Escola Tècnica Superior d'Enginyeria
Universitat Rovira i Virgili

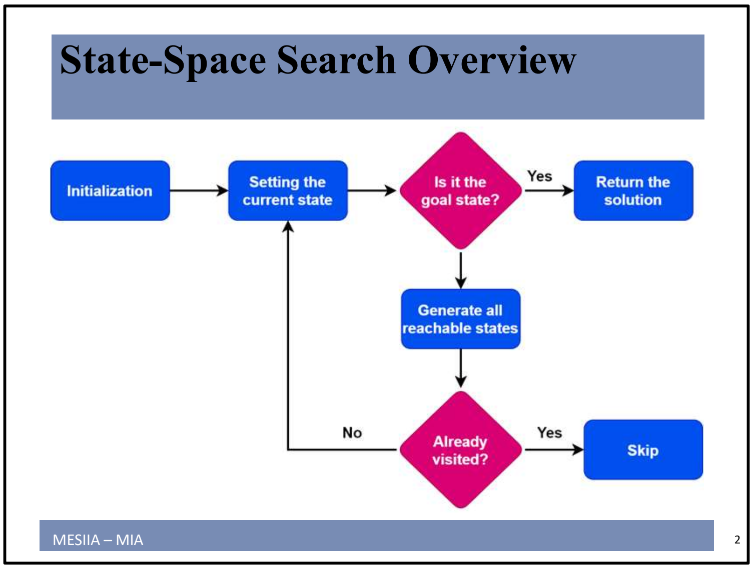
# AI Planning

**Hatem A. Rashwan**

# State-Space Search

A state space is a mathematical representation of a problem that defines all possible states that the problem can be in. Furthermore, in search algorithms, we use a state space to represent the current state of the problem, the initial state, and the goal state. Additionally, we represent each state in the state space by a set of variables.

**State space search is a method used widely in AI and computer science to find a solution to a problem by searching through the set of possible states of the problem.** Furthermore, a state space search algorithm uses the state space to navigate from the initial state to the goal state. Additionally, it generates and explores possible successors of the current state until we find a solution.



## State-Space Search Overview

Initialization

Setting the current state

Is it the goal state?

Yes

Return the solution

Generate all reachable states

Already visited?

No

Yes

Skip

MESIIA – MIA

2

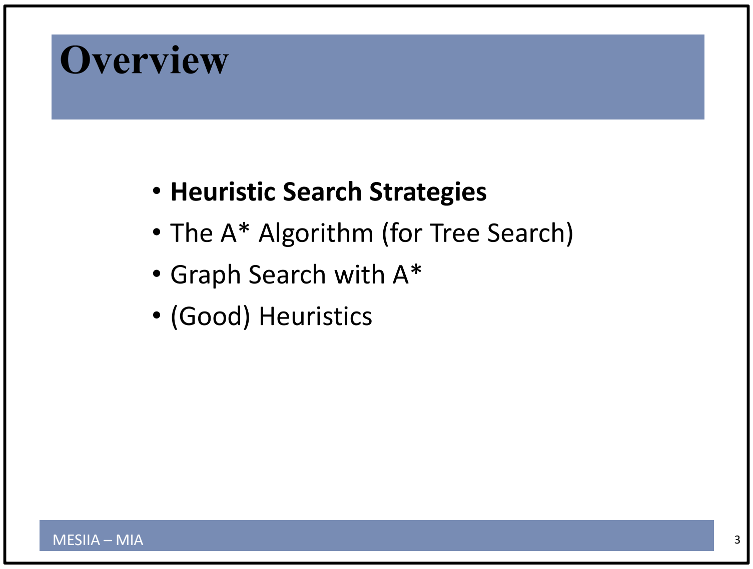
### Overview

The first step is to initialize the search by setting the initial state as the current state. After the initialization step, we check if the current state is a goal state. Finally, if the current state is a goal state, we terminate the algorithm and return the result.

However, if the current state is not the goal state, we generate the set of possible states that can be reached from the current state. Additionally, these states are known as successor states. Furthermore, for each successor state, we check if it has been previously visited. If the state is already explored, we skip the state. If it has not been visited, we add it to the queue of states to be visited.

Moving forward, we set the next state in the queue as the current state and check if it’s a goal state. If we find the goal state, we return the result. Otherwise, we repeat the previous step until we find the goal state or finish exploring all the states. Furthermore, if all possible states have been explored and we can’t reach the target state, we return with no solution.

The specific implementation details of the algorithm depend on the problem. Additionally, the algorithm’s performance depends on the data structures we use to represent the states and keep track of the search.



## Overview

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

**Overview**

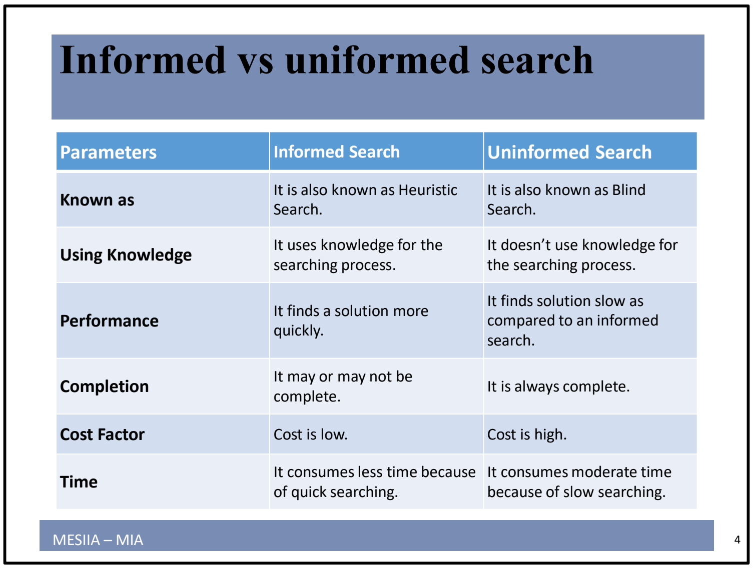
**➢Heuristic Search Strategies**

**•The A* Algorithm (for Tree Search)**

**•Properties of A***

**•Graph Search with A***

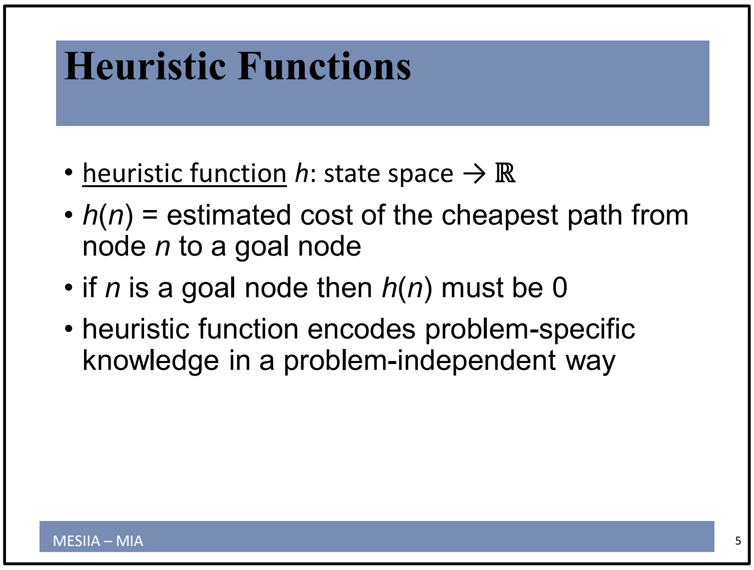
**•(Good) Heuristics**



# 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. |

**We explained in the last lecture the uniformed search (Search Node in a queue) that its strategy depends on LIFO or FIFO (e.g.,** Dijkstra search)
**Today, we will give hits about informed search (Heuristic Search) that can encode problem-specific knowledge in a problem-independent way using heuristic function (e.g.,** A* search)**.**



## Heuristic Functions

- <u>heuristic function</u> *h*: state space → 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

MESIIA – MIA 5

**In the heuristic search, the information about nodes can be encoded in heuristic function.**

**Heuristic Functions**

**•<u>heuristic function</u> *h*: state space → ℝ**

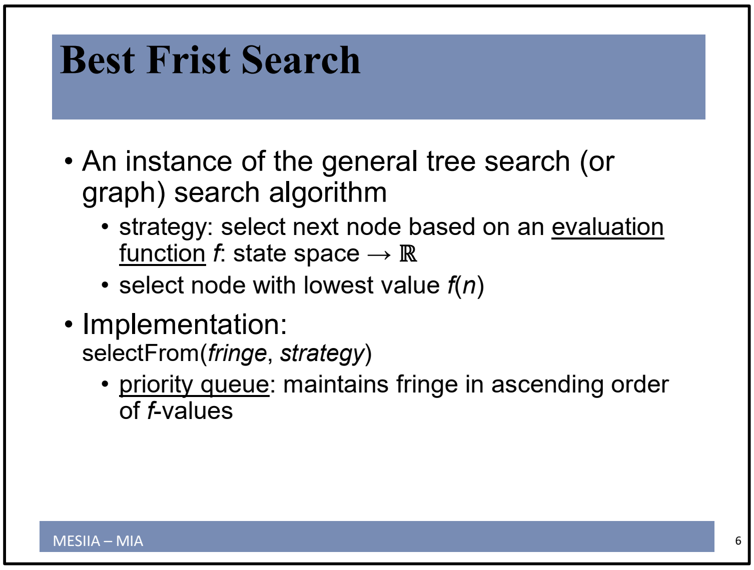
**•*h*(*n*) = estimated cost of the cheapest path from node *n* to a goal node**

**•if *n* is a goal node then *h*(*n*) must be 0**

**•heuristic function encodes problem-specific knowledge in a problem-independent way**

•difference between evaluation function and heuristic function:

- •good evaluation function makes sure nodes are expanded in an order that leads straight to the optimal solution
- •good heuristic function always gives the correct distance to the nearest goal node
- •evaluation function is not problem-specific, but uses heuristic function which is problem-specific
- •For each problem we are looking at, we can define a different heuristic function, which is why the heuristic function is a problem specific
- • but whatever search space we are looking at the heuristic function will always give us a numeric value for each state.
- •And the fact that it is a number is a problem independent.
- •A perfect heuristic function would always give us the correct distance to the goal node.
- •But to find perfect heuristic function is difficult.

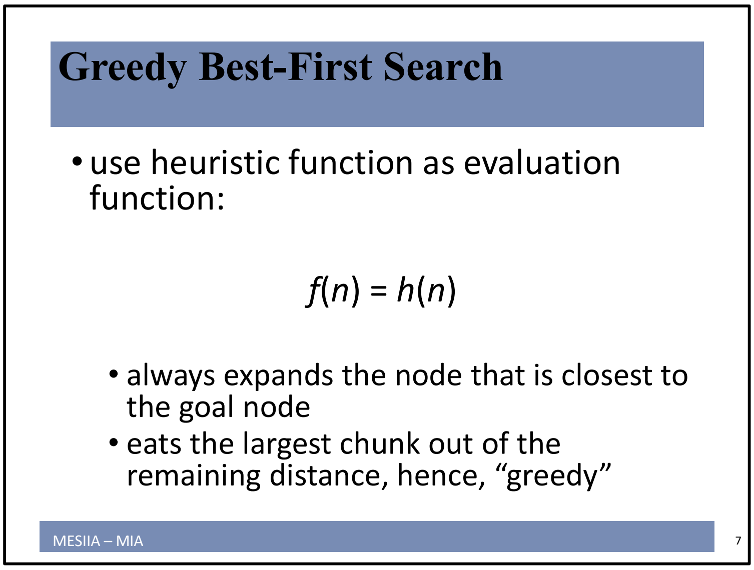


## Best Frist Search

- An instance of the general tree search (or graph) search algorithm
  - strategy: select next node based on an <u>evaluation function</u> *f*: state space → ℝ
  - select node with lowest value *f*(*n*)
- Implementation:
  selectFrom(*fringe*, *strategy*)
  - <u>priority queue</u>: maintains fringe in ascending order of *f*-values

**Best-First Search**

- **an instance of the general tree search (or graph) search algorithm**
  - tree or graph search: both possible; difference only lies in test for repeated states
  - **strategy: select next node based on an <u>evaluation function</u> *f*: state space → ℝ**
    - evaluation function: determines the search strategy
    - intuition: choose function that estimates the distance to the goal
  - **select node with lowest value *f*(*n*)**
    - lowest *f*-value means best node: hence best-first search
- **implementation: selectFrom(*fringe, strategy*)**
  - **<u>priority queue</u>: maintains fringe in ascending order of *f*-values**
    - implementation as binary tree: nodes can be added/retrieved in log-time (still expensive)



## Greedy Best-First Search

- use heuristic function as evaluation function:

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

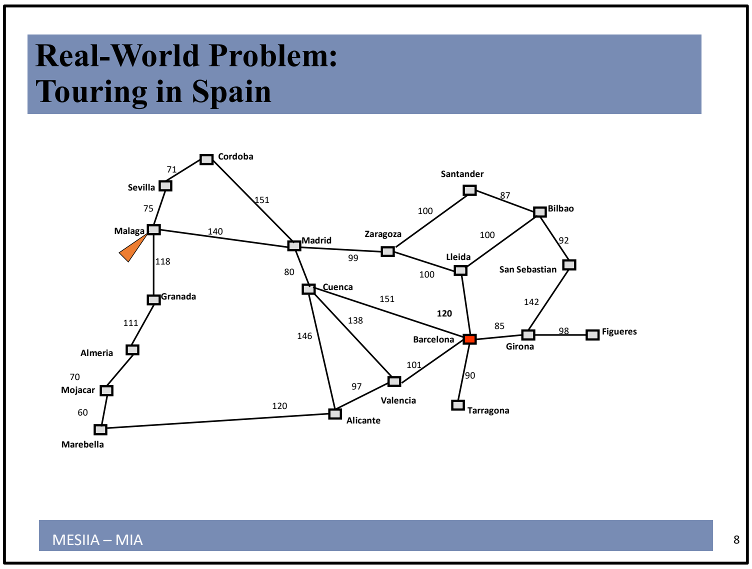
  - always expands the node that is closest to the goal node
  - eats the largest chunk out of the remaining distance, hence, “greedy”

**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”**



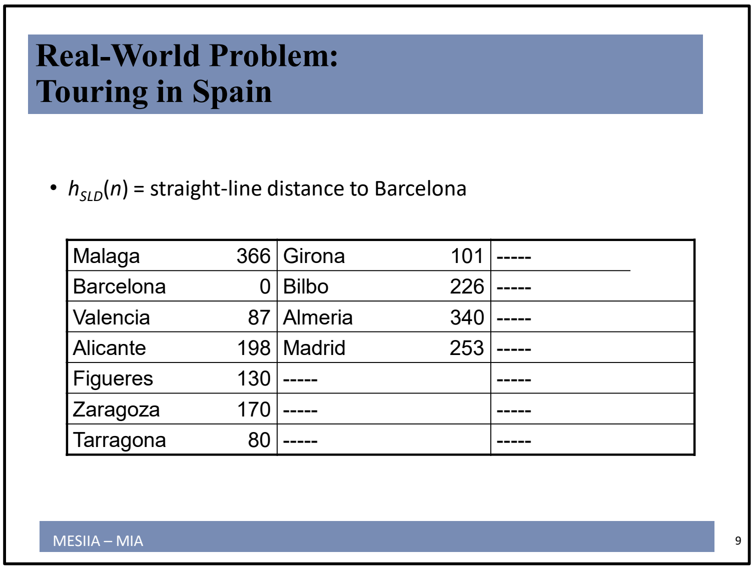
## Real-World Problem: Touring in Spain

# Real-World Problem: Touring in Spain

•shown: rough map of Spain

•initial state: on vacation in Malaga, Spain

•goal? actions? -- “Touring Spain” cannot readily be described in terms of possible actions, goals, and path cost



## 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 | ----- | | ----- |

**Touring in Romania: Heuristic**

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

•straight-line distance: Euclidean distance

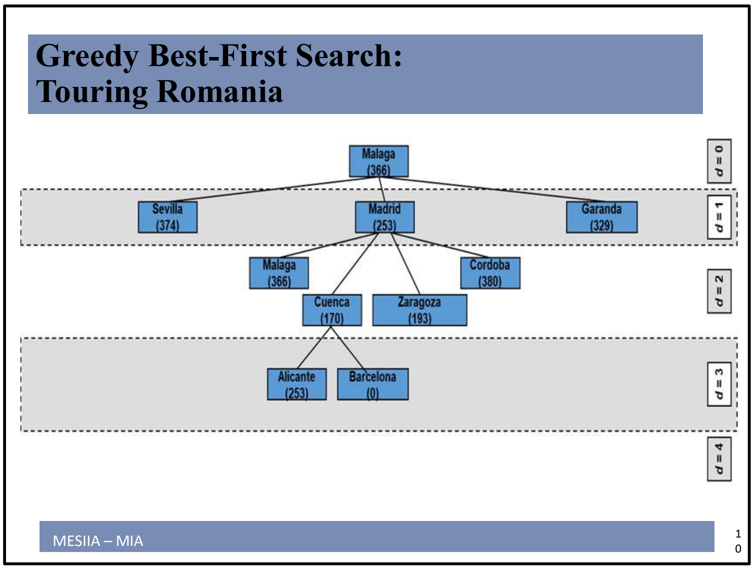
•distance to Barcelona because our goal is to be in Barcelona

**•[table]**

•$h_{SLD}$(Barcelona) = 0

•$h_{SLD}$(Zaragoza) = 176 < 211 driving distance

•$h_{SLD}(n)$ cannot be computed from the problem description, it represents additional information



## Greedy Best-First Search:
## Touring Romania

**Greedy Best-First Search: Touring Spain**

- •values are values of evaluation function = heuristic function

•select Malaga; expand Malaga
•select Madrid; expand Madrid

- •Cuenca has lowest *f*-value of all fringe nodes

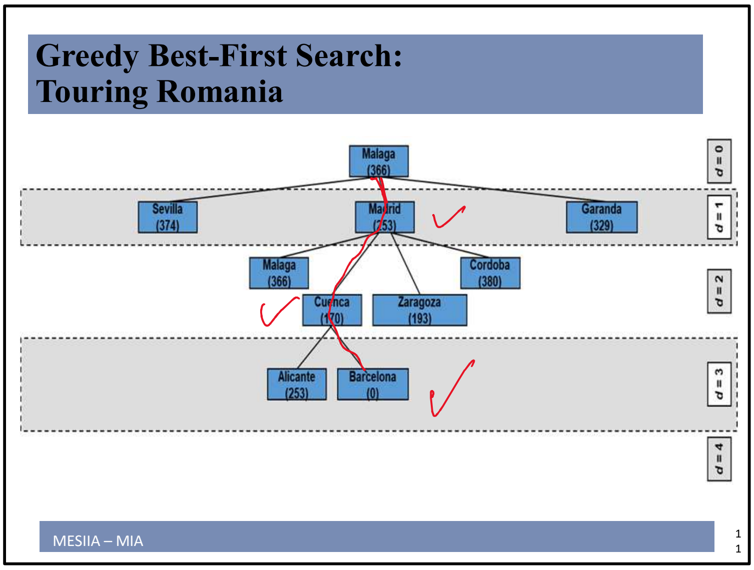
•select Cuenca; expand Cuenca
•select Barcleona – goal node

•for this problem: search proceeds straight to the goal node:

- •minimal search cost
- •but not the optimal path

•uniform-cost search vs. greedy best-first search: both expand node with lowest number:

- •UCS: numbers start from 0 and increase – tendency to expand earlier nodes – breadth-first tendency
- •GBFS: number start from high and decreases – tendency to expand later nodes – depth-first tendency



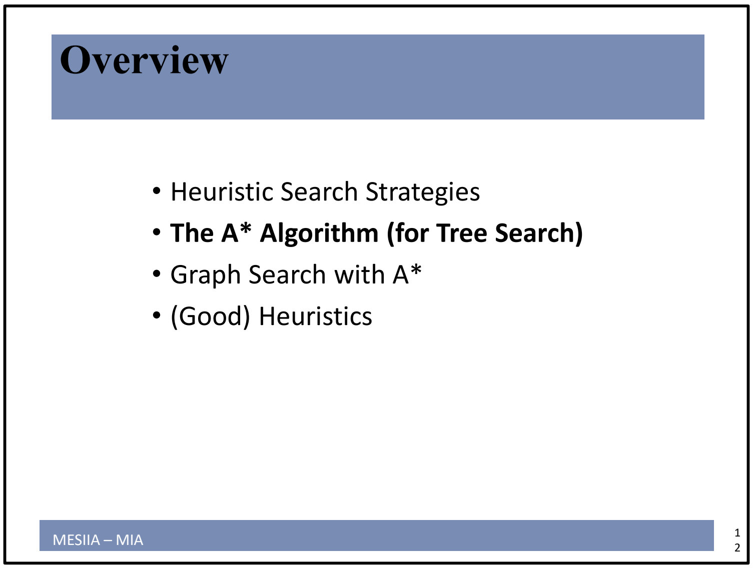
Greedy Best-First Search:
Touring Romania

MESIIA – MIA

11

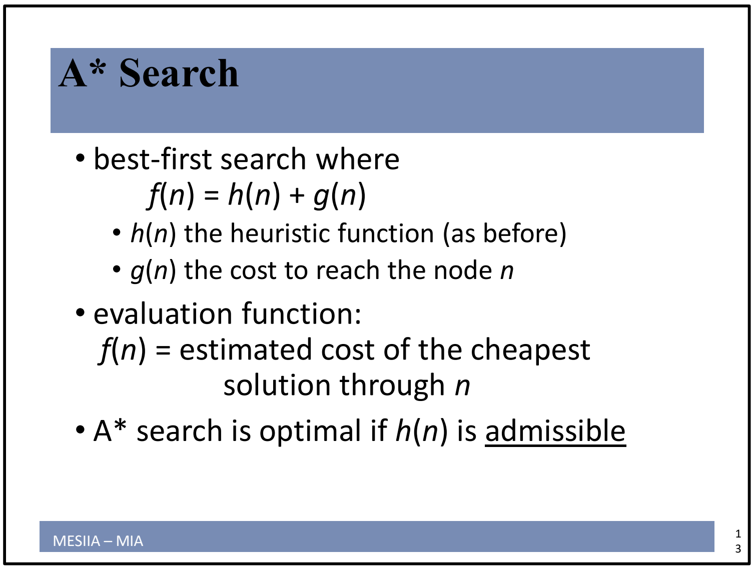
## Greedy Best-First Search: Touring Spain

- So the solution is marked in RED, by following the smallest value of heuristic vales.



# 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 <u>admissible</u>

**A* Search**

**•best-first search where *f*(*n*) = *h*(*n*) + *g*(*n*)**

**•*h*(*n*) the heuristic function (as before)**

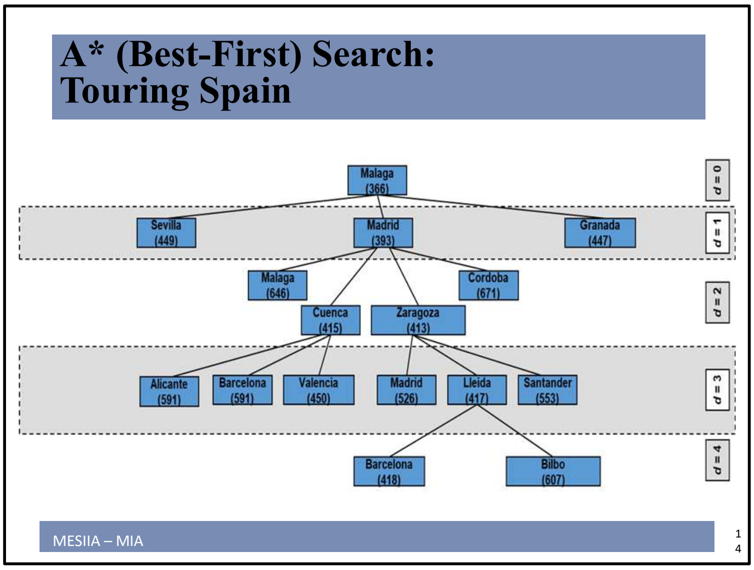
**•*g*(*n*) the cost to reach the node *n***

•adds a breadth-first component to GBFS

**•evaluation function: *f*(*n*) = estimated cost of the cheapest solution through *n***

•expand that node next which is on the cheapest path to a goal node

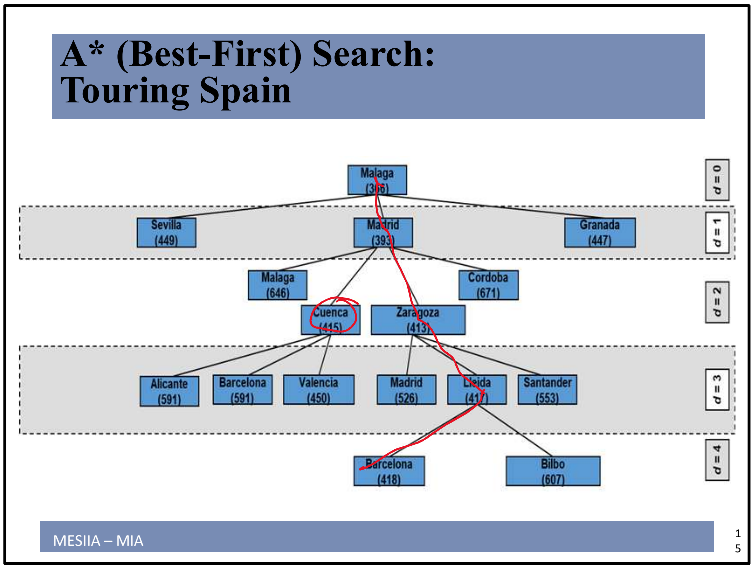
**•A* search is optimal if *h*(*n*) is <u>admissible</u>**



# A* (Best-First) Search: Touring Spain

**A* (Best-First) Search: Touring Spain**

**Now the value of each node includes h(n) + g(n)**



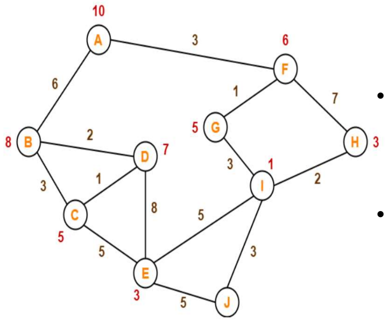
A* (Best-First) Search:
Touring Spain

## A* (Best-First) Search: Touring Spain

**The final solution is marked in RED**

# A* (Best-First) Search: Example

Consider the following graph-

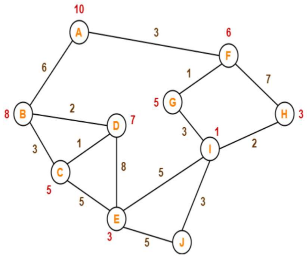


- The numbers written on edges represent the distance between the nodes.

- 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.

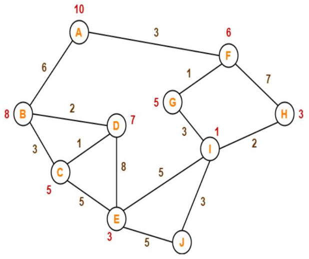
A* Algorithm calculates f(B) and f(F).

- 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.

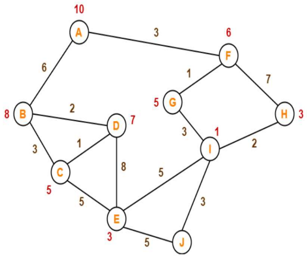
A* Algorithm calculates f(G) and f(H).

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



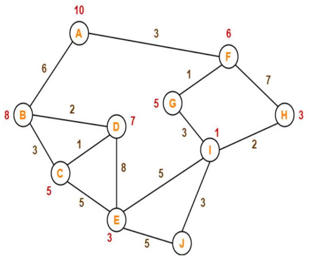
**Step-03:**

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

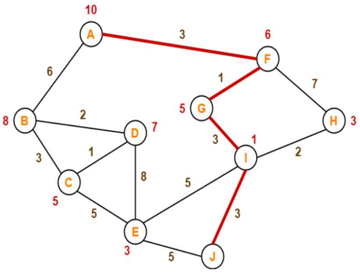
Since f(J) is least, so it decides to go to node J.

# A* (Best-First) Search: Example

**Final:.**

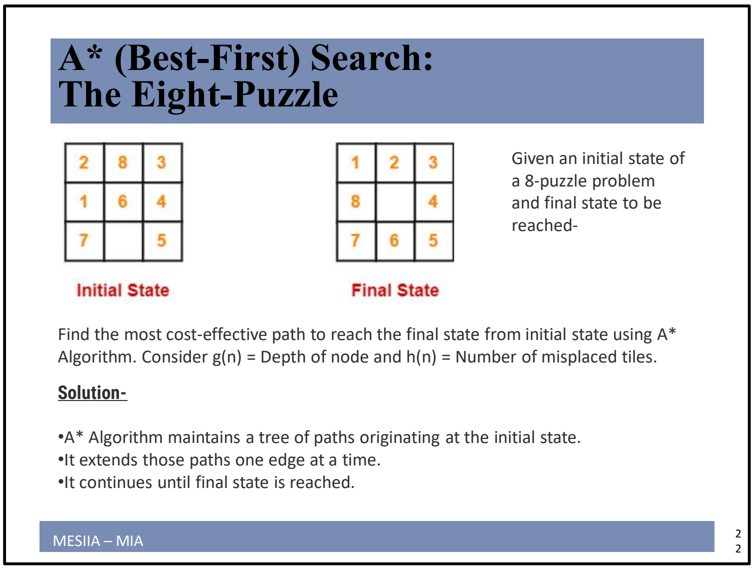
**Path- A → F → G → I → J**

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-**

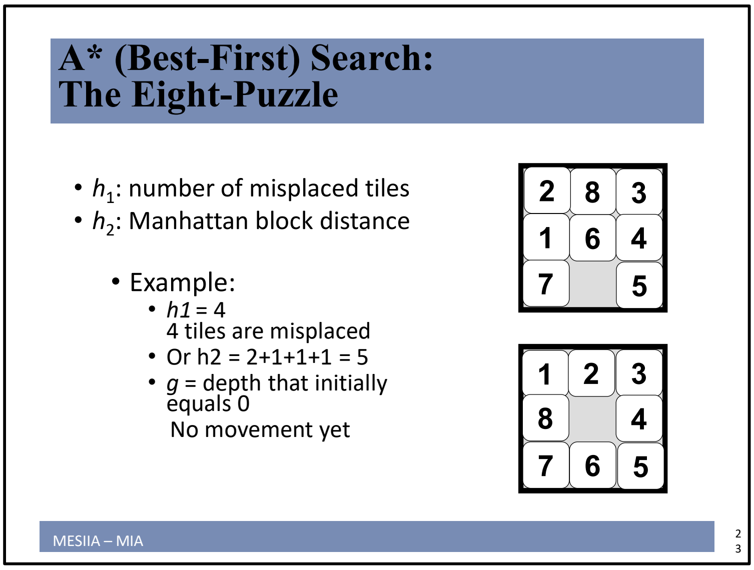
•A* Algorithm maintains a tree of paths originating at the initial state.
•It extends those paths one edge at a time.
•It continues until final state is reached.

The graph shown  in the pervious example is not very interesting, because it is a relatively small search space.

### The Eight-Puzzle

The 8-puzzle problem is **a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s**. It is played on a 3-by-3 grid with 8 square blocks labelled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in order.

The goal is to **rearrange the tiles so that they are in row-major order, using as few moves as possible**. You are permitted to slide tiles either horizontally or vertically into the blank square.



## 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

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| 7 | | 5 |

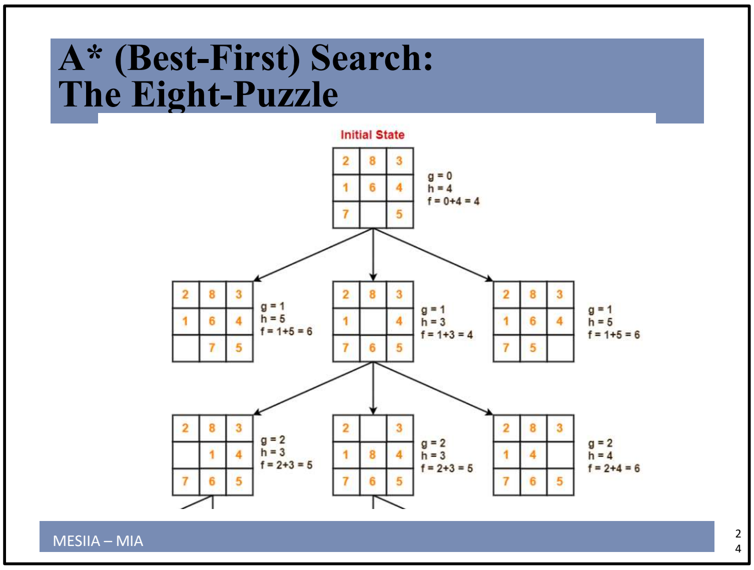
| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |

### Heuristics for the Eight-Puzzle

•Both heuristics are admissible;
•Cost of the optimal solution: 26; both heuristics underestimate;

•H function can be calculated by number of misplaced that is 4 tiles are misplaced (2, 8, 6, 1)
•H function also could be calculated based on the sum of the distance between the initial place of each number from its target place (e.g., number 1,2, and 6 need one movement to reach their locations in the goal state, 3, 4, 5, and 7 do not need any movement, and 8 needs two movements to reach its correct place. So g = 1+1+1+0+0+0+0+2 = 5.

• G function depend on the depth in the tree, (at the beginning is 0 as no movement).



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

Initial State

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| 7 | | 5 |

g = 0
h = 4
f = 0+4 = 4

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| | 7 | 5 |

g = 1
h = 5
f = 1+5 = 6

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | | 4 |
| 7 | 6 | 5 |

g = 1
h = 3
f = 1+3 = 4

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| 7 | 5 | |

g = 1
h = 5
f = 1+5 = 6

| | | |
|---|---|---|
| 2 | 8 | 3 |
| | 1 | 4 |
| 7 | 6 | 5 |

g = 2
h = 3
f = 2+3 = 5

| | | |
|---|---|---|
| 2 | | 3 |
| 1 | 8 | 4 |
| 7 | 6 | 5 |

g = 2
h = 3
f = 2+3 = 5

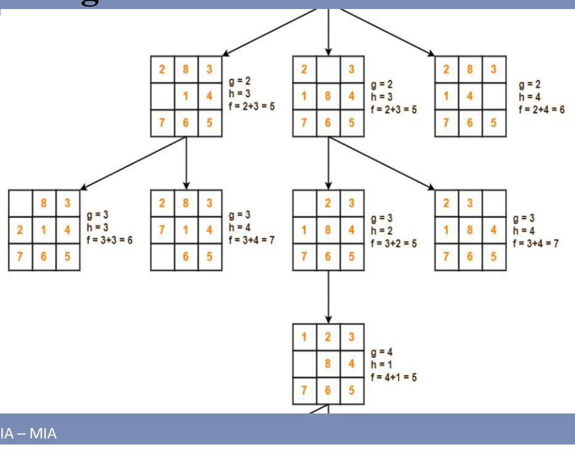
| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 4 | |
| 7 | 6 | 5 |

g = 2
h = 4
f = 2+4 = 6

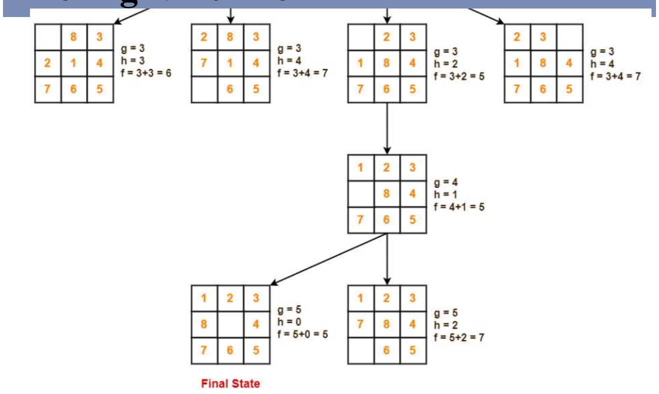
In the first step, you have three options:
1 – to move 5 to left, 2- to move 6 to down, and 3- to move 7 to right.

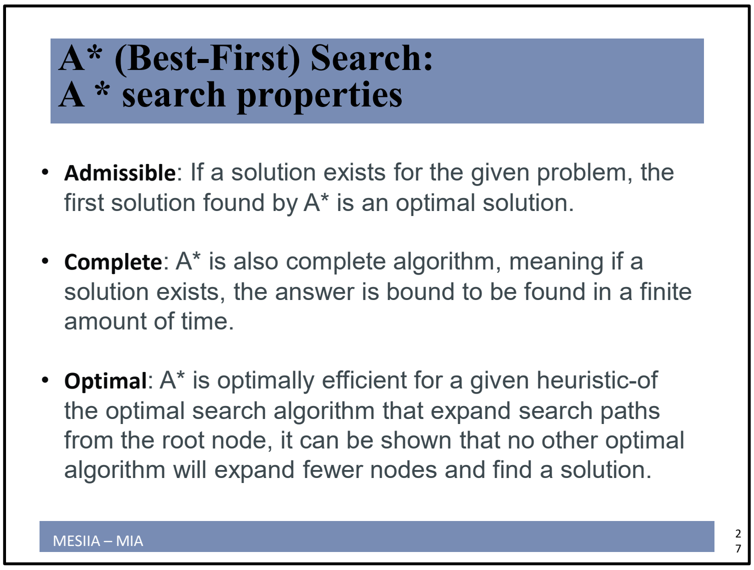
Then calculate function f = h + g to select the min value of f and so on.

# 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.

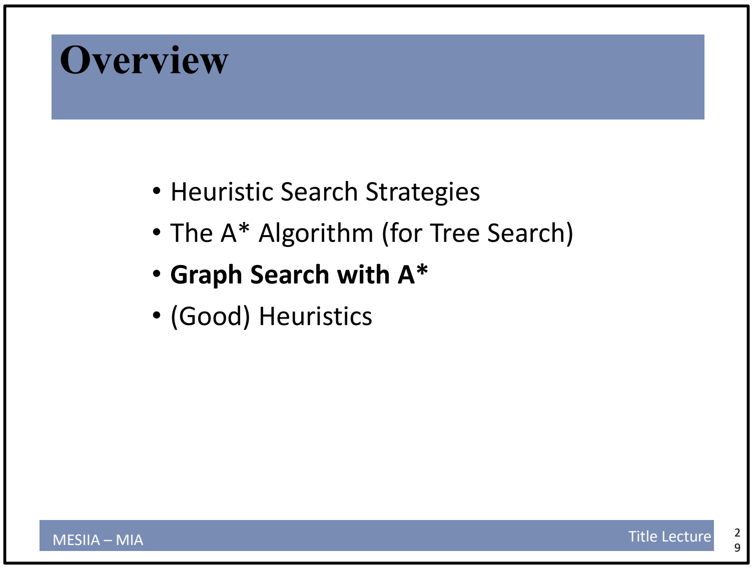
- **Optimal**: A* is optimally efficient for a given heuristic-of the optimal search algorithm that expand search paths from the root node, it can be shown that no other optimal algorithm will expand fewer nodes and find a solution.

In computer science, specifically in algorithms related to pathfinding, a heuristic function is said to be admissible if it never overestimates the cost of reaching the goal, i.e. the cost it estimates to reach the goal is not higher than the lowest possible cost from the current point in the path.

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

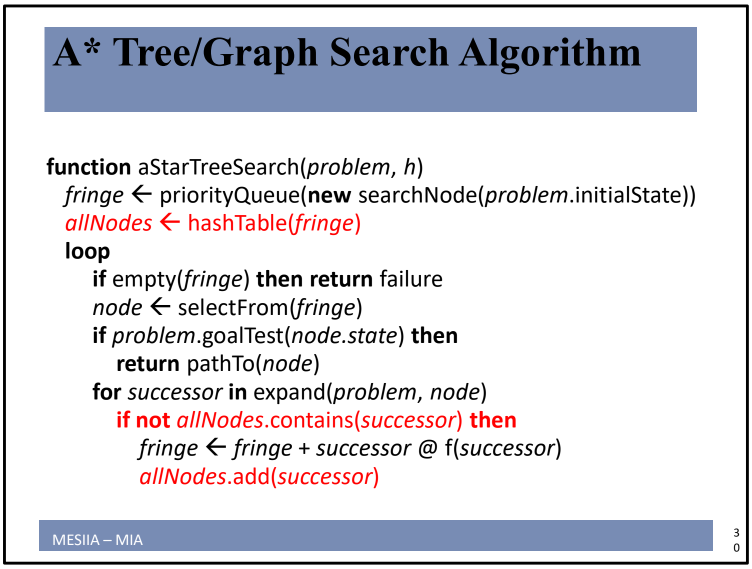
- **Advantages**:
  - It is optimal search algorithm in terms of heuristics.
  - It is one of the best heuristic search techniques in AI.
  - It is used to solve complex search problems.
  - There is no other optimal algorithm guaranteed to expand fewer nodes than A*.

- **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



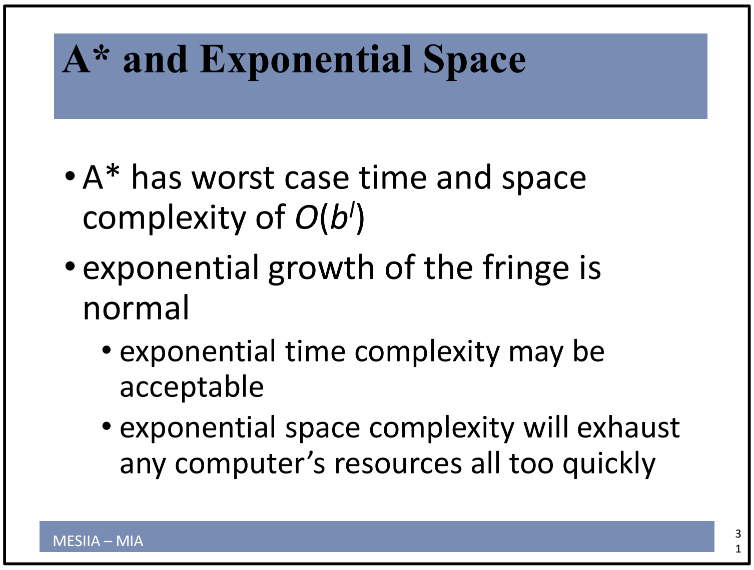
## A* Tree/Graph Search Algorithm

```
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* Tree/Graph Search Algorithm

•steps in red for graph search
•Steps in black for tree search
•The only difference between tree search and graph search is that tree search does not need to store the explored set, because we are guaranteed never to attempt to visit the same state twice.
•also needed: detecting short-cuts if heuristic is not admissible

* Note: the graph does not guarantee that will generate an optimal solution, the reason for that occurs in the line of (***if not allNodes.contains(successor) then).*** Namely if we generate the successors of a node, we may generate a new node that we have seen before. But the second time we generate the node, we actually discover a shorter path than the first time we generated it. So we need to add a small condition to check if the path is shorter or not. So it will generate an optimal solution.



# 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

**A* and Exponential Space**

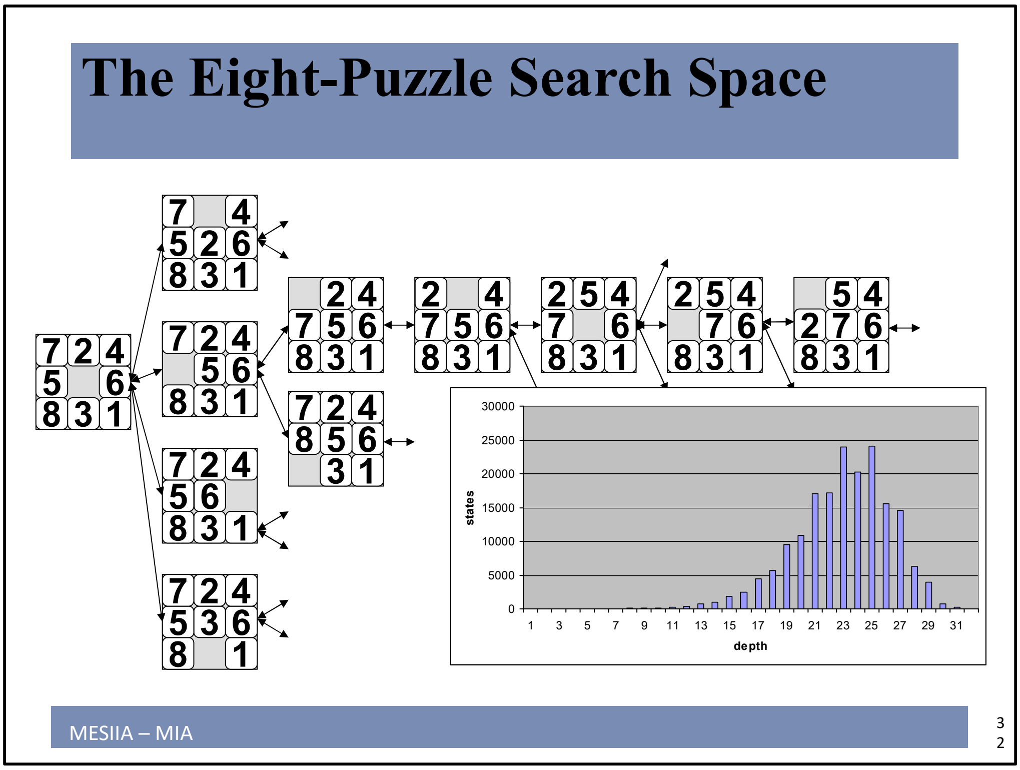
•**A* has worst case time and space complexity of $O(b^l)$,** where b is the number of successes we have per node on average and l is the length of the path we are looking for.

•**exponential growth of the fringe is normal**

  •**exponential time complexity may be acceptable**

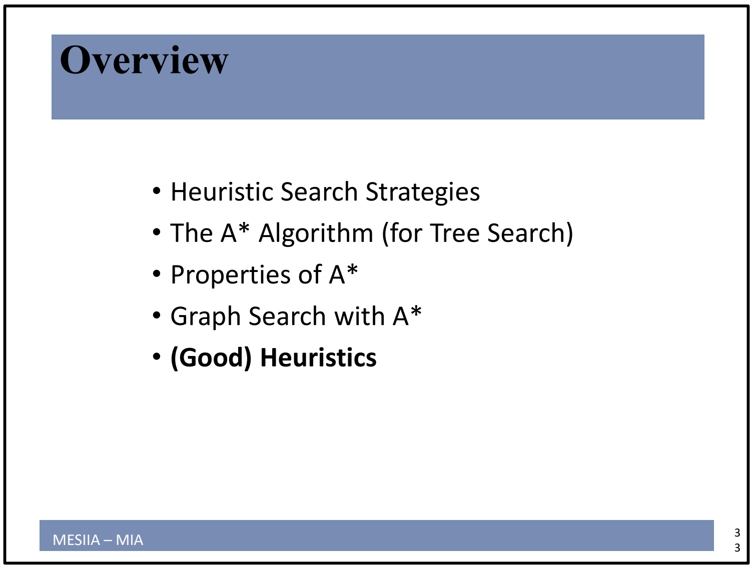
  •**exponential space complexity will exhaust any computer’s resources all too quickly**

    •and with the memory exhausted A* cannot continue and fails – no solution will be found



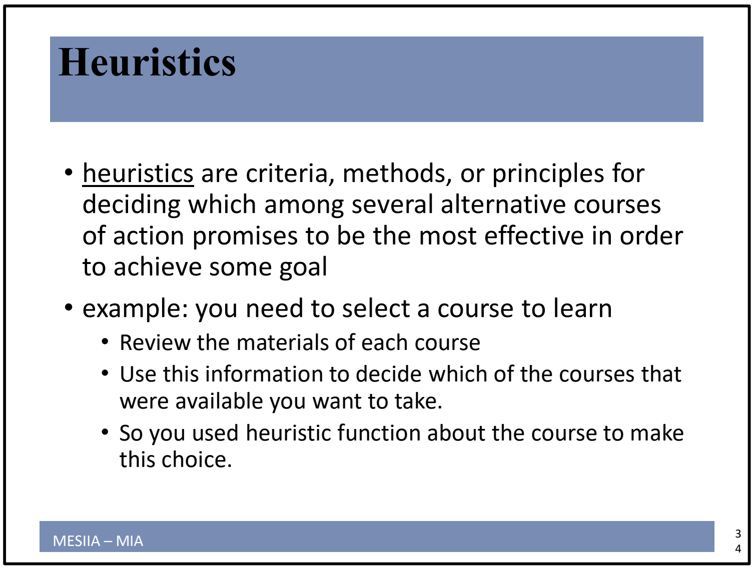
# The Eight-Puzzle Search Space

## 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

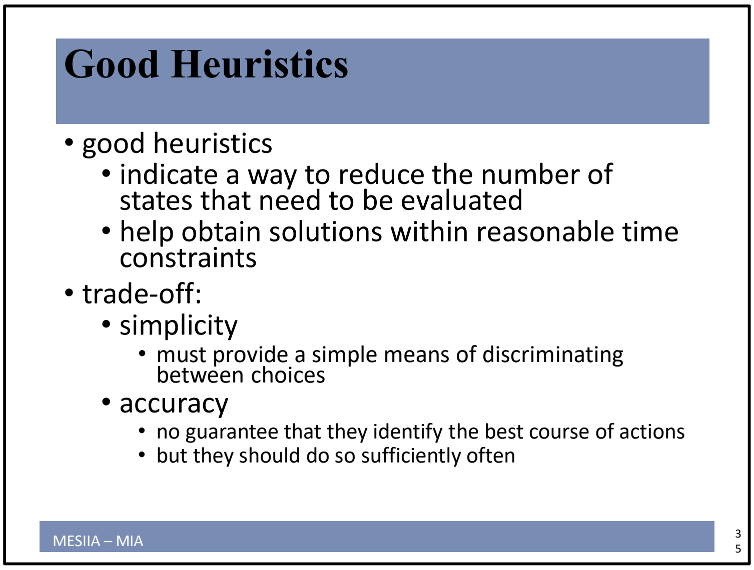
- heuristics are criteria, methods, or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal
- example: you need to select a course to learn
  - Review the materials of each course
  - Use this information to decide which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

**Heuristics**

•Heuristic (colloquial): **a rule of thumb**;

•Heuristic (general term) vs. heuristic function (technical term) $h(n)$ = estimated cost of the cheapest path from node $n$ to a goal node;

•An example: what made you (the student) choose this course over others?



## Good Heuristics

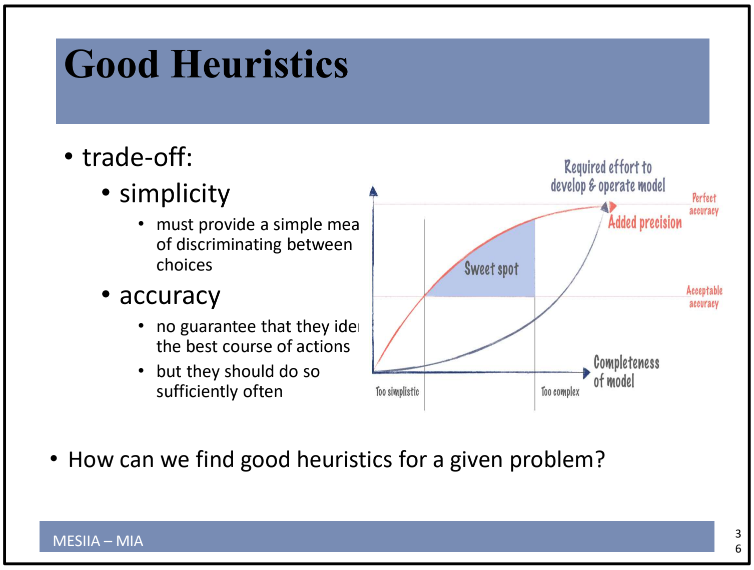
- good heuristics
  - indicate a way to reduce the number of states that need to be evaluated
  - help obtain solutions within reasonable time constraints
- trade-off:
  - simplicity
    - must provide a simple means of discriminating between choices
  - accuracy
    - no guarantee that they identify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

**Good Heuristics**

•Simplicity: easy to compute;

•Accuracy: should result in the best course of actions;

•The question is how can we find good heuristics for a given problem? And can this process be automated?



## Good Heuristics

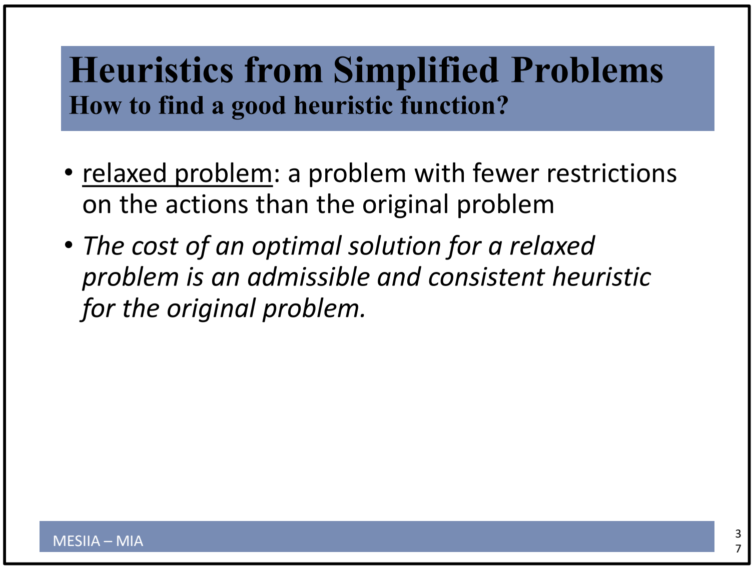
- trade-off:
  - simplicity
    - must provide a simple mea of discriminating between choices
  - accuracy
    - no guarantee that they ide the best course of actions
    - but they should do so sufficiently often

Required effort to develop & operate model
Perfect accuracy
Added precision
Sweet spot
Acceptable accuracy
Completeness of model
Too simplistic
Too complex

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

**Good Heuristics**
•Simplicity: easy to compute;
•Accuracy: should result in the best course of actions;

•The question is how can we find good heuristics for a given problem? And can this process be automated?



## 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.*

**Heuristics from Simplified Problems**

•Admissibility:

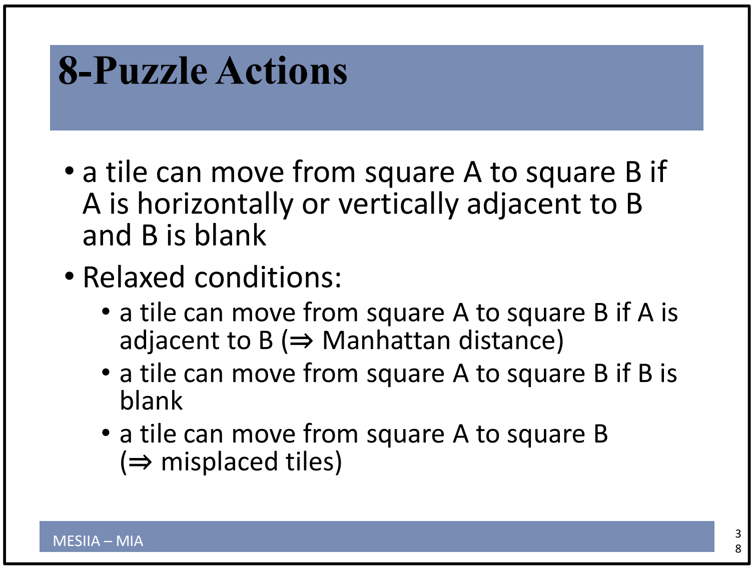
- •Optimal solution to original problem is also solution to relaxed problem;
- •Therefore: optimal solution to original problem at least as expensive as solution to relaxed problem;

•Consistency:

- •Because derived heuristic is exact cost for relaxed problem (finds “short cuts”) the triangle inequality must hold;

•ABSOLVER: program that generates heuristics based on the relaxed problem method;

- •Found best heuristic for 8-puzzle and first useful heuristic for Rubik’s cube;

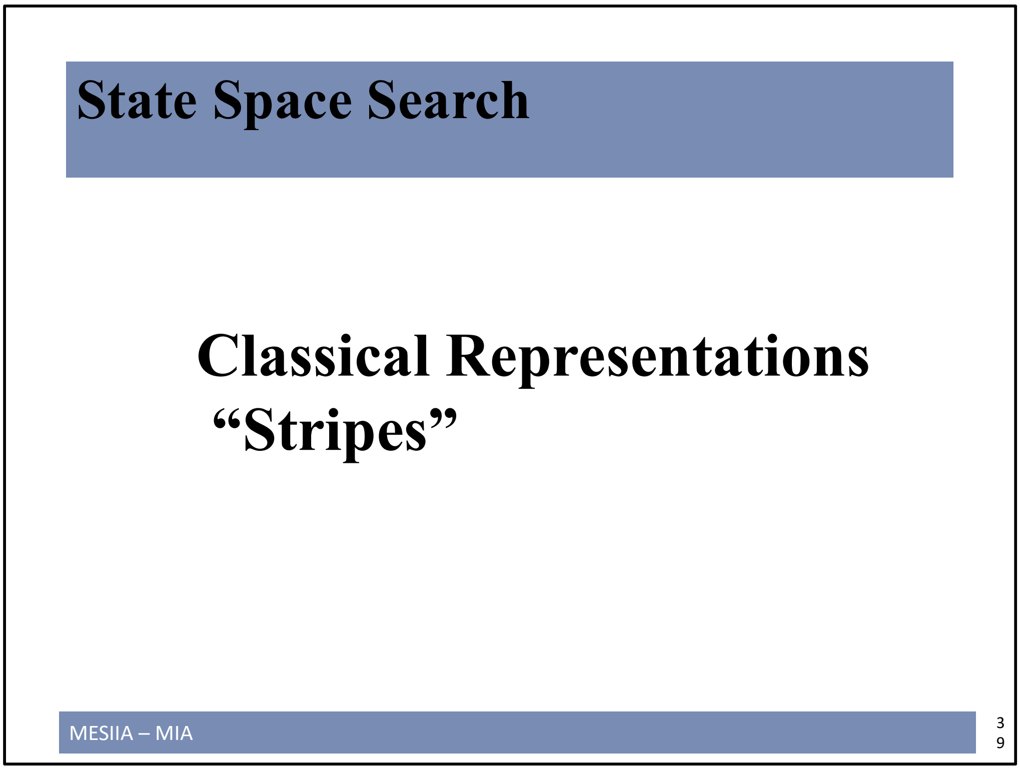


## 8-Puzzle Actions

- 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)

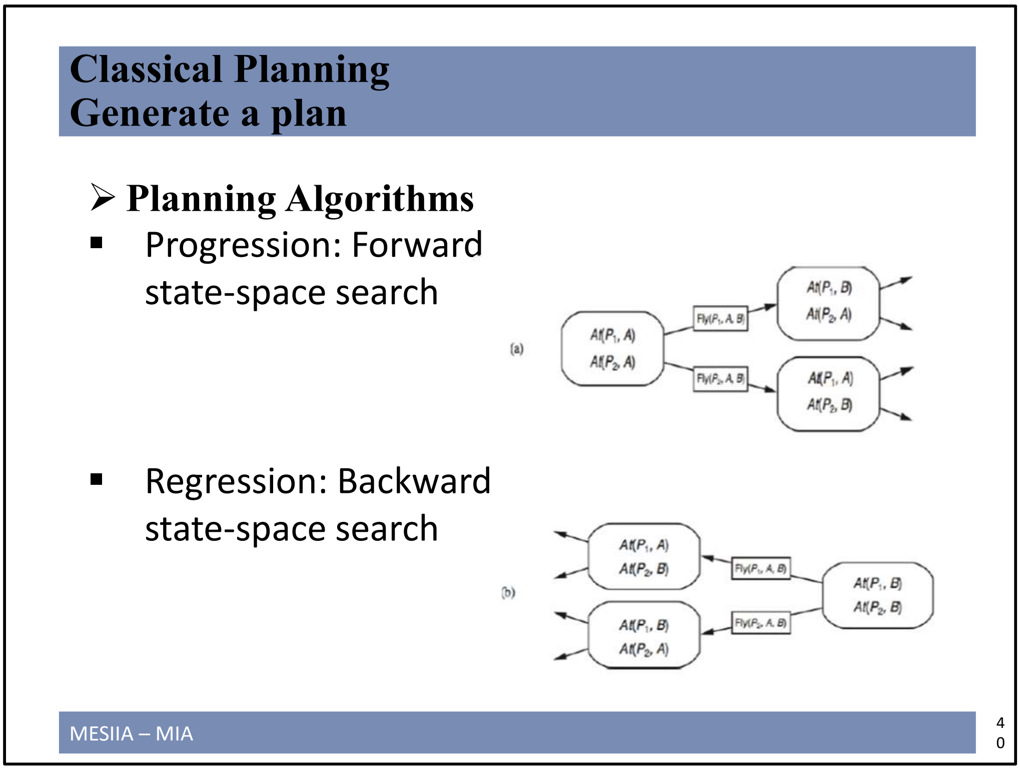
**8-Puzzle Actions**

•Heuristics are estimates for costs of actions still expected, hence modify constraints on actions



# State Space Search

## Classical Representations “Stripes”



## Classical Planning
## Generate a plan

➤ **Planning Algorithms**

- Progression: Forward state-space search
- Regression: Backward state-space search

MESIIA – MIA

40

The goal of generate a planner is to generate a plan that can be able to transform from the initial state to the goal state.
There are two types of search: one is based state-space and plan-space search

**In classical planning, we will start with state-space planning.**
With *state-space* planning, we generate plans by searching through state space.

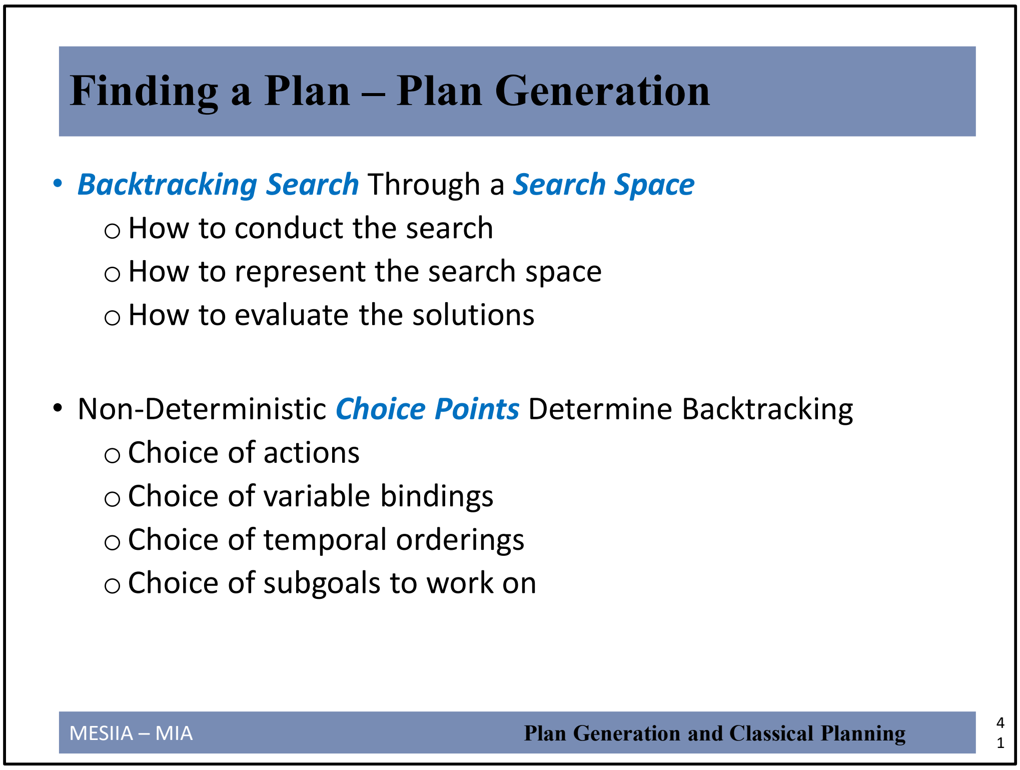
**Searching for plans**

The basic idea is to apply standard search algorithms (e.g., bread-first, depth-first, A*, etc.) to the planning problem.
**search space is a subset of the state space**
nodes correspond to world states
arcs correspond to state transitions
path in the search space corresponds to plan

**Forward search** starts from the initial state to the goal state.
It is *sound* (if a plan is returned, it will indeed be a solution) and it is *complete* (if a solution exists, it will be found).

**Backward Search**
Alternatively, we can search backwards from a goal state to the initial state.

See the example shown in the figure, we have two packages that are in location A, and we aim to transfer them to Location B, the plan could be achieve in forward or backward.



## Finding a Plan – Plan Generation

- ***Backtracking Search*** Through a ***Search Space***
  - ○ How to conduct the search
  - ○ How to represent the search space
  - ○ How to evaluate the solutions

- Non-Deterministic ***Choice Points*** Determine Backtracking
  - ○ Choice of actions
  - ○ Choice of variable bindings
  - ○ Choice of temporal orderings
  - ○ Choice of subgoals to work on

MESIIA – MIA **Plan Generation and Classical Planning** 41

- There are three main algorithmic techniques for solving constraint satisfaction problems (CSP) (e.g., Find plans which has clear and well-defined constraints on any objective solution): **backtracking search, local search, and dynamic programming**.

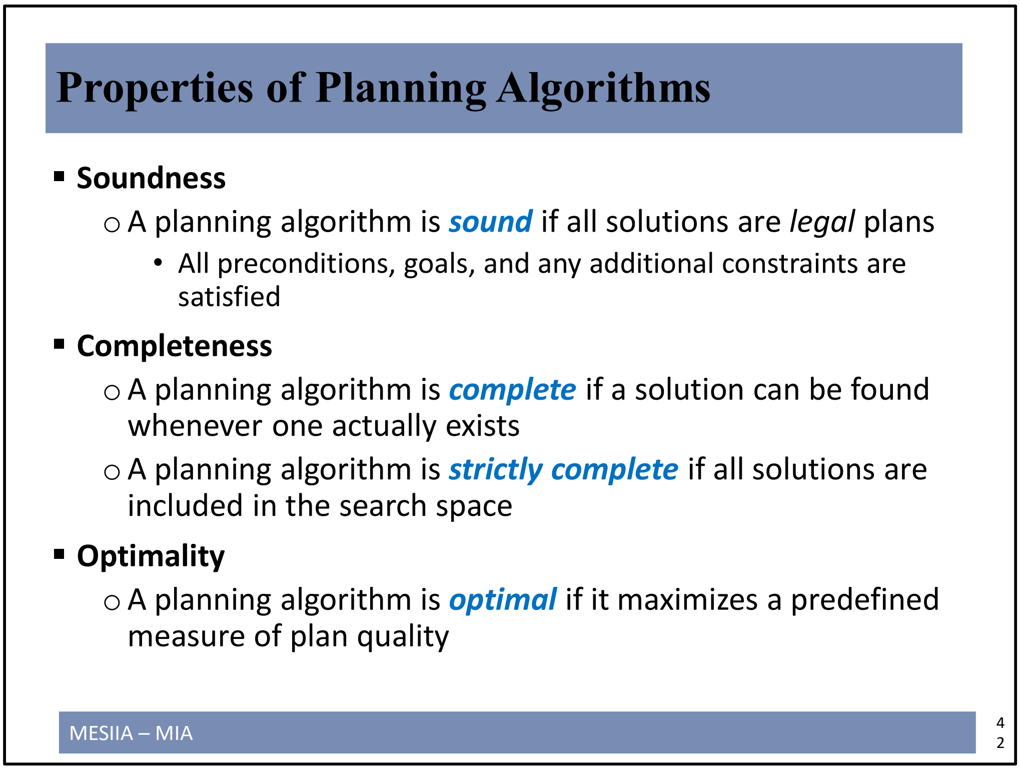
- An algorithm for solving (CSP) can be either complete or incomplete, e.g., **Local searching could be incomplete or partially complete, but backtracking search, and dynamic programming are complete algorithms.**
  - ○ **Complete Planning Solutions mean that the solution will facilitate your development, bring all the loose ends together, and ensuring your goals are achieved.**

- **What is the difference between Backtracking search vs. dynamic programming?**

  —backtracking search algorithms are currently **the most important in practice**.
  —backtracking search algorithms **work on only one solution at a time** and thus need only **a polynomial amount of space** (i.e., polynomial space is a way of characterizing the complexity of an algorithm). You usually need backtracking to process a non-deterministic algorithm, since (by the definition of non-deterministic) you don't know which path to take at a particular time in your processing, but instead you must try several.
  —Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.
  —In contrast, Dynamic programming is called Greedy algorithms.
  —**The drawbacks of dynamic programming** approaches are that they often require **an exponential amount of time and space**, and they do unnecessary work by finding, or making it possible to easily generate, all solutions to a CSP.

- **What is the relation between non-deterministic actions and backtracking? Here we need to define non-deterministic algorithms.**

—For instance, a **nondeterministic** algorithm is which can specify, at certain points in the algorithm (called "**choice points**"),
—Various alternatives for the algorithm flow. Unlike an if-then statement, the method of choice between these alternatives is not directly specified by the algorithm’ programmer; the algorithm must decide at runtime between the alternatives, via some general method applied to all choice points.
—A programmer specifies a limited number of alternatives, but the algorithm must later choose between them. ("Choose" is, in fact, a typical name for the nondeterministic operator.)
—A hierarchy of choice points may be formed, with higher-level choices leading to branches that contain lower-level choices within them.
—One method of choice is embodied in backtracking systems, in which some alternatives may "fail", causing the algorithm to backtrack and try other alternatives. If all alternatives fail at a particular choice point, then an entire branch fails, and the algorithm will backtrack further, to an older choice point.
—One complication is that, because any choice is tentative and may be remade, the system must be able to restore old program states by undoing side-effects caused by partially executing a branch that eventually failed.



## 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

MESIIA – MIA 42

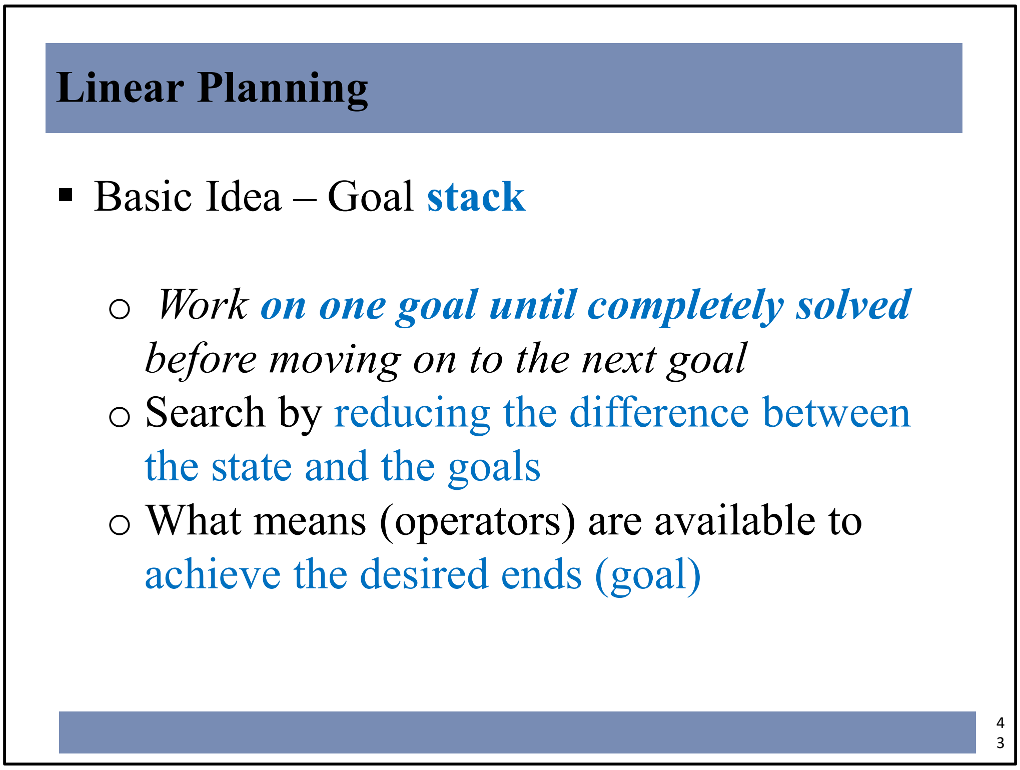
- To say this planner is acceptable or not?
- Three main **Properties of Planning Algorithms** are that planner should be **sound, complete and optimal**.

- Optimality, this expression can be complicated. For instance, we may have many factors affecting optimality of a subway trip (stations with escalators, crowds on trains, etc.). To find the “Best”, something is very complex.

- An algorithm is **sound** if, anytime it returns an answer, that answer is true." means the same as "Basically, soundness (of an algorithm) means that **the algorithm doesn't yield any results that are untrue**.“

- An algorithm is **complete** if it guarantees to return a correct answer for any arbitrary input (or, if no answer exists, **it guarantees to return failure**).

- An algorithm can be said to be **optimal** if the function that describes its time complexity in the worst case is a lower bound of the function that describes the time complexity in the worst case of a problem that the algorithm in question solves. “ **asymptotically solve the problem in less time**”.

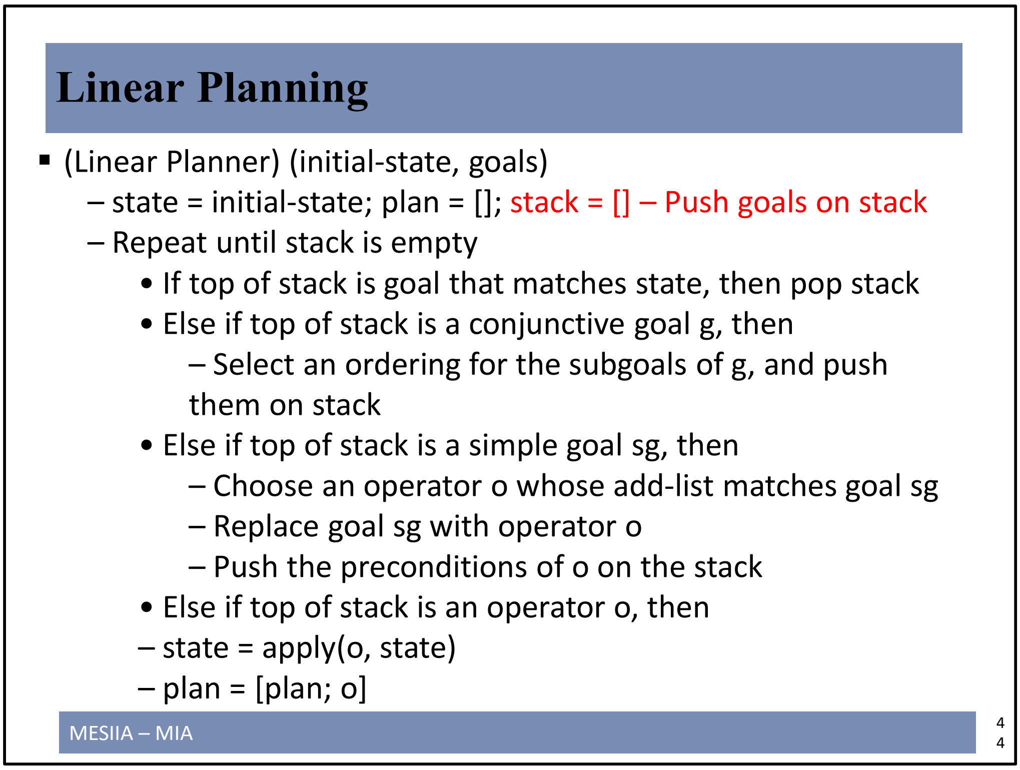


## Linear Planning

- 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)

- The approximation of a solution of any problem is a linear solution. The Liner planning system is a prototype implementation of the linear backward chaining (LBC) planning algorithm, which is based on the linear connection method.
- A linear planner is one in which the order in which sub-problems are solved is linearly related to the order in which the actions of the plan are executed. Examples of linear planner as we said in the first lecture is STRIPS (Means-Ends analysis).
- Means-Ends reasoning
  - Means-end reasoning is concerned with finding the means for achieving goals. (actions = means, goals = ends).
  - The basic idea is a simple one: to achieve a goal, we consider an action that would achieve it under some specified circumstances (conditions) and then try to find a way of putting ourselves in those circumstances (conditions) in order to achieve the goal by performing the action.
  - Putting ourselves in those circumstances becomes a subgoal.
  - The idea is to work backward from the goal through subgoals until we arrive at subgoals that are already achieved.
  - The resulting sequence of actions constitutes a plan for achieving the goal.

- A precise logical theory of plan-construction is formulated that completely characterizes means-end reasoning.



## 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]

MESIIA – MIA

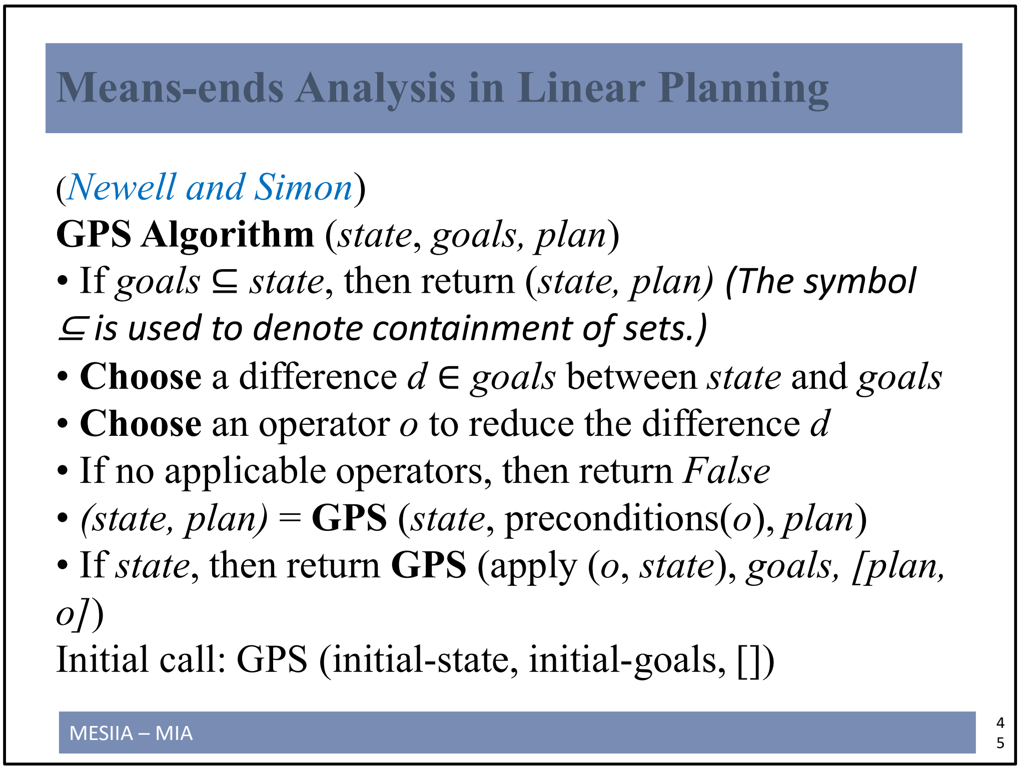
4
4

- The algorithm of the classical linear planner

- STRIPS (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, [])

MESIIA – MIA

45

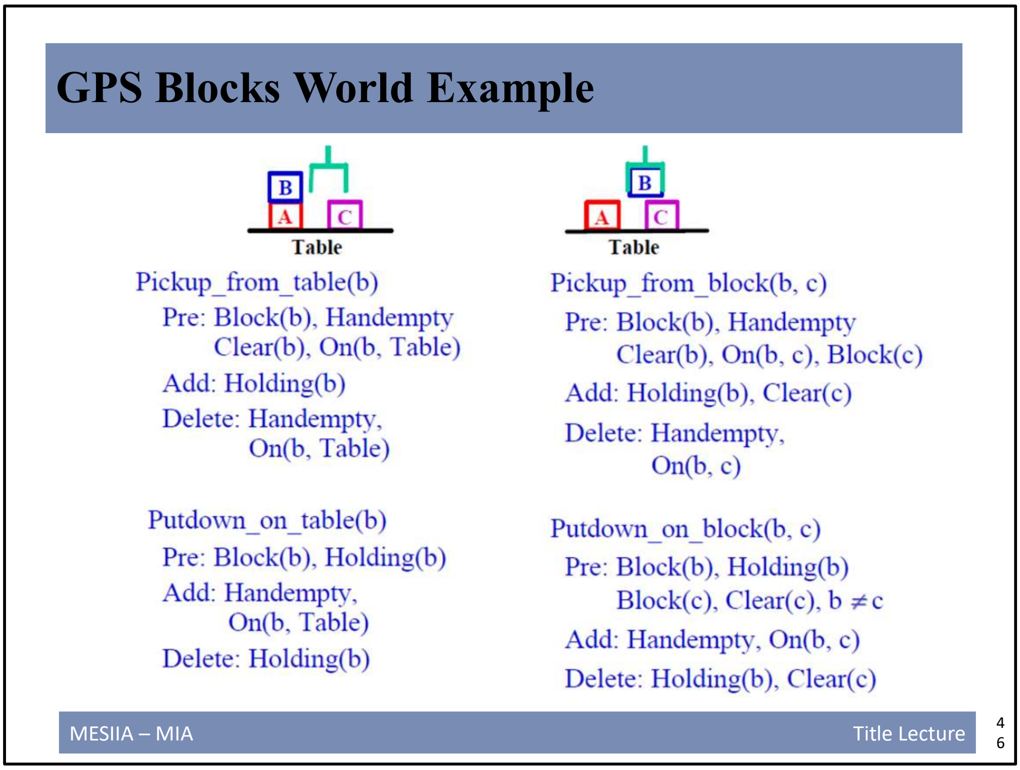
- **G**eneral **P**roblem **S**olver (GPS) proposed by Newell & Simon, CMU.
    - ❖ The General Problem Solver (GPS) was a theory of human problem solving stated in the form of a simulation program (Ernst & Newell, 1969; Newell & Simon, 1972). This program and the associated theoretical framework had a significant impact on the subsequent direction of cognitive psychology. It also introduced the use of productions as a method for specifying cognitive models.
    - ❖ The theoretical framework was information processing and attempted to explain all behaviour as a function of memory operations, control processes and rules. The methodology for testing the theory involved developing a computer simulation and then comparing the results of the simulation with human behaviour in a given task. Such comparisons also made use of protocol analysis (Ericsson & Simon, 1984) in which the verbal reports of a person solving a task are used as indicators of cognitive processes.
    - ❖ GPS was intended to provide a core set of processes that could be used to solve a variety of different types of problems.
    - ❖ The critical step in solving a problem with GPS is the definition of the problem space in terms of the goal to be achieved and the transformation rules (actions).
    - ❖ Using a means-end-analysis approach, GPS would divide the overall goal into subgoals and attempt to solve each of those.
    - ❖ Some of the basic solution rules include:
        - ▪ (1) transform one state into another,
        - ▪ (2) reduce the different between two states, and
        - ▪ (3) apply an operator to an state.

- One of the key elements need by GPS to solve problems was an operator-difference table that specified what transformations were possible.

- The MEA technique is a strategy to control search in problem-solving. Given a current state and a goal state, an action is chosen which will reduce the difference between the two. The action is performed on the current state to produce a new state, and the process is recursively applied to this new state and the goal state.

- Note that, in order for MEA to be effective, the goal-seeking system must have a means of associating to any kind of detectable difference those actions that are relevant to reducing that difference. It must also have means for detecting the progress it is making (the changes in the differences between the actual and the desired state), as some attempted sequences of actions may fail and, hence, some alternate sequences may be tried.

- When knowledge is available concerning the importance of differences, the most important difference is selected first to further improve the average performance of MEA over other brute-force search strategies. However, even without the ordering of differences according to importance, MEA improves over other search heuristics (again in the average case) by focusing the problem solving on the actual differences between the current state and that of the goal.



## GPS Blocks World Example

B
A
C
Table
B
A
C
Table

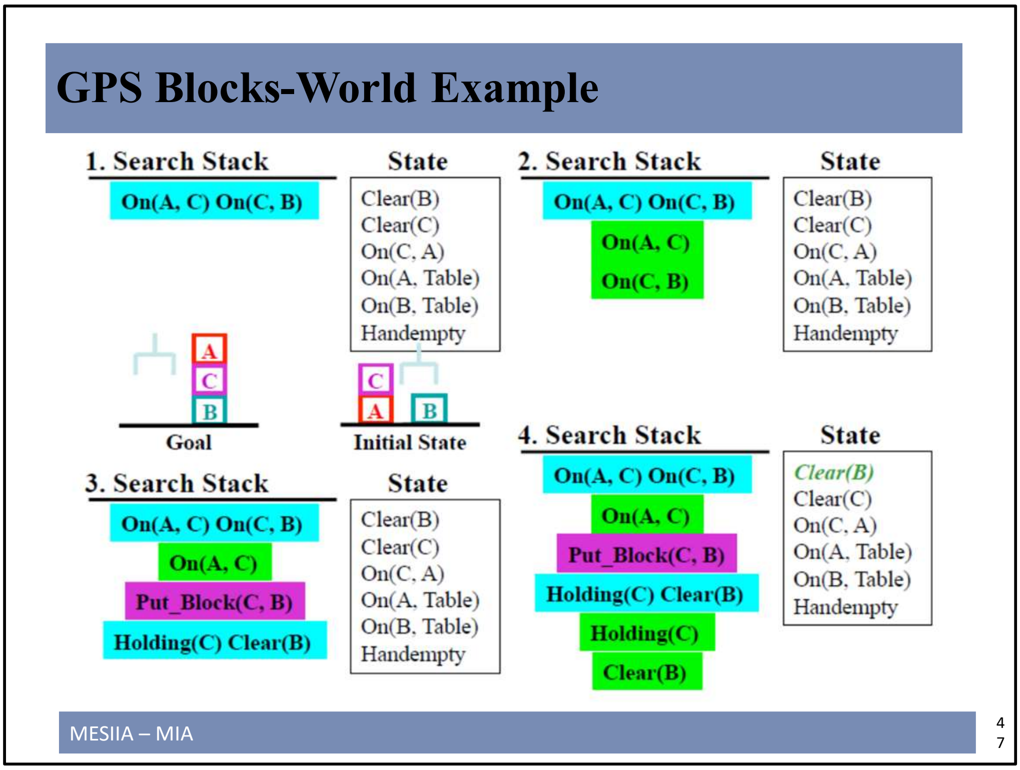
Pickup_from_table(b)
Pre: Block(b), Handempty
Clear(b), On(b, Table)
Add: Holding(b)
Delete: Handempty,
On(b, Table)

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)

- As you see in the blocks world problem, to have the final goal that indicates that C on B and finally A on C.
- As clear, we have two subgoals: 1) C on B and 2) A on C.
- We try to solve the two sub-problems.
- Firstly, C on B (be sure the top of B is empty and then hold C by the arm and finally put C on B.
- Here’s an illustration of the challenge– we have the two subgoals– C on B and A on C.
- This makes it harder to get a good heuristic, as we did in search.



## 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

**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)

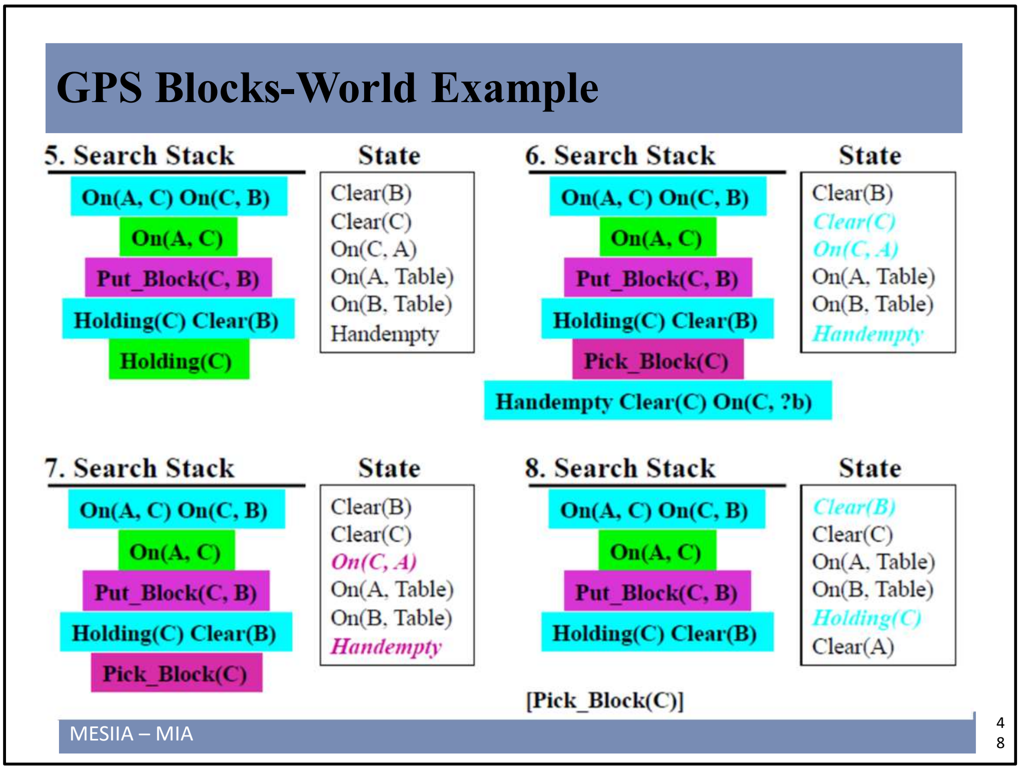
**State**

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

MESIIA – MIA

47

- In the figures,
  1) Cyan: Goal.
  2) Green: Difference between init and goals,
  3) We took all available actions **(Put_on_Block, Pick_from_Block, Put_on_Table, Pick_from_Table)** with On(C,B) as postcondition and picked one (in magenta). Now we push the precondition of Put_Block(C,B) and add it to stack as a subgoal.
  4) We're on state = GPS(init, precond(Put_Block(C,B)) Compute the differences– add the two greens to the stack (**Holding(C) and Clear(B)**).



## GPS Blocks-World Example

**5. Search Stack**

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

**State**

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

**6. Search Stack**

On(A, C) On(C, B)
On(A, C)
Put_Block(C, B)
Holding(C) Clear(B)
Pick_Block(C)
Handempty Clear(C) On(C, ?b)

**State**

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

**7. Search Stack**

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

**State**

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

**8. Search Stack**

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

[Pick_Block(C)]

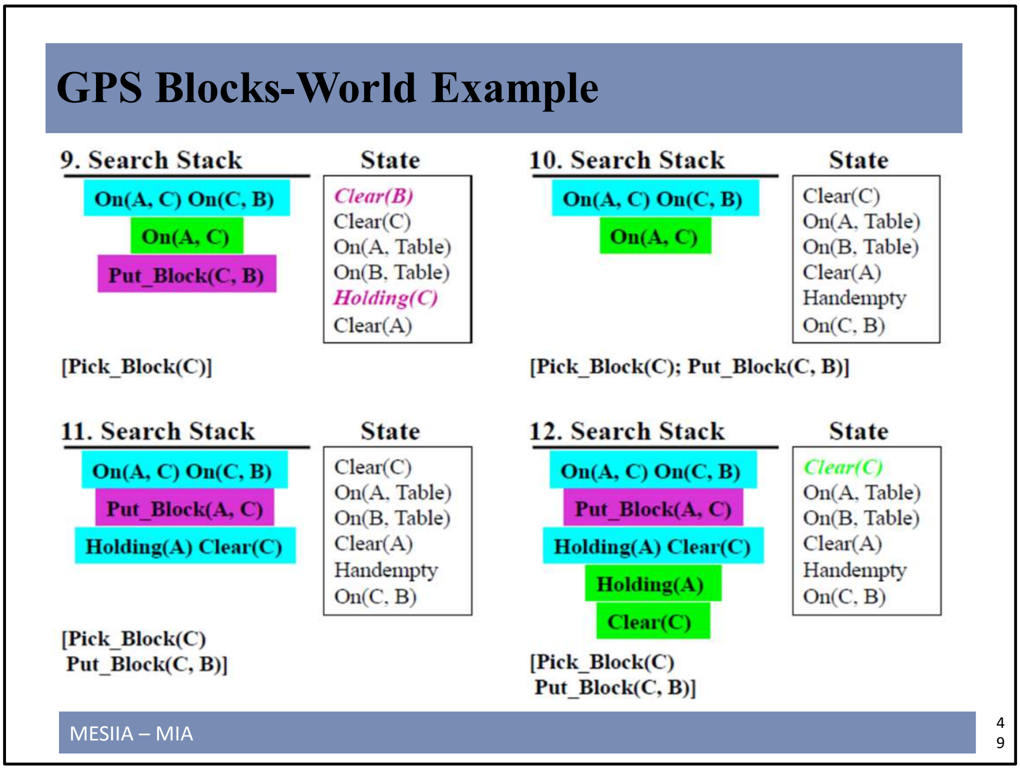
**State**

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

MESIIA – MIA

48

5) Now take Clear(B) from the stack since it’s in the state already. Pick_Block(C) is the next operator,
6) Call GPS with cyan as new goals.
7) All of Handempty, Clear(C), and Clear (C)  is in the state, so return true. Apply PB(C), and
8) Now call GPS(PB(C)(init-state), current goals). Where current goals are the top of the stack– holding(C) and clear(B). These are already in, return true again.



## GPS Blocks-World Example

**9. Search Stack**

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

**State**

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

[Pick_Block(C)]

**10. Search Stack**

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

**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**

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

**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**

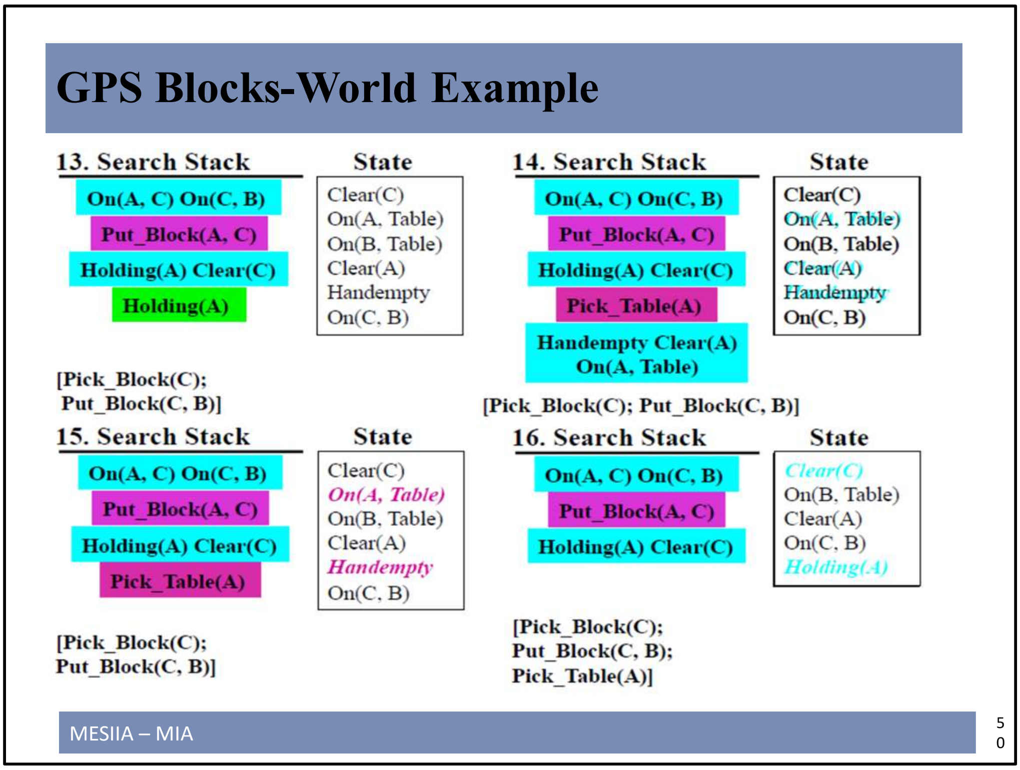
On(A, C) On(C, B)
Put_Block(A, C)
Holding(A) Clear(C)
Holding(A)
Clear(C)

**State**

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

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

9) Pick_Block(C)
10) Pick_Block(C); Put_Block(C, B)
11) Pick_Block(C) Put_Block(C, B)
12) 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)

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

State

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

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

State

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

15. Search Stack

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

Put_Block(A, C)

Holding(A) Clear(C)

Pick_Table(A)

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

State

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

16. Search Stack

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

Put_Block(A, C)

Holding(A) Clear(C)

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

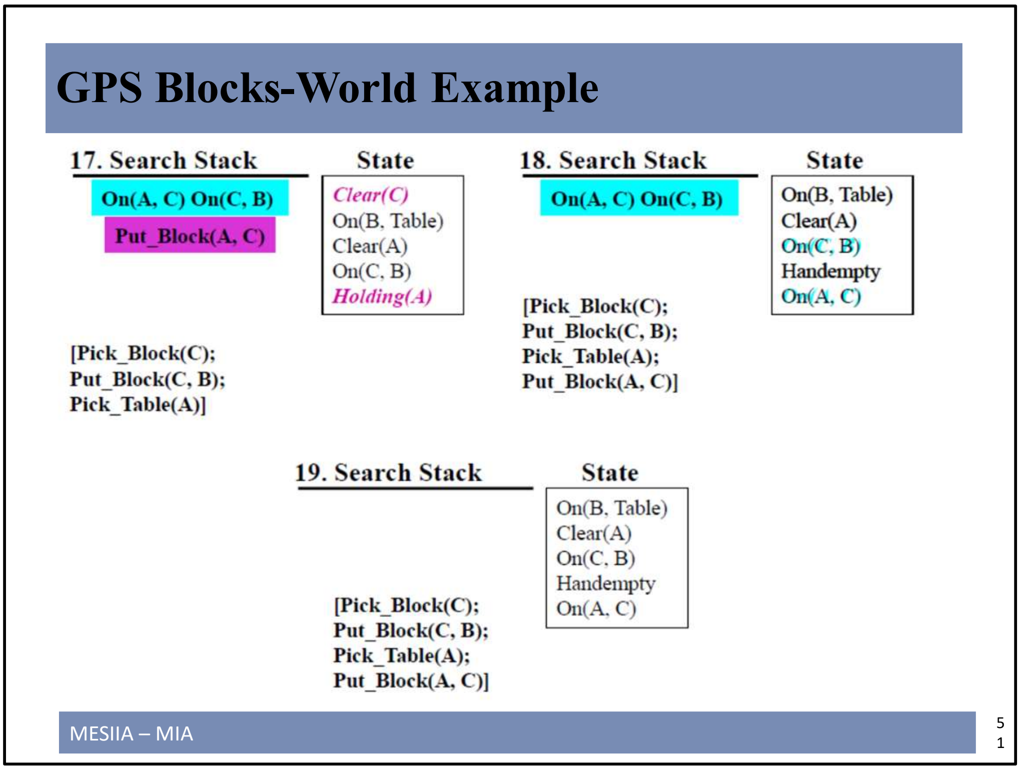
State

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

MESIIA – MIA

50

13) Now call GPS on the action, so our new goal diff is Holding(A),
14) Recurse back, try to get to holding(A),
15) Preconds to holding(A) were fulfilled already, so add that to the actions, and
16) Add PT(A) to the actions, and move down to the next goals on the stack. Oh look, they’re already in our state.



# GPS Blocks-World Example

17. Search Stack

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

Put_Block(A, C)

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

State

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

18. Search Stack

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

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

State

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

19. Search Stack

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

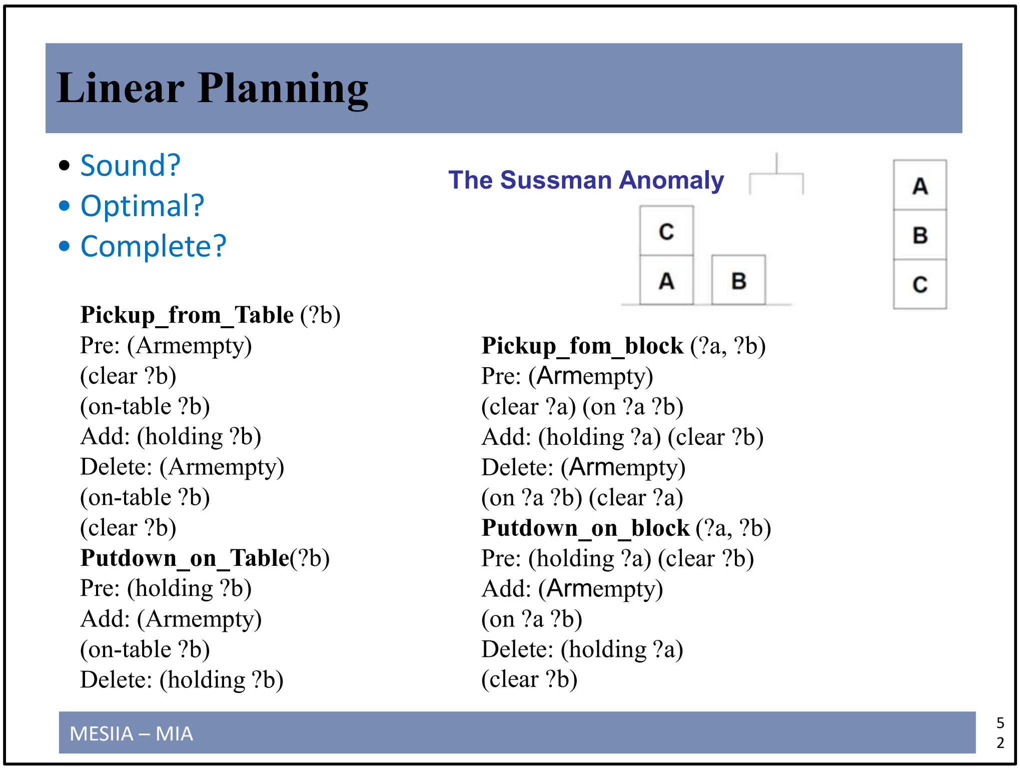
State

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

17) [Pick_Block(C); Put_Block(C, B); Pick_Table(A)] **So that means we want our action PB(A,C). Add to our actions.**

18) [Pick_Block(C); Put_Block(C, B); Pick_Table(A); Put_Block(A, C)] **Now we're down to our original goals.**

19) [Pick_Block(C); Put_Block(C, B); Pick_Table(A); Put_Block(A, C)] **And since our goals are a subset of our state, we return true. Since our stack's empty we're done.**



# 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)

MESIIA – MIA 52

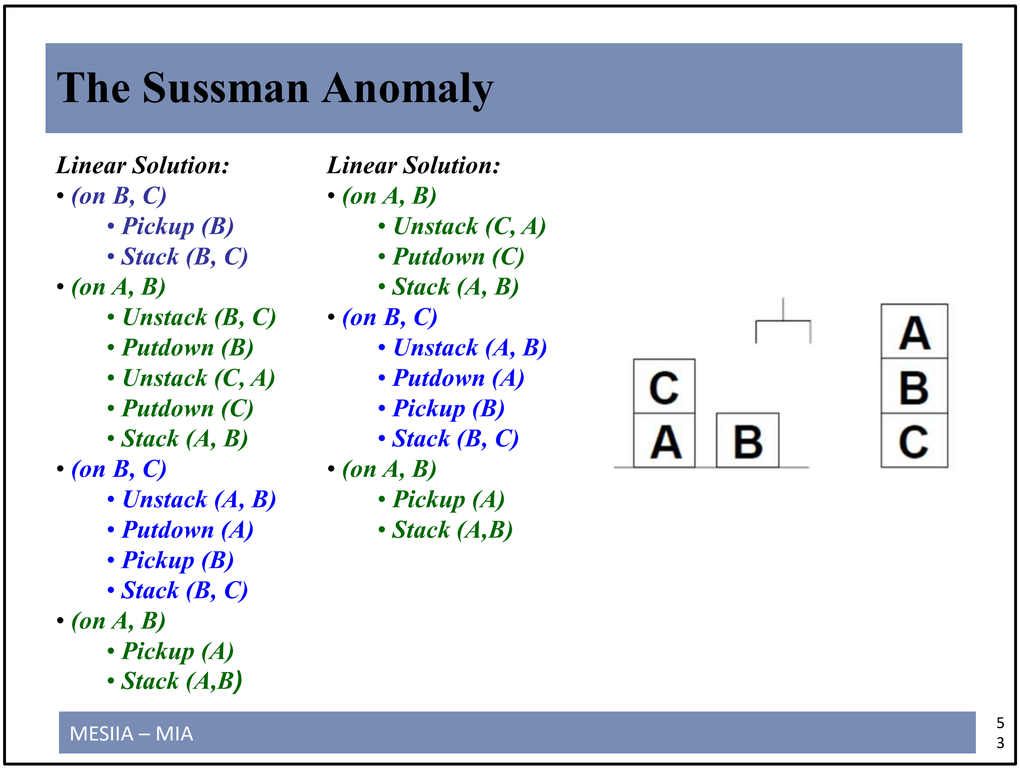
- Here you get really stuck because you're only working on one goal at a time.

- Our goal is On(A,B) and On(B,C).

- Because "C on table" isn't a goal, we don't do the obvious.

- So, when you have multiple goals, the ordering really affects what you do.

- In the figure, an example of he block world problem that known as the Sussman Anomaly that the problem was considered anomalous as the non-interleaved planners of the early 1970 could not solve it.



# The Sussman Anomaly

*Linear Solution:*

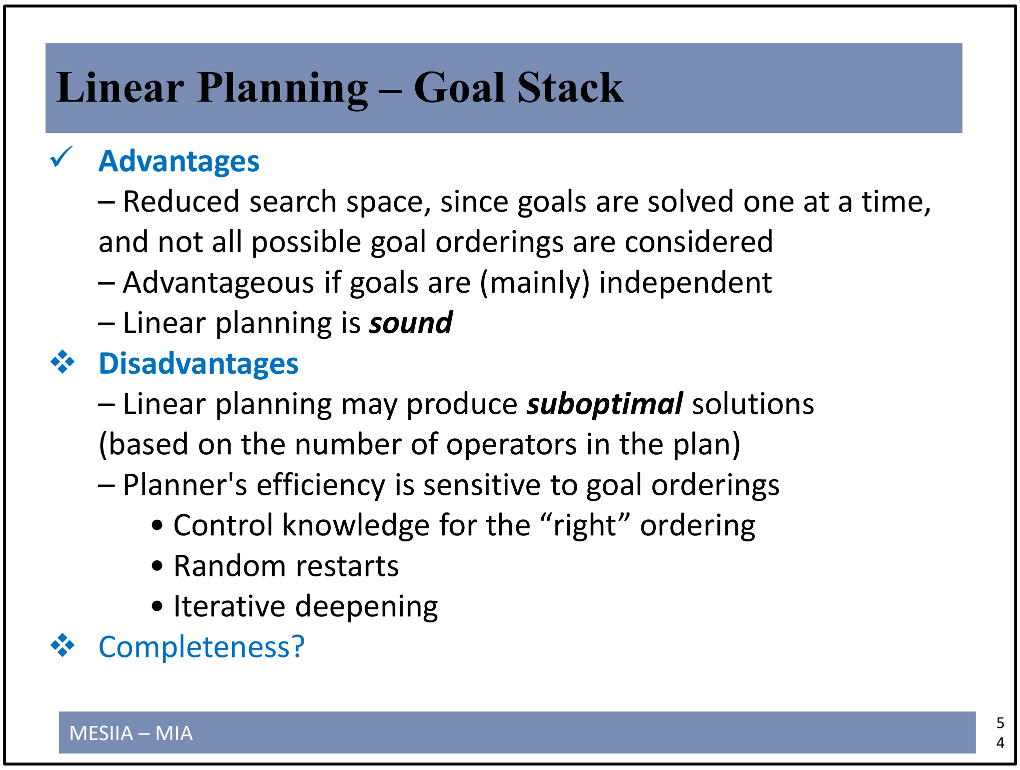
- *(on B, C)*
    - *Pickup (B)*
    - *Stack (B, C)*
- *(on A, B)*
    - *Unstack (B, C)*
    - *Putdown (B)*
    - *Unstack (C, A)*
    - *Putdown (C)*
    - *Stack (A, B)*
- *(on B, C)*
    - *Unstack (A, B)*
    - *Putdown (A)*
    - *Pickup (B)*
    - *Stack (B, C)*
- *(on A, B)*
    - *Pickup (A)*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

This is tow examples of the solution of the Sussman Anomaly block problem.

1) The initial state is blocks B and A on table and C on A and blocks B and C are clear.
2) The goal state must be C on table and B on C, A on B and A must be clear.
3) Here final state requires ON(A,B) and On(B,C) and top diagram tries to focus on subgoal: On(B,C). Now trying to put A on top of B cannot doing without undoing On(B,C) and On(A,B) first. Now it is trying to put B on top of C would cause On(A,B) to be undone.



## 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?

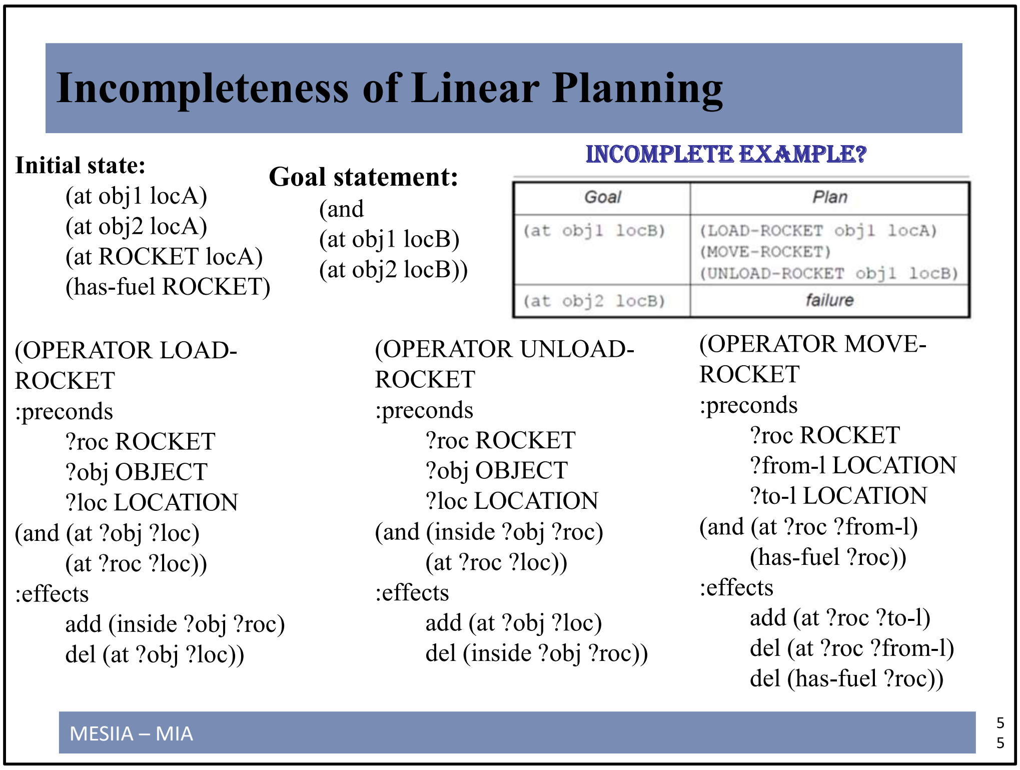
**Linear planning is a goal stacking**

- Linear planners can be sub-optimal
- Planner's efficiency is sensitive to goal orderings
  - – Control knowledge for the “right” ordering
  - – Random restarts
  - – Iterative deepening
- Planner keeps a small search space by not considering all the possible goal orderings.

- In computer science, iterative deepening search or more specifically iterative deepening depth-first search[2] (IDS or IDDFS) is a state space/graph search strategy in which a depth-limited version of depth-first search is run repeatedly with increasing depth limits until the goal is found. IDDFS is optimal like breadth-first search, but uses much less memory; at each iteration, it visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited is effectively breadth-first.

- Note that the random restart mechanism will enable the algorithm to escape from local minima of the evaluation function and can hence be expected to improve its performance.

- So, the linear planning is sound and suboptimal but is it complete?



# 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)<br>(MOVE-ROCKET)<br>(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))
```

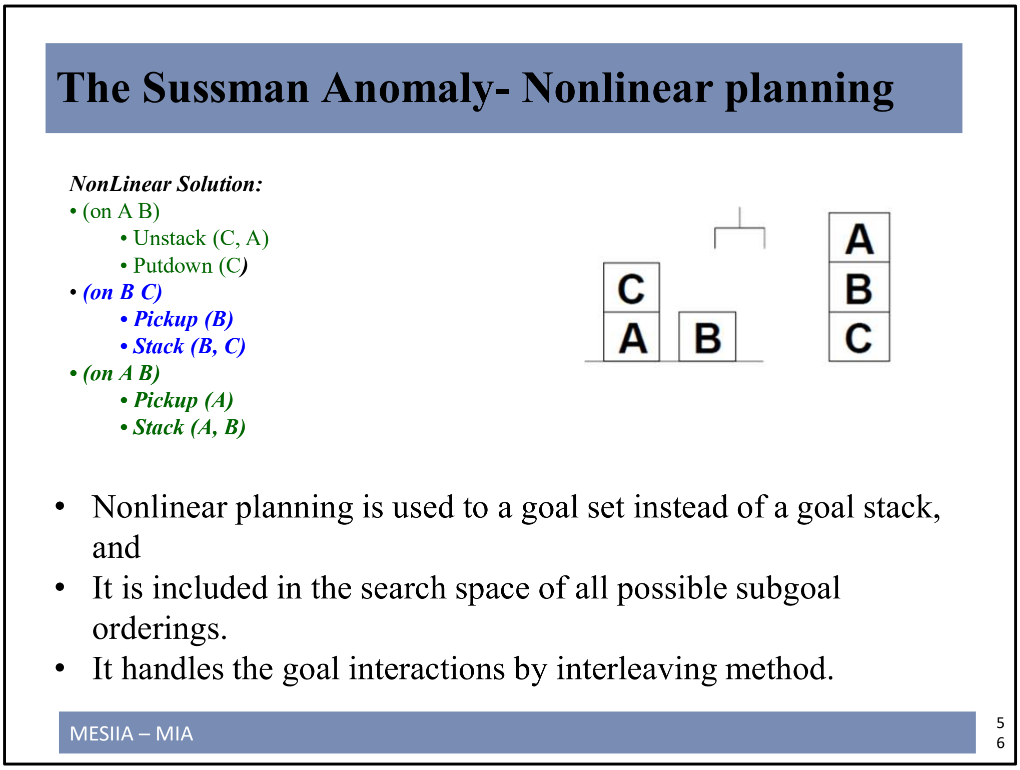
```
(OPERATOR UNLOAD-ROCKET
:preconds
    ?roc ROCKET
    ?obj OBJECT
    ?loc LOCATION
(and (inside ?obj ?roc)
    (at ?roc ?loc))
:effects
    add (at ?obj ?loc)
    del (inside ?obj ?roc))
```

```
(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))
```

Sussman's anomaly showed "sub-optimal".

But sometimes linear planning can even be not complete.

Here, in this example of "One-Way Rocket "
you run out of fuel– we have non-reversible actions, unlike in the block world where you can always return to a past state.



## 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.

MESIIA – MIA

56

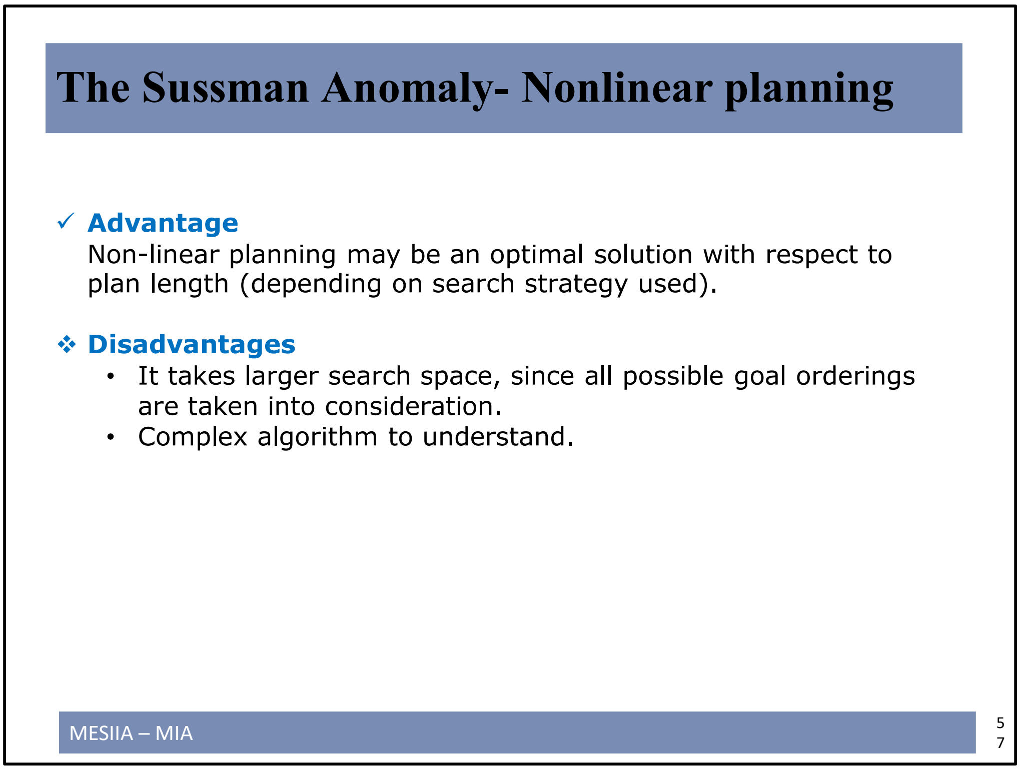
A nonlinear plan is **order inconsistent if and only if the causal links and safety conditions of the plan define** a cycle in the plan steps.

The only way to solve this is goal switching.

- When you're on the way to (on A B), you interrupt it and switch to (on B C).
- Goals are now a goal set, rather than a goal stack.
- Branching factor increases dramatically, but you can actually solve the problem.
- Basic Idea
  - – Use a goal set instead of a goal stack
  - – Include in the search space all possible subgoal orderings
    - Handles goal interactions by ***interleaving***

```
Non-Linear Planning (initial-state, goals)
– state = initial-state; plan = []; goalset = goals; opstack = []
– Repeat until goalset is empty
        • Choose a goal g from the goalset
        • If g does not match state, then
                – Choose an operator (Action) o whose add-list matches goal g
                – Push o on the operator stack
                – Add the preconditions of o to the goalset
```

- While all preconditions of operator on top of operator stack  are met in state
    - – Pop operator **o** from top of opstack
    - – state = apply(o, state)
    - – plan = [plan; o]



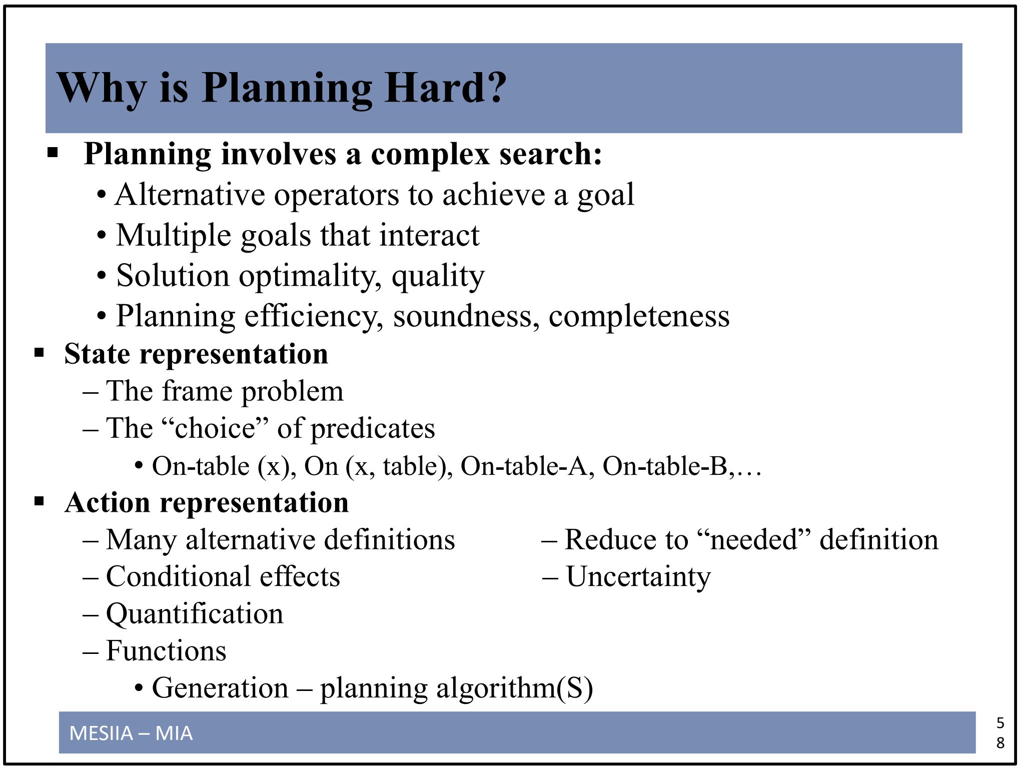
## The Sussman Anomaly- Nonlinear planning

✓ **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.

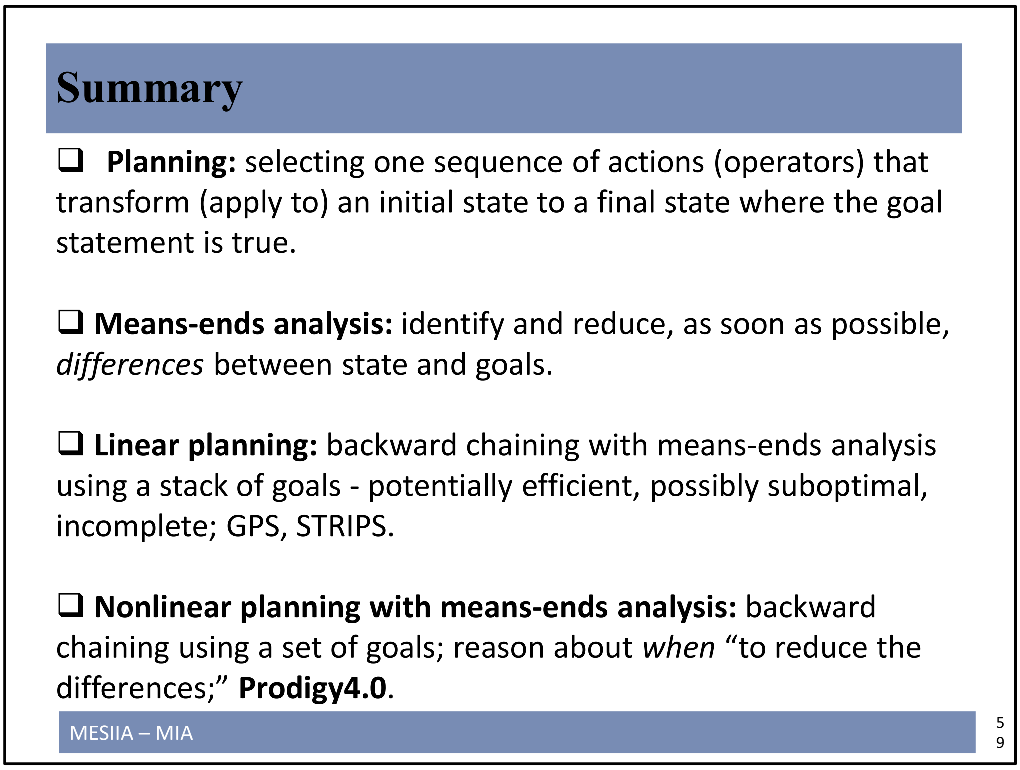
- Advantages
  - – Non-linear planning is sound
  - – Non-linear planning is complete
  - – Non-linear planning may be optimal with respect to plan length (depending on search strategy employed)

- Disadvantages
  - – Larger search space, since all possible goal orderings may have to be considered
  - – Somewhat more complex algorithm; More bookkeeping



## 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