Lecture 11 — Planning

TDT4136: Introduction to Artificial Intelligence

Xavier F. C. Sánchez Díaz

Department of Computer Science
Faculty of Information Technology and Electrical Engineering
Norwegian University of Science and Technology

October 31, 2024

Outline

What is Planning?

Classical Planning

Find a sequence of actions to accomplish a goal in an environment that is:

- discrete
- deterministic
- static
- fully observable

What is Planning?

Classical Planning

Find a sequence of actions to accomplish a goal in an environment that is:

- discrete
- deterministic
- static
- fully observable

In other words, what we have discussed before while studying **search**!

Why do we plan?

What is Planning?

- To address a new situation or problem

Why do we plan?
What is Planning?

- To address a new situation or problem
- When the task is complex

Why do we plan?

What is Planning?

- To address a new situation or problem
- When the task is complex
- when the environment imposes a high risk or cost!

Why do we plan?

What is Planning?

- To address a new situation or problem
- When the task is complex
- when the environment imposes a high risk or cost!
- when collaborating with others

Why do we plan?

What is Planning?

- To address a new situation or problem
- When the task is complex
- when the environment imposes a high risk or cost!
- when collaborating with others

Why do we plan?

What is Planning?

- To address a new situation or problem
- When the task is complex
- when the environment imposes a high risk or cost!
- when collaborating with others

We *explicitly* plan when it is strictly necessary.

How do we do it?

What is Planning?

There are multiple ways to define a planning problem, and several ways of finding a valid plan.

How do we do it?

What is Planning?

There are multiple ways to define a planning problem, and several ways of finding a valid plan.

So far, we have covered:

- Problem-solving by **searching**

How do we do it?

What is Planning?

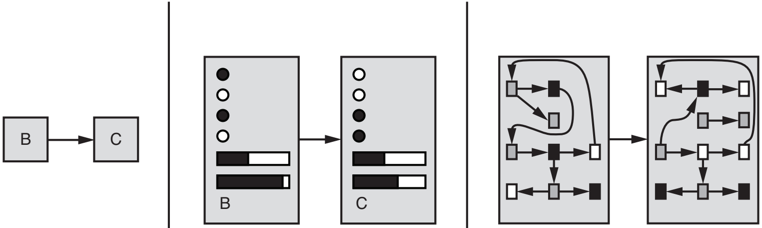
There are multiple ways to define a planning problem, and several ways of finding a valid plan.

So far, we have covered:

- Problem-solving by **searching**
- **Logic** and **satisfiability**

Representing the world

Representing the world



(a) Atomic (b) Factored (b) Structured

The **state of the world** can be described in different ways.

An example: Wumpus World

Representing the world

- A **partially observable** world, with **sensors** and a limited set of actions
- We act rationally by updating **our belief** of the world
- The world is **stored as facts in a knowledge base** (a logical agent can solve this!)

`https://thiagodnf.github.io/wumpus-world-simulator/`

What happens when we move?

What happens when we move?

Time is important!

Dealing with time(steps)

Representing the world

- Some aspects of the world change from time to time. We call them **fluents**.

Dealing with time(steps)

Representing the world

- Some aspects of the world change from time to time. We call them **fluents**.
- For example: the state of a grid cell—‘free’ or ‘occupied’ at time 0

Dealing with time(steps)

Representing the world

- Some aspects of the world change from time to time. We call them **fluents**.
- For example: the state of a grid cell—‘free’ or ‘occupied’ at time 0
- (First Order) Logic cannot deal with it

Dealing with time(steps)

Representing the world

- Some aspects of the world change from time to time. We call them **fluents**.
- For example: the state of a grid cell—‘free’ or ‘occupied’ at time 0
- (First Order) Logic cannot deal with it
  - There are extensions that can handle it, like **linear temporal** or **computational tree** logics!

Dealing with time(steps)

Representing the world

- Some aspects of the world change from time to time. We call them **fluents**.
- For example: the state of a grid cell—‘free’ or ‘occupied’ at time 0
- (First Order) Logic cannot deal with it
  - There are extensions that can handle it, like **linear temporal** or **computational tree** logics!
- An agent’s actions can change aspects of the world (fluents) but not all

Dealing with time(steps)

Representing the world

- Some aspects of the world change from time to time. We call them **fluents**.
- For example: the state of a grid cell—‘free’ or ‘occupied’ at time 0
- (First Order) Logic cannot deal with it
  - There are extensions that can handle it, like **linear temporal** or **computational tree** logics!
- An agent’s actions can change aspects of the world (fluents) but not all
- The agent needs to keep track of fluents, and know what remains unchanged!

A snapshot of the world

Representing the world

Time 0

I am at *cellA*1 facing *east*, I feel a *breeze* and have 1 *arrow*.

A snapshot of the world

Representing the world

Time 0

I am at *cellA*1 facing *east*, I feel a *breeze* and have 1 *arrow*.

If we decide to move *Forward*, then in logic:

$$Location^0_{cellA1} \wedge FacingEast^0 \wedge Forward^0 \implies Location^1_{cellA2} \wedge \neg Location^1_{cellA1}$$

...and although the *arrows* and *breeze* percepts were not modelled, we would have 4 directions $\times T$ time steps $\times n^2$ locations.

A snapshot of the world

Representing the world

Time 0

I am at *cellA*1 facing *east*, I feel a *breeze* and have 1 *arrow*.

If we decide to move *Forward*, then in logic:

$$Location^0_{cellA1} \wedge FacingEast^0 \wedge Forward^0 \implies Location^1_{cellA2} \wedge \neg Location^1_{cellA1}$$

...and although the *arrows* and *breeze* percepts were not modelled, we would have 4 directions $\times\, T$ time steps $\times\, n^2$ locations.

It is extremely expensive and inefficient!

Planning Domain Definition Language
PDDL

Instead, we can use **PDDL**, which uses actions with preconditions and **effects**:

Planning Domain Definition Language

PDDL

Instead, we can use **PDDL**, which uses actions with preconditions and **effects**:

$$
\begin{aligned}
&\mathit{Action}(\mathit{Move}(\mathit{who}, \mathit{from}, \mathit{to})\\
&\qquad \mathit{Precond} : \mathit{At}(\mathit{who}, \mathit{from}) \land \mathit{Adj}(\mathit{from}, \mathit{to}) \land \neg \mathit{Pit}(\mathit{to})\\
&\qquad \mathit{Effect} : \neg \mathit{At}(\mathit{who}, \mathit{from}) \land \mathit{At}(\mathit{who}, \mathit{to}))
\end{aligned}
$$

Planning Domain Definition Language

PDDL

$$
\begin{aligned}
&\mathit{Action}(\mathit{Move}(\mathit{who}, \mathit{from}, \mathit{to}) \\
&\qquad \mathit{Precond} : \mathit{At}(\mathit{who}, \mathit{from}) \wedge \mathit{Adj}(\mathit{from}, \mathit{to}) \wedge \neg \mathit{Pit}(\mathit{to}) \\
&\qquad \mathit{Effect} : \neg \mathit{At}(\mathit{who}, \mathit{from}) \wedge \mathit{At}(\mathit{who}, \mathit{to}))
\end{aligned}
$$

- *Move* is the action being defined
- *who*, *from* and *to* are variables
- *Precond* describes the state of the world needed for the action *to occur*
- *Effect* describes the *resulting* state after acting

States in PDDL

PDDL

- The world is closed—any fluents not mentioned are *False*

States in PDDL

PDDL

- The world is closed—any fluents not mentioned are *False*
- Unique names—different literals refer to different entities (like *cellA*1 and *cellA*2)

States in PDDL

PDDL

- The world is closed—any fluents not mentioned are *False*
- Unique names—different literals refer to different entities (like *cellA*1 and *cellA*2)
- No uncertain or negated literals

States in PDDL

PDDL

- The world is closed—any fluents not mentioned are *False*
- Unique names—different literals refer to different entities (like *cellA*1 and *cellA*2)
- No uncertain or negated literals

States in PDDL

PDDL

- The world is closed—any fluents not mentioned are *False*
- Unique names—different literals refer to different entities (like $cellA1$ and $cellA2$)
- No uncertain or negated literals

As with search, we also need **starting** and **goal** states.

Example: Block world

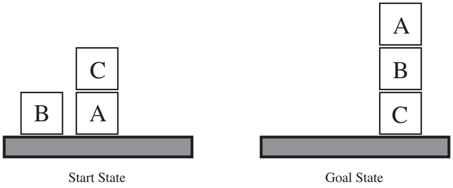
PDDL

Start State

Goal State

What would the **start and goal** states look like in **PDDL**?

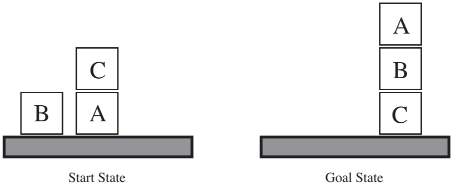
Example: Block world



- **Start**: $On(A, Table), On(B, Table), On(C, A), Clear(B), Clear(C)$[1]
- **Goal**: $On(A, B), On(B, C)$

[1] Commas are a shorthand for *AND*s ($\wedge$) and semicolons for *OR*s ($\vee$)

Example: Block world

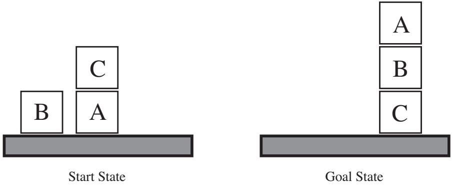


- **Start**: $On(A, Table), On(B, Table), On(C, A), Clear(B), Clear(C)$[1]
- **Goal**: $On(A, B), On(B, C)$

What would the **actions** look like in **PDDL**?

[1] Commas are a shorthand for *AND*s ($\wedge$) and semicolons for *OR*s ($\vee$)

Example: Block world



- **Start**: $On(A, Table), On(B, Table), On(C, A), Clear(B), Clear(C)$
- **Goal**: $On(A, B), On(B, C)$

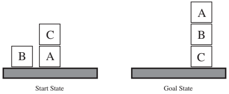
$$Action(Move(block, x, y)$$
$$PRECOND : On(block, x), Clear(block), Clear(y), Block(block), Block(y)$$
$$EFFECT : On(block, y), Clear(x), \neg On(block, x), \neg Clear(y))$$

Notice how any variable in the **effect** must appear in the **precondition**!

Example: Block world

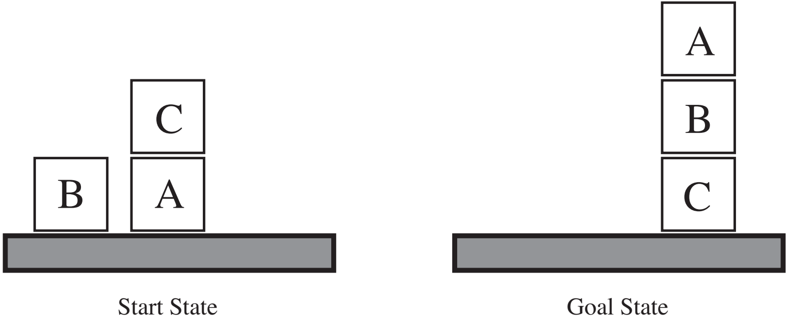
- **Start**: $On(A, Table), On(B, Table), On(C, A), Clear(B), Clear(C)$
- **Goal**: $On(A, B), On(B, C)$

$$
\begin{aligned}
&Action(Move(block, x, y) \\
&\qquad PRECOND: On(block, x), Clear(block), Clear(y) \\
&\qquad EFFECT: On(block, y), Clear(x), \neg On(block, x), \neg Clear(y))
\end{aligned}
$$



$$
\begin{aligned}
&Action(MoveToTable(block, x) \\
&\qquad PRECOND: On(block, x), Clear(block), Clear(Table) \\
&\qquad EFFECT: On(block, Table), Clear(x), \neg On(block, x))
\end{aligned}
$$

Example: Block world



Solution: [$MoveToTable(C, A)$, $Move(B, Table, C)$, $Move(A, Table, B)$]

Adding and deleting

PDDL

$$
\begin{aligned}
&Action(Move(block, x, y) \\
&\qquad PRECOND : On(block, x), Clear(block), Clear(y) \\
&\qquad EFFECT : On(block, y), Clear(x), \neg On(block, x), \neg Clear(y))
\end{aligned}
$$

At **each step** of the plan, the **state of the world** is **modified** depending on the **effect** of the action taken:

Adding and deleting

PDDL

$$Action(Move(block, x, y)$$
$$PRECOND : On(block, x), Clear(block), Clear(y)$$
$$EFFECT : On(block, y), Clear(x), \neg On(block, x), \neg Clear(y))$$

At **each step** of the plan, the **state of the world** is **modified** depending on the **effect** of the action taken:

- Positive fluents are **added**

Adding and deleting

PDDL

$$
\begin{aligned}
&Action(Move(block, x, y)\\
&\quad PRECOND : On(block, x), Clear(block), Clear(y)\\
&\quad EFFECT : On(block, y), Clear(x), \neg On(block, x), \neg Clear(y))
\end{aligned}
$$

At **each step** of the plan, the **state of the world** is **modified** depending on the **effect** of the action taken:

- Positive fluents are **added**
- Negative fluents are **deleted**

Adding and deleting

PDDL

$$
\begin{aligned}
&Action(Move(block, x, y)\\
&\qquad PRECOND : On(block, x), Clear(block), Clear(y)\\
&\qquad EFFECT : On(block, y), Clear(x), \neg On(block, x), \neg Clear(y))
\end{aligned}
$$

At **each step** of the plan, the **state of the world** is **modified** depending on the **effect** of the action taken:

- Positive fluents are **added**
- Negative fluents are **deleted**

Adding and deleting

PDDL

$$
\begin{aligned}
&Action(Move(block, x, y) \\
&\qquad PRECOND: On(block, x), Clear(block), Clear(y) \\
&\qquad EFFECT: On(block, y), Clear(x), \neg On(block, x), \neg Clear(y))
\end{aligned}
$$

At **each step** of the plan, the **state of the world** is **modified** depending on the **effect** of the action taken:

- Positive fluents are **added**
- Negative fluents are **deleted**

These are known as the ADD and DEL lists, and allow us to calculate the state s at the next time step after taking action a:

$$s^{(t+1)} = (s^{(t)} \setminus DEL(a)) \cup ADD(a)$$

Designing actions

PDDL

There are some other things to consider.

- How would the system differentiate between $\mathit{MoveToTable}(\mathit{block}, x)$ and $\mathit{Move}(\mathit{block}, x, \mathit{Table})$?

Designing actions

PDDL

There are some other things to consider.

- How would the system differentiate between $MoveToTable(block, x)$ and $Move(block, x, Table)$?
  - Larger-than-necessary search space!

Designing actions
PDDL

There are some other things to consider.

- How would the system differentiate between $MoveToTable(block, x)$ and $Move(block, x, Table)$?
  - Larger-than-necessary search space!
- Some operators are a bit dumb, i.e., $Move(B, C, C)$ as it introduces an inconsistent $Clear(C), \neg Clear(C)$

Designing actions

PDDL

There are some other things to consider.

- How would the system differentiate between $MoveToTable(block, x)$ and $Move(block, x, Table)$?
  - Larger-than-necessary search space!
- Some operators are a bit dumb, i.e., $Move(B, C, C)$ as it introduces an inconsistent $Clear(C), \neg Clear(C)$

Designing actions

PDDL

There are some other things to consider.

- How would the system differentiate between $MoveToTable(block, x)$ and $Move(block, x, Table)$?
  - Larger-than-necessary search space!
- Some operators are a bit dumb, i.e., $Move(B, C, C)$ as it introduces an inconsistent $Clear(C), \neg Clear(C)$
- A solution is to use the predicate $Block$ so that in order to $Move(x, y, z)$ we need $Block(x), Block(y)$

Designing actions
PDDL

There are some other things to consider.

- How would the system differentiate between $MoveToTable(block, x)$ and $Move(block, x, Table)$?
  - Larger-than-necessary search space!
- Some operators are a bit dumb, i.e., $Move(B, C, C)$ as it introduces an inconsistent $Clear(C), \neg Clear(C)$
- A solution is to use the predicate $Block$ so that in order to $Move(x, y, z)$ we need $Block(x), Block(y)$
- Another approach is to use $\neg(y = z)$ as a precondition for $Move(x, y, z)$

Designing actions
PDDL

There are some other things to consider.

- How would the system differentiate between $\mathit{MoveToTable}(\mathit{block}, x)$ and $\mathit{Move}(\mathit{block}, x, \mathit{Table})$?
  - Larger-than-necessary search space!
- Some operators are a bit dumb, i.e., $\mathit{Move}(B, C, C)$ as it introduces an inconsistent $\mathit{Clear}(C), \neg \mathit{Clear}(C)$
- A solution is to use the predicate Block so that in order to $\mathit{Move}(x, y, z)$ we need $\mathit{Block}(x), \mathit{Block}(y)$
- Another approach is to use $\neg(y = z)$ as a precondition for $\mathit{Move}(x, y, z)$

Designing actions
PDDL

There are some other things to consider.

- How would the system differentiate between $MoveToTable(block, x)$ and $Move(block, x, Table)$?
  - Larger-than-necessary search space!
- Some operators are a bit dumb, i.e., $Move(B, C, C)$ as it introduces an inconsistent $Clear(C), \neg Clear(C)$
- A solution is to use the predicate $Block$ so that in order to $Move(x, y, z)$ we need $Block(x), Block(y)$
- Another approach is to use $\neg(y = z)$ as a precondition for $Move(x, y, z)$

It is **not easy**!

How to plan?

There are several ways to come up with a feasible plan, and the **search space** might be different on each approach.

How to plan?

There are several ways to come up with a feasible plan, and the **search space** might be different on each approach.

- **State-space planning**: search through nodes representing states of the world. A plan is a **path** through the space
- **Plan-space planning**: search through partially instantiated operators and constraints—starts with a partial, *possibly incorrect* plan and then apply changes to correct it.
- **Heuristic planning**: search for a sequence of actions and evaluate your plan using an objective function.

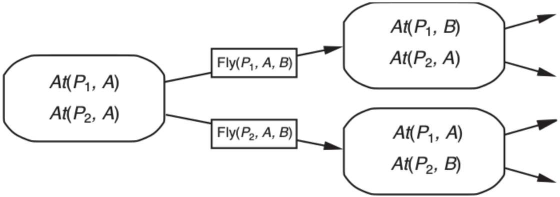
Algorithms for classical planning

How to plan?

- **Forward** (progression) search
- **Backward** (regression) search
- **Logical Inference**

Forward planning

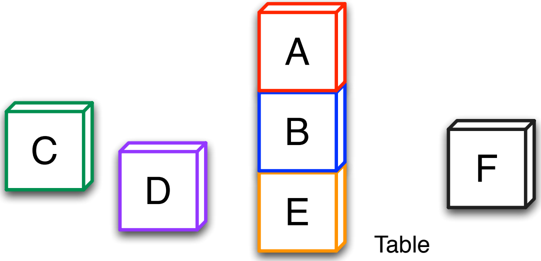
How to plan?



1. Determine all actions applicable
2. **Ground**[2] the actions by replacing any variable with constants
3. Choose an action to apply
4. Determine the new state of the world and update the knowledge based according to the action description
5. Repeat this process until the goal state is reached

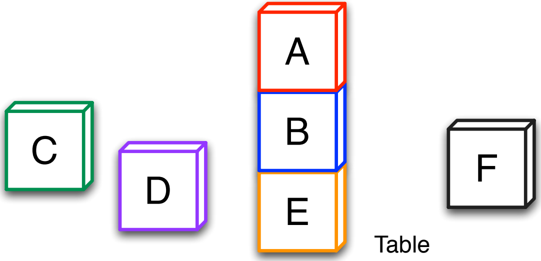
[2]Instantiate, if you will

Forward planning



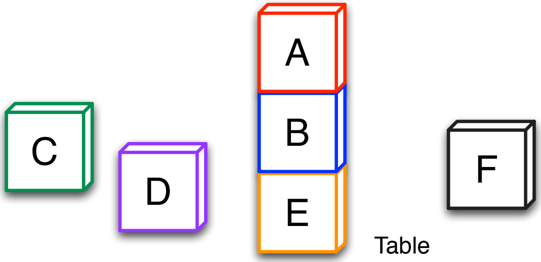
What is your plan?

Forward planning



How many possible **first actions** are there?

Forward planning



How many possible **first actions** are there?

How do we know which one is the *best* one?

Branching Factor
Forward Search

- **Forward search** can have a **very large** branching factor

Branching Factor
Forward Search

- **Forward search** can have a **very large** branching factor
  - Many applicable dumb actions that do not progress towards our goal

Branching Factor

Forward Search

- **Forward search** can have a **very large** branching factor
  - Many applicable dumb actions that do not progress towards our goal
- The search algorithms we have covered can waste a lot of time here

Branching Factor

Forward Search

- **Forward search** can have a **very large** branching factor
  - Many applicable dumb actions that do not progress towards our goal
- The search algorithms we have covered can waste a lot of time here

Branching Factor

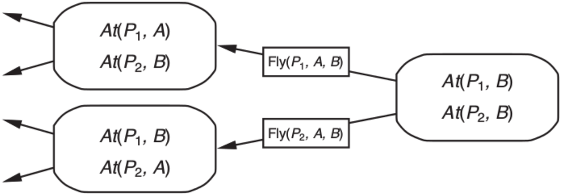
Forward Search

- **Forward search** can have a **very large** branching factor
  - Many applicable dumb actions that do not progress towards our goal
- The search algorithms we have covered can waste a lot of time here

It needs a good (domain-specific) heuristic or pruning procedure!

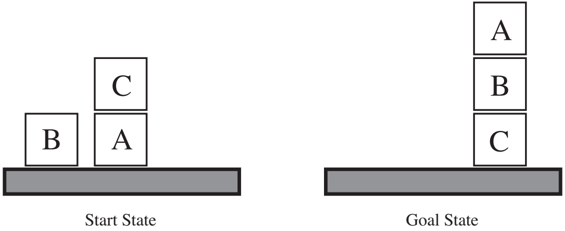
Backward search

How to plan?



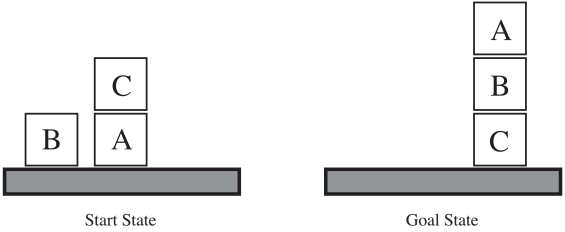
1. Choose a **relevant** action that satisfies (*some*) goal propositions
2. Make a new goal by applying an action *a* **backwards**:
   - *DEL* satisfied conditions of goal
   - *ADD* preconditions of *a*
   - Keep unsolved goal propositions
3. Repeat until the goal is satisfied by the start state

Backward search



What are the **relevant actions** here?

Backward search



What are the **relevant actions** here?

How do we know which one is the *best* one?

Branching factor

Backward search

- **Backward search** is not guided towards any **specific** **subgoal**

Branching factor
Backward search

- **Backward search** is not guided towards any **specific** **subgoal**
- The **order** in which we try to achieve the subgoals (and do search) matters

Branching factor

Backward search

- **Backward search** is not guided towards any **specific subgoal**
- The **order** in which we try to achieve the subgoals (and do search) matters
  - It impacts the efficiency of the search

Branching factor

Backward search

- **Backward search** is not guided towards any **specific subgoal**
- The **order** in which we try to achieve the subgoals (and do search) matters
  - It impacts the efficiency of the search
  - A wrong order can make the plan unfeasible

Sub-plans: Total and partial orders

How to plan

So far, we have only looked at algorithms that generate **complete** plans, with a **strict** order. However, there may exist **subplans**—sequence of actions that can be partially ordered.

Sub-plans: Total and partial orders

How to plan

So far, we have only looked at algorithms that generate **complete** plans, with a **strict** order. However, there may exist **subplans**—sequence of actions that can be partially ordered.

Think about it as when you put your socks and shoes on every morning:

Sub-plans: Total and partial orders

How to plan

So far, we have only looked at algorithms that generate **complete** plans, with a **strict** order. However, there may exist **subplans**—sequence of actions that can be partially ordered.

Think about it as when you put your socks and shoes on every morning:

- Which *sock* should go on first?

Sub-plans: Total and partial orders

How to plan

So far, we have only looked at algorithms that generate **complete** plans, with a **strict** order. However, there may exist **subplans**—sequence of actions that can be partially ordered.

Think about it as when you put your socks and shoes on every morning:

- Which *sock* should go on first?
- Do you do the subsequence $sock_1 \rightarrow shoe_1$ like a lunatic or $sock_1 \rightarrow sock_2$ first?

If we had enough hands I guess we could do $sock_i \rightarrow shoe_i$ in **parallel**!

Heuristic Planning

As usual, the **real world** is a much scarier place.

Heuristic Planning

As usual, the **real world** is a much scarier place.

- Problems (instances) can be huge (number of variables, number of constraints)

Heuristic Planning

As usual, the **real world** is a much scarier place.

- Problems (instances) can be huge (number of variables, number of constraints)
- We have limited resources / memory / time / money

Heuristic Planning

As usual, the **real world** is a much scarier place.

- Problems (instances) can be huge (number of variables, number of constraints)
- We have limited resources / memory / time / money
- We need faster solutions

Heuristic Planning

As usual, the **real world** is a much scarier place.

- Problems (instances) can be huge (number of variables, number of constraints)
- We have limited resources / memory / time / money
- We need faster solutions

Heuristic Planning

As usual, the **real world** is a much scarier place.

- Problems (instances) can be huge (number of variables, number of constraints)
- We have limited resources / memory / time / money
- We need faster solutions

We need **heuristic** solutions!

General idea: relax
Heuristic Planning

Most heuristics rely on **relaxing** the problem to make it easier.

General idea: relax

Heuristic Planning

Most heuristics rely on **relaxing** the problem to make it easier.

In planning:

- Add edges and **group** nodes (subplans)

General idea: relax

Heuristic Planning

Most heuristics rely on **relaxing** the problem to make it easier.

In planning:

- Add edges and **group** nodes (subplans)
  - State abstraction, pattern DBs, symmetry reduction. . .

General idea: relax

Heuristic Planning

Most heuristics rely on **relaxing** the problem to make it easier.

In planning:

- Add edges and **group** nodes (subplans)
  - State abstraction, pattern DBs, symmetry reduction…
- Ignore restrictions

General idea: relax

Heuristic Planning

Most heuristics rely on **relaxing** the problem to make it easier.

In planning:

- Add edges and **group** nodes (subplans)
  - State abstraction, pattern DBs, symmetry reduction…
- Ignore restrictions
  - Either **all** or **some** of them

General idea: relax

Heuristic Planning

Most heuristics rely on **relaxing** the problem to make it easier.

In planning:

- Add edges and **group** nodes (subplans)
  - State abstraction, pattern DBs, symmetry reduction…
- Ignore restrictions
  - Either **all** or **some** of them
- Ignore **negative fluents**

General idea: relax

Heuristic Planning

Most heuristics rely on **relaxing** the problem to make it easier.

In planning:

- Add edges and **group** nodes (subplans)
  - State abstraction, pattern DBs, symmetry reduction. . .
- Ignore restrictions
  - Either **all** or **some** of them
- Ignore **negative fluents**
- Weigh actions to have **preferred** actions

General idea: relax

Heuristic Planning

Most heuristics rely on **relaxing** the problem to make it easier.

In planning:

- Add edges and **group** nodes (subplans)
  - State abstraction, pattern DBs, symmetry reduction...
- Ignore restrictions
  - Either **all** or **some** of them
- Ignore **negative fluents**
- Weigh actions to have **preferred** actions
- Find **serialisable subplans**

General idea: relax

Most heuristics rely on **relaxing** the problem to make it easier.

In planning:

- Add edges and **group** nodes (subplans)
  - State abstraction, pattern DBs, symmetry reduction...
- Ignore restrictions
  - Either **all** or **some** of them
- Ignore **negative fluents**
- Weigh actions to have **preferred** actions
- Find **serialisable subplans**
  - Achieving one will not undo the others.

General idea: relax

Heuristic Planning

Most heuristics rely on **relaxing** the problem to make it easier.

In planning:

- Add edges and **group** nodes (subplans)
  - State abstraction, pattern DBs, symmetry reduction...
- Ignore restrictions
  - Either **all** or **some** of them
- Ignore **negative** **fluents**
- Weigh actions to have **preferred** actions
- Find **serialisable subplans**
  - Achieving one will not undo the others.

General idea: relax

Heuristic Planning

Most heuristics rely on **relaxing** the problem to make it easier.

In planning:

- Add edges and **group** nodes (subplans)
  - State abstraction, pattern DBs, symmetry reduction...
- Ignore restrictions
  - Either **all** or **some** of them
- Ignore **negative fluents**
- Weigh actions to have **preferred** actions
- Find **serialisable subplans**
  - Achieving one will not undo the others.

...or try a metaheuristic!

Examples of relaxation

Heuristic Planning

- **State abstraction**: group subtasks into one bigger, more abstract task

Examples of relaxation

Heuristic Planning

- **State abstraction**: group subtasks into one bigger, more abstract task
  - All packages for Trondheim (from Oslo) count as one big package

Examples of relaxation

Heuristic Planning

- **State abstraction**: group subtasks into one bigger, more abstract task
  - All packages for Trondheim (from Oslo) count as one big package
  - Worry about getting those packages to Trondheim first. You can worry about individual deliveries afterwards!

Examples of relaxation

Heuristic Planning

- **State abstraction**: group subtasks into one bigger, more abstract task
  - All packages for Trondheim (from Oslo) count as one big package
  - Worry about getting those packages to Trondheim first. You can worry about individual deliveries afterwards!
  - Decompose costs: $Cost(P) = Cost(P_i) + Cost(P_j)$ where i, j are subgoals

Examples of relaxation

Heuristic Planning

- **State abstraction**: group subtasks into one bigger, more abstract task
  - All packages for Trondheim (from Oslo) count as one big package
  - Worry about getting those packages to Trondheim first. You can worry about individual deliveries afterwards!
  - Decompose costs: $Cost(P) = Cost(P_i) + Cost(P_j)$ where i, j are subgoals
- **Ignore restrictions**: 'a perfect plan *if* this road r were not closed'

Examples of relaxation

- **State abstraction**: group subtasks into one bigger, more abstract task
  - All packages for Trondheim (from Oslo) count as one big package
  - Worry about getting those packages to Trondheim first. You can worry about individual deliveries afterwards!
  - Decompose costs: $Cost(P) = Cost(P_i) + Cost(P_j)$ where i, j are subgoals

- **Ignore restrictions**: 'a perfect plan *if* this road r were not closed'
  - We will try finding an alternative route for the r segment later!

Examples of relaxation

Heuristic Planning

- **State abstraction**: group subtasks into one bigger, more abstract task
  - All packages for Trondheim (from Oslo) count as one big package
  - Worry about getting those packages to Trondheim first. You can worry about individual deliveries afterwards!
  - Decompose costs: $Cost(P) = Cost(P_i) + Cost(P_j)$ where i, j are subgoals
- **Ignore restrictions**: ‘a perfect plan *if* this road r were not closed’
  - We will try finding an alternative route for the r segment later!
- **Serialisable subplans**: achieving a subgoal (putting on $shoe_1$) does not interfere with other goals (putting on $shoe_2$)

Classical Planning vs IRL

Planning in Complex Environments

So far, we have covered how to do **classical planning**:

Classical Planning vs IRL

Planning in Complex Environments

So far, we have covered how to do **classical planning**:

1. The environment is fully observable

Classical Planning vs IRL

Planning in Complex Environments

So far, we have covered how to do **classical planning**:

1. The environment is fully observable
   - We can “see” what happens at any state

Classical Planning vs IRL

Planning in Complex Environments

So far, we have covered how to do **classical planning**:

1. The environment is fully observable
   - We can "see" what happens at any state
2. The environment is deterministic

Classical Planning vs IRL

Planning in Complex Environments

So far, we have covered how to do **classical planning**:

1. The environment is fully observable
   - We can “see” what happens at any state
2. The environment is deterministic
   - Our actions always give us the same effect

Classical Planning vs IRL

Planning in Complex Environments

So far, we have covered how to do **classical planning**:

1. The environment is fully observable
   - We can "see" what happens at any state
2. The environment is deterministic
   - Our actions always give us the same effect
3. The environment is static

Classical Planning vs IRL

Planning in Complex Environments

So far, we have covered how to do **classical planning**:

1. The environment is fully observable
   - We can “see” what happens at any state
2. The environment is deterministic
   - Our actions always give us the same effect
3. The environment is static
   - It never changes, unless we act

Classical Planning vs IRL

Planning in Complex Environments

So far, we have covered how to do **classical planning**:

1. The environment is fully observable
   - We can "see" what happens at any state
2. The environment is deterministic
   - Our actions always give us the same effect
3. The environment is static
   - It never changes, unless we act

Classical Planning vs IRL

Planning in Complex Environments

So far, we have covered how to do **classical planning**:

1. The environment is fully observable
   - We can "see" what happens at any state
2. The environment is deterministic
   - Our actions always give us the same effect
3. The environment is static
   - It never changes, unless we act

How do we plan when we encounter more complex environments?

Planning in Complex Environments

1. I cannot see

Planning in Complex Environments

1. I cannot see
   - So we make a **sensorless plan**

Planning in Complex Environments

1. I cannot see
   - So we make a **sensorless plan**
2. I can see but I am not sure what will happen

Planning in Complex Environments

1. I cannot see
   - So we make a **sensorless plan**
2. I can see but I am not sure what will happen
   - So we make a **contingency plan**

Planning in Complex Environments

1. I cannot see
   - So we make a **sensorless plan**
2. I can see but I am not sure what will happen
   - So we make a **contingency plan**
3. I can see and am sure what will happen but need to keep an eye

Planning in Complex Environments

1. I cannot see
   - So we make a **sensorless plan**
2. I can see but I am not sure what will happen
   - So we make a **contingency plan**
3. I can see and am sure what will happen but need to keep an eye
   - Then we need **online planning**

I cannot see

Sensorless planning

- I need to **make sure** that all **preconditions are met**
- I will then **carry out all operations** that will lead me to the goal

Example: paint a chair and a table with <u>the same</u> colour. How?

I cannot see

Sensorless planning

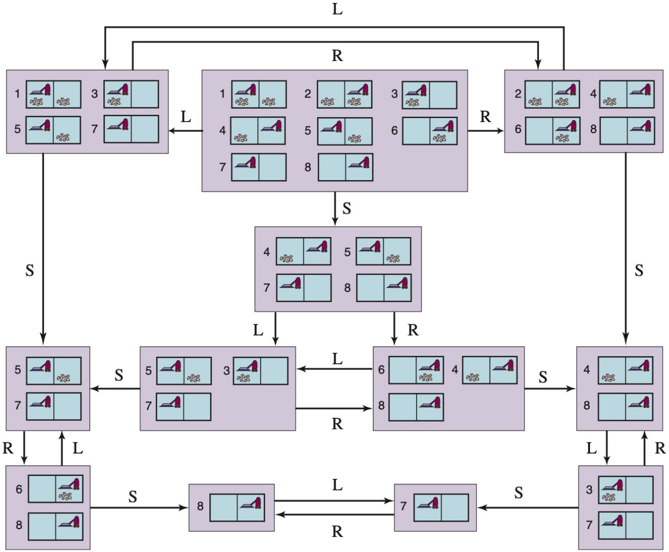
- I need to **make sure** that all **preconditions are met**
- I will then **carry out all operations** that will lead me to the goal

Example: paint a chair and a table with the same colour. How?

$$[RemoveLid(Can), Paint(Chair, Can), Paint(Table, Can)]$$

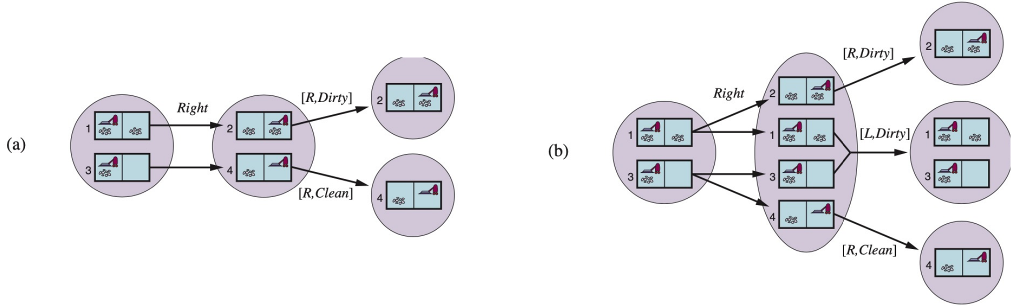
Recall

Belief space in sensorless planning



Recall

Belief space in non-deterministic world

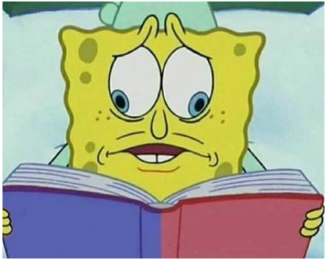


- The agent knows where it is and see the dirt (if any) on its spot
- The **transition model** becomes a **function** of a belief state, an action, and a **another** belief state
  - In case of nondeterminism (right), we do like Dr. Strange and consider possible outcomes on different universes. **How**?

I can see but I am not sure what will happen

Contingency planning

- I need to **make sure** to know where I am by **looking around**
- Then, and depending on where I am, I will **carry out** necessary operations **conditionally**

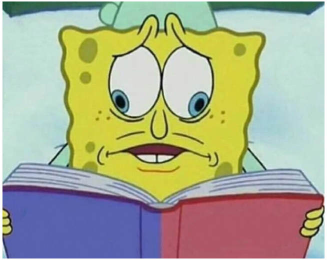


Example: paint a chair and a table with the same colour. How?

I can see but I am not sure what will happen

Contingency planning

- I need to **make sure** to know where I am by **looking around**
- Then, and depending on where I am, I will **carry out** necessary operations **conditionally**



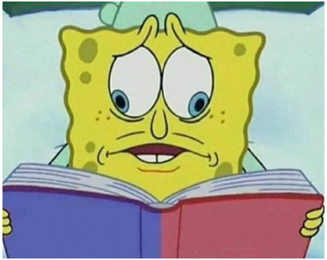
Example: paint a chair and a table with <u>the same</u> colour. How?

$$\begin{aligned} & \mathit{FullPlan} = \\ & \qquad [\mathit{LookAt}(\mathit{Table}), \mathit{LookAt}(\mathit{Chair}), \\ & \qquad \textbf{if } \mathit{Color}(\mathit{Table}, c) \wedge \mathit{Color}(\mathit{Chair}, c), \textbf{then } \mathit{NoOP} \\ & \qquad \textbf{else } \mathit{ContingencyPlan}] \end{aligned}$$

I can see but I am not sure what will happen

Contingency planning

- I need to **make sure** to know where I am by **looking around**
- Then, and depending on where I am, I will **carry out** necessary operations **conditionally**



Example: paint a chair and a table with <u>the same</u> colour. How?

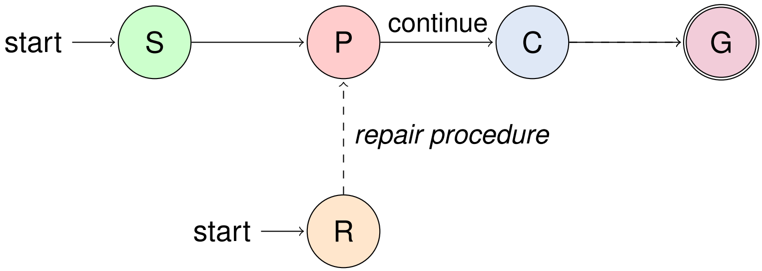
$$
\begin{aligned}
&ContingencyPlan = \\
&\qquad [RemoveLid(Can), LookAt(Can), \\
&\qquad \textbf{if } Color(Table, c) \wedge Color(Can, c) \textbf{ then } Paint(Chair, can) \\
&\qquad \textbf{else if } Color(Chair, c) \wedge Color(can, c) \textbf{then } Paint(Table, can) \\
&\qquad \textbf{else}[Paint(Chair, Can), Paint(Table, Can)]]
\end{aligned}
$$

I need to keep an eye

Online planning

The world is **dynamic**, and it can change in unpredictable ways.

- **Action monitoring**: check that all **preconditions hold**
- **Plan monitoring**: check if the plan can **succeed** (or the goal is **satisfiable**)
- **Goal monitoring**: check if there is a *better* goal



The Job-shop Scheduling Problem

Figure 1

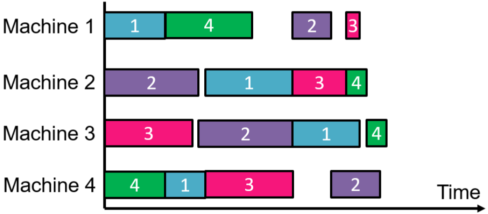


Figure 2

Machine	O1	O2	O3	O4
J1	1	4	2	3
J2	2	3	1	4
J3	3	4	2	1
J4	4	1	2	3

Table 1 Machine Sequence

Time	O1	O2	O3	O4
J1	3	2	4	3
J2	4	5	2	2
J3	4	4	3	1
J4	3	4	1	1

Table 2 Processing Time

Image from Ataç, 2023: Job Shop Scheduling Problem and Solution Algorithms

JSP: heuristic approach

The Job-shop Scheduling Problem

- Assign the **earliest** and **latest** possible start for each action

JSP: heuristic approach

The Job-shop Scheduling Problem

- Assign the **earliest** and **latest** possible start for each action
- Then duration of the plan is that of the **critical path**

JSP: heuristic approach

The Job-shop Scheduling Problem

- Assign the **earliest** and **latest** possible start for each action
- Then duration of the plan is that of the **critical path**
  - i.e., the path with the **longest duration**. There cannot be a shorter plan than this!

JSP: heuristic approach

The Job-shop Scheduling Problem

- Assign the **earliest** and **latest** possible start for each action
- Then duration of the plan is that of the **critical path**
  - i.e., the path with the **longest duration**. There cannot be a shorter plan than this!
- The JSP is an $\mathcal{NP}$-problem

JSP: heuristic approach

The Job-shop Scheduling Problem

- Assign the **earliest** and **latest** possible start for each action
- Then duration of the plan is that of the **critical path**
  - i.e., the path with the **longest duration.** There cannot be a shorter plan than this!
- The JSP is an $\mathcal{NP}$-problem
  - The optimisation version (finding how) is an $\mathcal{NP}$-hard problem

JSP: heuristic approach

The Job-shop Scheduling Problem

- Assign the **earliest** and **latest** possible start for each action
- Then duration of the plan is that of the **critical path**
  - i.e., the path with the **longest duration**. There cannot be a shorter plan than this!
- The JSP is an $\mathcal{NP}$-problem
  - The optimisation version (finding how) is an $\mathcal{NP}$-hard problem

JSP: heuristic approach

The Job-shop Scheduling Problem

- Assign the **earliest** and **latest** possible start for each action
- Then duration of the plan is that of the **critical path**
  - i.e., the path with the **longest duration**. There cannot be a shorter plan than this!
- The JSP is an $\mathcal{NP}$-problem
  - The optimisation version (finding how) is an $\mathcal{NP}$-hard problem

In practice we use **metaheuristics**!