that "simplicity", these problems are computationally hard! Typically harder than **NP** . . .

Examples of Classical Search Problems

Just to name a few:

- **Route planning** (e.g. Google Maps).
- **Puzzles** (Rubik's Cube, 15-Puzzle, Towers of Hanoi . . .).
- **Detecting bugs** in software and hardware. Actions = executing instructions
- **Non-player-characters** in computer games.
- **Travelling Salesman Problem (TSP)**. Actions = moves in the graph.
- **Robot assembly sequencing**. Planning of the assembly of complex objects. Actions = robot activities.
- **Attack planning**. Finding a hack into a secured network. Used for regular security testing. Actions = exploits.
- **Query optimization in databases**. Actions = rewriting operations.
- **Sequence alignment** in Bioinformatics. Actions = re-alignment operations.
- **Natural language sentence generation**. Actions = add another word to a partial sentence.

Our Agenda for This Chapter

- **What (Exactly) Is a "Problem":** How are they formally defined?
  $\rightarrow$ **Get ourselves on firm ground.**
- **Basic Concepts of Search:** What are search spaces?
  $\rightarrow$ **Sets the stage for the consideration of search strategies.**
- **(Non-Trivial) Blind Search Strategies:** How to guarantee optimality? How to make the best use of time and memory?
  $\rightarrow$ **Blind search serves to get started, and is used in some applications.**
- **Heuristic Functions:** How are heuristic functions h defined? What are relevant properties of such functions? How can we obtain them in practice?
  $\rightarrow$ **Which "problem knowledge" do we wish to give the computer?**
- **Systematic Search** How to use a heuristic function h while still guaranteeing completeness/optimality of the search.
  $\rightarrow$ **How to exploit the knowledge in a systematic way?**
  $\rightarrow$ **How to exploit the knowledge in a greedy way?**

$\rightarrow$ **Some implementation details, as well as plain breadth-first search and depth-first search, are moved to the "Background" and "Lookup Section" and won't be discussed.**

Before We Begin

→ To precisely specify how we solve search problems algorithmically, we first need a **formal definition**.

That definition really is quite simple:

- The underlying base concept are **state spaces**.
- State spaces are (annotated) directed **graphs**.
- Paths to goal states correspond to **solutions**.
- Cheapest such paths correspond to **optimal** solutions.

A **directed graph** consists of

- a set of **nodes**
- a set of **arcs** (ordered pairs of nodes)

State-Space Problem

Definition (State Space). *A* **state space** *is a 6-tuple* $\Theta = (S, A, c, T, I, S^G)$ *where:*

- S *is a finite set of* **states**.
- A *is a finite set of* **actions**.
- $c : A \mapsto \mathbb{R}_0^+$ *is the* **cost function**.
- $T \subseteq S \times A \times S$ *is the* **transition relation**. *We require that* T *is* **deterministic**, *i.e., for all* $s \in S$ *and* $a \in A$, *there is at most one state* s' *such that* $(s, a, s') \in T$. *If such* (s, a, s') *exists, then* a *is* **applicable** *to* s.
- $I \in S$ *is the* **initial state** *(also called start state).*
- $S^G \subseteq S$ *is the set of* **goal states**.

We say that Θ **has the transition** (s, a, s') *if* $(s, a, s') \in T$. *We also write* $s \xrightarrow{a} s'$, *or* $s \to s'$ *when not interested in* a.
We say that Θ *has* **unit costs** *if, for all* $a \in A$, $c(a) = 1$.

A **Solution** consists of

- For any given start state, a sequence of actions that lead to a goal state
- (optional) a sequence of actions with minimal cost
- (optional) a sequence of actions leading to a goal state with maximal value

State Spaces Terminology

Some commonly used terms:

- s' **successor** of s if $s \rightarrow s'$; s **predecessor** of s' if $s \rightarrow s'$.
- s' **reachable** from s if there exists a sequence of transitions:

$$s = s_0 \xrightarrow{a_1} s_1, \ldots, s_{n-1} \xrightarrow{a_n} s_n = s'$$

  - $n = 0$ possible; then $s = s'$.
  - $a_1, \ldots, a_n$ is called **path** from s to s'.
  - $s_0, \ldots, s_n$ is also called **path** from s to s'.
  - The **cost** of that path is $\sum_{i=1}^{n} c(a_i)$.
- s' **reachable** (without reference state) means reachable from I.
- s is **solvable** if some $s' \in S^G$ is reachable from s; else, s is a **dead end**.

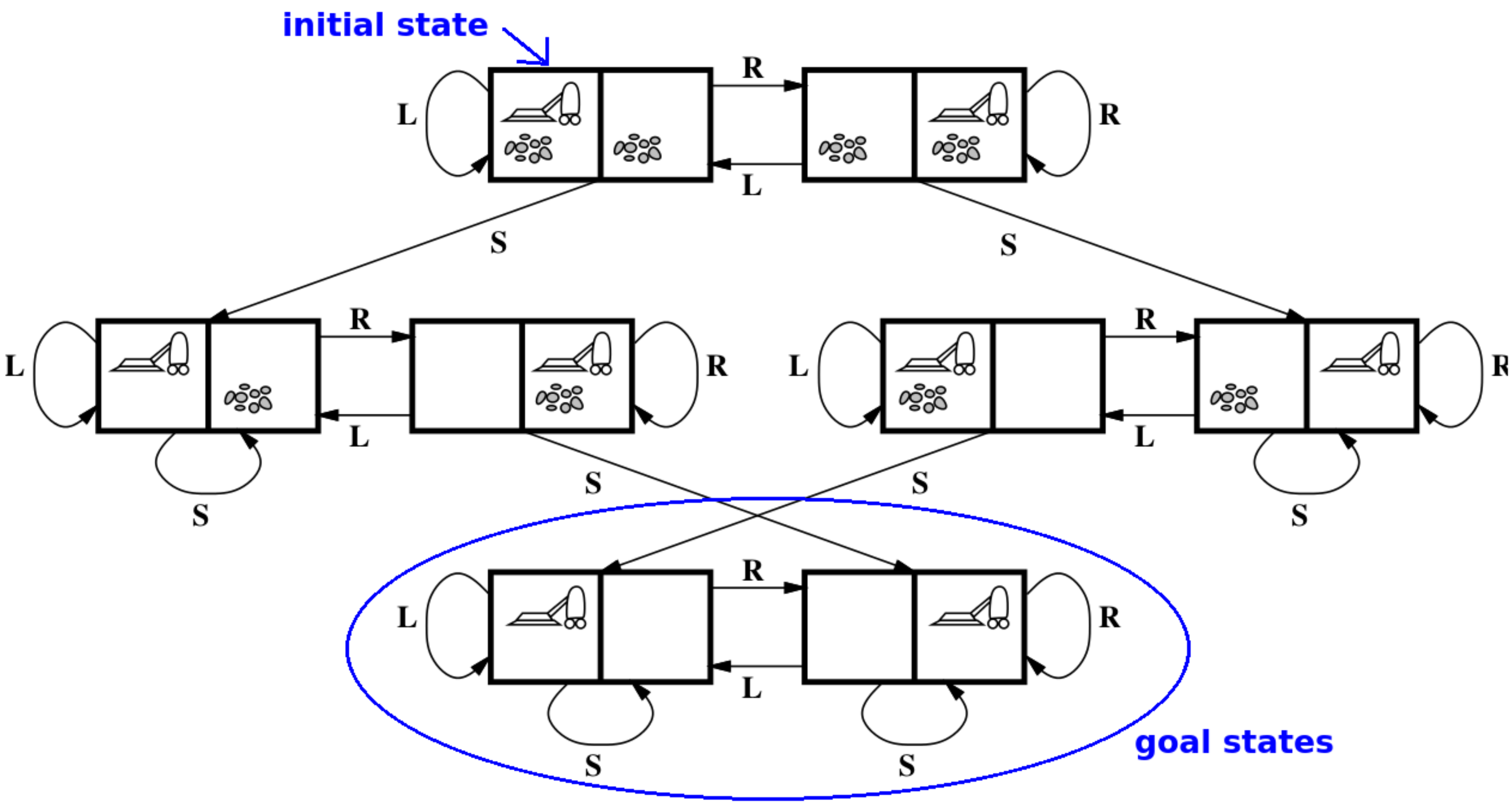
Definition (State Space Solutions). *Let $\Theta = (S, A, c, T, I, S^G)$ be a state space, and let $s \in S$. A* **solution** *for s is a path from s to some $s' \in S^G$. The solution is* **optimal** *if its cost is minimal among all solutions for s. A solution for I is called a* **solution for** Θ. *If a solution exists, then Θ is* **solvable**.

→ **Unsolvable Θ do occur naturally!**

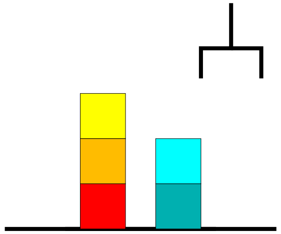
Example Vacuum Cleaner

Example Vacuum Cleaner: State Space

- Starting from state 1 (dirty!) . . .
- . . . go right(R), left (L), or suck (S) . . .
- . . . to clean the apartment.
- Performance measure: Minimize number of actions.



So, Why All the Fuss? Example Blocksworld



- n blocks, 1 hand.
- A single action either takes a block with the hand or puts a block we're holding onto some other block/the table.

blocks	states	blocks	states
1	1	9	4596553
2	3	10	58941091
3	13	11	824073141
4	73	12	12470162233
5	501	13	202976401213
6	4051	14	3535017524403
7	37633	15	65573803186921
8	394353	16	1290434218669921

→ State spaces may be huge. In particular, the state space is typically exponentially large in the size of its specification.

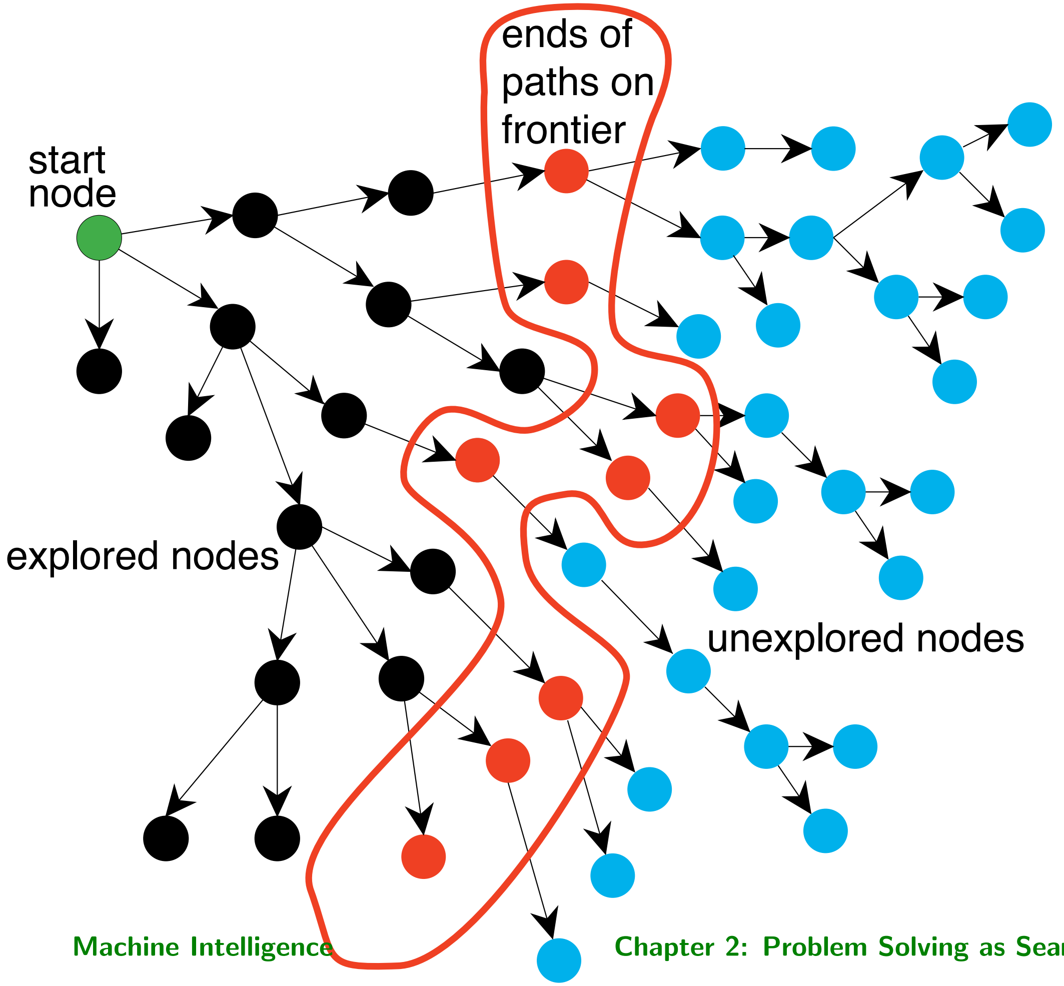
→ In other words: Search problems typically are computationallly hard (e.g., optimal Blocksworld solving is NP-complete).

Graph Search

A state-space problem can be solved by searching in the state-space graph for paths from start states to goal states.

How to "search"? Start at the **initial state**. Then, step-by-step, **expand** a state by generating its successors . . .

→ This does not require the whole graph at once. Only the **Search space**.



Generic Search Algorithm: Best-first search

Input: a graph API(*),
frontier := $\{\langle$InitialState()$\rangle\}$;
explored := $\{\}$;
while *frontier* is not empty:
  select and **remove** node $\langle s_0, \dots, s_k \rangle$ from *frontier*;
  if $GoalTest(s_k)$
    return $\langle s_0, \dots, s_k \rangle$;
  if $s_k \in$ *explored*
    continue
  add s_k to *explored*
  for every action a in Actions(s_k)
    add $\langle s_0, \dots, s_k,$ ChildState$(s, a)\rangle$ to *frontier* ;
end while

(*) The algorithm does not require the complete graph as input. Only needed are:

- **InitialState**(): Returns the initial state of the problem.
- **GoalTest**(s): Returns a Boolean, "true" iff state s is a goal state.
- **Actions**(s): Returns the set of actions that are applicable to state s.
- **ChildState**(s, a): Requires that action a is applicable to state s, i.e., there is a transition $s \xrightarrow{a} s'$. Returns the outcome state s'.
- **Cost**(a): Returns the cost of action a.

→Some variants perform GoalTest and closed list operations at generation time

Search Terminology

Search node n: Contains a *state* reached by the search, plus information about how it was reached.

Path cost $g(n)$: The cost of the path reaching n.

Optimal cost g^*: The cost of an optimal solution path. For a state s, $g^*(s)$ is the cost of a cheapest path reaching s.

Node expansion: Generating all successors of a node, by applying all actions applicable to the node's state s. Afterwards, the *state* s itself is also said to be expanded.

Search strategy: Method for deciding which node is expanded next.

Open list: Set of all *nodes* that currently are candidates for expansion. Also called **frontier**.

Closed list: Set of all *states* that were already expanded. Used only in **graph search**, not in **tree search** (up next). Also called **explored set**.

Tree Search vs. Graph Search

Duplicate Elimination:

- Maintain a closed list.
- Check for each generated state s' whether s' is in the closed list. If so, discard s'.

Tree Search:

- ... is another word for "don't use duplicate elimination".
- Search space is "tree-like": We do not consider the possibility that the same state may be reached from more than one predecessor.
- The same state may appear in many search nodes.
- Main advantage: lower memory consumption (no closed list needed).

Graph Search:

- ... is another word for "use duplicate elimination".
- Search space is "graph-like": We do consider said possibility.

Criteria for Evaluating Search Strategies

Guarantees:

Completeness: Is the strategy guaranteed to find a solution when there is one?

Optimality: Are the returned solutions guaranteed to be optimal?

Complexity:

Time Complexity: How long does it take to find a solution? (Measured in **expanded or generated nodes/states**.)

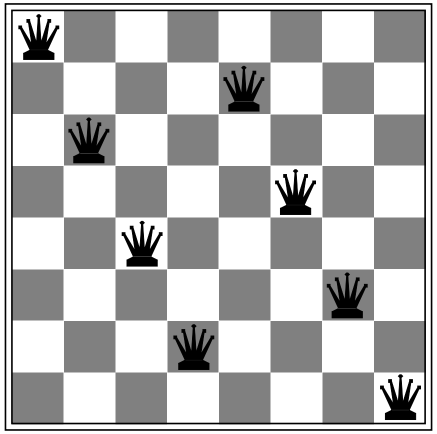
Space Complexity: How much memory does the search require? (Measured in **states**.)

Typical state space features governing complexity:

Branching factor b: How many successors does each state have?

Goal depth d: The number of actions required to reach the shallowest goal state.

Questionnaire



- Chess board, numbering the 8 columns $C_1, \ldots, C_8$ from left to right.
- 8 queens $Q_1, \ldots, Q_8$, each Q_i to be placed "in its own" column C_i.
- We fill the columns left to right, i.e., the actions allow to place Q_i somewhere in C_i, provided all of $Q_1, \ldots, Q_{i-1}$ have already been placed.
- Goal: Placement where no queens attack each other.

Question!

Tree search always terminates in?

(A): 15-Puzzle. **(B): Route Finding.**

(C): Vacuum Cleaning. **(D): 8-Queens.**

→ (A, B, C): No. Tree search does not check for repeated states, so if there are cycles in the state space it may not terminate. For example, in Vacuum Cleaning an infinite search path just keeps moving the robot from left to right and back.

→ (D): Yes, because after adding 8 queens to the board there are no more applicable actions. That is, the *maximum length of a path in the state space* is bounded by 8.

Preliminaries

Blind search vs. informed search:

- **Blind search** does not require any input beyond the problem API.

  Pros and Cons: Pro: No additional work for the programmer. Con: It's not called "blind" for nothing ... same expansion order regardless what the problem actually is. Rarely effective in practice.

- **Informed search** requires as additional input a **heuristic function** h that maps states to estimates of their **goal distance**.

  Pros and Cons: Pro: Typically more effective in practice. Con: Somebody's gotta come up with/implement h.

  $\rightarrow$ Note: In **planning**, h is generated automatically from the declarative problem description (**Chapters 11**).

Preliminaries, ctd.

Blind search strategies covered:

- **Breadth-first search**, **depth-first search**.
- **Uniform-cost search**. Optimal for non-unit costs.
- **Iterative deepening search**. Combines advantages of breadth-first search and depth-first search.

Blind search strategy not covered:

- **Bi-directional search**. Two separate search spaces, one forward from the initial state, the other backward from the goal. Stops when the two search spaces overlap.

Content I will not talk about:

- Breadth-first search and depth-first search.
- The pseudo-code in what follows will use some basic functions.

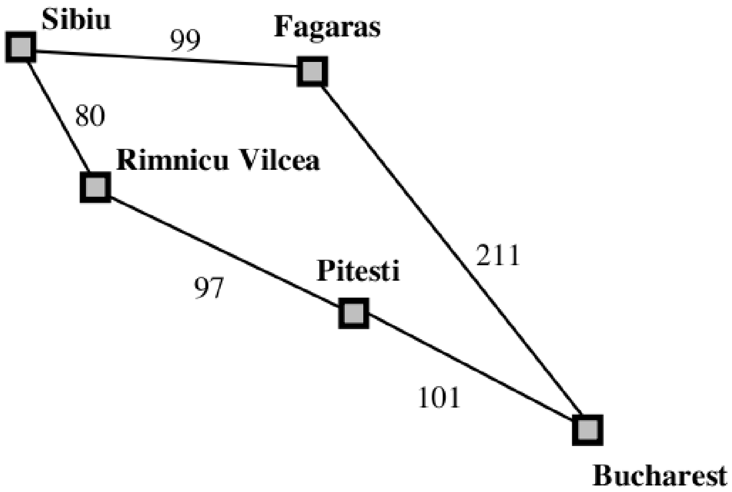
$\rightarrow$ **Both are in the "Background Section". I strongly recommend you read that section. Post any questions you may have in Moodle.**

Uniform-Cost Search: Pseudo-Code

function Uniform-Cost Search(*problem*) **returns** a solution, or failure
  node ← a node n with n.*State*=*problem.InitialState*
  frontier ← a priority queue ordered by **ascending** g, only element n
  explored ← empty set of states
  loop do
    if *Empty?(frontier)* **then return** failure
    n ← *Pop(frontier)*
    if *problem.GoalTest*(n.*State*) **then return** *Solution*(n)
    explored ← *explored*∪n.State
    for each *action* a **in** *problem.Actions*(n.*State*) **do**
      n' ← *ChildNode*(*problem*,n,a)
      if n'.**State**∉ [**explored** ∪ **States**(**frontier**)] **then** *Insert*(n', $g(n')$, *frontier*)
      else if ex. n'' ∈**frontier s.t.** n''.**State**= n'.**State** and $g(n') < g(n'')$ **then**
        replace n'' **in frontier with** n'

- Goal test at node-expansion time.
- Duplicates in frontier replaced in case of cheaper path.

Route Planning in Romania: Uniform-Cost Search



Search protocol:

1. Expand Sibiu, generating Rimnicu $g = 80$, Fagaras $g = 99$.
2. Expand Rimnicu, generating Pitesti $g = 80 + 97 = 177$ (as well as Sibiu which is already explored and thus pruned).
3. Expand Fagaras, generating **Bucharest** $g = 99 + 211 = 310$.
4. Expand Pitesti, generating Bucharest $g = 177 + 101 = 278$;
   Replace Bucharest $g = 310$ **with Bucharest** $g = 278$ **in frontier!**
5. Expand Bucharest $g = 278$.

Uniform-Cost Search: Guarantees and Complexity

Lemma. *Uniform-cost search is equivalent to Dijkstra's algorithm on the state space graph.* (Obvious from the definition of the two algorithms.)

→ **The only differences are: (a) we generate only a part of that graph incrementally, whereas Dijkstra inputs and processes the whole graph[1]; (b) we stop when we reach any goal state (rather than a fixed target state given in the input).**

Theorem. *Uniform-cost search is optimal.* (Because Dijkstra's algorithm is optimal.)

- **Completeness:** Yes, thanks to duplicate elimination and our assumption that the state space is finite.
- **Time complexity:** $O(b^{1+\lfloor g^*/\epsilon \rfloor})$ where g^* denotes the cost of an optimal solution, and ϵ is the positive cost of the cheapest action.
- **Space complexity:** Same as time complexity.

[1] Interesting historical fun fact: this is not necessarily how Dijkstra thought of it **?**.

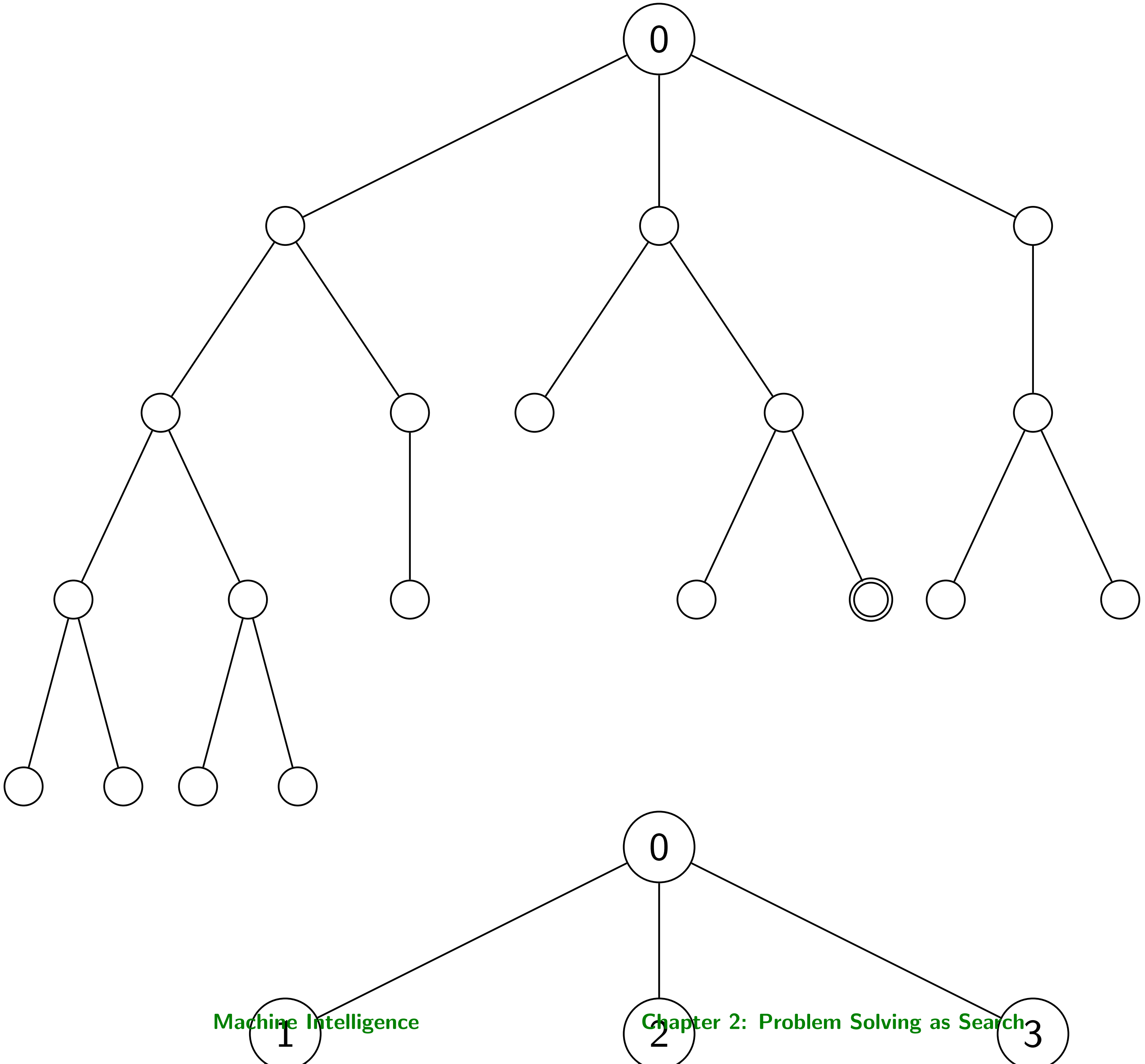
Iterative Deepening Search: Pseudo-Code

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

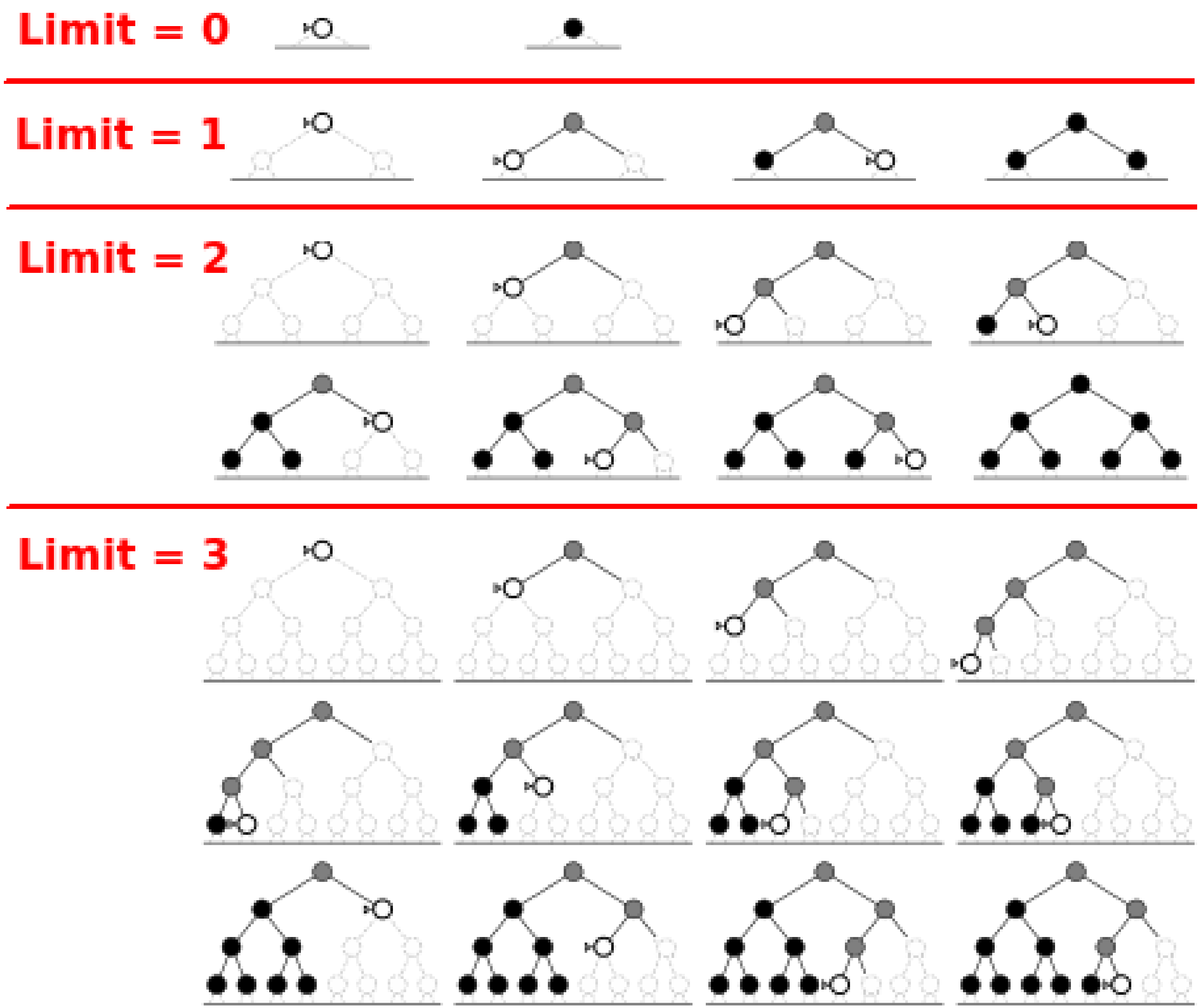
```
function Depth-Limited Search(problem, limit) returns a solution, or failure/cutoff
  node ← a node n with n.state=problem.InitialState
  return Recursive-DLS(node, problem, limit)

function Recursive-DLS(n, problem, limit) returns a solution, or failure/cutoff
  if problem.GoalTest(n.State) then return the empty action sequence
  if limit = 0 then return cutoff
  cutoffOccured ← false
  for each action a in problem.Actions(n.State) do
    n′ ← ChildNode(problem,n,a)
    result ← Recursive-DLS(n′, problem, limit−1)
    if result = cutoff then cutoffOccured ← true
      else if result ≠ failure then return a ○ result
  if cutoffOccured then return cutoff else return failure
```

Iterative deepening: an example



Iterative Deepening Search: Illustration



Iterative Deepening Search: Guarantees and Complexity

"Iterative Deepening Search=
Keep doing the same work over again until you find a solution."

BUT: **Optimality?** Yes![2] **Completeness?** Yes! **Space complexity?** $O(b\,d)$.
Repeated computation: depth-bounded search k repeats computations of depth-bounded search $k-1$. **How bad is it?**

Question!

Assume branching factor $b = 10$, and goal depth $d = 5$. By which factor we increase the amount of explored states with respect to breadth-first search?

(A): $\approx 10\%$ **(B):** $\approx 50\%$ **(C):** $\approx 100\%$ **(D):** $\approx 1000\%$

Not as problematic as it looks!: constant overhead of $(b/(b-1))$.
Time complexity:

Breadth-First-Search	$b + b^2 + \cdots + b^{d-1} + b^d \in \mathbf{O(b^d)}$
Iterative Deepening Search	$(d)b + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + 1b^d \in \mathbf{O(b^d)}$

Example: $b = 10, d = 5$

Breadth-First Search	$10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$
Iterative Deepening Search	$50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$

$\rightarrow$ IDS combines the advantages of breadth-first and depth-first search. It may be the preferred blind search method in large state spaces with unknown solution depth.

$\rightarrow$ Videos illustrating vs. depth-first search: `http://movingai.com/dfid.html`

[2] For unit costs. Extension to general action costs possible.

Blind Search Strategies: Overview

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]
Time	$O(b^d)$	$O(b^{1+\lfloor g^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor g^*/\epsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$

b	finite branching factor
d	goal depth
m	maximum depth of the search tree
l	depth limit
g^*	optimal solution cost
$\epsilon > 0$	minimal action cost

Footnotes:
[a] if b is finite
[b] if action costs $\geq \epsilon > 0$
[c] if action costs are unit
[d] if both directions use breadth-first search

(Not) Playing Stupid

→ **Problem: Find a route to Madrid.**

- **"Look at all locations 10km distant from Aalborg, look at all locations 20km distant from Aalborg, . . ." = Breadth-first search/Uniform-cost search.**
- **"Just keep choosing arbitrary roads, following through until you hit an ocean, then back up . . ." = Depth-first search.**
- **"Focus on roads that go the right direction." = Informed search!**

Informed Search: Basic Idea

Recall: Search strategy=**how to choose the next node to expand?**

- **Blind Search**: Rigid procedure using the same expansion order no matter which problem it is applied to.

  → Blind search has 0 knowledge of the problem it is solving.

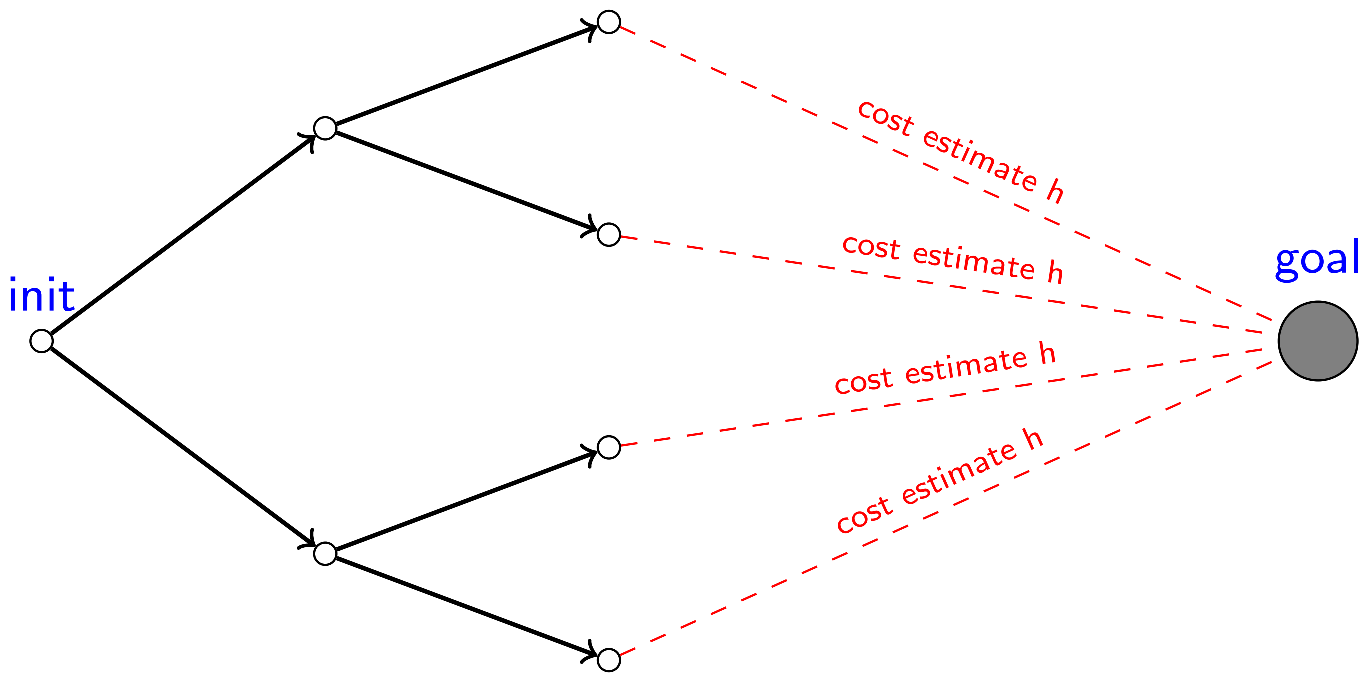
  → It can't "focus on roads that go the right direction", because it has no idea what "the right direction" is.

- **Informed Search**: Knowledge of the "goodness" of expanding a state s is given in the form of a **heuristic function** $h(s)$, which estimates the cost of an optimal (cheapest) path from s to the goal.

  → "$h(s)$ larger than where I came from $\implies$ seems s is not the right direction."

→ Informed search is a way of giving the computer knowledge about the problem it is solving, thereby stopping it from doing stupid things.

Informed Search: Basic Idea, ctd.



→ Heuristic function h estimates the cost of an optimal path from a state s to the goal; search prefers to expand states s with small $h(s)$.

Heuristic Functions

Definition (Heuristic Function). *Let Π be a problem with states S. A* **heuristic function**, *short* **heuristic**, *for Π is a function $h : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ so that, for every goal state s, we have $h(s) = 0$.*

The **perfect heuristic** *h^* is the function assigning every $s \in S$ the cost of a cheapest path from s to a goal state, or ∞ if no such path exists.*

Notes:

- We also refer to $h^*(s)$ as the **goal distance** of s.
- $h(s) = 0$ on goal states: If your estimator returns "I think it's still a long way" on a goal state, then its "intelligence" is, um ...
- Return value ∞: To indicate dead ends, from which the goal can't be reached anymore.
- The value of h depends only on the *state* s, not on the *search node* (i.e., the path we took to reach s). I'll sometimes abuse notation writing "$h(n)$" instead of "$h(n.\text{State})$".

Heuristic Functions: The Eternal Trade-Off

Distance "estimate"? (h is an arbitrary function in principle!)

- We want h to be **accurate** (aka: **informative**), i.e., "close to" the actual goal distance.
- We also want it to be fast, i.e., a small **overhead** for computing h.
- **These two wishes are in contradiction!**

  $\rightarrow$ **Extreme cases?** $h = 0$: no overhead at all, completely un-informative.
  $h = h^*$: perfectly accurate, overhead=solving the problem in the first place.

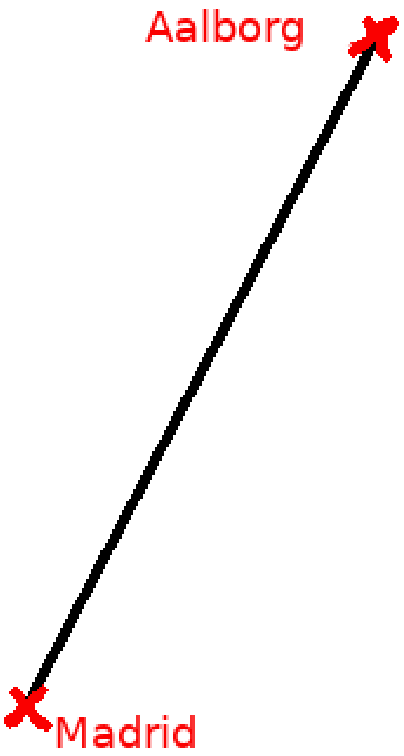
$\rightarrow$ We need to trade off the accuracy of h against the overhead for computing $h(s)$ on every search state s.

So, how to? $\rightarrow$ **Given a problem Π, a heuristic function h for Π can be obtained as goal distance within a simplified (relaxed) problem Π'.**

Heuristic Functions from Relaxed Problems: Example 1

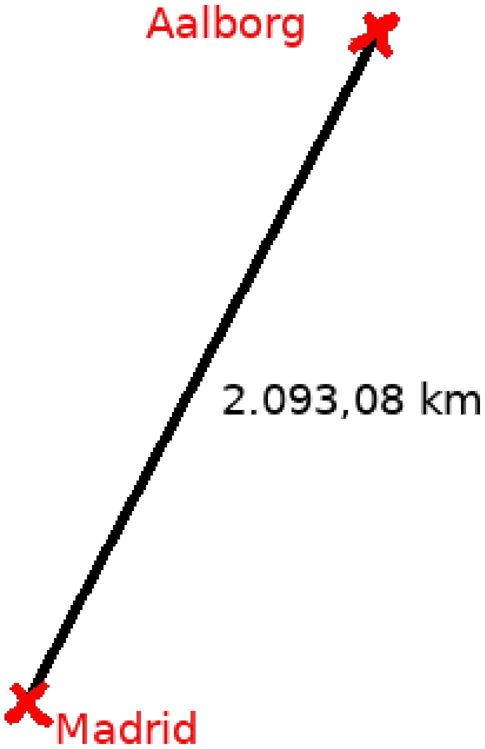
Problem II: Find a route from Aalborg to Madrid.

Heuristic Functions from Relaxed Problems: Example 1



Relaxed Problem Π': Throw away the map.

Heuristic Functions from Relaxed Problems: Example 1



Heuristic function h: Straight line distance.

Heuristic Functions from Relaxed Problems: Example 2

9	2	12	6
5	7	14	13
3	4	1	11
15	10	8	

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

- **Problem Π: Move tiles to transform left state into right state.**
- **Relaxed Problem Π': Allow to move each tile to any cell in a single move, regardless of the situation.**
- Heuristic function h: Number of **misplaced tiles**. Here: 13.

Heuristic Functions from Relaxed Problems: Example 3

9	2	12	6
5	7	14	13
3	4	1	11
15	10	8	

—

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

- **Problem Π: Move tiles to transform left state into right state.**
- **Relaxed Problem Π': Allow to move each tile to any neighbor cell, regardless of the situation.**
- Heuristic function h: **Manhattan distance**. Here: 36.

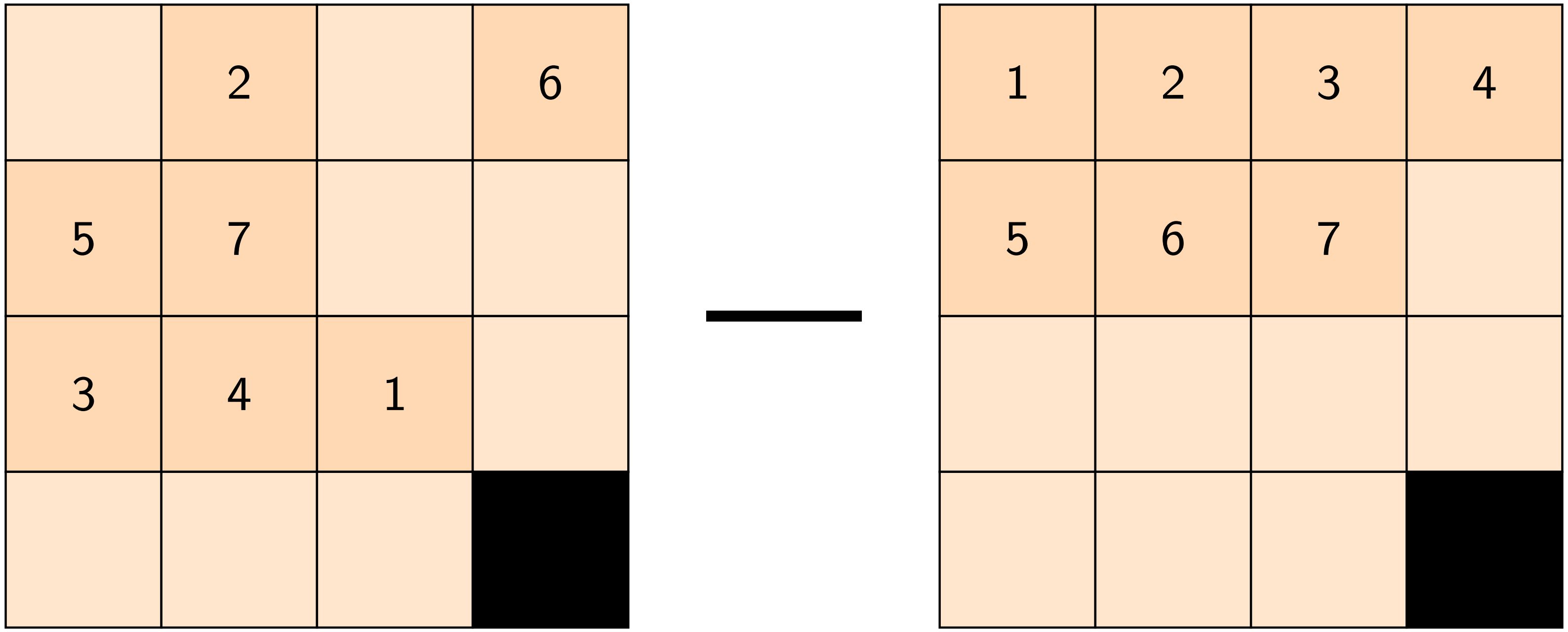
Heuristic Functions from Relaxed Problems: Example 4

9	2	12	6
5	7	14	13
3	4	1	11
15	10	8	

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

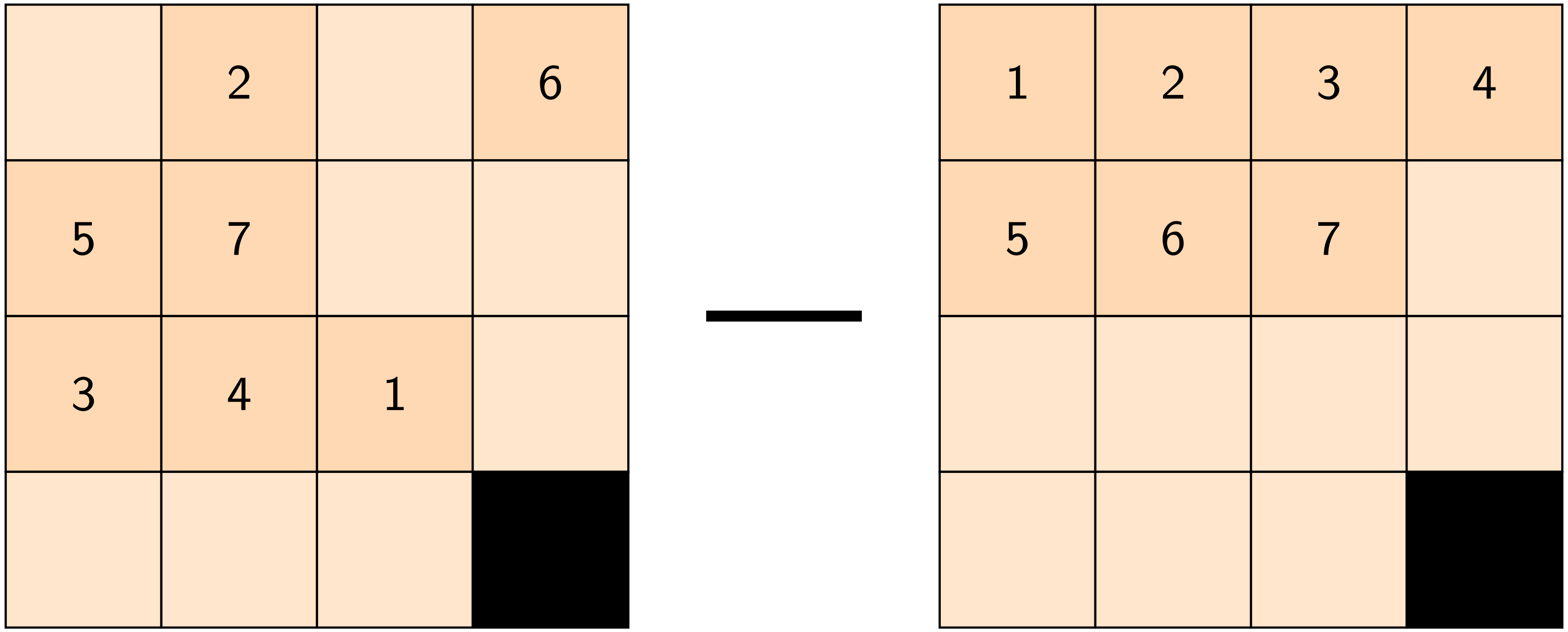
Problem Π: Move tiles to transform left state into right state.

Heuristic Functions from Relaxed Problems: Example 4



Relaxed Problem Π': Don't distinguish tiles 8–15.

Heuristic Functions from Relaxed Problems: Example 4



Heuristic function h: Length of solution to reduced puzzle.

Heuristic Function Pitfalls: Example Path Planning

h^*:

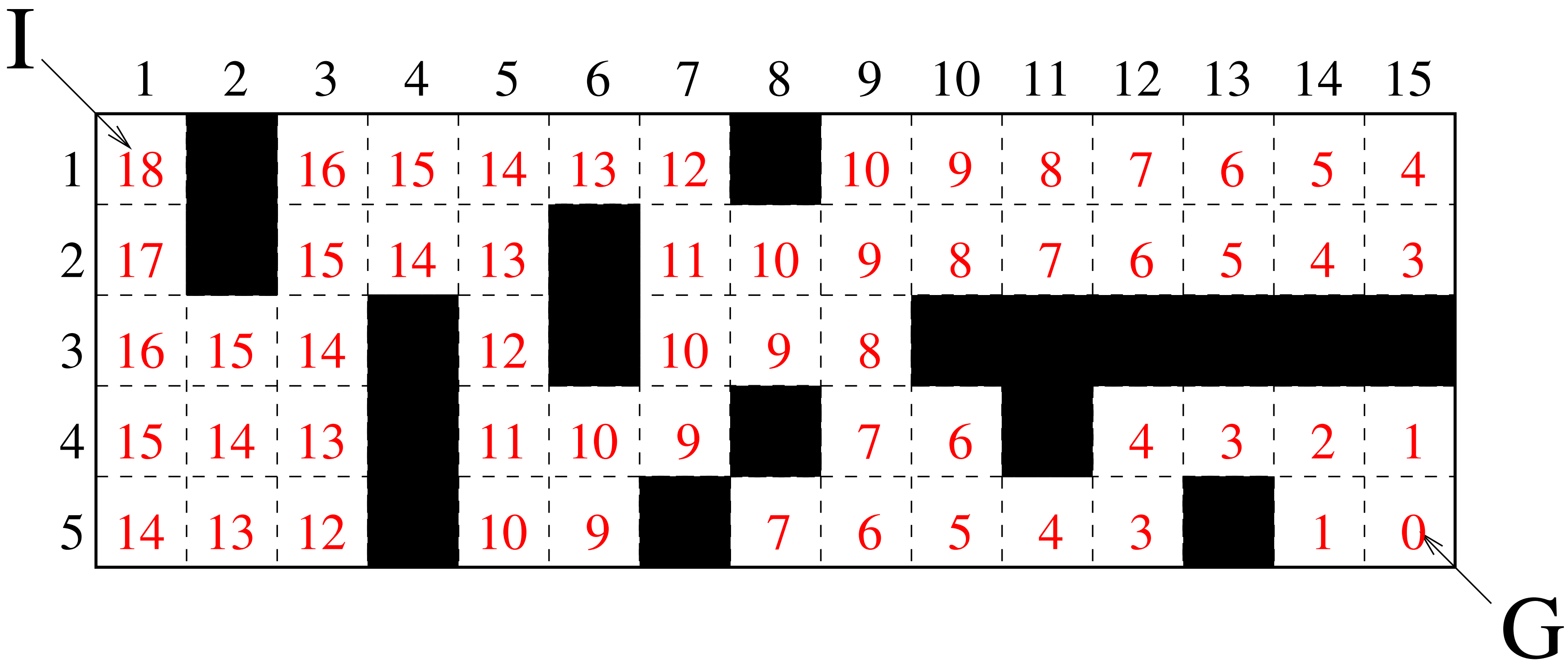
I

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	24		18	17	16	15	14		12	13	14	15	16	17	18
2	23		19	18	17		13	12	11	12	13	14	15	16	17
3	22	21	20		16		12	11	10						
4	23	22	21		15	14	13		9	8		4	3	2	1
5	24	23	22		16	15		9	8	7	6	5		1	0

G

Heuristic Function Pitfalls: Example Path Planning

Manhattan Distance, "accurate h":



Heuristic Function Pitfalls: Example Path Planning

Manhattan Distance, “inaccurate h”:

I

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	18		16	15	14	13	12	11	10	9	8	7	6	5	4
2	17		15												3
3	16		14		12	11	10		8	7	6		4		2
4	15	14	13		11		9		7		5		3		1
5	14	13	12	11	10		8	7	6		4	3	2		0

G

Important! Properties of Heuristic Functions

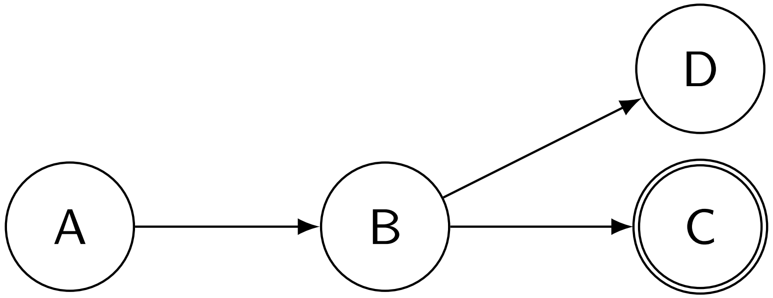
Definition (Admissibility). *Let Π be a problem with state space Θ and states S, and let h be a heuristic function for Π. We say that h is* **admissible** *if, for all $s \in S$, we have $h(s) \leq h^*(s)$.*

$\rightarrow$Admissible heuristics never overestimate the real cost (they are a **lower bound** on goal distance).

Definition (Consistency). *We say that h is* **consistent** *if, for all transitions $s \xrightarrow{a} s'$ in Θ, we have $h(s) - h(s') \leq c(a)$.*

$\rightarrow$With consistent heuristics, when applying an action a, the heuristic value cannot decrease by more than the cost of a.

Questionnaire



	A	B	C	D
h_1	0	1	0	1
h_2	0	1	0	100
h_3	1	2	0	0
h_4	2	0	0	0

Question!

What heuristics are admissible?

(A): h_1 **(B):** h_2

(C): h_3 **(D):** h_4

Question!

What heuristics are consistent?

(A): h_1 **(B):** h_2

(C): h_3 **(D):** h_4

→ (h_1): Consistent and Admissible
→ (h_2): Consistent and Admissible
→ (h_3): Inconsistent ($B \to C$) and Inadmissible ($h_3(B) = 2 > 1$)
→ (h_4): Inconsistent ($A \to B$) and Admissible

Properties of Heuristic Functions, ctd.

Proposition (Consistency $\Longrightarrow$ Admissibility). *Let Π be a problem, and let h be a heuristic function for Π. If h is consistent, then h is admissible.*

Proof. We need to show that $h(s) \leq h^*(s)$ for all s. For states s where $h^*(s) = \infty$, this is trivial. For all other states, we show the claim by induction over the length of the cheapest path to a goal state.

Base case: s is a goal state. Then $h(s) = 0$ by definition of heuristic functions, so $h(s) \leq h^*(s) = 0$ as desired.

Step case: Assume the claim holds for all states s' with a cheapest goal path of length n. Say s has a cheapest goal path of length $n + 1$, the first transition of which is $s \xrightarrow{a} s'$. By consistency, we have $h(s) - h(s') \leq c(a)$ and thus (a) $h(s) \leq h(s') + c(a)$. By construction, s' has a cheapest goal path of length n and thus, by induction hypothesis, (b) $h(s') \leq h^*(s')$. By construction, (c) $h^*(s) = h^*(s') + c(a)$. Inserting (b) into (a), we get $h(s) \leq h^*(s') + c(a)$. Inserting (c) into the latter, we get $h(s) \leq h^*(s)$ as desired.

Properties of Heuristic Functions: Examples

Admissibility and consistency:

- **Is straight line distance admissible/consistent?** Yes. Consistency: If you drive 100km, then the straight line distance to Madrid can't decrease by more than 100km.
- **Is goal distance of the "reduced puzzle" (slide 15) admissible/consistent?** Yes. Consistency: Moving a tile can't decrease goal distance in the reduced puzzle by more than 1. Same for misplaced tiles/Manhattan distance.
- **Can somebody come up with an admissible but inconsistent heuristic?** To-Madrid with $h(Aalborg) = 1000$, $h(Aarhus) = 100$.

$\rightarrow$ In practice, admissible heuristics are typically consistent.

Consistency/Admissibility Proofs

How to prove that a heuristic h is NOT admissible?

$\rightarrow$Find a counter-example: a state for which you can compute $h(s)$ and it is greater than the goal distance

How to prove that a heuristic is admissible?

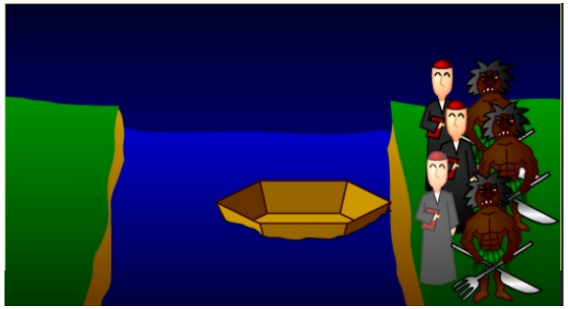
$\rightarrow$Prove that h is consistent

How to prove that a heuristic is consistent?

$\rightarrow$Show that it is 0 for goal states. Then, show that for each action the changes to the state cannot make the heuristic to decrease more than the action cost

For example, Manhattan distance in the N-puzzle is consistent because with every move it cannot reduce the Manhattan distance by more than 1.

Questionnaire



- 3 missionaries, 3 cannibals.
- Boat that holds ≤ 2.
- Never leave k missionaries alone with $> k$ cannibals.

Question!

Is $h :=$ number of persons at right bank consistent/admissible?

(A): Only consistent. **(B): Only admissible.**

(C): None. **(D): Both.**

$\rightarrow$ (A): No: If h is consistent then it is admissible, so "only consistent" can't happen (for any heuristic).

$\rightarrow$ (B): No: h is not admissible because a single move of the boat may get more than 1 person to the desired bank (example: 1 missionary and 1 cannibal at the wrong bank, with the boat).

$\rightarrow$ (C): Yes: h is not admissible so it can't be consistent either.

$\rightarrow$ (D): No, see above.

Before We Begin

Systematic search vs. local search:

- **Systematic search strategies**: No limit on the number of search nodes kept in memory at any point in time.

  → Guarantee to consider all options at some point, thus complete.
- **Local search strategies**: Keep only one (or a few) search nodes at a time.

  → No systematic exploration of all options, thus incomplete.

Tree search vs. graph search:

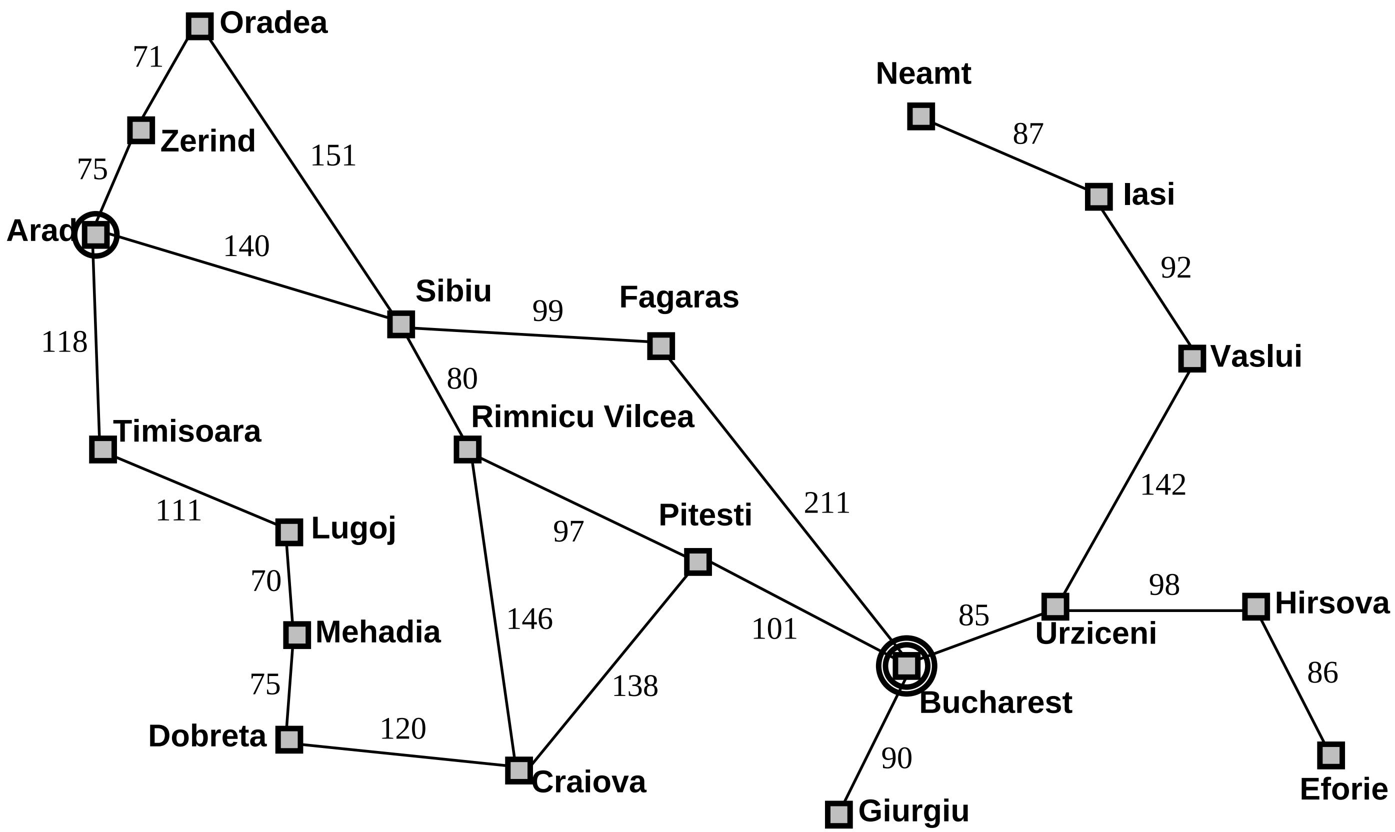
- For the systematic search strategies, we consider graph search algorithms exclusively, i.e., we use duplicate pruning.
- There also are tree search versions of these algorithms. These are easier to understand, but aren't used in practice. (Maintaining a complete open list, the search is memory-intensive anyway.)

Greedy Best-First Search

function **Greedy Best-First Search**(*problem*) **returns** a solution, or failure
  node $\leftarrow$ a node n with n.*state*=*problem.InitialState*
  frontier $\leftarrow$ a priority queue ordered by **ascending** h, only element n
  explored $\leftarrow$ empty set of states
  loop do
    if *Empty?(frontier)* **then return** failure
    $n \leftarrow$ *Pop(frontier)*
    if *problem.GoalTest*(n.*State*) **then return** *Solution*(n)
    explored $\leftarrow$ *explored* $\cup n$.State
    for each *action* a **in** *problem.Actions*(n.*State*) **do**
      $n' \leftarrow$ *ChildNode*(*problem*,n,a)
      if n'.**State**$\notin$**explored** $\cup$ **States**(**frontier**) **then** *Insert*(n', $h(n')$, *frontier*)

- Frontier ordered by ascending h.
- Duplicates checked at successor generation, against both the frontier and the explored set.

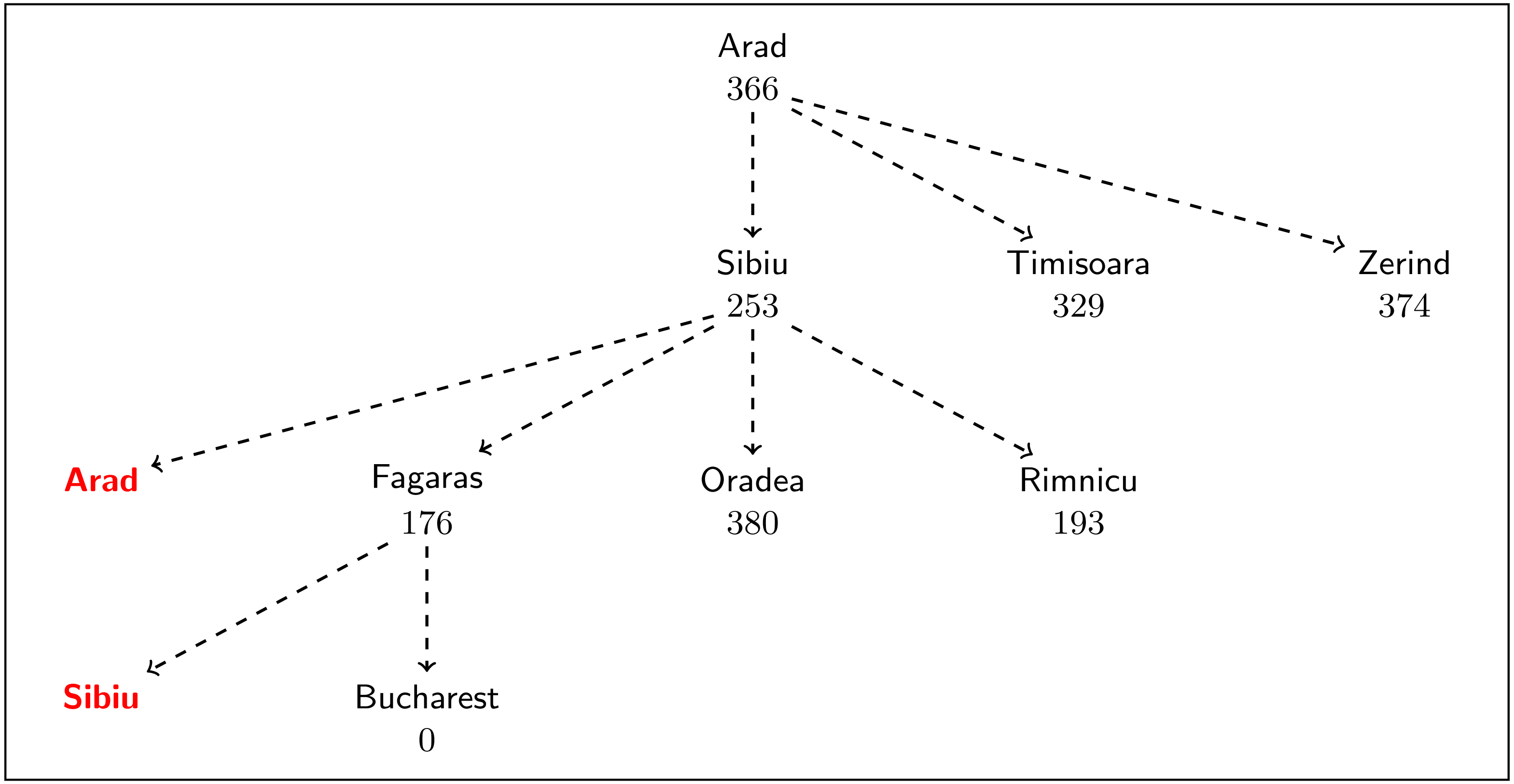
Greedy Best-First Search: Route to Bucharest



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

Greedy Best-First Search: Route to Bucharest

Subscripts: h. Red nodes: removed by duplicate pruning.



Greedy Best-First Search: Guarantees

- **Completeness:** Yes, thanks to duplicate elimination and our assumption that the state space is finite.
- **Optimality?** No (h might lead us to Madrid via Amsterdam).

Can we do better than this?

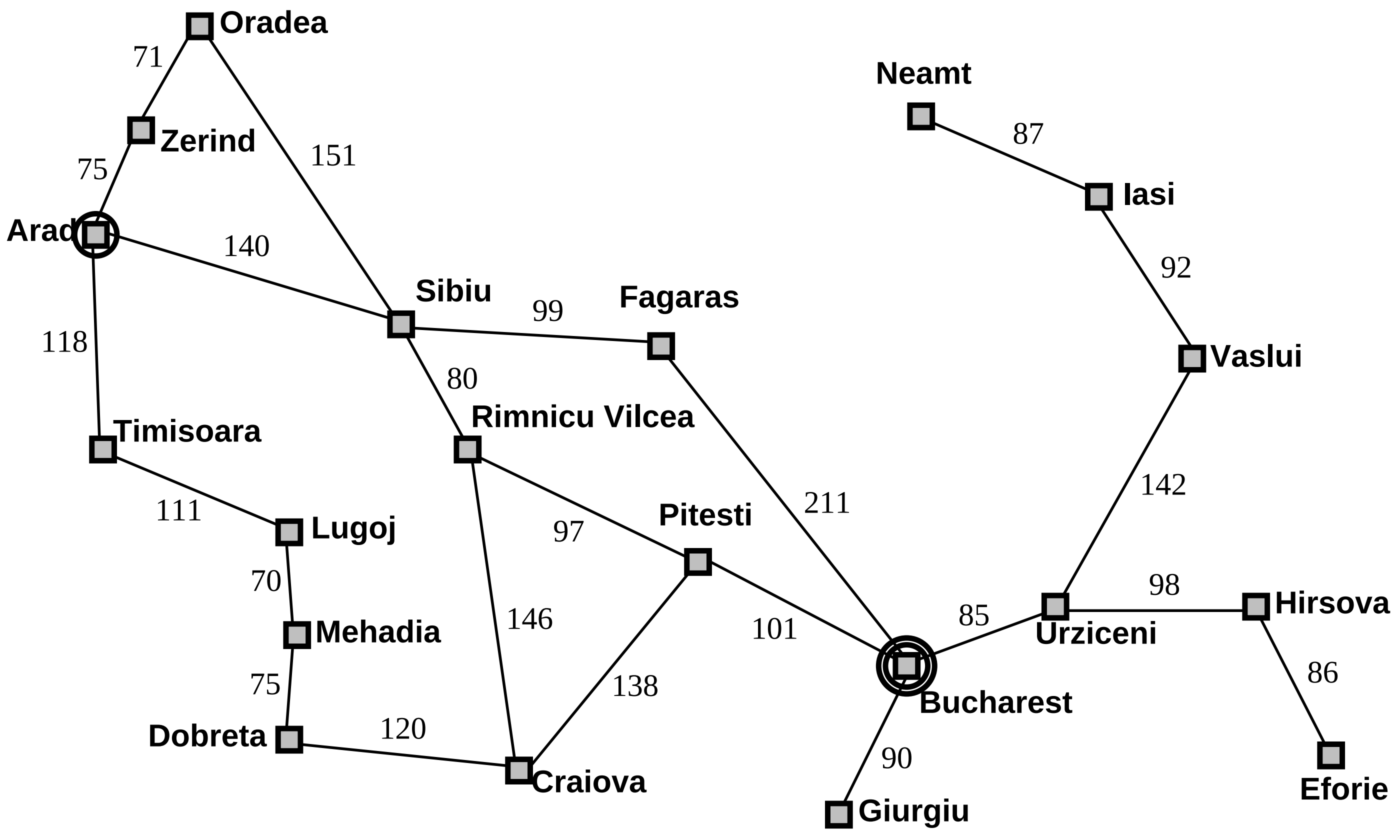
$\rightarrow$ **Yes: $\mathbf{A}^*$ is complete and optimal.**

A*

function **A*** *(problem)* **returns** a solution, or failure
  node ← a node n with n.*State*=*problem.InitialState*
  frontier ← a priority queue ordered by **ascending** $g + h$, only element n
  explored ← empty set of states
  loop do
    if *Empty?(frontier)* **then return** failure
    n ← *Pop(frontier)*
    if *problem.GoalTest*(n.*State*) **then return** *Solution*(n)
    explored ← *explored* $\cup n$. State
    for each *action* a **in** *problem.Actions*(n.*State*) **do**
      n' ← *ChildNode(problem,*n,a*)*
      if n'**.State**$\notin$**explored** $\cup$ **States(frontier)** **then**
        Insert(n', $g(n') + h(n')$, *frontier)*
      else if ex. n'' $\in$**frontier s.t.** n''**.State**$=$ n'**.State** and $g(n') < g(n'')$ **then**
        replace n'' **in frontier with** n'
      else if ex. n'' $\in$**explored s.t.** n''**.State**$=$ n'**.State** and $g(n') < g(n'')$ **then**
        Insert(n', $g(n') + h(n')$, **frontier)**

- Frontier ordered by ascending $g + h$.
- Duplicates handled **similarly as in uniform-cost search**. We may perform **node re-opening**: inserting a explored node in the frontier again if a better path is found.

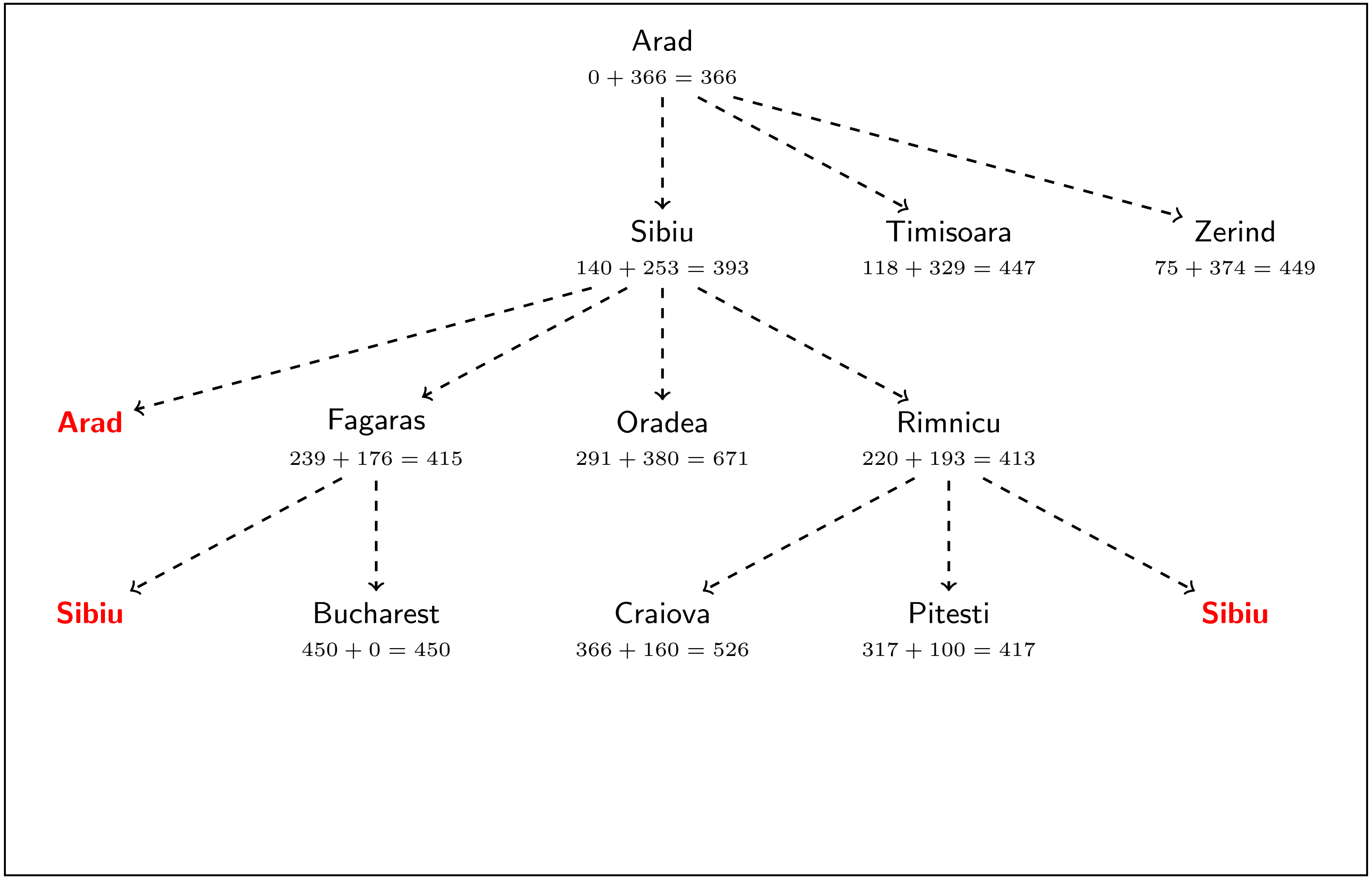
A^*: Route to Bucharest



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

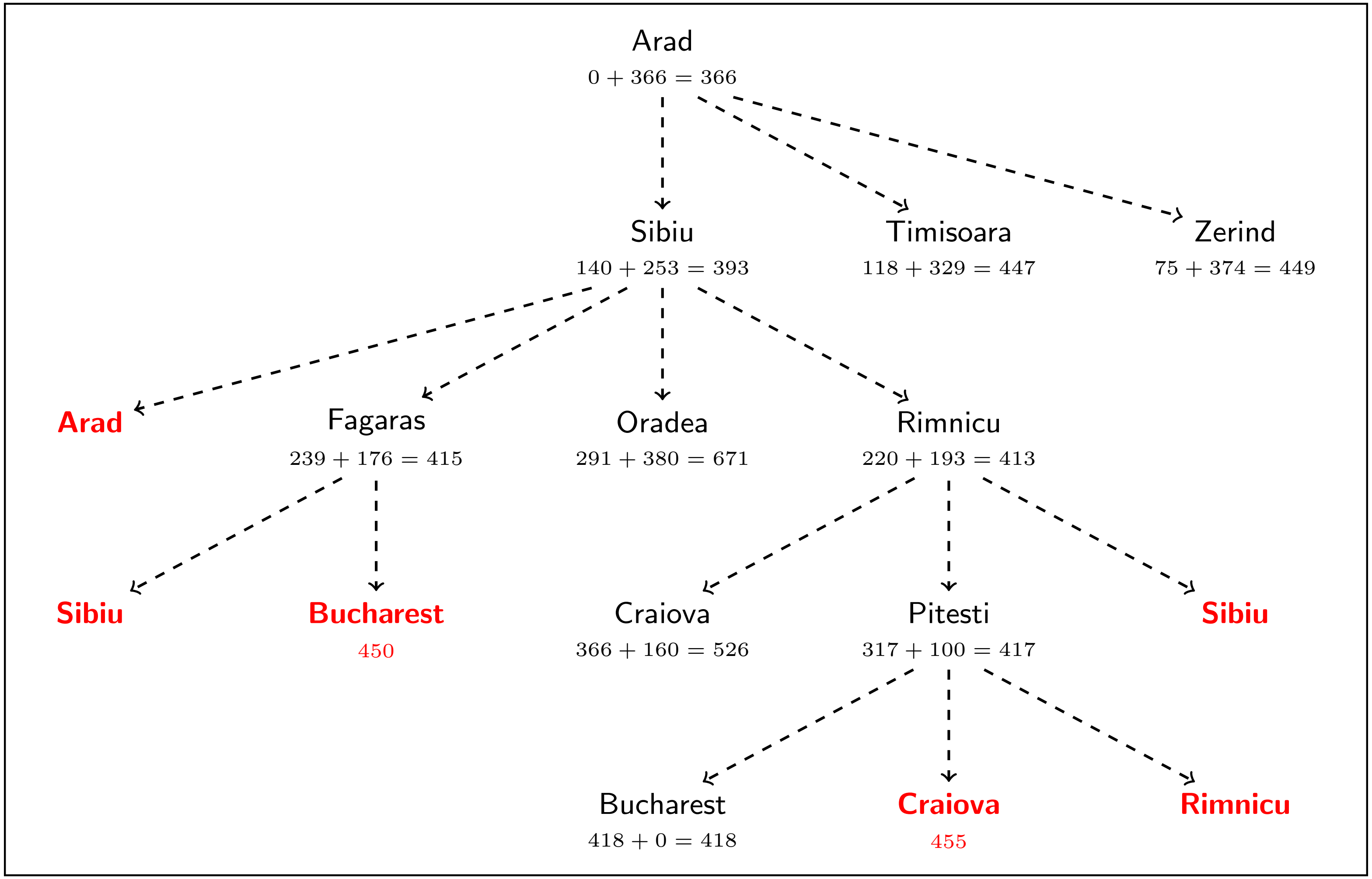
A^*: Route to Bucharest

Subscripts: $g + h$. Red nodes: removed by duplicate pruning (without subscript), or because of better path (with subscript g).



A^*: Route to Bucharest

Subscripts: $g + h$. Red nodes: removed by duplicate pruning (without subscript), or because of better path (with subscript g).



Questionnaire

Question!

If we set $h(s) := 0$ for all states s, what does greedy best-first search become?

(A): Breadth-first search **(B): Depth-first search**

(C): Uniform-cost search **(D): Depth-limited search**

$\rightarrow$ h implies no node ordering at all. The search order is determined by how we break ties in the open list. We *basically* get (A) with FIFO, (B) with LIFO, and (C) when ordering on g (in each case, differences remain in the handling of duplicate states etc).

Question!

If we set $h(s) := 0$ for all states s, what does $\mathbf{A}^*$ become?

(A): Breadth-first search **(B): Depth-first search**

(C): Uniform-cost search **(D): Depth-limited search**

$\rightarrow$ (C): The *only* difference between A^* and uniform-cost search is the use of $g + h$ instead of g to order the open list.

Optimality of A^*

Theorem (Optimality of $\mathbf{A}^*$). *Let Π be a problem, and let h be a heuristic function for Π. If h is admissible, then the solution returned by A^* (if any) is optimal.*

Proof Sketch. $f(n) = g(n) + h(n)$ is a lower bound on the cost of a solution through that node, so when a goal is expanded (has minimal f value), no other node can lead to a better path.

Note: $\mathrm{A}^*$is only optimal with admissible but inconsistent heuristics if it performs node-reopening (re-introduce a node in the frontier whenever a better path to it is found, even if it was already explored).
If the heuristic is consistent, node re-opening is not necessary because every time we expand a node we have already found an optimal path to that state.

A*is the best!

Optimal Efficiency of A*: With consistent heuristics, A*expands a minimal amount of nodes to find the optimal solution. No other algorithm can expand fewer nodes, unless it has access to other sources of information.

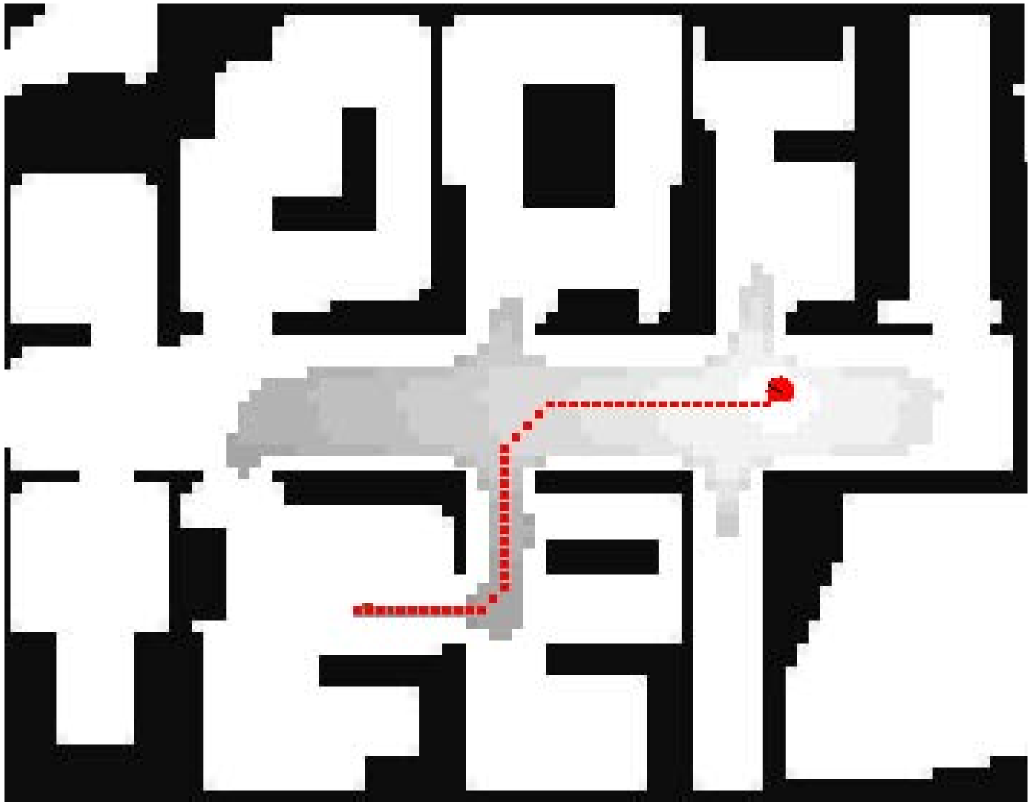
- If the heuristic is consistent, A*expands all states with $f(s) = g(s) + h(s) < C^*$ where C^* is the cost of the optimal path from the initial state to the goal
- Depending on tie-breaking it will expand some states with $f(s) = C^*$, but it won't expand any state with $f(s) > C^*$.
- Any algorithm that does not expand some state with $f(s) < C^*$, cannot guarantee that the solution is optimal

Empirical Performance: A^* in the 8-Puzzle

Without Duplicate Elimination; d = length of solution:

	Number of search nodes generated		
	Iterative	A^* with	
d	Deepening Search	misplaced tiles h	Manhattan distance h
2	10	6	6
4	112	13	12
6	680	20	18
8	6384	39	25
10	47127	93	39
12	3644035	227	73
14	-	539	113
16	-	1301	211
18	-	3056	363
20	-	7276	676
22	-	18094	1219
24	-	39135	1641

Empirical Performance: A^* in Path Planning



Live Demo vs. Breadth-First Search:

http://qiao.github.io/PathFinding.js/visual/

Greedy Best-First vs. A*: Illustration Path Planning

A*($g+h$), "accurate h":

I

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	18		22	22	22	22	22		24	24	24	24	24	24	24
2	18		20	20	20		22	22	22	22	22	22	22	22	22
3	18	18	18		20		22	22	22						
4	18	18	18		20	20	20		22	22		24	24	24	24
5	18	18	18		20	20		24	22	22	22	22		24	24

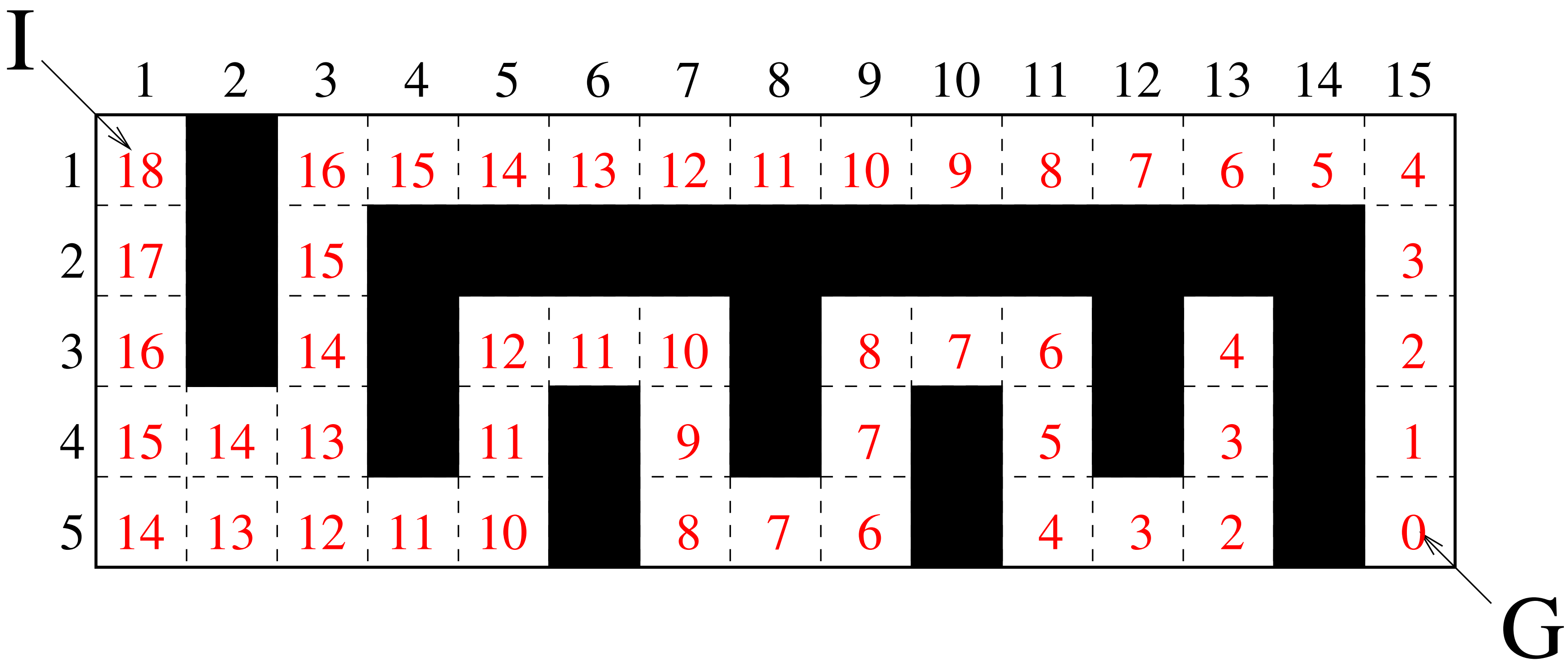
G

→ In A* with a consistent heuristic, $g+h$ always increases monotonically (h cannot *de*crease by more than g *in*creases).

→ We need more search, in the "right upper half". This is typical: Greedy best-first search tends to be faster than A*.

Greedy Best-First vs. A^*: Illustration Path Planning

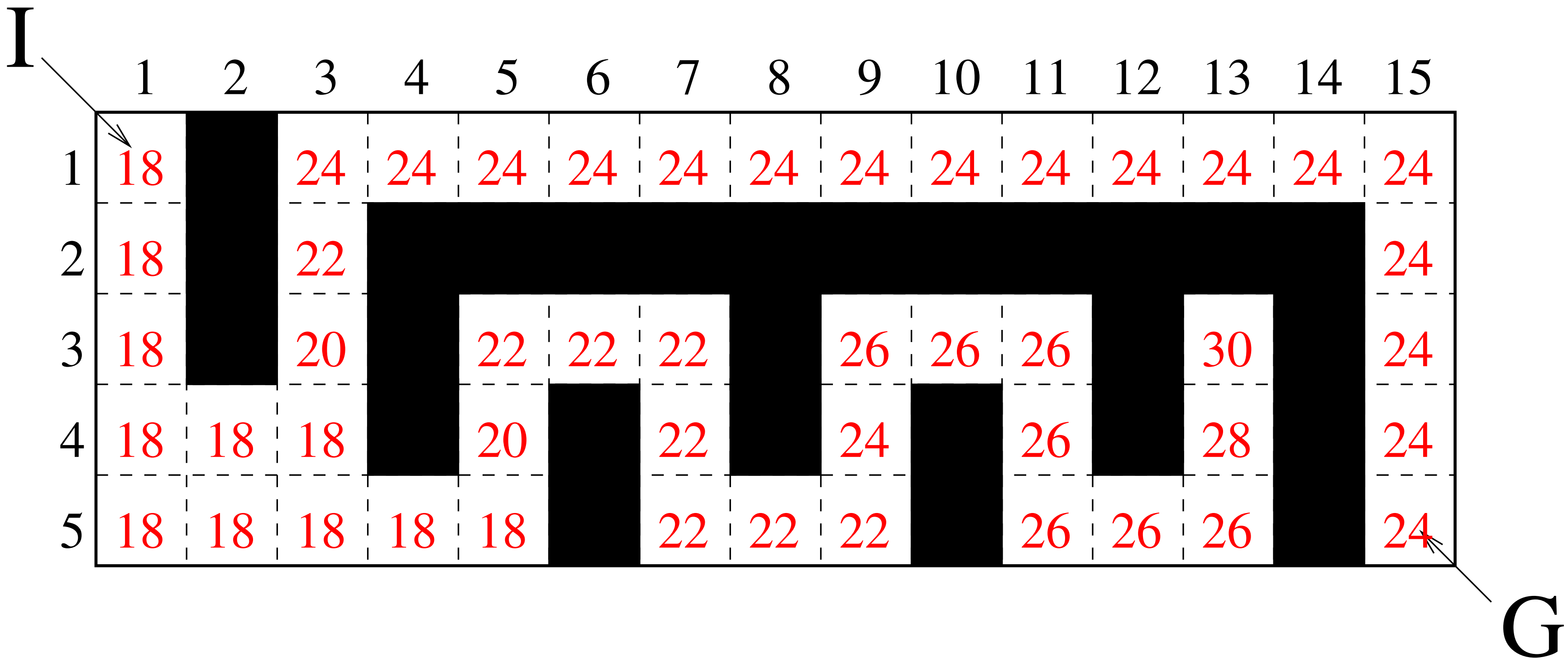
Greedy best-first search, “inaccurate h”:



$\rightarrow$ Search will be mis-guided into the “dead-end street”.

Greedy Best-First vs. A^*: Illustration Path Planning

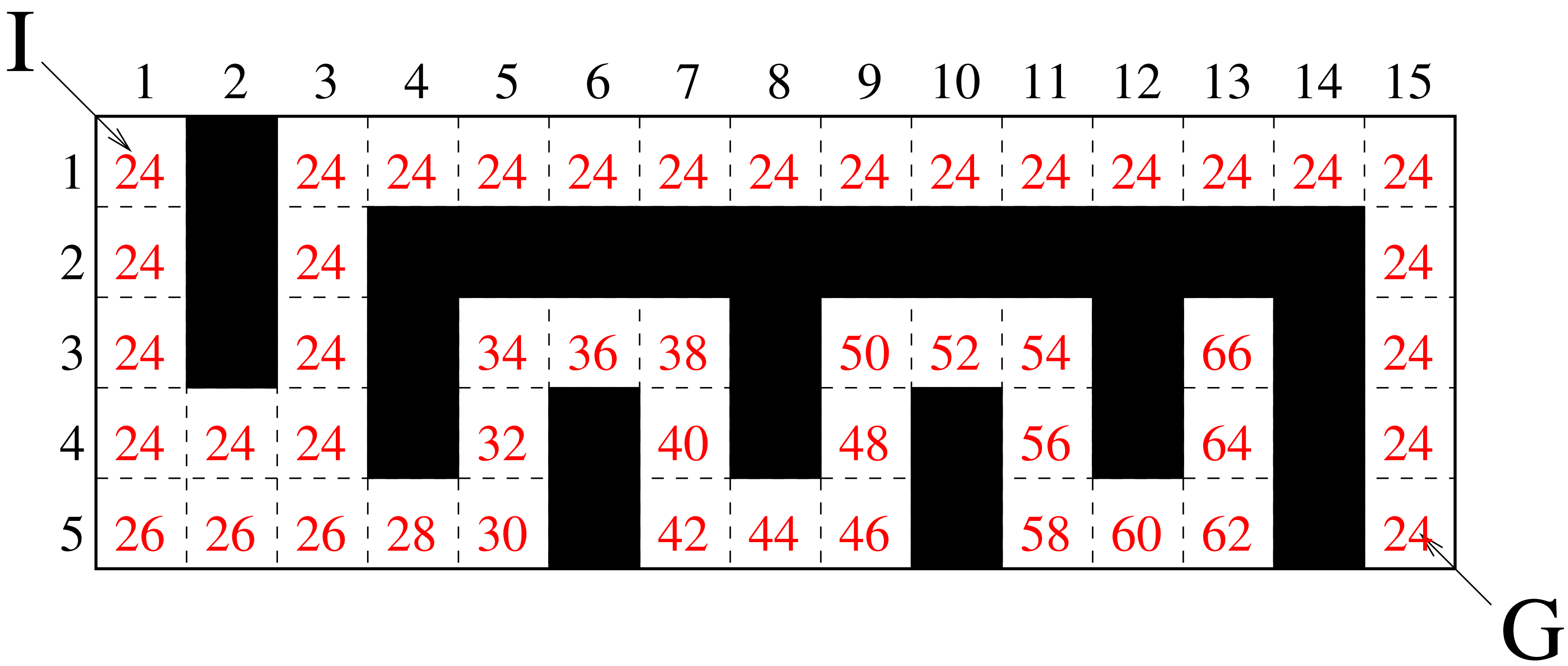
$\mathbf{A}^*(g+h)$, "inaccurate h":



$\rightarrow$ We will search less of the "dead-end street". For very "bad heuristics", $g+h$ gives better search guidance than h, and A^* is faster.

Greedy Best-First vs. A^*: Illustration Path Planning

$\mathrm{A}^*(g+h)$ using h^*:



$\rightarrow$ With $h = h^*$, $g + h$ remains constant on optimal paths.

Questionnaire

Question!

1. Is A^* always at least as fast as uniform-cost search? 2. Does it always expand at most as many states?

(A): No and no. **(B): Yes and no.**

(C): No and Yes. **(D): Yes and yes.**

$\rightarrow$ Regarding 1.: No, simply because computing h takes time. So the overall runtime may get worse.

$\rightarrow$ Regarding 2.: "Yes, but". Setting $h(s) := 0$ for uniform-cost search, both algorithms expand *only* states s where $g^*(s) + h(s) \leq g^*$, and *must* expand all states where $g^*(s) + h(s) < g^*$.
Non-zero h can only reduce the latter. Which s with $g^*(s) + h(s) = g^*$ are explored depends on the tie-breaking used (which state to expand if there is more than one state with minimal $g + h$ in the open list). So the answer is "yes but only if the tie-breaking in both algorithms is the same".

Best-First Search Algorithms: Overview

Algorithm	Uniform-Cost	GBFS	A*	WA*
Criteria	$g(n)$	$h(n)$	$g(n) + h(n)$	$g(n) + wh(n)$
Complete?	Yes	Yes	Yes[a]	Yes[a]
Optimal?	Yes	No	Yes[b]	No[c]

Note: we assume that b is finite, action costs are ≥ 0, and the state space is finite.

Footnotes:

[a] if h is safe (only returns ∞ for dead-end states)

[b] if h is consistent or if h is admissible and we re-open nodes when a better path has been found

[c] No, but if guarantees that solution cost is only sub-optimal by a factor of w (assuming [b])

Summary

- **Classical search problems** require to find a path of actions leading from an initial state to a goal state.
- They assume a single-agent, fully-observable, deterministic, static environment. Despite this, they are ubiquitous in practice.
- **Search strategies** differ (amongst others) in the order in which they **expand search nodes**, and in the way they use **duplicate elimination**. Criteria for evaluating them are **completeness**, **optimality**, **time complexity**, and **space complexity**.
- **Uniform-cost search** is optimal and works like Dijkstra, but building the graph incrementally. **Iterative deepening search** uses linear space only and is often the preferred blind search algorithm.
- **Heuristic functions** h map each state to an estimate of its goal distance. This provides the search with knowledge about the problem at hand, thus making it more focussed.
- h is **admissible** if it lower-bounds goal distance. h is **consistent** if applying an action cannot reduce its value by more than the action's cost. Consistency implies admissibility. In practice, admissible heuristics are typically consistent.
- **Greedy best-first search** explores states by increasing h. It is complete but not optimal.
- $\mathbf{A}^*$ explores states by increasing $g + h$. It is complete. If h is consistent, then A^* is optimal. (If h is admissible but not consistent, then we need to use **re-opening** to guarantee optimality.)
- **Local search** takes decisions based on its direct neighborhood. It is neither complete nor optimal, and suffers from **local minima** and **plateaus**. Nevertheless, it is often successful in practice.

Topics We Didn't Cover Here

- **Bounded Sub-optimal Search:** Giving a guarantee weaker than "optimal" on the solution, e.g., within a constant factor W of optimal.
- **Limited-Memory Heuristic Search:** Hybrids of A^* with depth-first search (using linear memory), algorithms allowing to make best use of a given amount M of memory, ...
- **External Memory Search:** Store the open/closed list on the hard drive, group states to minimize the number of drive accesses.
- **Search on the GPU:** How to use the GPU for part of the search work?
- **Real-Time Search:** What if there is a fixed deadline by which we must return a solution? (Often: fractions of seconds ...)
- **Lifelong Search:** When our problem changes, how can we re-use information from previous searches?
- **Non-Deterministic Actions:** What if there are several possible outcomes?
- **Partial Observability:** What if parts of the world state are unknown?
- **Reinforcement Learning Problems:** What if, a priori, the solver does not know anything about the world it is acting in?

Reading

- Chapter 3 Searching for Solutions
  We covered from 3.1 to 3.6, Section 3.8 explains Branch and Bound, which is another important search algorithm that we are not covering here.
- The Moving AI website (`https://www.movingai.com`) has a lot of resources.
  - Here, we have covered only a few basic algorithms, we could spend the whole course on this topic (`https://www.movingai.com/SAS/class.html`).
  - Of special interest are the interactive demos (`https://www.movingai.com/SAS/index.html`):
  - You can execute Dijkstra/$A^*$and WA^* step by step in a graph (`https://www.movingai.com/SAS/ASG/`) and in a grid (https://www.movingai.com/SAS/ASM/).

References I

Ariel Felner. Position paper: Dijkstra's algorithm versus uniform cost search or a case against dijkstra's algorithm. In *Proceedings of the Fourth Annual Symposium on Combinatorial Search, SOCS 2011, Castell de Cardona, Barcelona, Spain, July 15.16, 2011*, 2011.

John Gaschnig. Exactly how good are heuristics?: Toward a realistic predictive theory of best-first search. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI'77)*, pages 434–441, Cambridge, MA, August 1977. William Kaufmann.

Malte Helmert and Gabriele Röger. How good is almost perfect? In Dieter Fox and Carla Gomes, editors, *Proceedings of the 23rd National Conference of the American Association for Artificial Intelligence (AAAI'08)*, pages 944–949, Chicago, Illinois, USA, July 2008. AAAI Press.

Implementation: What Is a Search Node?

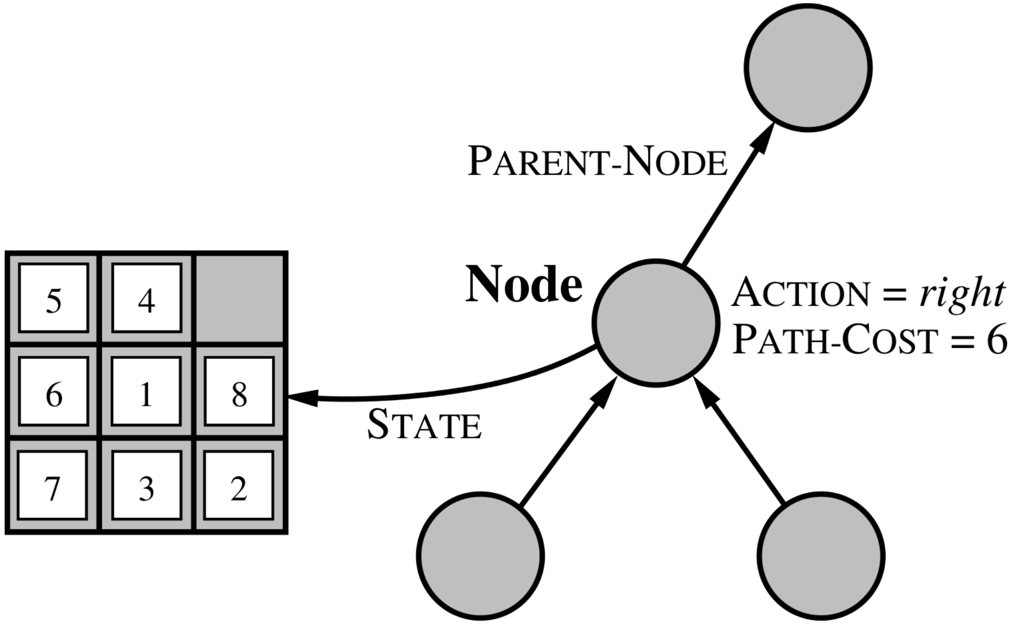
Data Structure for Every Search Node n

n.State: The state (from the state space) which the node contains.

n.Parent: The node in the search tree that generated this node.

n.Action: The action that was applied to the parent to generate the node.

n.PathCost: $g(n)$, the cost of the path from the initial state to the node (as indicated by the parent pointers).



Implementation, ctd: Operations on Search Nodes

Operations on Search Nodes

Solution(n): Returns the path to node n. (By backchaining over the n.Parent pointers and collecting n.Action in each step.)

ChildNode(problem,n,a): Generates the node n' corresponding to the application of action a in state n.State. That is: n'.State:=problem.ChildState(n.State, a); n'.Parent:= n; n'.Action:= a; n'.PathCost:= n.PathCost+problem.Cost(a).

Implementation, ctd: Operations for the Open List

Operations for the Open List

- Empty?(frontier): Returns true iff there are no more elements in the open list.
- Pop(frontier): Returns the first element of the open list, and removes that element from the list.
- Insert(element, frontier): Inserts an element into the open list.

→ Crucial point: Where "Insert(element, frontier)" inserts the new element. Different implementations yield different search strategies.

Direction of search

- The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes.
- Search complexity is b^n. Should use forward search if forward branching factor is less than backward branching factor, and vice versa.
- Note: sometimes when graph is dynamically constructed, you may not be able to construct the backwards graph.

Bi-directional search

- You can search backward from the goal and forward from the start simultaneously.
- This wins as $2b^{k/2} \ll b^k$. This can result in an exponential saving in time and space.
- The main problem is making sure the frontiers meet.
- This is often used with one breadth-first method that builds a set of locations that can lead to the goal. In the other direction another method can be used to find a path to these interesting locations.

Provable Performance Bounds: Extreme Case

Let's consider an extreme case: What happens if $h = h^*$?

Greedy Best-First Search:

- If all action costs are strictly positive, when we expand a state, at least one of its successors has strictly smaller h. The search space is linear in the length of the solution.
- If there are 0-cost actions, the search space may still be exponentially big (e.g., if all actions costs are 0 then $h^* = 0$).

$\mathbf{A}^*$:

- If all action costs are strictly positive, *and* we break ties ($g(n) + h(n) = g(n') + h(n')$) by smaller h, then the search space is linear in the length of the solution.
- Otherwise, the search space may still be exponentially big.

Provable Performance Bounds: More Interesting Cases?

"Almost perfect" heuristics:

$$|h^*(n) - h(n)| \leq c \textbf{ for a constant } c$$

- Basically the only thing that lead to <u>some</u> interesting results.
- If the state space is a tree (only one path to every state), and there is only one goal state: linear in the length of the solution [**?**].
- But if these additional restrictions do not hold: **exponential even for very simple problems and for $c = 1$ [?]!**

→ Systematically analyzing the practical behavior of heuristic search remains one of the biggest research challenges.

→ There is little hope to prove practical sub-exponential-search bounds. (But there are some interesting insights one *can* gain).