Planning in a Nutshell

We consider classical planning problems, which consist of:

- An initial state $s_i$ – all “world properties” true in the beginning.
- A set of available actions – how world states can be changed.
- A goal description $g$ – all properties we’d like to hold.

What do we want?

→ Find a plan that transforms $s_i$ into $g$. 

![Diagram of planning process](Image)
Planning Games like the Sliding Tile Puzzle

Initial State

Goal State

Planning Robots like the Mars Rovers

- The mixed-initiative planning system MAPGEN was used to generate rovers’ plans offline.
- These are then executed by the rovers (i.e., they do not run planners).

Planning Automated Factories like a Greenhouse

Source: https://www.lemnatec.com/
Copyright: With kind permission from LemnaTec GmbH
Further reading:
- Helmert2010Scanalyzer
- The IPC Scanalyzer Domain in PDDL (see paper above).

Planning: A Domain-Independent Approach

- Automated Planning is a domain-independent approach!
- As mentioned in the beginning, the integral part is:
  - The state descriptions: Which state properties exist?
  - Actions: What can be done and how does this change states?
- Planning technology is agnostic against specific applications!
- Research bases on an abstract high-level description language.
  Example action in a domain controlling Satellites:

```plaintext
(def-durative :action turn_to
 :duration (= ?duration 5)
 :condition (and (at-start (pointing ?s ?d_prev))
 (over-all (not (= ?d_new ?d_prev))))
 :effect (and (at-end (pointing ?s ?d_new))
 (at-start (not (pointing ?s ?d_prev))))
)
```
Domain-Independence: Pros vs. Cons

Advantages of Domain-independence:
- Use (well-tested) standard solvers:
  - Cost-effective: only write the model, not new software
  - Most likely there are less bugs
- Optimality guarantees of solutions (find the cheapest).
- Exploit further planning technology, e.g., automated support for:
  - Model can be checked for problems.
  - Existing techniques for proving unsolvability can be used.
  - Plan explanation techniques can be exploited.
  - Check solutions.

Disadvantages of Domain-independence:
- You need a planning expert to model the domain.
  (But we will have many more in just like 60 minutes!)
- Potential inefficiency: a domain-specific might be more efficient than a domain-independent one.

Math Recap: Sets versus Lists

We will heavily base upon sets.
- Sets are invariant against repetition.
  - E.g., \{a, a, b, b, c\} = \{a, b, c\}.
- Sets are invariant against re-ordering.
  - E.g., \{a, b, a, b, c, b\} = \{a, a, b, b, c\}.
  - Thus, we also get \{a, b, a, b, c, b\} = \{a, a, b, b, c\} = \{a, c, b\} = \{b, a, c\} = \{b, c, a\} = \{c, a, b\} = \{c, b, a\}
- Therefore, sets are fundamentally different from lists!
  - Lists are also called sequences or tuples.
  - E.g., the list \((a, b, c)\) is different from both \((a, c, b)\) and \((a, a, b, c)\)

The empty set \(\{\}\) is usually always denoted by the symbol \(\emptyset\).
- In contrast, the empty list (i.e., empty sequence) is denoted by \(\varepsilon\).
- Let \(X\) be a finite set. Then, \(2^X\) denotes its “power set” that contains all subsets.
  - E.g., if \(X = \{a, b, c\}\), then \(2^X = \{\{a, b, c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a\}, \{b\}, \{c\}, \emptyset\}\)
- The “cardinality” of a set \(X\) is the number of its elements and denoted by \(|X|\). Cardinality is not recursive!
  - E.g., if \(X = \{a, b, c\}\), then \(|X| = 3\).
  - Regarding non-recursiveness: if, e.g., \(X = \{a, b\}\), then \(|X| = 2\), because it has two elements, not three.
- Note that for all possible \(X\) it holds that \(|2^X| = 2^{|X|}|.\)
  - E.g., if \(X = \{a, b, c\}\), then \(|2^X| = 2^{|X|} = 2^3 = 8\)
Math Recap: Set Notation, cont’d

- We write \( a \in X \) if \( a \) is contained in \( X \), i.e., an element of \( X \). We say \( a \notin X \) if \( a \) is not contained in \( X \). Again, this is not recursive!
  - E.g., if \( X = \{ \{a, b\}, c\} \), then only \( \{a, b\} \in X \) and \( c \in X \), but \( a \notin X \) and \( b \notin X \). Also \( \{c\} \notin X \).

- We write \( Y \subseteq X \) if \( Y \) is a (not necessarily strict) subset of \( X \). We write \( Y \nsubseteq X \) if this does not hold.
  - E.g., \( \{a, b, c\} \subseteq \{a, b, c\} \) (in fact, \( X \subseteq X \) for all sets \( X \)).
  - E.g., \( \{a, c\} \subseteq \{a, b, c\} \).
  - E.g., if \( X = \{\{a, b\}, c\} \), then \( \{c\} \subseteq X \) and \( \{\{a, b\}\} \subseteq X \), but \( \{\{a, c\}\} \nsubseteq X \) and \( \{a, b\} \nsubseteq X \).

- Let \( X, Y, Z \) be sets. Then \( X \times Y \times Z \) is the cartesian product of these sets. It’s the set of tuples with one element from each set.
  - Formally, \( X \times Y \times Z = \{(x, y, z) \mid x \in X, y \in Y, z \in Z\} \).
  - One normally uses standard math notation, e.g., \( X^2 = X \times X \). We use \( X^n \) for exactly \( n \) times \( X \), and \( X^\ast \) for all numbers in \( \mathbb{N} \cup \{0\} \).

Math Recap: Set Operations

- \( X \cup Y \) denotes the set union of \( X \) and \( Y \), i.e., \( X \cup Y \) contains all elements that occur in any of those sets.
  - Formally: \( X \cup Y = \{z \mid z \in X \text{ or } z \in Y\} \).
- \( X \cap Y \) denotes the set intersection, i.e., \( X \cap Y \) contains all elements that are contained in both sets.
  - Formally: \( X \cap Y = \{z \mid z \in X \text{ and } z \in Y\} \).
- \( X \setminus Y \) denotes the set subtraction, i.e., \( X \setminus Y \) contains all elements that are contained in \( X \), but not (“minus the ones”) in \( Y \).
  - Formally: \( X \setminus Y = \{z \in X \mid z \notin Y\} \).

Problem Definition: Assumptions made in Classical Planning

We focus on the “base case” of AI planning: Classical Planning
- **Discrete:** only instantaneous state changes (no time)
- **Deterministic:** outcomes of actions are known and unique
- **Fully observable:** no hidden information anywhere
- **Single-agent:** “the planner” controls all actions
A classical planning problem \( \mathcal{P} = (V, A, s_i, g) \) consists of:

- \( V \) is a finite set of state variables.
  - \( S = 2^V \) is called the state space. (The set of all states.)
  - States are sets consisting of state variables (also called facts).
  - We assume the closed world assumption, i.e., all variables not mentioned in a state do not hold in that state (in contrast to: it's not known whether they hold or not).
  - E.g., if \( a \in s \) then \( a \) is true (does hold) in \( s \), but if \( a \notin s \) then \( a \) is false (does not hold) in \( s \).

- \( A \) is a finite set of actions.
  - Each action \( a \in A \) is a tuple \((\text{pre}, \text{add}, \text{del}, c) \in 2^V \times 2^V \times 2^V \times \mathbb{R}^+ \) consisting of a precondition, add and delete list, and action costs.

\[ V = \{\text{CrateAtLoc1}, \text{CrateInCrane}, \text{CrateInTruck}, \text{TruckAtLoc1}, \text{TruckAtLoc2}\} \]
Problem Definition: Formalism

A classical planning problem $\mathcal{P} = (V, A, s_i, g)$ consists of:
- $A$ is a finite set of actions. Each action $a \in A$ is a tuple $(pre, add, del, c) \in 2^V \times 2^V \times 2^V$ consisting of a precondition, add and delete list, and action costs. For convenience, we write $pre(a)$, $add(a)$, $del(a)$, and $c(a)$.

- $s_i \in S$ is the initial state (complete state description).
- $g \subseteq V$ is the goal description.
  - Each state $s \in S$ with $s \supseteq g$ is called a goal state.
  - We abbreviate the set of goal states with $G = \{ s \in S \mid s \supseteq g \}$
  - $s_i = \{ \text{CrateAtLoc1}, \text{TruckAtLoc2} \} = s_0$
  - $g = \{ \text{CrateInCrane}, \text{TruckAtLoc2} \}$, thus: $G = \{ s_0 \}$ since $s_0 \supseteq g$.

Problem Definition: Formalism, cont’d

- An action $a \in A$ is called applicable (or executable) in a state $s \in S$ if and only if $pre(a) \subseteq s$.

- If $pre(a) \subseteq s$ holds, its application results into the successor state $\gamma(a, s) = (s \setminus del(a)) \cup add(a)$. $\gamma : A \times S \rightarrow S$ is called the state transition function.

  → Example: The action...
  - $\text{take}$ pre: $\{ \text{CrateAtLoc1} \}$
    add: $\{ \text{CrateInCrane} \}$
    del: $\{ \text{CrateAtLoc1} \}$
  - ... is applicable in state $\{ \text{CrateAtLoc1}, \text{TruckAtLoc2} \}$ resulting into $\{ \text{TruckAtLoc2}, \text{CrateInCrane} \}$. 
Problem Definition: Formalism, cont’d I

An action \( a \in A \) is called applicable (or executable) in a state \( s \in S \) if and only if \( \text{pre}(a) \subseteq s \).

If \( \text{pre}(a) \subseteq s \) holds, its application results into the successor state
\[
\gamma(a, s) = (s \setminus \text{del}(a)) \cup \text{add}(a). \quad \gamma : A \times S \to S
\]
is called the state transition function.

An action sequence \( \alpha = a_0, \ldots, a_{n-1} \) is applicable in a state \( s_0 \) if and only if
- for all \( 0 \leq i \leq n - 1 \), \( a_i \) is applicable in \( s_i \), where for all \( 1 \leq i \leq n \), \( s_i \) denotes the resulting state of applying \( a_0, \ldots, a_i \) to \( s_0 = s_i \).
- This means: Each action is applicable in its predecessor state.

We extend the state transition function to work on action sequences as well, i.e., \( \gamma : A^* \times S \to S \). (Definition omitted.)

Problem Definition: Formalism, cont’d II

This is everything about the classical planning formalism! I.e.,
- Formal definition of the “planning problem”.
- Formal definition of any “plan”, i.e., solution.

Most notably, this includes the definition of action application.

Questions so far?

Introduction  Notation  AI Planning Problems  State Transition Systems  Blocksworld  Summary and Outlook

State Transition Systems
**What's a State Transition System?**

- State transition systems are the “underlying semantics” of classical planning problems.
- They **explicitly** show all states and how they can be traversed by actions.
- We use them to give an intuition on how hard solving planning problems can become (and how easy it is to model them)!

---

**Formal Definition of State Transition System**

**Definition (State Transition System)**

A state transition system is a 6-tuple \((S, L, c, T, I, G)\), where

- \(S\) is a finite set of states.
- \(L\) is a finite set of transition labels.
- \(c : L \rightarrow \mathbb{R}^+\) is a cost function.
- \(T \subseteq S \times L \times S\) is the transition relation.
- \(I \in S\) is the initial state.
- \(G \subseteq S\) is the set of goal states.

---

**Example for a State Transition System**

- **State transition system** is just a graph consisting of states and labeled edges
- with a designated initial state and designated goal states

---

**Size Increase of the State Space in Blocksworld**

- **We have:** \(n\) blocks, 1 gripper, and two actions, each takes a top-most block with the gripper and
  - puts it immediately onto some other top-most block
  - or onto the table, respectively.
- **We want:** transform the initial towers into another, given set of towers.

---

<table>
<thead>
<tr>
<th>blocks</th>
<th>states</th>
<th>blocks</th>
<th>states</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>58,941,091</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>11</td>
<td>824,073,141</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>12</td>
<td>12,470,162,233</td>
</tr>
<tr>
<td>4</td>
<td>73</td>
<td>13</td>
<td>202,976,401,213</td>
</tr>
<tr>
<td>5</td>
<td>501</td>
<td>14</td>
<td>3,535,017,524,403</td>
</tr>
<tr>
<td>6</td>
<td>4,051</td>
<td>15</td>
<td>65,573,803,186,921</td>
</tr>
<tr>
<td>7</td>
<td>37,633</td>
<td>16</td>
<td>1,290,434,218,669,921</td>
</tr>
<tr>
<td>8</td>
<td>394,353</td>
<td>17</td>
<td>26,846,616,451,246,353</td>
</tr>
<tr>
<td>9</td>
<td>4,596,553</td>
<td>18</td>
<td>588,633,468,315,403,843</td>
</tr>
</tbody>
</table>
Size of Planning Problems vs. State Transition Systems

- We can thus see that planning problems are much more compact representations of state transition systems.
- Compare, e.g., the size of blocksworld domain with \( n = 5 \) blocks (which will have only a few actions) to the state size of \( > 501 \).
- **Exercise!** We will model this simple blocksworld problem!
- We give some details here, but then use an online PDDL (Planning Domain Description Language) editor.

Propositional Model: Required State Variables

- We have 5 blocks called \( A, B, C, D, E \).
- Actions can use the gripper to:
  - take a top-most block from a tower of size \( \geq 2 \), or
  - take a block that lies on the table (tower of size 1).
- Actions can also use the gripper to:
  - place its block onto another top-most block, or
  - place the block in it onto the table.

So, which state variables do we need?

- \( \text{AisTopMost} \), \( \text{BisTopMost} \), etc. – to check whether we can grab it
- \( \text{AonB} \), \( \text{AonC} \), etc. – so we can make the next block top-most
- \( \text{AonTable} \), \( \text{BonTable} \), etc. – for the lowest block in each tower
- \( \text{holdingA} \), \( \text{holdingB} \), etc. – to know what the gripper is holding
- \( \text{gripperFree} \) – so we know whether we can take a block

Propositional Model: Modeling the Stack Actions

- Now we model putting one block on another:
  - Say we have block \( A \) in the gripper.
  - We need support (i.e., an action) for each other block \( b \in \{ B, C, D, E \} \) since that one could be on top.
  - Now let's do it!
    - Open editor.planning.domains
    - Choose File, then Load. Choose groundBlocksworldDomain.pddl from the zip for this course that can be downloaded from my website.
    - Before you do the exercise, take a look at the actions \text{take}-A-from-table and \text{place}-A-on-table.
- **Solution:**
  ```
  (:action stack-A-onto-B
   :precondition (and (holdingA) (BisTopMost))
   :effect (and (not (holdingA)) (gripperFree)
              (AonB) (AisTopMost)
              (not (BisTopMost))))
  ```
Propositional Model: Modeling the Unstack Actions

Now we model removing one block from another:
- Say we want to take block A into the gripper.
- We need support (i.e., an action) for each other block \( b \in \{B, C, D, E\} \) since that one could be beneath A – we need this since we need to state that this one will be at top next.
- Back to editor.planning.domains!

Solution:
\[
(:action \text{unstack}\_A\_from\_B
:precondition (and (gripperFree) (AonB) (AisTopMost)))
:effect (and (not (gripperFree)) (holdingA) (not (AonB)) (not (AisTopMost)) (BisTopMost)))
\]

Lifted Model: A “Lifted” Blocksworld Model

- We have seen that modeling still requires many actions!
  - Each stack and unstack action requires \( n \times (n-1) \) different variants when there are \( n \) blocks! (i.e. \( 5 \times 4 = 20 \) actions just for stack and unstack for \( n = 5 \) blocks).
  - Also the number of existing state variables (defined in the domain file) was quadratic! (36 for \( n = 5 \) blocks)
  - (Although that’s much better than the exponential search space increase (> 501 states for \( n = 5 \)), we can still improve on that!)
- We will now regard lifted planning problems, where one can specify variables. This leads to an even more compact representation! (In general, this gives an exponential size decrease.)

Propositional Model: Modeling the Initial State

We now take a look at the problem definition.

We have 5 blocks called \( A, B, C, D, E \).
- Actions can use the gripper to:
  - take a top-most block from a tower of size \( \geq 2 \), or
  - take a block that lies on the table (tower of size 1).
- Actions can also use the gripper to:
  - place its block onto another top-most block, or
  - place the block in it onto the table.

Which state variables predicates do we need? Let ?b and ?b’ be variables.
- \( \text{topMost(?b)} \) – to check whether we can grab ?b
- \( \text{on(?b, ?b')} \) – so we can make ?b’ the next top-most block
- \( \text{onTable(?b)} \) – for the lowest block in each tower
- \( \text{holding(?b)} \) – to know what the gripper is holding
- \( \text{gripperFree()} \) – so we know whether we can take a block

→ The problem instance lists all blocks as “objects”
Lifted Model: Modeling the (Lifted) Unstack Action

Now we model removing one block from another:
- Say we want to take block $b$ into the gripper.
- We need support (i.e., an action) for each other block $b' \in \{A, B, C, D, E\}$ since that one could be beneath $b$ – we need this since we need to state that $b'$ one will be at top next.
- Again, do it!
  - Choose File, then Load. Choose liftedBlocksworldDomain.pddl.
  - You can again check the syntax by looking at the other actions.

Solution:
```pddl
:action unstack
 :parameters (?b1 ?b2 - block)
 :precondition (and (gripperFree)
 (on ?b1 ?b2) (topMost ?b1))
 :effect (and (not (gripperFree)) (holding ?b1)
 (not (on ?b1 ?b2)) (not (topMost ?b1))
 (topMost ?b2))
```

Lifted Model: Solving Blocksworld

Now, solve it!
- Load the file liftedBlocksworldProblem-Instance1.pddl.
- Use the Solve button and select the right files.

A B C D E

1 (unstack A B)
2 (place-on-table A)
3 (unstack B C)
4 (place-on-table B)
5 (unstack D E)
6 (stack D C)
7 (take-from-table A)
8 (stack A D)
9 (take-from-table E)
10 (stack E B)

Lifted Model: Size of the Lifted Model

How large does this (very compact) model become (now)? ($n$ blocks)

- Propositional model:
  - $O(n^2)$ many actions and state variables.
- Lifted model:
  - Only 4 $\in O(1)$ actions and 5 $\in O(1)$ predicates.
  - $n \in O(n)$ blocks (as a simple list in the problem instance).

$\rightarrow$ Every blocksworld problem can be modeled with just 4 actions and listing the $n$ blocks. (Instead of specifying the state transition system, which grows exponentially.)
Summary: What did we do today?

- We’ve learned the formal foundations of *Classical Planning* problems.
- We’ve learned how they can be modeled using the *Planning Domain Description Language (PDDL)*.
- We took a brief glance at planning.domains, which features (among others) a tool for:
  - Modeling planning problems in PDDL.
  - Running a solver on these models.

Outlook: What *didn’t* we do today?

- There are (so!) many extensions of the classical model, e.g.,
  - Uncertainty! Partial observability and probabilistic effects.
  - Time (how long do actions take, and what happens when?).
  - Resource consumption and production.
  - Complex state trajectory constraints.
  - Hierarchies among the actions. (My main research area!)
- How so actually solve planning problems? (My research area!)
- Complexity analysis: How hard is it to solve a problem? (My favorite research area!)
- So much more, e.g.,
  - Proving unsolvability.
  - Plan explanations or explaining unsolvability.
  - Modeling support.