Escola Tècnica Superior d'Enginyeria
Universitat Rovira i Virgili

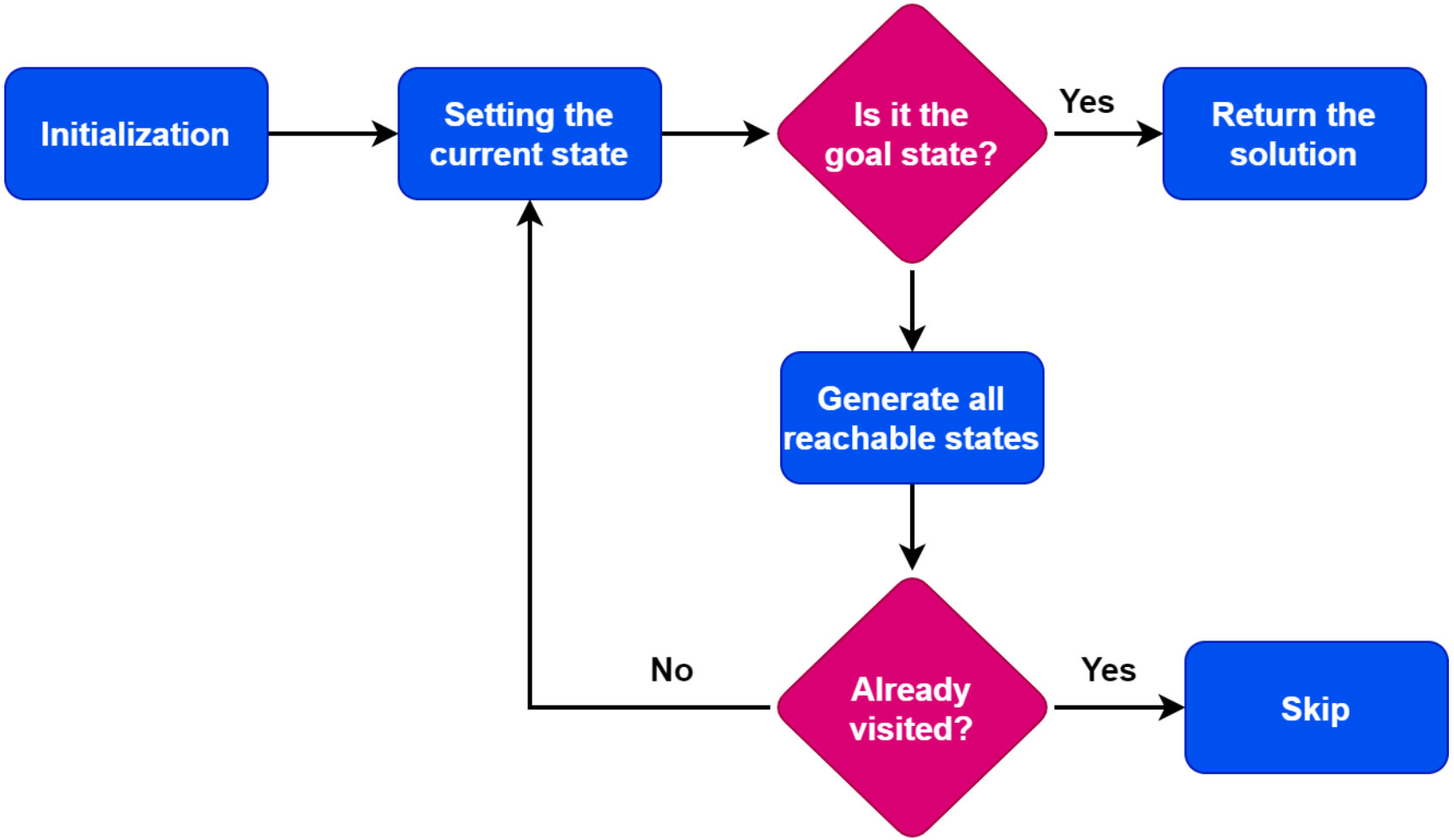
[DΣIM]

AI Planning

Hatem A. Rashwan

State-Space Search

State-Space Search Overview



Overview

- **Heuristic Search Strategies**
- The A* Algorithm (for Tree Search)
- Graph Search with A*
- (Good) Heuristics

Informed vs uniformed search

Parameters	Informed Search	Uninformed Search
Known as	It is also known as Heuristic Search.	It is also known as Blind Search.
Using Knowledge	It uses knowledge for the searching process.	It doesn't use knowledge for the searching process.
Performance	It finds a solution more quickly.	It finds solution slow as compared to an informed search.
Completion	It may or may not be complete.	It is always complete.
Cost Factor	Cost is low.	Cost is high.
Time	It consumes less time because of quick searching.	It consumes moderate time because of slow searching.

Heuristic Functions

- heuristic function h: state space → $\mathbb{R}$
- $h(n)$ = estimated cost of the cheapest path from node n to a goal node
- if n is a goal node then $h(n)$ must be 0
- heuristic function encodes problem-specific knowledge in a problem-independent way

Best Frist Search

- An instance of the general tree search (or graph) search algorithm
  - strategy: select next node based on an evaluation function f: state space $\rightarrow \mathbb{R}$
  - select node with lowest value $f(n)$
- Implementation:
  selectFrom(*fringe*, *strategy*)
  - priority queue: maintains fringe in ascending order of f-values

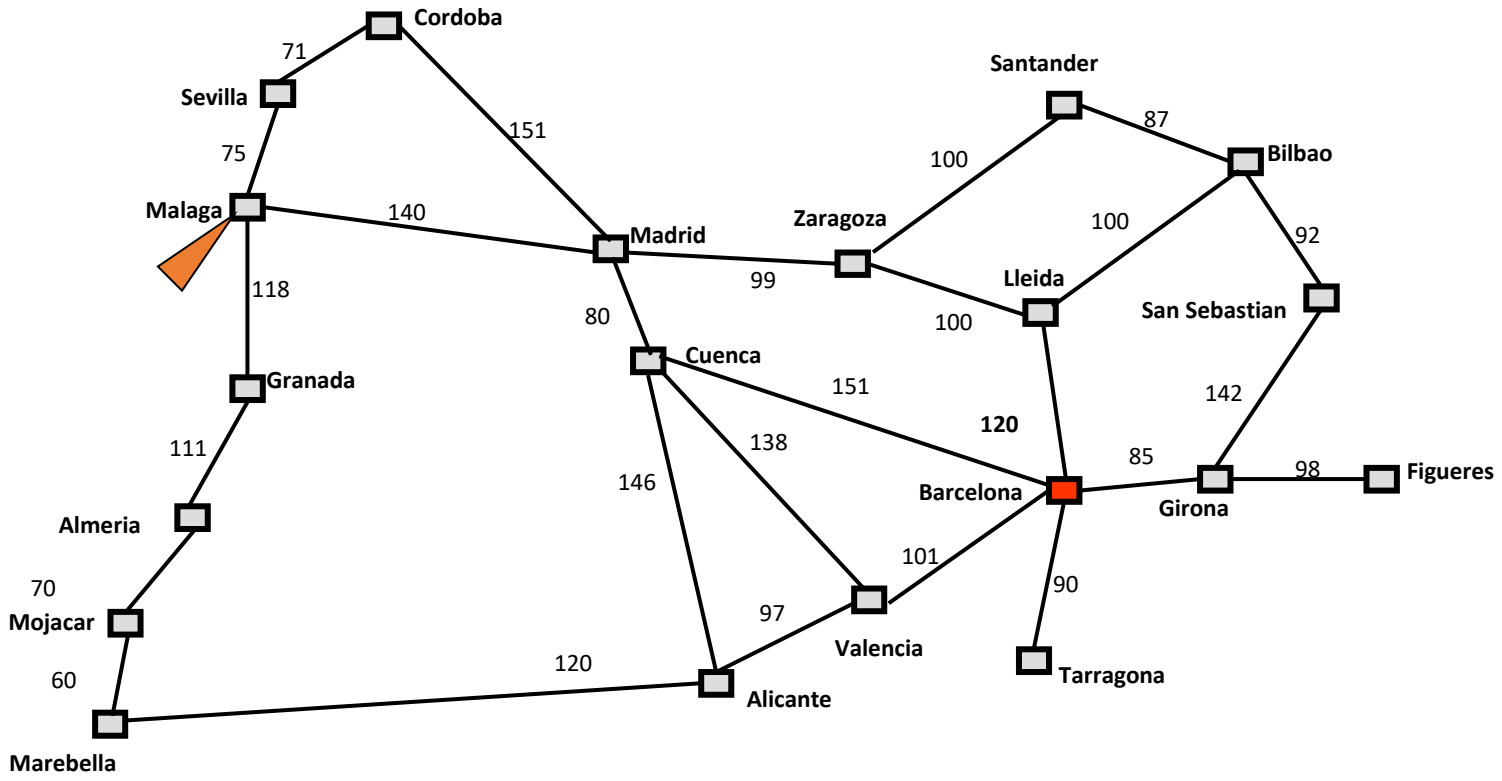
Greedy Best-First Search

- use heuristic function as evaluation function:

$$f(n) = h(n)$$

  - always expands the node that is closest to the goal node
  - eats the largest chunk out of the remaining distance, hence, “greedy”

Real-World Problem: Touring in Spain

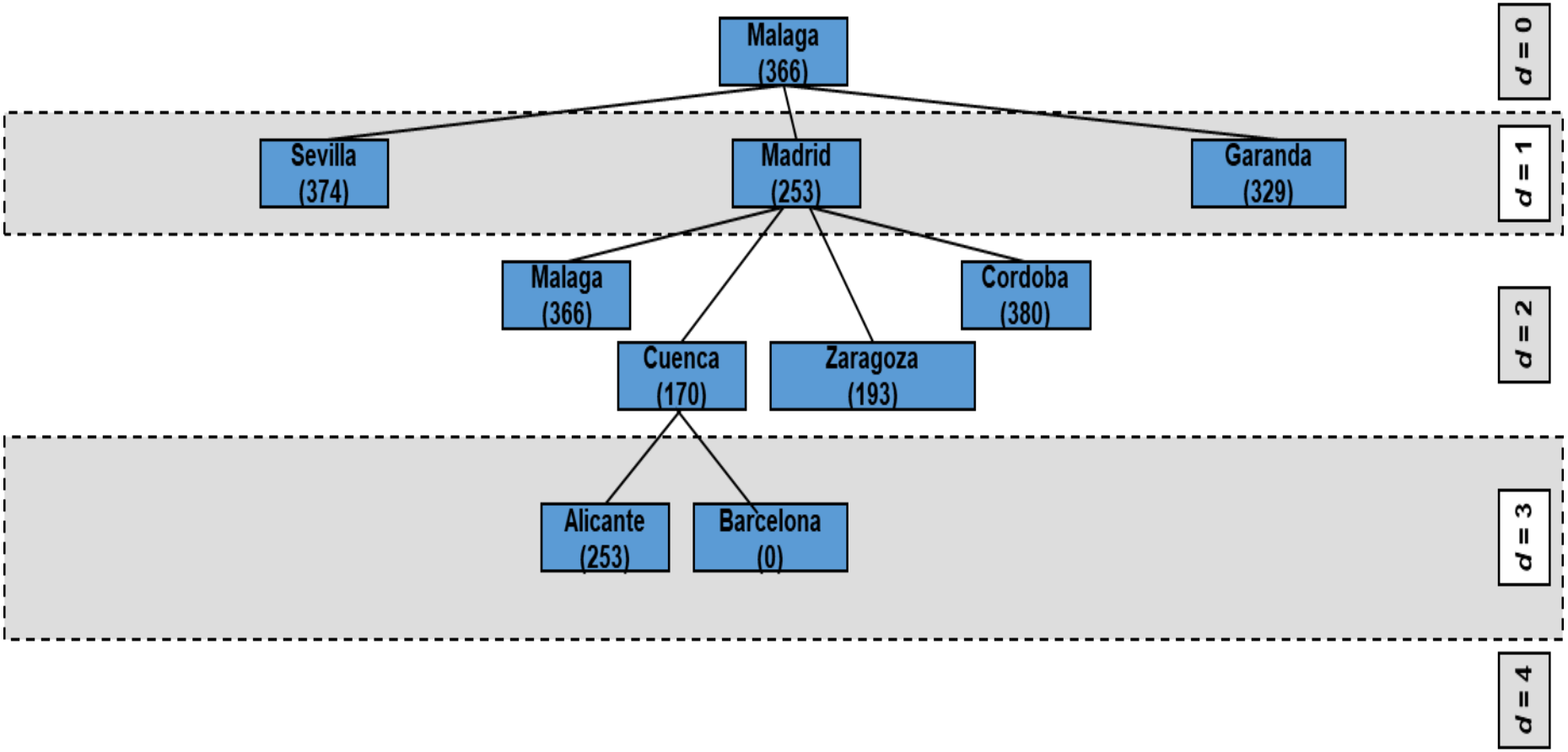


Real-World Problem: Touring in Spain

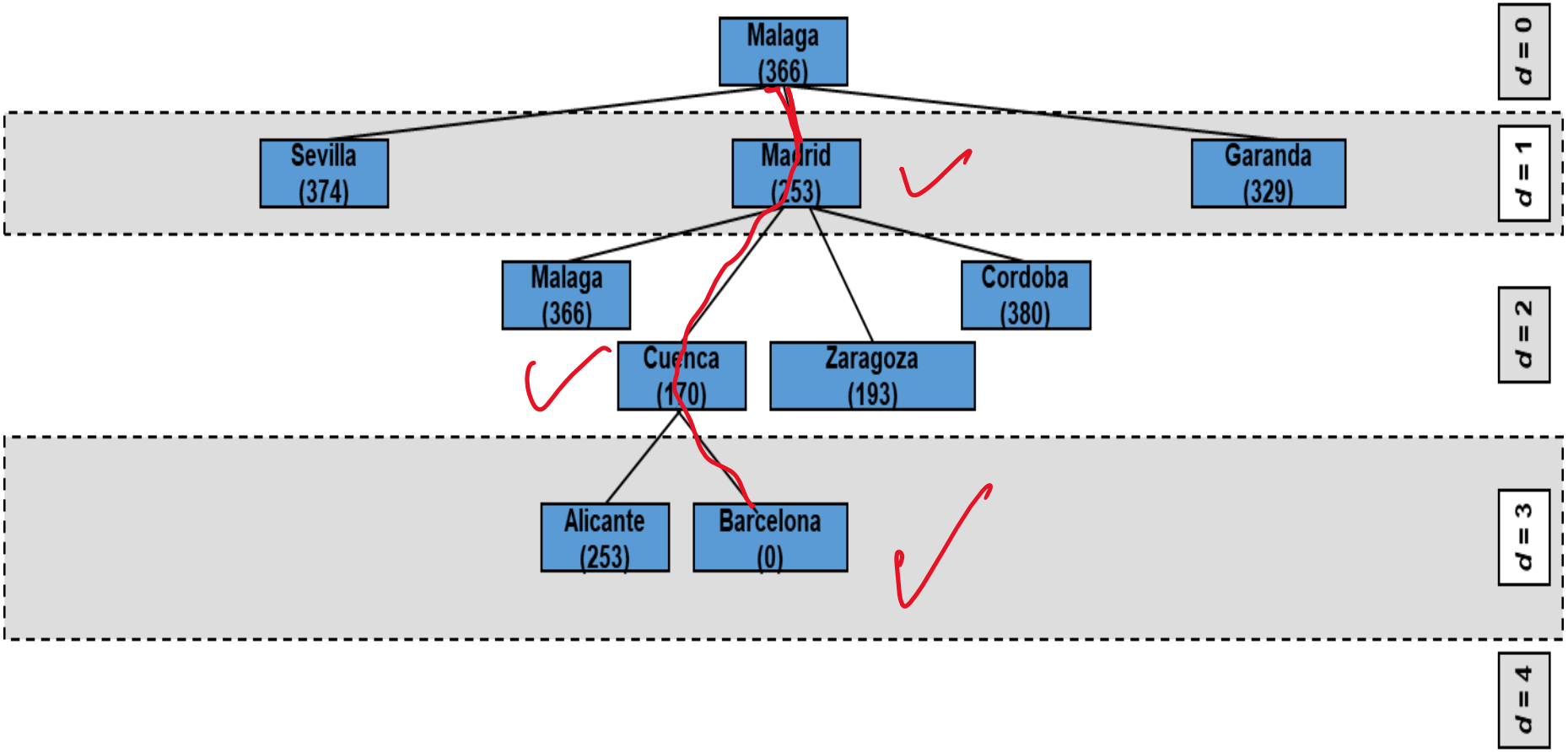
- $h_{SLD}(n)$ = straight-line distance to Barcelona

Malaga	366	Girona	101	-----
Barcelona	0	Bilbo	226	-----
Valencia	87	Almeria	340	-----
Alicante	198	Madrid	253	-----
Figueres	130	-----		-----
Zaragoza	170	-----		-----
Tarragona	80	-----		-----

Greedy Best-First Search: Touring Romania



Greedy Best-First Search: Touring Romania



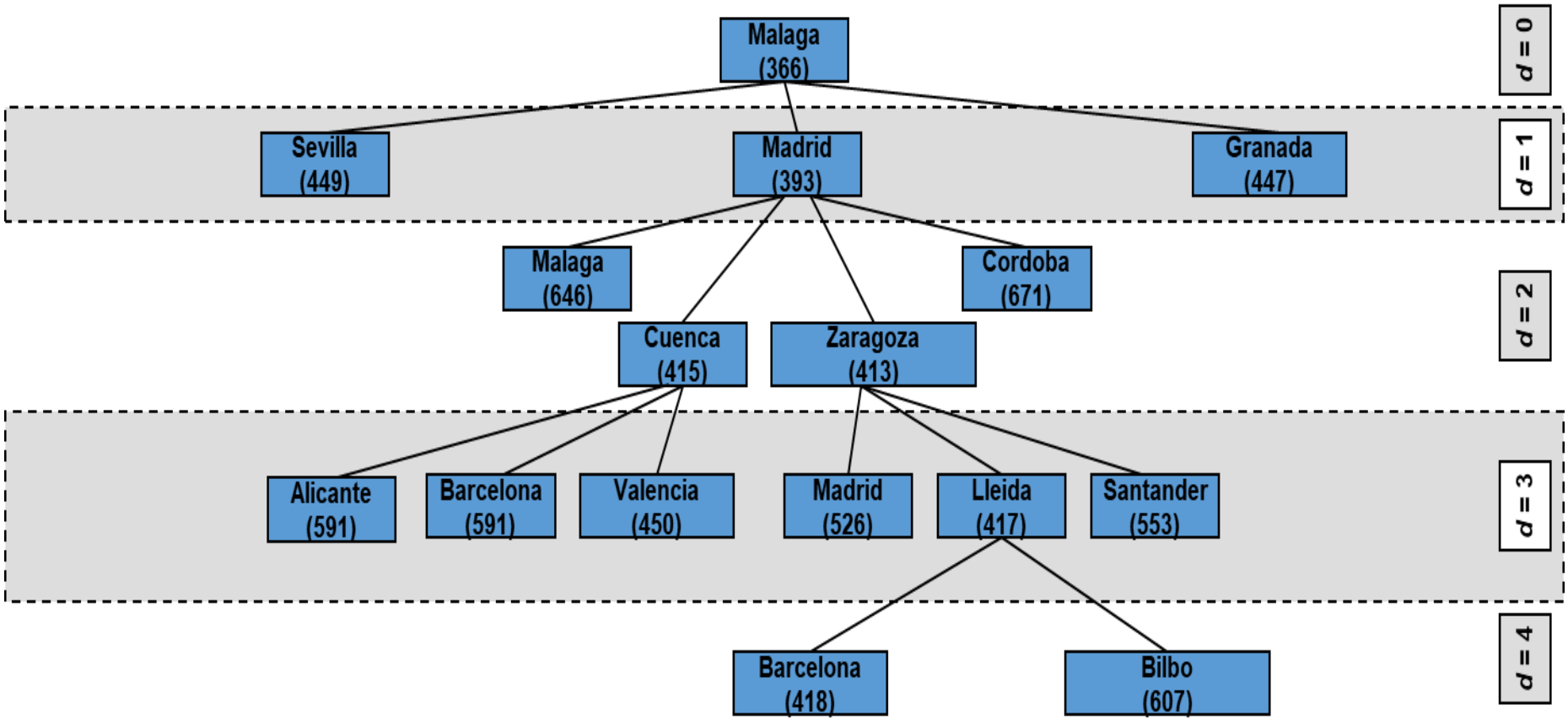
Overview

- Heuristic Search Strategies
- **The A* Algorithm (for Tree Search)**
- Graph Search with A*
- (Good) Heuristics

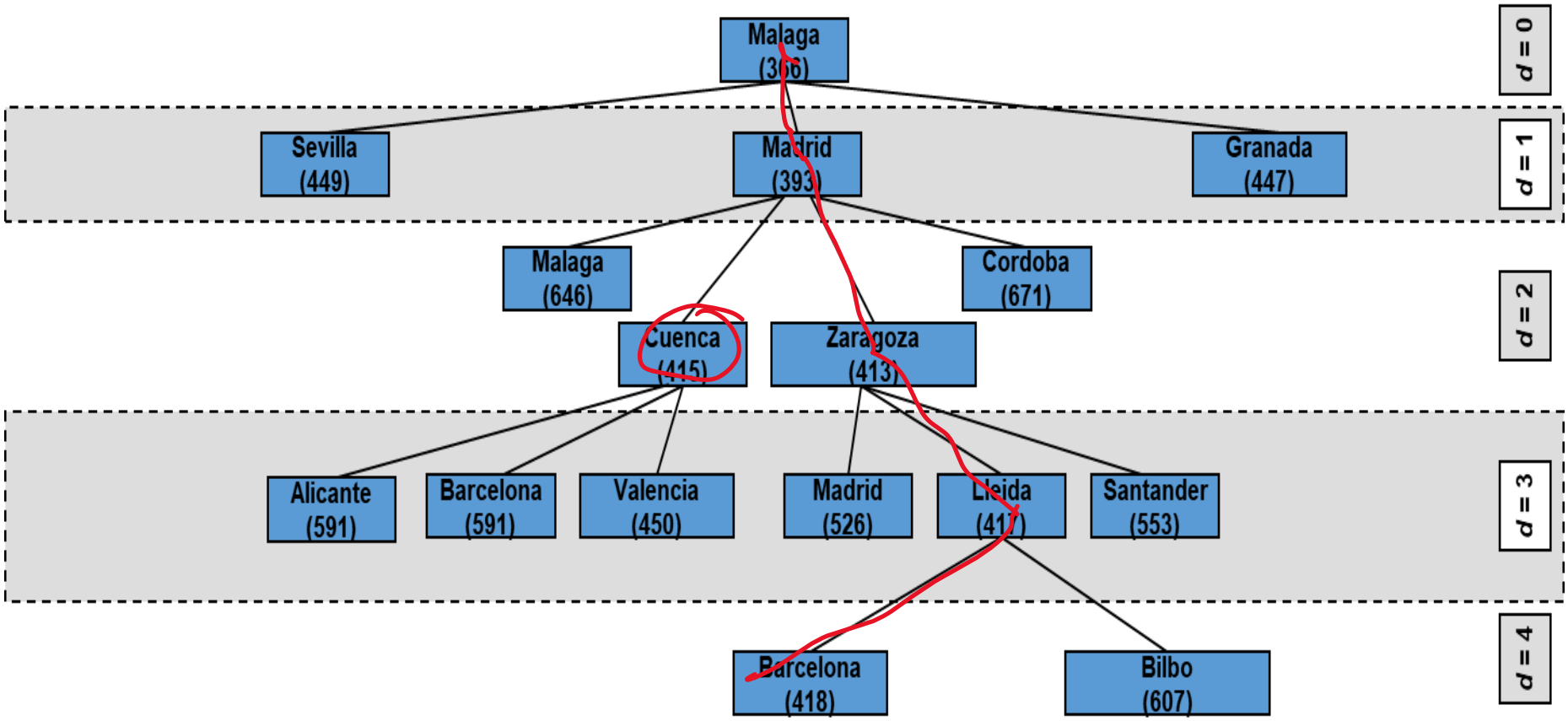
A* Search

- best-first search where
  $f(n) = h(n) + g(n)$
  - $h(n)$ the heuristic function (as before)
  - $g(n)$ the cost to reach the node n
- evaluation function:
  $f(n)$ = estimated cost of the cheapest solution through n
- A* search is optimal if $h(n)$ is admissible

A* (Best-First) Search: Touring Spain

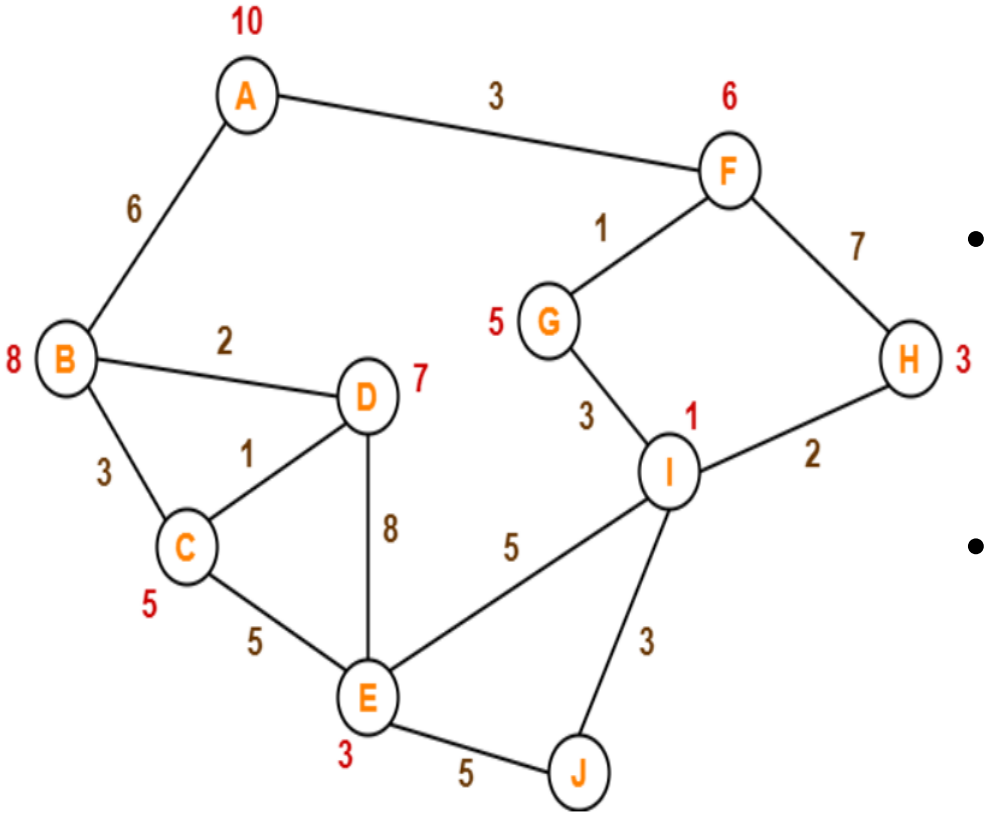


A* (Best-First) Search: Touring Spain



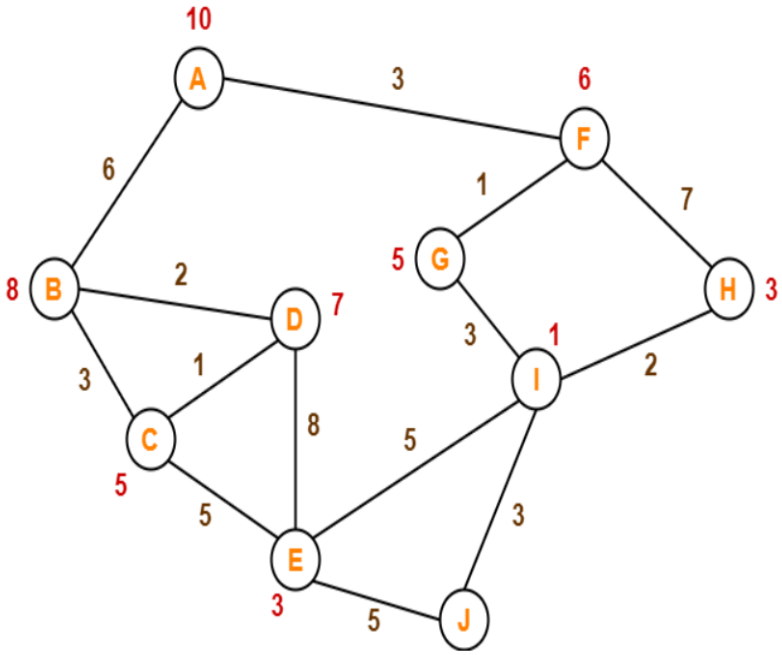
A* (Best-First) Search: Example

Consider the following graph-



- **The numbers written on edges represent the distance between the nodes.**
- **The numbers written on nodes represent the heuristic value.**
- **Find the most cost-effective path to reach from start state A to final state J using A* Algorithm.**

A* (Best-First) Search: Example



Step-01:

We start with node A.

- Node B and Node F can be reached from node A.

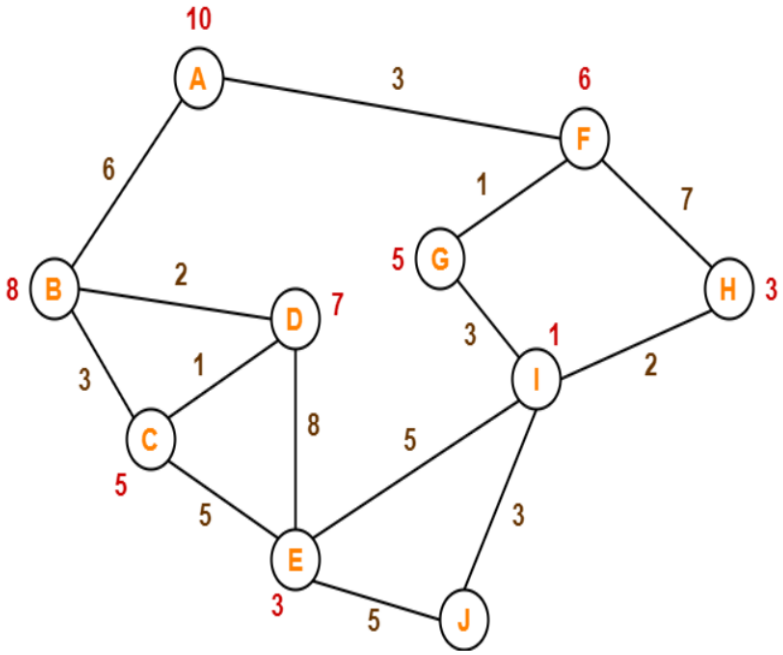
A* Algorithm calculates f(B) and f(F).

- f(B) = 6 + 8 = 14
- f(F) = 3 + 6 = 9

Since f(F) < f(B), so it decides to go to node F.

Path- A → F

A* (Best-First) Search: Example



Step-02:

Node G and Node H can be reached from node F.

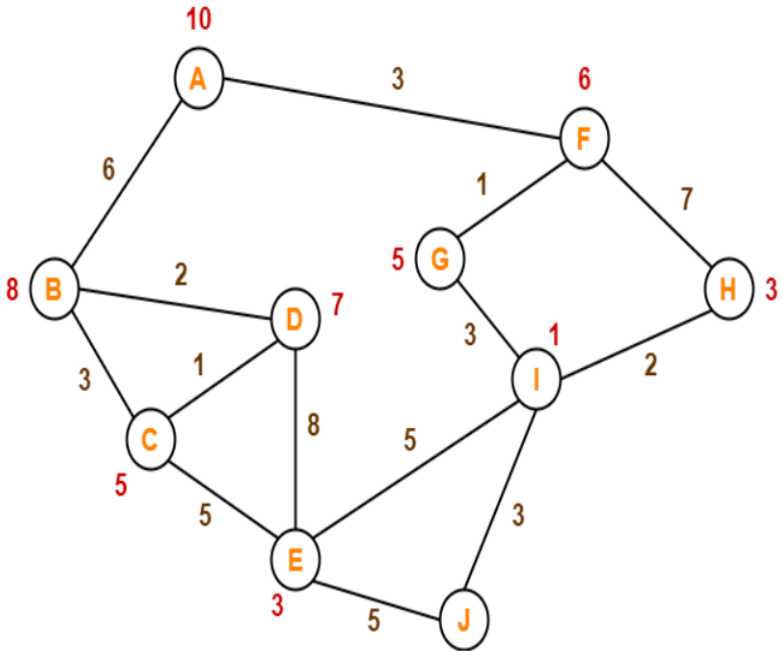
A* Algorithm calculates f(G) and f(H).

f(G) = (3+1) + 5 = 9
f(H) = (3+7) + 3 = 13

Since f(G) < f(H), so it decides to go to node G.

Path- A → F → G

A* (Best-First) Search: Example



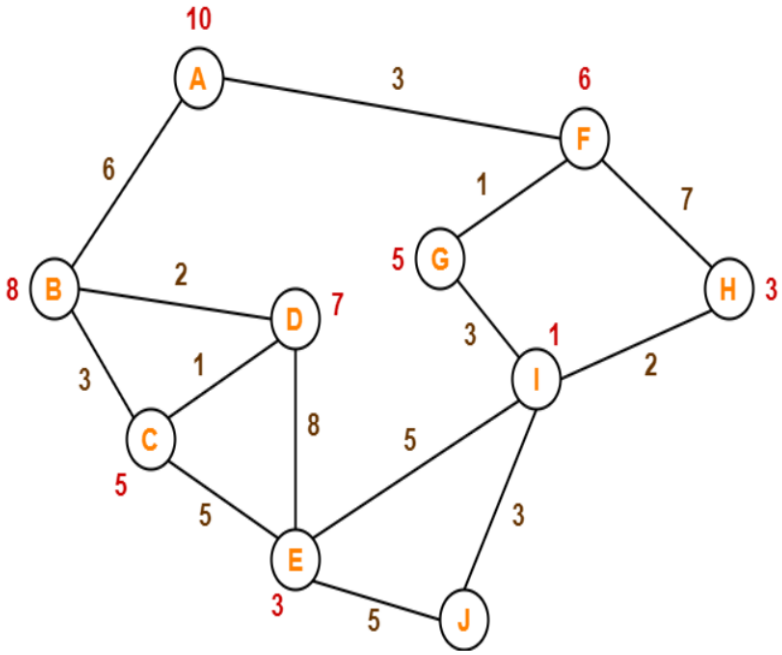
Step-03:

Node I can be reached from node G.

A* Algorithm calculates f(I).
f(I) = (3+1+3) + 1 = 8
It decides to go to node I.

Path- A → F → G → I

A* (Best-First) Search: Example



Step-04:

Node E, Node H and Node J can be reached from node I.

A* Algorithm calculates f(E), f(H) and f(J).

- f(E) = (3+1+3+5) + 3 = 15
- f(H) = (3+1+3+2) + 3 = 12
- f(J) = (3+1+3+3) + 0 = 10

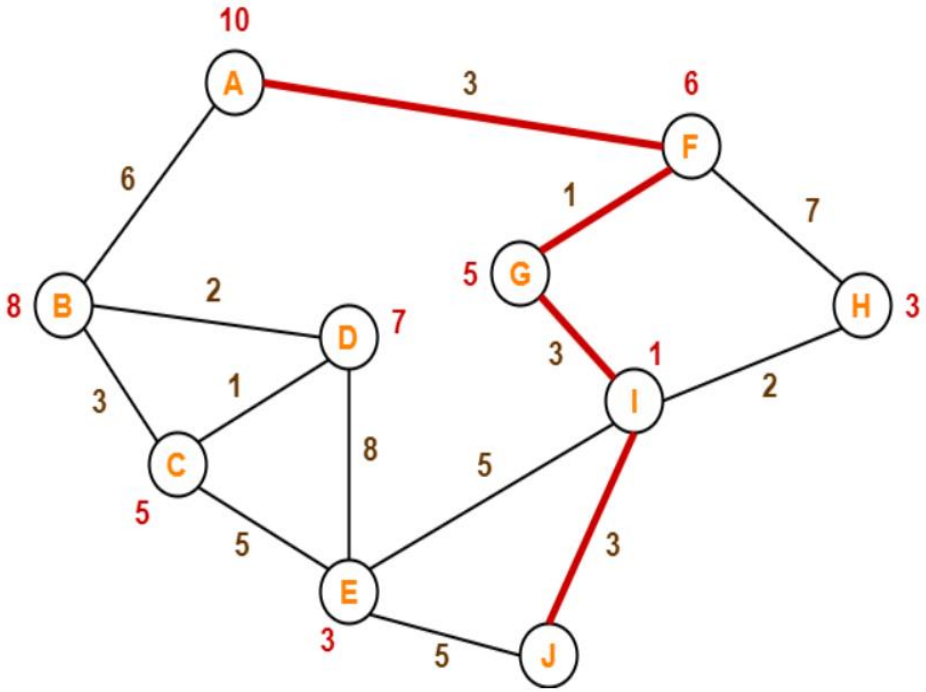
Since f(J) is least, so it decides to go to node J.

A* (Best-First) Search: Example

Final:.

Path- A → F → G → I → J

This is the required shortest path from node A to node J.



Important Note-

- A* Algorithm is one of the best path finding algorithms.
- But it does not produce the shortest path always.
- This is because it heavily depends on heuristics.

A* (Best-First) Search: The Eight-Puzzle

2	8	3
1	6	4
7		5

Initial State

1	2	3
8		4
7	6	5

Final State

Given an initial state of a 8-puzzle problem and final state to be reached-

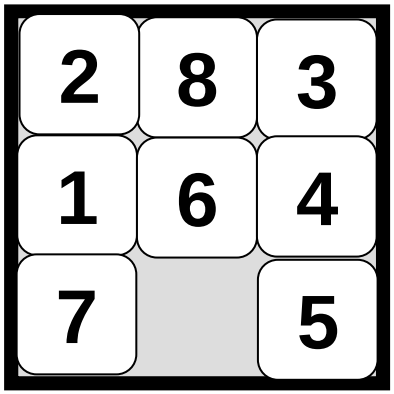
Find the most cost-effective path to reach the final state from initial state using A* Algorithm. Consider g(n) = Depth of node and h(n) = Number of misplaced tiles.

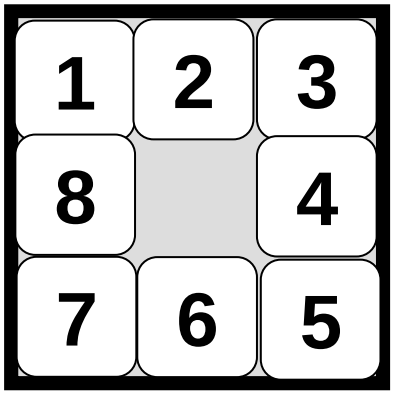
Solution-

- A* Algorithm maintains a tree of paths originating at the initial state.
- It extends those paths one edge at a time.
- It continues until final state is reached.

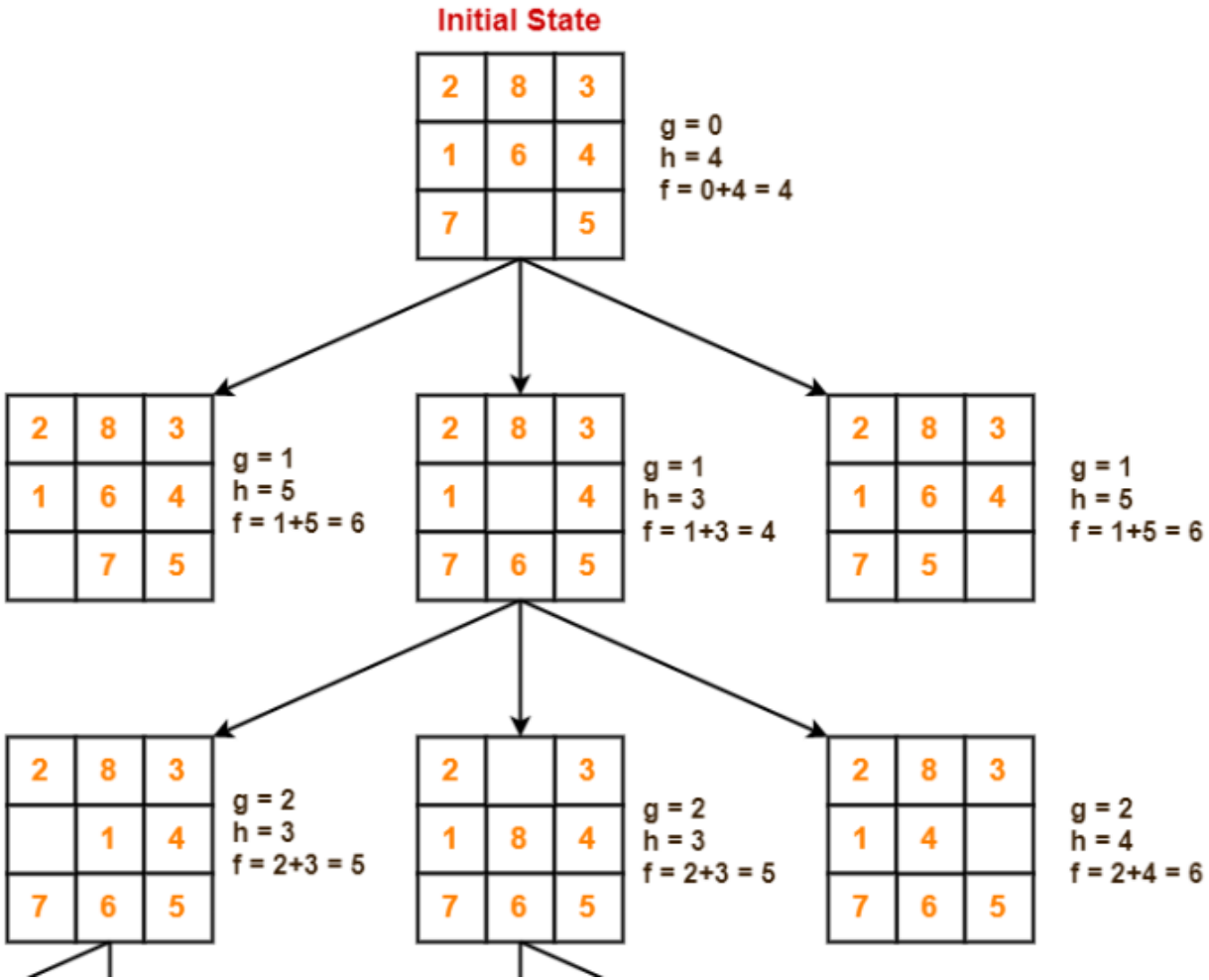
A* (Best-First) Search: The Eight-Puzzle

- h_1: number of misplaced tiles
- h_2: Manhattan block distance
  - Example:
    - *h1* = 4
      4 tiles are misplaced
    - Or h2 = 2+1+1+1 = 5
    - *g* = depth that initially equals 0
      No movement yet

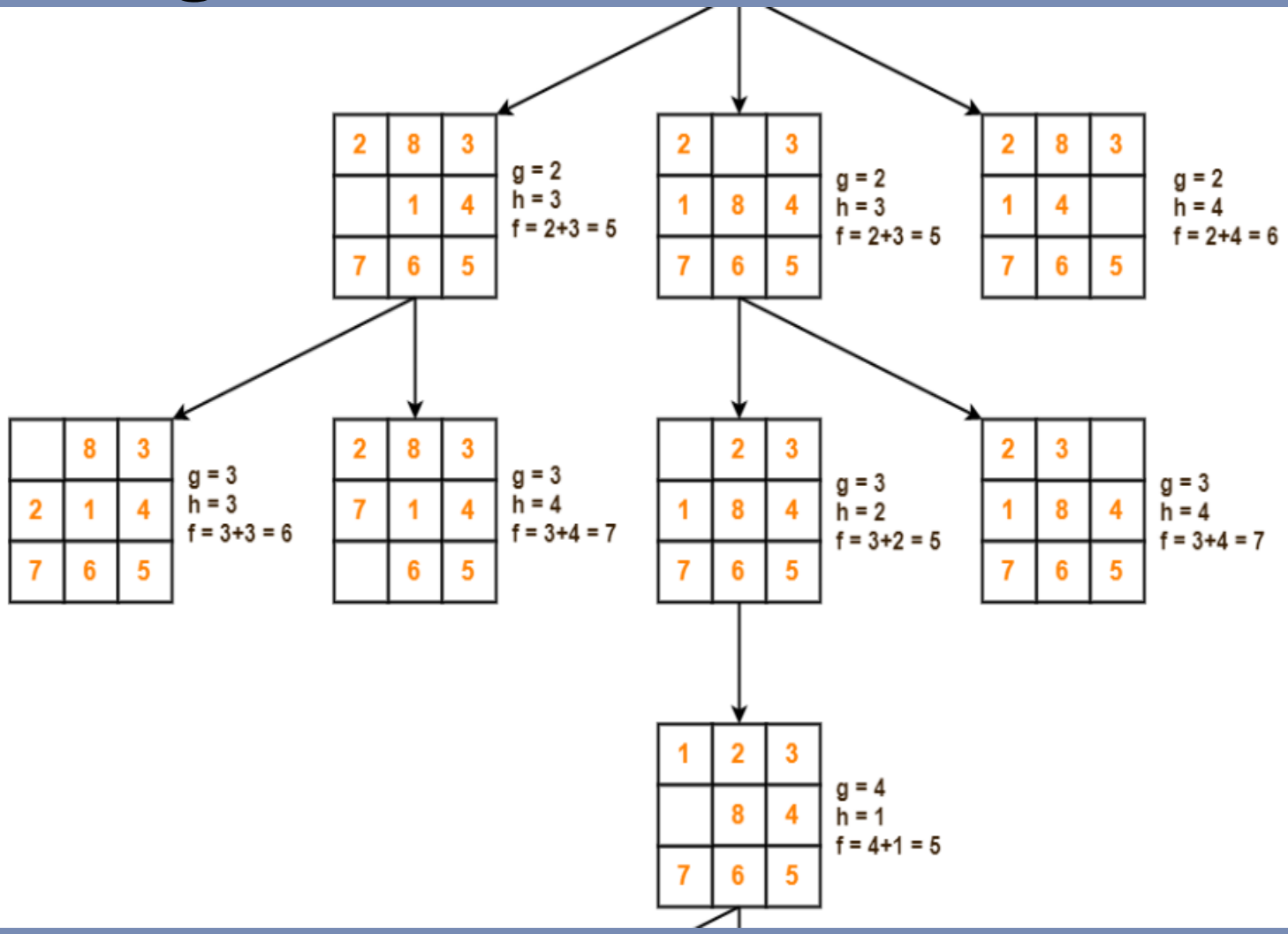




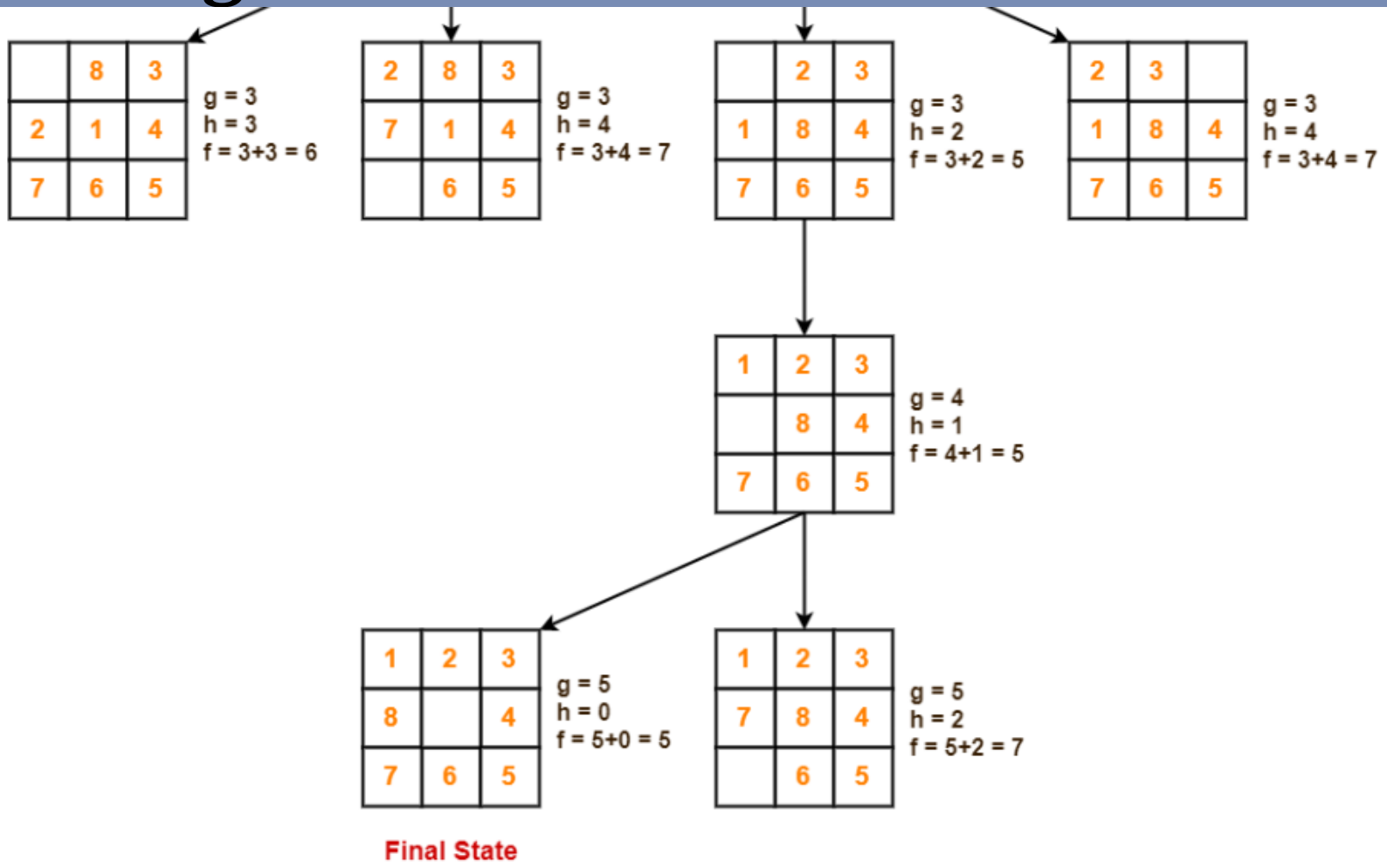
A* (Best-First) Search: The Eight-Puzzle



A* (Best-First) Search: The Eight-Puzzle



A* (Best-First) Search: The Eight-Puzzle



A* (Best-First) Search: A * search properties

- **Admissible**: If a solution exists for the given problem, the first solution found by A* is an optimal solution.
- **Complete**: A* is also complete algorithm, meaning if a solution exists, the answer is bound to be found in a finite amount of time.
- **Optimal**: A* is optimally efficient for a given heuristic-of the optimal search algorithm that expand search paths from the root node, it can be shown that no other optimal algorithm will expand fewer nodes and find a solution.

A* (Best-First) Search: Advantages/disadvantages

- **Advantages**:
  - It is optimal search algorithm in terms of heuristics.
  - It is one of the best heuristic search techniques in AI.
  - It is used to solve complex search problems.
  - There is no other optimal algorithm guaranteed to expand fewer nodes than A*.

- **Disadvantages**:
  - This algorithm is complete if the branching factor is finite and every action has fixed cost.
  - The performance of A* search is dependent on accuracy of heuristic algorithm used to compute the function h(n).

Overview

- Heuristic Search Strategies
- The A* Algorithm (for Tree Search)
- **Graph Search with A***
- (Good) Heuristics

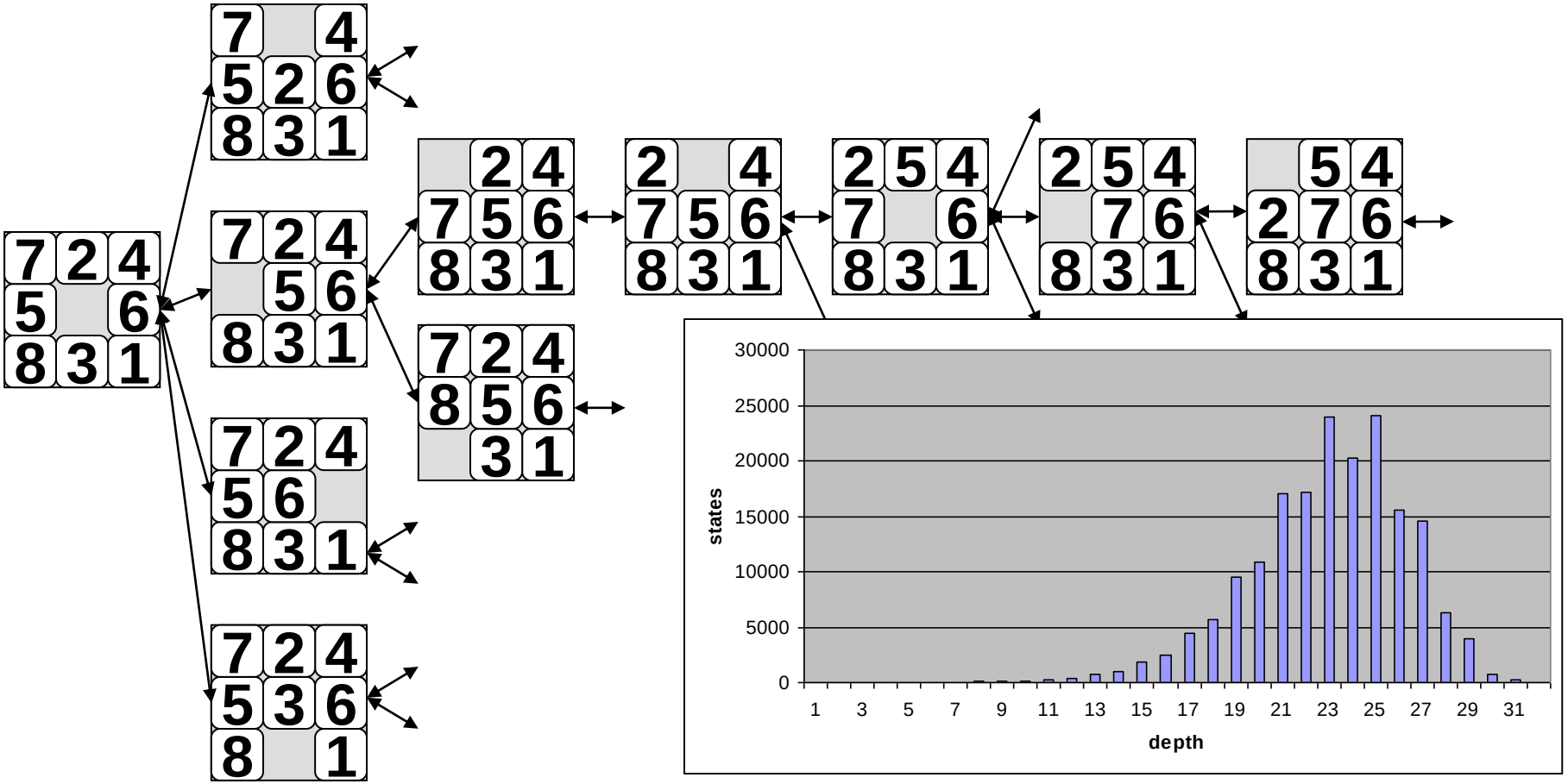
A* Tree/Graph Search Algorithm

```
function aStarTreeSearch(problem, h)
  fringe ← priorityQueue(new searchNode(problem.initialState))
  allNodes ← hashTable(fringe)
  loop
    if empty(fringe) then return failure
    node ← selectFrom(fringe)
    if problem.goalTest(node.state) then
      return pathTo(node)
    for successor in expand(problem, node)
      if not allNodes.contains(successor) then
        fringe ← fringe + successor @ f(successor)
        allNodes.add(successor)
```

A* and Exponential Space

- A* has worst case time and space complexity of $O(b^l)$
- exponential growth of the fringe is normal
  - exponential time complexity may be acceptable
  - exponential space complexity will exhaust any computer's resources all too quickly

The Eight-Puzzle Search Space



Overview

- Heuristic Search Strategies
- The A* Algorithm (for Tree Search)
- Properties of A*
- Graph Search with A*
- **(Good) Heuristics**

Heuristics

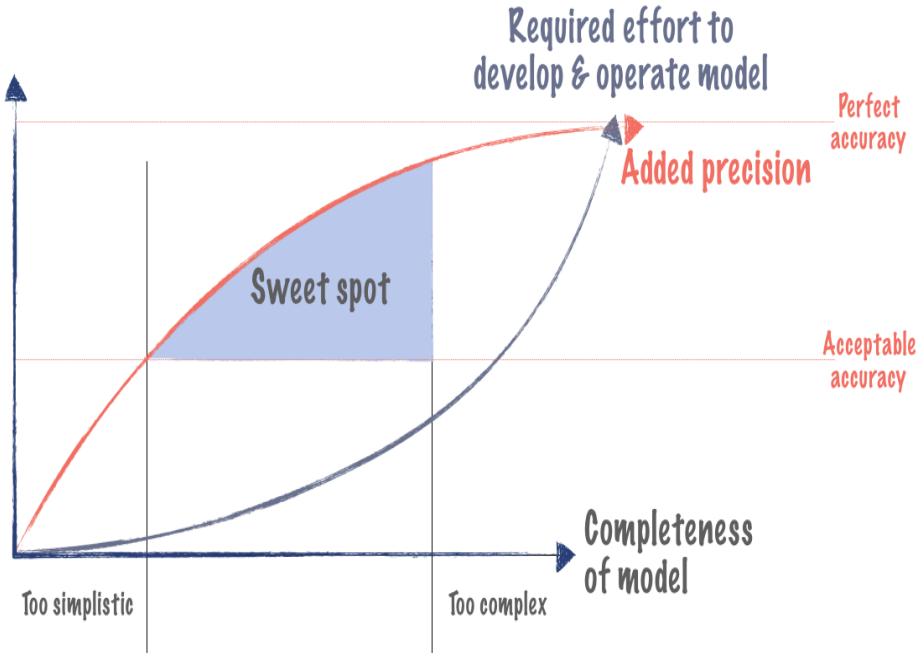
- heuristics are criteria, methods, or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal
- example: you need to select a course to learn
  - Review the materials of each course
  - Use this information to decide which of the courses that were available you want to take.
  - So you used heuristic function about the course to make this choice.

Good Heuristics

- good heuristics
  - indicate a way to reduce the number of states that need to be evaluated
  - help obtain solutions within reasonable time constraints
- trade-off:
  - simplicity
    - must provide a simple means of discriminating between choices
  - accuracy
    - no guarantee that they identify the best course of actions
    - but they should do so sufficiently often

Good Heuristics

- trade-off:
  - simplicity
    - must provide a simple mea of discriminating between choices
  - accuracy
    - no guarantee that they ide the best course of actions
    - but they should do so sufficiently often



- How can we find good heuristics for a given problem?

Heuristics from Simplified Problems
How to find a good heuristic function?

- relaxed problem: a problem with fewer restrictions on the actions than the original problem
- *The cost of an optimal solution for a relaxed problem is an admissible and consistent heuristic for the original problem.*

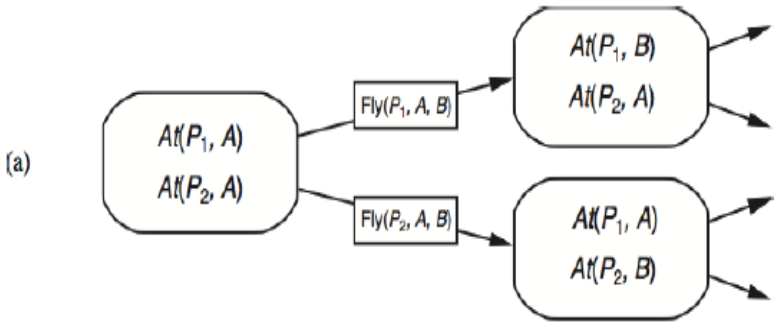
8-Puzzle Actions

- a tile can move from square A to square B if A is horizontally or vertically adjacent to B and B is blank
- Relaxed conditions:
  - a tile can move from square A to square B if A is adjacent to B (⇒ Manhattan distance)
  - a tile can move from square A to square B if B is blank
  - a tile can move from square A to square B (⇒ misplaced tiles)

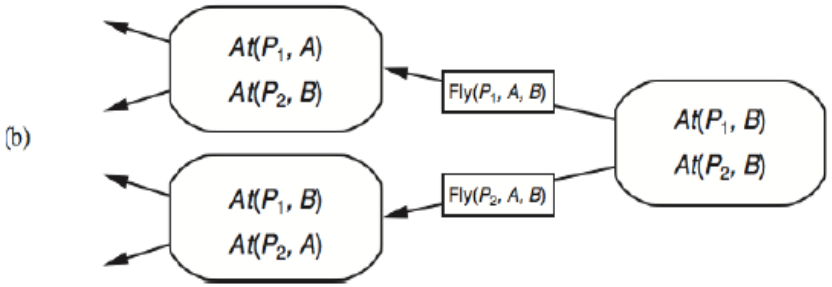
Classical Representations
"Stripes"

Classical Planning
Generate a plan

- **Planning Algorithms**
  - Progression: Forward state-space search



  - Regression: Backward state-space search



Properties of Planning Algorithms

- **Soundness**
  - A planning algorithm is ***sound*** if all solutions are *legal* plans
    - All preconditions, goals, and any additional constraints are satisfied
- **Completeness**
  - A planning algorithm is ***complete*** if a solution can be found whenever one actually exists
  - A planning algorithm is ***strictly complete*** if all solutions are included in the search space
- **Optimality**
  - A planning algorithm is ***optimal*** if it maximizes a predefined measure of plan quality

Linear Planning

- Basic Idea – Goal **stack**
  - *Work **on one goal until completely solved** before moving on to the next goal*
  - Search by reducing the difference between the state and the goals
  - What means (operators) are available to achieve the desired ends (goal)

Linear Planning

- (Linear Planner) (initial-state, goals)
  - state = initial-state; plan = []; stack = [] – Push goals on stack
  - Repeat until stack is empty
    - If top of stack is goal that matches state, then pop stack
    - Else if top of stack is a conjunctive goal g, then
      - Select an ordering for the subgoals of g, and push them on stack
    - Else if top of stack is a simple goal sg, then
      - Choose an operator o whose add-list matches goal sg
      - Replace goal sg with operator o
      - Push the preconditions of o on the stack
    - Else if top of stack is an operator o, then
    - state = apply(o, state)
    - plan = [plan; o]

Means-ends Analysis in Linear Planning

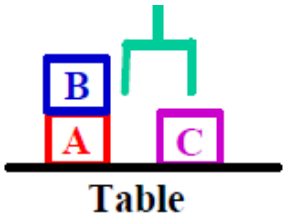
(*Newell and Simon*)

GPS Algorithm (*state*, *goals*, *plan*)

- If *goals* ⊆ *state*, then return (*state*, *plan*) *(The symbol ⊆ is used to denote containment of sets.)*
- **Choose** a difference *d* ∈ *goals* between *state* and *goals*
- **Choose** an operator *o* to reduce the difference *d*
- If no applicable operators, then return *False*
- *(state, plan)* = **GPS** (*state*, preconditions(*o*), *plan*)
- If *state*, then return **GPS** (apply (*o*, *state*), *goals*, *[plan, o]*)

Initial call: GPS (initial-state, initial-goals, [])

GPS Blocks World Example



Pickup_from_table(b)

Pre: Block(b), Handempty
Clear(b), On(b, Table)

Add: Holding(b)

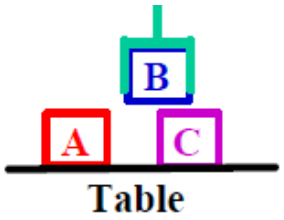
Delete: Handempty,
On(b, Table)

Putdown_on_table(b)

Pre: Block(b), Holding(b)

Add: Handempty,
On(b, Table)

Delete: Holding(b)



Pickup_from_block(b, c)

Pre: Block(b), Handempty
Clear(b), On(b, c), Block(c)

Add: Holding(b), Clear(c)

Delete: Handempty,
On(b, c)

Putdown_on_block(b, c)

Pre: Block(b), Holding(b)
Block(c), Clear(c), $b \neq c$

Add: Handempty, On(b, c)

Delete: Holding(b), Clear(c)

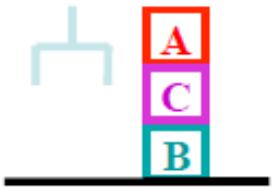
GPS Blocks-World Example

1. Search Stack

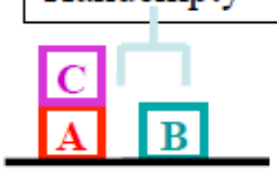
On(A, C) On(C, B)

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty



Goal



Initial State

2. Search Stack

On(A, C) On(C, B)

On(A, C)

On(C, B)

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

3. Search Stack

On(A, C) On(C, B)

On(A, C)

Put_Block(C, B)

Holding(C) Clear(B)

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

4. Search Stack

On(A, C) On(C, B)

On(A, C)

Put_Block(C, B)

Holding(C) Clear(B)

Holding(C)

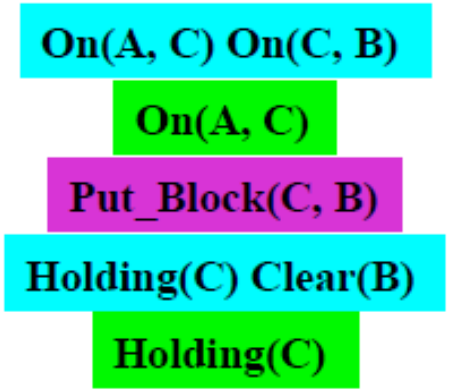
Clear(B)

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

GPS Blocks-World Example

5. Search Stack



State

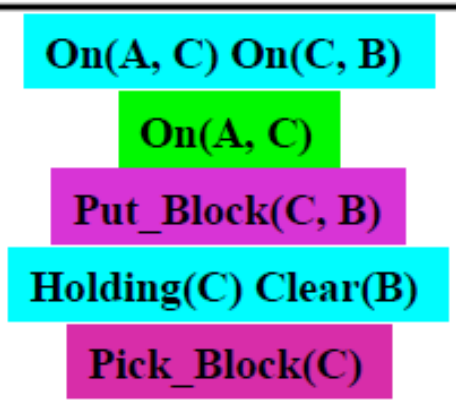
Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

6. Search Stack

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

7. Search Stack



State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

8. Search Stack

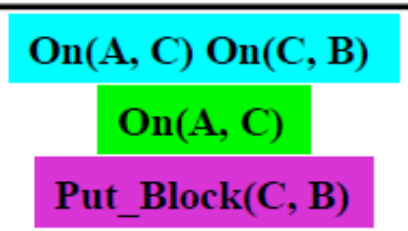
State

Clear(B)
Clear(C)
On(A, Table)
On(B, Table)
Holding(C)
Clear(A)

[Pick_Block(C)]

GPS Blocks-World Example

9. Search Stack

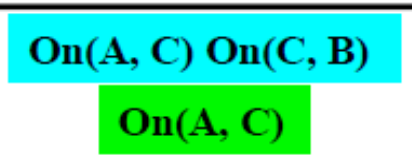


State

Clear(B)
Clear(C)
On(A, Table)
On(B, Table)
Holding(C)
Clear(A)

[Pick_Block(C)]

10. Search Stack

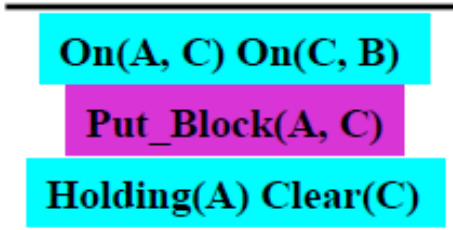


State

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

[Pick_Block(C); Put_Block(C, B)]

11. Search Stack



State

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

**[Pick_Block(C)
Put_Block(C, B)]**

12. Search Stack

State

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

**[Pick_Block(C)
Put_Block(C, B)]**

GPS Blocks-World Example

13. Search Stack

- On(A, C) On(C, B)
- Put_Block(A, C)
- Holding(A) Clear(C)
- Holding(A)

State

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

[Pick_Block(C);
Put_Block(C, B)]

14. Search Stack

- On(A, C) On(C, B)
- Put_Block(A, C)
- Holding(A) Clear(C)
- Pick_Table(A)
- Handempty Clear(A) On(A, Table)

State

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

[Pick_Block(C); Put_Block(C, B)]

15. Search Stack

- On(A, C) On(C, B)
- Put_Block(A, C)
- Holding(A) Clear(C)
- Pick_Table(A)

State

Clear(C)
On(A, Table)
On(B, Table)
Clear(A)
Handempty
On(C, B)

[Pick_Block(C);
Put_Block(C, B)]

16. Search Stack

- On(A, C) On(C, B)
- Put_Block(A, C)
- Holding(A) Clear(C)

State

Clear(C)
On(B, Table)
Clear(A)
On(C, B)
Holding(A)

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A)]

GPS Blocks-World Example

17. Search Stack

On(A, C) On(C, B)

Put_Block(A, C)

State

Clear(C)
On(B, Table)
Clear(A)
On(C, B)
Holding(A)

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A)]

18. Search Stack

On(A, C) On(C, B)

State

On(B, Table)
Clear(A)
On(C, B)
Handempty
On(A, C)

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A);
Put_Block(A, C)]

19. Search Stack

State

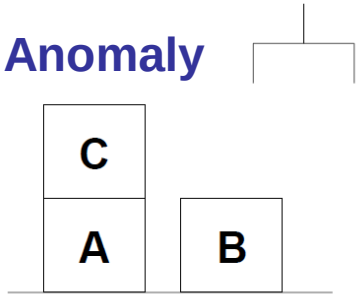
On(B, Table)
Clear(A)
On(C, B)
Handempty
On(A, C)

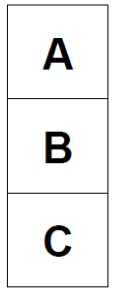
[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A);
Put_Block(A, C)]

Linear Planning

- Sound?
- Optimal?
- Complete?

The Sussman Anomaly





Pickup_from_Table (?b)
Pre: (Armempty)
(clear ?b)
(on-table ?b)
Add: (holding ?b)
Delete: (Armempty)
(on-table ?b)
(clear ?b)

Putdown_on_Table(?b)
Pre: (holding ?b)
Add: (Armempty)
(on-table ?b)
Delete: (holding ?b)

Pickup_fom_block (?a, ?b)
Pre: (Armempty)
(clear ?a) (on ?a ?b)
Add: (holding ?a) (clear ?b)
Delete: (Armempty)
(on ?a ?b) (clear ?a)

Putdown_on_block (?a, ?b)
Pre: (holding ?a) (clear ?b)
Add: (Armempty)
(on ?a ?b)
Delete: (holding ?a)
(clear ?b)

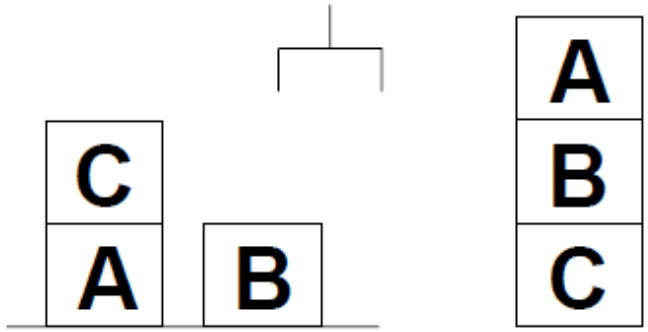
The Sussman Anomaly

Linear Solution:

- *(on B, C)*
  - *Pickup (B)*
  - *Stack (B, C)*
- *(on A, B)*
  - *Unstack (B, C)*
  - *Putdown (B)*
  - *Unstack (C, A)*
  - *Putdown (C)*
  - *Stack (A, B)*
- *(on B, C)*
  - *Unstack (A, B)*
  - *Putdown (A)*
  - *Pickup (B)*
  - *Stack (B, C)*
- *(on A, B)*
  - *Pickup (A)*
  - *Stack (A,B)*

Linear Solution:

- *(on A, B)*
  - *Unstack (C, A)*
  - *Putdown (C)*
  - *Stack (A, B)*
- *(on B, C)*
  - *Unstack (A, B)*
  - *Putdown (A)*
  - *Pickup (B)*
  - *Stack (B, C)*
- *(on A, B)*
  - *Pickup (A)*
  - *Stack (A,B)*



Linear Planning – Goal Stack

- ✓ **Advantages**
  - – Reduced search space, since goals are solved one at a time, and not all possible goal orderings are considered
  - – Advantageous if goals are (mainly) independent
  - – Linear planning is ***sound***
- ❖ **Disadvantages**
  - – Linear planning may produce ***suboptimal*** solutions (based on the number of operators in the plan)
  - – Planner's efficiency is sensitive to goal orderings
    - Control knowledge for the “right” ordering
    - Random restarts
    - Iterative deepening
- ❖ Completeness?

Incompleteness of Linear Planning

Initial state:

(at obj1 locA)
(at obj2 locA)
(at ROCKET locA)
(has-fuel ROCKET)

Goal statement:

(and
(at obj1 locB)
(at obj2 locB))

INCOMPLETE EXAMPLE?

Goal	*Plan*
(at obj1 locB)	(LOAD-ROCKET obj1 locA) (MOVE-ROCKET) (UNLOAD-ROCKET obj1 locB)
(at obj2 locB)	*failure*

(OPERATOR LOAD-ROCKET
:preconds
?roc ROCKET
?obj OBJECT
?loc LOCATION
(and (at ?obj ?loc)
(at ?roc ?loc))
:effects
add (inside ?obj ?roc)
del (at ?obj ?loc))

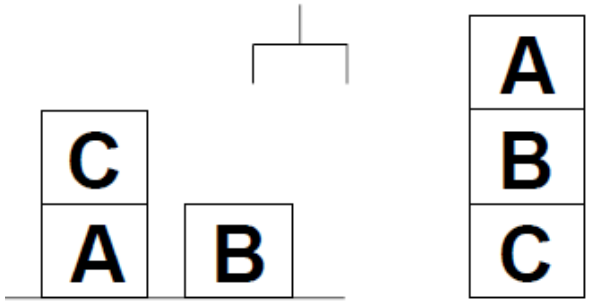
(OPERATOR UNLOAD-ROCKET
:preconds
?roc ROCKET
?obj OBJECT
?loc LOCATION
(and (inside ?obj ?roc)
(at ?roc ?loc))
:effects
add (at ?obj ?loc)
del (inside ?obj ?roc))

(OPERATOR MOVE-ROCKET
:preconds
?roc ROCKET
?from-l LOCATION
?to-l LOCATION
(and (at ?roc ?from-l)
(has-fuel ?roc))
:effects
add (at ?roc ?to-l)
del (at ?roc ?from-l)
del (has-fuel ?roc))

The Sussman Anomaly- Nonlinear planning

NonLinear Solution:

- (on A B)
  - Unstack (C, A)
  - Putdown (C)
- ***(on B C)***
  - ***Pickup (B)***
  - ***Stack (B, C)***
- ***(on A B)***
  - ***Pickup (A)***
  - ***Stack (A, B)***



- Nonlinear planning is used to a goal set instead of a goal stack, and
- It is included in the search space of all possible subgoal orderings.
- It handles the goal interactions by interleaving method.

The Sussman Anomaly- Nonlinear planning

✓ **Advantage**

Non-linear planning may be an optimal solution with respect to plan length (depending on search strategy used).

❖ **Disadvantages**

- It takes larger search space, since all possible goal orderings are taken into consideration.
- Complex algorithm to understand.

Why is Planning Hard?

- **Planning involves a complex search:**
  - Alternative operators to achieve a goal
  - Multiple goals that interact
  - Solution optimality, quality
  - Planning efficiency, soundness, completeness
- **State representation**
  - The frame problem
  - The "choice" of predicates
    - On-table (x), On (x, table), On-table-A, On-table-B,…
- **Action representation**
  - Many alternative definitions
  - Conditional effects
  - Quantification
  - Functions
    - Generation – planning algorithm(S)
  - Reduce to "needed" definition
  - Uncertainty

Summary

- **Planning:** selecting one sequence of actions (operators) that transform (apply to) an initial state to a final state where the goal statement is true.

- **Means-ends analysis:** identify and reduce, as soon as possible, *differences* between state and goals.

- **Linear planning:** backward chaining with means-ends analysis using a stack of goals - potentially efficient, possibly suboptimal, incomplete; GPS, STRIPS.

- **Nonlinear planning with means-ends analysis:** backward chaining using a set of goals; reason about *when* "to reduce the differences;" **Prodigy4.0**.

End