L3

Wednesday, July 30, 2025 3:05 PM

L03 - Searching

# Solving Problems by Searching

Dr. Sandareka Wickramanayake

## Outline

- Problem-solving agents
- Problem formulation
- Example problems
- Uninformed Search Algorithms
- Informed Search Algorithms
- Heuristic Functions

## Problem-Solving Agents

- Problem-Solving Agent - An agent that plans ahead: considers a sequence of actions that form a path to a goal state.
- Search – The computational process undertaken by a problem-solving agent.
- Use atomic representations.
- Only the simplest environments: episodic, single agent, fully observable, deterministic, static, and discrete.

Search Algorithms

Uninformed Algorithms | Informed Algorithms

**1. Problem-Solving Agent**

This is a type of intelligent agent that doesn't just react to its immediate environment. Instead, it **thinks ahead**.

- **How it works:** The agent evaluates its current state, considers various possible actions, and chooses a sequence of actions (a **plan**) that it predicts will lead it to a desirable **goal state**.
- **Analogy:** Imagine you're in a maze (current state). A simple agent might just always turn right. A problem-solving agent would mentally map out a path from its current location to the exit (goal state) before taking a single step.

**2. Search**

This is the core "thinking" process the agent uses to find that plan.

- It's the algorithm that systematically explores sequences of actions (the **paths**) from the initial state to find one that reaches the goal.
- The "computational process" means it involves calculations, memory, and following a set of rules to find a solution.

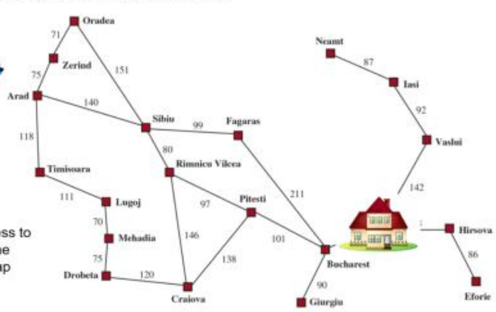
**3. Atomic Representations**

This refers to how the agent models the world for the search process.

- **Atomic** means the states of the world are considered as indivisible, whole units without any internal structure.
- The agent sees the world as a set of distinct states (e.g., "Room A", "Room B", "Room C") and doesn't reason about the properties inside those rooms. It's a simplified, high-level model that makes the search problem computationally manageable.

- **Uninformed (Blind) Search Algorithms:** These algorithms have no additional information about the problem beyond its definition. They don't know which node is closer to the goal. They systematically but often inefficiently explore possibilities until they stumble upon the goal.
  - *Examples:* Breadth-First Search, Depth-First Search, Uniform-Cost Search.
- **Informed (Heuristic) Search Algorithms:** These algorithms use extra information in the form of a **heuristic** (a rule of thumb or an educated guess) to estimate how close a state is to the goal. This allows them to explore in a more promising direction first, making them much more efficient.
  - *Example:* Greedy Best-First Search, A* Search (which is very important and efficient).

## A Vacation in Romania




The agent has access to information about the world, such as a map

A simplified road map of part of Romania, with road distances in miles.

## The Problem-Solving Process

- GOAL FORMULATION: Goals organize behavior by limiting the objectives and hence the actions to be considered.
- PROBLEM FORMULATION: The agent devises a description of the states and actions necessary to reach the goal.
- SEARCH: Before taking any action in the real world, the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal (*solution*).
- EXECUTION: The agent can now execute the actions in the solution, one at a time.

## A Vacation in Romania

- Agent on holiday in Romania; currently in Arad.
- Needs to catch a flight taking off from Bucharest
- Formulate goal:
  - be in Bucharest
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest.

## Search Problems and Solutions

- A **search problem** has the following components:

- **State space -** A set of possible states that the environment can be in.
- **Initial state** – The state that the agent starts in.
  - E.g., Arad
- **Goal states**
  - One goal state (e.g., Bucharest)
  - A small set of alternative goal states (e.g., The goal of a vacuum cleaner is to have no dirt in any location.)
  - The goal is defined by a property that applies to many states

## Search Problems and Solutions

- A **search problem** has the following components:

- **Actions** - Actions available to the agent. Given a state $s$, $Action(s)$ returns a finite set of actions that can be executed in $s$. We say that each of these actions is **applicable** in $s$.
  - $ACTIONS(Arad) = \{ToSibiu, ToTimisoara, ToZerind\}$
- **Transition model** – Describes what each action does. $RESULT(s, a)$ returns the state that results from doing action $a$ in state $s$.
  - E.g., $RESULT(Arad, ToZerind) = Zerind$

# Search Problems and Solutions

- A search **problem** has the following components:

- **Action cost function** – The numeric cost of applying action $a$ in state $s$ to reach $s'$, $ACTION\_COST(s, a, s')$.
  - E.g., lengths in miles/ time it takes to complete the action
- **Path** – A sequence of states connected by a sequence of actions.
- **Solution** – A path from the initial state to a goal state.
- **Optimal solution** – The lowest path cost among all solutions.
- **Graph** – A representation of state space in which vertices are states and the directed edges between them are actions.

**Assumptions** - Action costs are additive, and all action costs will be positive

9

# Search Problems and Solutions

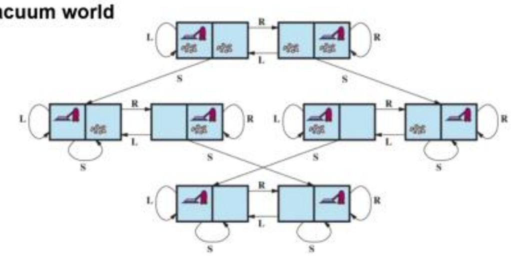
- A search **problem** has the following components:

- **Model** - An abstract mathematical description.
  - E.g., our formulation of the problem of getting to Bucharest.
- **Abstraction** – Removing details from a representation.
  - Real world is quite complex.
  - A good problem formulation has the right level of detail.

10

# Example Problems

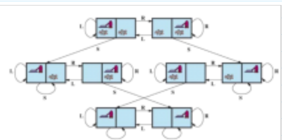
- **Vacuum world**



The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

11

# Example Problems



The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = Left, R = Right, S = Suck.

- **Vacuum world**
  - **STATES –** 8 states (Agent in cell 1, cell 1 has dirt, cell 2 has dirt, etc.)
  - **INITIAL STATE** - Any state can be designated as the initial state.
  - **ACTIONS** - *Suck, move Left*, and *move Right*.
  - **TRANSITION MODEL** - Suck removes any dirt from the agent's cell; *Forward moves* the agent ahead of one cell in the direction it is facing unless it hits a wall, in which case the action has no effect. *Backward* moves the agent in the opposite direction, while *TurnRight* and *TurnLeft* change the direction it is facing by $90^0$.
  - **GOAL STATES**: The states in which every cell is clean.
  - **ACTION COST:** Each action costs 1.

12

# Example Problems

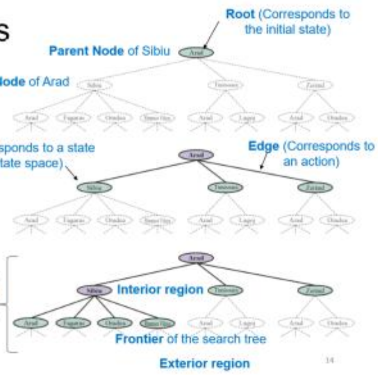
- **Route-finding problem – Travel-planning website**
  - **STATES** - Each state includes a location (e.g., an airport) and the current time.
  - **INITIAL STATE** - The user's home airport.
  - **ACTIONS** - Take any flight from the current location, in any seating class, leaving after the current time, leaving enough time for within-airport transfer if needed.
  - **TRANSITION MODEL**: The state resulting from taking a flight will have the flight's destination as the new location and the flight's arrival time as the new time.
  - **GOAL STATE**: A destination city. Sometimes the goal can be more complex, such as "arrive at the destination on a nonstop flight."
  - **ACTION COST**: Monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer reward points, etc.



# Search Algorithms

**Reached states**

**State Space**
All possible transitions among all the states.




**Search Tree**
Describes paths between the states toward the goal state.

**Expand –** Expand the node considering available actions.



# Redundant Paths

- How do we decide which node from the frontier to expand next?
- Redundant paths
  - **Repeated state –** Meeting a top node again.
  - **Redundant path**
    - We can get to Sibiu via the path Arad–Sibiu (140 miles long) or ~~the path Arad–Zerind–Oradea–Sibiu (297 miles long).~~
    - Eliminating redundant paths leads to faster solutions.



# Measuring Problem-Solving Performance

- Algorithms are evaluated along the following dimensions:
  - **Completeness**: Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?
  - **Cost optimality:** Does it find a solution with the lowest path cost of all solutions?
    - Also referred to as admissibility or optimality.
  - **Time complexity:** How long does it take to find a solution?
    - Can be measured in seconds, or more abstractly by the number of states and actions considered.
  - **Space complexity**: How much memory is needed to perform the search?

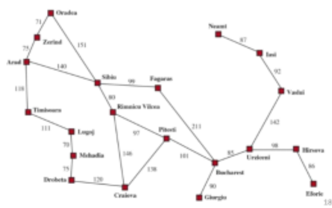
## Measuring Problem-Solving Performance

- Time and space complexity are measured in terms of
  - ***b***: maximum branching factor of the search tree (number of successors of a node that need to be considered)
  - ***d***: depth of the least-cost solution
  - ***m***: maximum number of actions in any path (maybe ∞)

17

## Uninformed Search Algorithms

- Have access only to the problem definition.
- No clue about how close a state is to the goal(s).
- Build a search tree to find a solution.




18

## Uninformed Search Algorithms

- Algorithms differ based on which node they expand first.
- Algorithms
  - **Breadth-first search** – Expands the shallowest nodes first.
    - Complete
    - Optimal for unit action costs.
    - Exponential space complexity.
  - **Uniform-cost search** – Expands the node with the lowest path cost.
    - Optimal for general action costs.

19

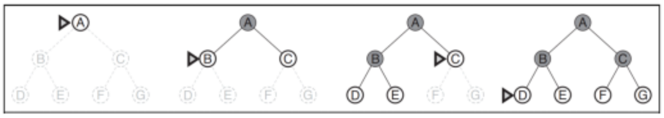
## Uninformed Search Algorithms

- Algorithms
  - **Depth-first Search** - Expands the deepest unexpanded node first.
    - Neither complete nor optimal
    - Linear time complexity
  - **Iterative Deepening Search** - Calls DFS with increasing depth limits until a goal is found.
    - Complete when full cycle checking is done
    - Optimal for unit action costs
    - Time complexity is comparable to BFS
    - Space complexity is linear.
  - **Bidirectional Search** - Expands two frontiers, one around the initial state and one around the goal, stopping when the two frontiers meet.

20

## Breadth-first Search

- Appropriate when all the actions have the same cost.



22

## Breadth-first Search - Evaluation

- **Complete?** Yes (if b is finite)
- **Time?** $1+b+b^2+ b^3 +\ldots + b^d = O(b^d)$, where $d$ is the depth of the solution.
- **Space?** $O(b^d)$ (keeps every node in memory)
- **Cost Optimal?** Yes (Only if path costs are identical)
- Space is the bigger problem (more than time)

- *Exponential complexity search problems cannot be solved by uninformed search for any but the smallest instances.*

23

## Breadth-first Search - Evaluation

| Depth | Nodes | Time | Memory |
|---|---|---|---|
| 2 | 110 | .11 milliseconds | 107 kilobytes |
| 4 | 11,110 | 11 milliseconds | 10.6 megabytes |
| 6 | $10^6$ | 1.1 seconds | 1 gigabyte |
| 8 | $10^8$ | 2 minutes | 103 gigabytes |
| 10 | $10^{10}$ | 3 hours | 10 terabytes |
| 12 | $10^{12}$ | 13 days | 1 petabyte |
| 14 | $10^{14}$ | 3.5 years | 99 petabytes |
| 16 | $10^{16}$ | 350 years | 10 exabytes |

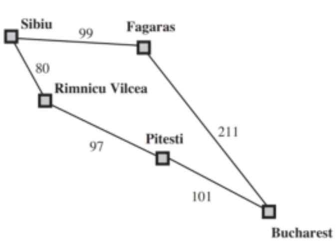
**Figure 3.13** Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

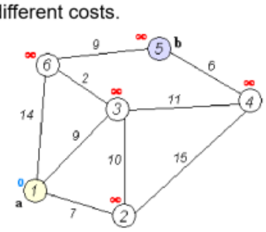
24

8

## Uniform-cost Search or Dijkstra's Algorithm

- Appropriate when the actions have different costs.




Source - https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

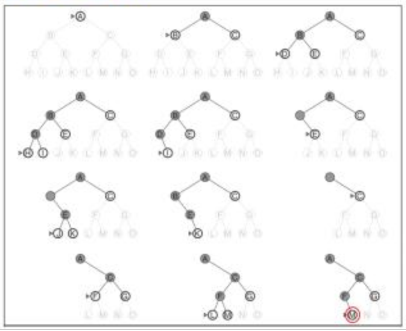
25

## Uniform-cost Search - Evaluation

- **Complete?** Yes (if b is finite)
- **Time?** $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
  - Where $C^*$ - The cost of the optimal solution and $\epsilon$ – lower bound on the cost of each action, with $\epsilon > 0$.
- **Space?** $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
- **Cost Optimal?** Yes



## Depth-first Search





## Depth-first Search - Evaluation

- **Complete?** Yes, for finite state spaces. No for infinite state spaces and spaces with loops.
- **Time?** $O(b^m)$, where m is the maximum depth.
- **Space?** O(bm)
- **Cost Optimal?** No: It returns the first solution it finds, even if it is not the cheapest.



## Depth-limited Search

- Keep DFS from wandering down an infinite path.
- A version of DFS which has a depth limit, $l$, and treats all nodes at depth $l$ as if they had no successors.
- Evaluation
  - **Complete?** No, a poor choice for $l$ makes the algorithm fail to reach the solution.
  - **Time?** $O(b^l)$
  - **Space?** O(bl)
  - **Cost Optimal?** No: It returns the first solution it finds, even if it is not the cheapest.

## Uninformed Search Algorithms Comparison

| 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$^{a,d}$ |
| Time | $O(b^d)$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(b^m)$ | $O(b^\ell)$ | $O(b^d)$ | $O(b^{d/2})$ |
| Space | $O(b^d)$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(bm)$ | $O(b\ell)$ | $O(bd)$ | $O(b^{d/2})$ |
| Optimal? | Yes$^c$ | Yes | No | No | Yes$^c$ | Yes$^{c,d}$ |

**Figure 3.21** Evaluation of tree-search strategies. $b$ is the branching factor; $d$ is the depth of the shallowest solution; $m$ is the maximum depth of the search tree; $l$ is the depth limit. Superscript caveats are as follows: $^a$ complete if $b$ is finite; $^b$ complete if step costs $\geq \epsilon$ for positive $\epsilon$; $^c$ optimal if step costs are all identical; $^d$ if both directions use breadth-first search.



## Informed Search Algorithms

- Uses domain-specific hints about the location of goals.
- Finds solutions more efficiently than an uninformed strategy.
- The hints come in the form of a **heuristic function**, $h(n)$.
- $h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.
  - In route-finding problems - the straight-line distance on the map between the current state and a goal.



## Informed Search Algorithms

- Algorithms
  - **Greedy Best-First Search** – Expands nodes with minimal $h(n)$
    - Not optimal
    - Efficient
  - **A* Search** – Expands nodes with minimal $f(n) = g(n) + h(n)$
    - Complete and optimal provided that $h(n)$ is admissible.
    - Bad space complexity.
  - **Bidirectional A* Search**
    - More efficient than A*
  - **Iterative Deepening A* Search** – An Iterative version of A*
    - Address the space complexity issue.
  - **Beam Search –** Puts a limit on the size of the frontier.
    - Incomplete and suboptimal
    - Efficient with reasonably good solutions.

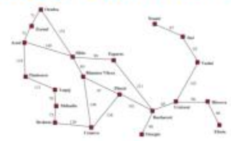


## Greedy Best-First Search

- **Best-First Search**
  - Idea: use an evaluation function *f* for each node *n*
    - *f(n)* estimates the "desirability" of node *n*
    - Expand the most desirable unexpanded node

- **Greedy Best-First Search**
  - Evaluation function *f(n)* = *h(n)*
    - where *h(n)* is some heuristic estimate of cost from *n* to *goal*
    - e.g., $h_{SLD}(n)$ = straight-line distance from *n* to Bucharest
  - Expands the node that appears to be closest to the goal
    - E.g., node *n*, such that $h_{SLD}(n)$ is minimum

## Greedy Best-First Search

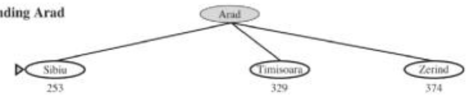


Straight-line distances to Bucharest.

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

**(a) The initial state**


**(b) After expanding Arad**






## Greedy Best-First Search

**(c) After expanding Sibiu**


**(d) After expanding Fagaras**




## Greedy Best-First Search - Evaluation

- **Complete**? No. Can lead to dead ends and the tree search version (not the graph search version) can go into infinite loops.




## Greedy Best-First Search - Evaluation

- **Worst case time**? $O(b^m)$ - can generate all nodes at depth $m$ before finding the solution.
- **Worst case space**? $O(b^m)$ - can generate all nodes at depth $m$ before finding the solution
  - But a good heuristic can dramatically improve the time and space needed
    - In our example, a solution was found without expanding any node not on the path to goal: Which very efficient in this case
- **Optimal**? No
  - Path found: Arad->Sibiu->Fagaras->Bucharest. Actual cost = 140+99+211=450
  - But the actual cost of, Arad->Sibiu->Rimnicu->Pitesti = 140+80+97+101=418

# A* Search

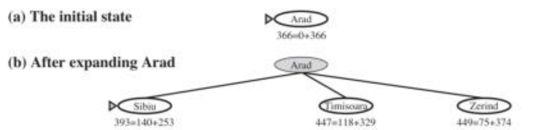
- Idea: avoid expanding paths that are already expensive.
- Evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach the node $n$.
- $f(n)$ - Estimated cost of the cheapest solution through $n$.
- A* is identical to Uniform-cost search except A* uses $g(n) + h(n)$ instead of $g(n)$.

38

# A* Search

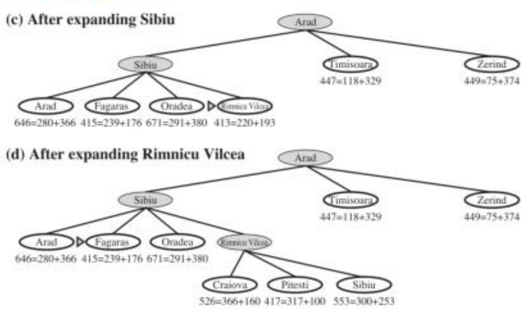
Straight-line distances to Bucharest.

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



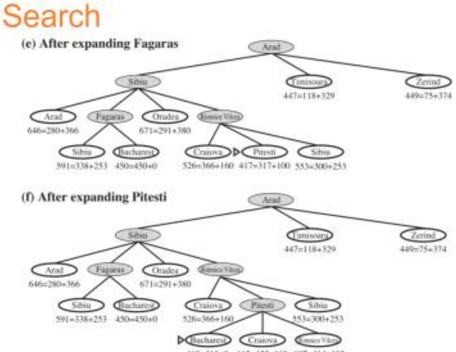

39

# A* Search




40

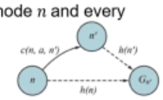
# A* Search




41

# A* Search - Evaluation

- Complete? Yes
- Optimal?
  - Depends on certain properties of the heuristics
  - **Admissibility**: an admissible heuristic never overestimates the cost of reaching a goal. (An admissible heuristic is therefore optimistic.)
  - *If the heuristic is admissible, A* is optimal.*
  - **Consistency**: A heuristic $h(n)$ is consistent if for every node $n$ and every successor $n'$ of $n$ generated by an action $a$, we have:
    $$h(n) \le c(n,a,n') + h(n')$$




  - Every consistent heuristic is admissible.
  - *If the heuristic is consistent, A* is optimal.*
  - *With an inadmissible heuristic, A* may or may not be cost-optimal.*

42

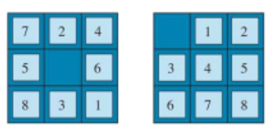
# A* Search - Evaluation

- Time? Exponential in the worst case.
- Space? Exponential in the worst case.
  - A good heuristic can reduce time and space complexity considerably.

43

# Heuristic Functions

- The performance of heuristic search algorithms depends on the quality of the heuristic function.
- One can sometimes construct good heuristics by
  - Relaxing the problem definition
  - Storing precomputed solution costs for subproblems in a pattern database
  - Defining landmarks
  - Learning from the experience with the problem class




Start State    Goal State

h1 = the number of misplaced tiles (blank not included). (An admissible heuristic)
h2 = the sum of the distances of the tiles from their goal positions. (An admissible heuristic)

44

**Heuristic 1: Number of Misplaced Tiles ($h_1$)**

- **What it is:** This heuristic simply counts how many tiles are **not in their goal position**. The blank tile is not included in the count.
- **Application to Start State:**
  Let's check each tile against the goal state:
  - Tile `1`: Is at (3,3), should be at (1,1). **Misplaced.**
  - Tile `2`: Is at (1,2), should be at (1,2). **Correct.**
  - Tile `3`: Is at (3,2), should be at (1,3). **Misplaced.**
  - Tile `4`: Is at (1,3), should be at (2,1). **Misplaced.**
  - Tile `5`: Is at (2,1), should be at (2,2). **Misplaced.**
  - Tile `6`: Is at (2,3), should be at (2,3). **Correct.**
  - Tile `7`: Is at (1,1), should be at (3,1). **Misplaced.**
  - Tile `8`: Is at (3,1), should be at (3,2). **Misplaced.**
  - Blank: Ignored.
  - **Total $h_1$ = 6** (Tiles 1, 3, 4, 5, 7, 8 are misplaced).
- **Why it's admissible:** Moving a tile into its correct position requires **at least one move**. Therefore, the number of misplaced tiles is a guaranteed *underestimate* of the number of moves needed to solve the puzzle. You will need at least 6 more moves, but almost certainly more because a single move only fixes one tile.

**Heuristic 2: Manhattan Distance ($h_2$)**

- **What it is:** This heuristic calculates the sum of the horizontal and vertical distances each tile is from its goal position. This is a more informed heuristic than $h_1$.
- **Application to Start State:**
  For each tile, calculate its Manhattan distance:
  - Tile `1`: From (3,3) to (1,1): |3-1| + |3-1| = 2 + 2 = 4
  - Tile `2`: From (1,2) to (1,2): |1-1| + |2-2| = 0 + 0 = 0
  - Tile `3`: From (3,2) to (1,3): |3-1| + |2-3| = 2 + 1 = 3
  - Tile `4`: From (1,3) to (2,1): |1-2| + |3-1| = 1 + 2 = 3
  - Tile `5`: From (2,1) to (2,2): |2-2| + |1-2| = 0 + 1 = 1
  - Tile `6`: From (2,3) to (2,3): 0 = 0
  - Tile `7`: From (1,1) to (3,1): |1-3| + |1-1| = 2 + 0 = 2
  - Tile `8`: From (3,1) to (3,2): |3-3| + |1-2| = 0 + 1 = 1
  - Total $h_2$ = 4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14
- **Why it's admissible:** To get a tile to its goal position, you must move it at least its Manhattan distance away. Since you can only move one tile one step at a time (and the moves of different tiles can interfere), the sum of these individual distances is a guaranteed *underestimate* of the total number of moves required. The true solution will require at least 14 moves.

# Generating Heuristics from Relaxed Problems

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

45

**4. Why Heuristic Quality Matters**

The slide states: "*The performance of heuristic search algorithms depends on the quality of the heuristic function.*"

- **A better heuristic** (like $h_2$) is **closer to the true cost** without ever exceeding it.
- **Why this is good:** A better heuristic guides the search algorithm more directly toward the goal. It results in **fewer nodes being expanded** (making the search faster) and requires **less memory**.
- **Comparison:** In the 8-puzzle, $h_2$ (Manhattan Distance) is **dominant** over $h_1$ (Misplaced Tiles). This means that for any given node, `h₂(n) >= h₁(n)`. A search algorithm using $h_2$ will always be more efficient than one using $h_1$.

**5. How to Create Good Heuristics**

The slide lists methods for constructing good heuristics:

1. **Relaxing the problem definition:** Make the problem easier by removing rules. The heuristics $h_1$ and $h_2$ are derived from relaxed problems:
   - **$h_1$ (Misplaced Tiles)** comes from a problem where a tile can magically move to its goal spot in one move, even if the way is blocked.
   - **$h_2$ (Manhattan Distance)** comes from a problem where a tile can move to any adjacent square in one move, even if it's occupied (it just phases through other tiles).
2. **Pattern Databases:** This is an advanced technique where you precompute and store the exact solution costs for subproblems (e.g., the cost to get tiles 1, 2, 3, 4 into place). The heuristic for a full state is then the maximum or sum of these stored costs.
3. **Landmarks:** For problems like pathfinding on a map, a landmark heuristic might precompute the distance from every node to a few key "landmark" points. The heuristic `h(n)` can then be derived from these precomputed distances.
4. **Learning from experience:** A machine learning model can be trained on many examples of the problem to learn a function that accurately estimates the cost to the goal. This is common in very complex games like Go.

## Generating Heuristics from Subproblems: Pattern Databases

- Admissible heuristics can also be derived from the solution cost of a subproblem of a given problem.
- E.g., Subproblem of 8-puzzle example
  - The cost of the optimal solution to this subproblem is a lower bound on the cost of the complete problem.
- **Pattern databases** - Stores these exact solution costs for every possible subproblem instance.