Planning

Chapter 10
Outline

• Search vs. planning
• Using PDDL for planning
Search vs. planning

- Consider the task get milk, bananas, and a cordless drill

- Problems:
  - Enormous search space
  - Actions are complex objects: They have preconditions and they change the world
  - Simple goal test is inadequate
Search vs. planning

- Consider the task get milk, bananas, and a cordless drill
- Standard search algorithms seem to fail miserably:

```
Start
Go To Pet Store  Go To School  Go To Supermarket
Talk to Parrot  Buy a Dog  Buy Tuna Fish
Buy Arugula  Buy Milk  Go To Class
Go To Sleep  Sit in Chair  Read A Book
Sit Some More  Etc. Etc. ...
```

Problems:
- Enormous search space
- Actions are complex objects: They have preconditions and they change the world
- Simple goal test is inadequate
Search vs. planning

- Consider the task get milk, bananas, and a cordless drill
- Standard search algorithms seem to fail miserably:

  - Problems:
    - *Enormous* search space
    - Actions are complex objects: They have preconditions and they change the world
    - Simple goal test is inadequate
Planning systems do the following:

1. open up action and goal representation to allow selection
2. divide-and-conquer by subgoaling
Search vs. planning contd.

Planning systems do the following:

1. open up action and goal representation to allow selection
2. divide-and-conquer by subgoaling

Compare:

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Planning</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>Data structures</td>
<td>Logical sentences (positive ground literals)</td>
</tr>
<tr>
<td>Actions</td>
<td>Code</td>
<td>Preconditions/outcomes in a schema</td>
</tr>
<tr>
<td>Goal</td>
<td>Test</td>
<td>Logical sentence (conjunction of literals)</td>
</tr>
<tr>
<td>Plan</td>
<td>Sequence from $S_0$</td>
<td>Sequence from $S_0$</td>
</tr>
</tbody>
</table>
PDDL

PDDL (Planning Domain Definition Language) is a planning system derived from STRIPS, which goes back to 1971(!).
PDDL

PDDL (= Planning Domain Definition Language) is a planning system derived from STRIPS, which goes back to 1971(!).

• The world is assumed to be fully observable, deterministic, static and with a single agent.
PDDL

PDDL (≡ Planning Domain Definition Language) is a planning system derived from STRIPS, which goes back to 1971(!).

- The world is assumed to be fully observable, deterministic, static and with a single agent.
  - Describe the state of the world by a set of facts (ground atomic formulas).
  - Facts that can change their truth value are called *fluents*.
PDDL

PDDL (= Planning Domain Definition Language) is a planning system derived from STRIPS, which goes back to 1971(!).

- The world is assumed to be fully observable, deterministic, static and with a single agent.
  - Describe the state of the world by a set of facts (ground atomic formulas).
  - Facts that can change their truth value are called fluents
- Action schemas describe general actions.
PDDL

PDDL (= Planning Domain Definition Language) is a planning system derived from STRIPS, which goes back to 1971(!).

- The world is assumed to be fully observable, deterministic, static and with a single agent.
  - Describe the state of the world by a set of facts (ground atomic formulas).
  - Facts that can change their truth value are called *fluents*
- *Action schemas* describe general actions.
  - Schemas are *instantiated* to specific action instances
  - An action instance transforms the world description.
PDDL

PDDL (= Planning Domain Definition Language) is a planning system derived from STRIPS, which goes back to 1971(!).

- The world is assumed to be fully observable, deterministic, static and with a single agent.
  - Describe the state of the world by a set of facts (ground atomic formulas).
  - Facts that can change their truth value are called *fluents*
- *Action schemas* describe general actions.
  - Schemas are *instantiated* to specific action instances
  - An action instance transforms the world description.
- Given an *initial world description*, find a sequence of action instances that achieves a given *goal*.
- No explicit mention is made of time.
World States

• The world or domain is described as a variable-free set of atomic formulas.

• Example:
  \{ Block(a), Block(b), \ldots, \\
  \hspace{1cm} On(a, b), OnTable(b), \ldots, Clear(c), \ldots \} \\

• Uses \textit{database semantics}: If a fact doesn’t appear in the list, it is assumed to be false.
  • E.g. If \( On(b, c) \) isn’t in the domain description \( \neg On(b, c) \) is assumed to hold.

• Constants are assumed to denote distinct individuals, i.e. \( a \neq b \).
An action schema consists of

- the action name,
- a list of variables used in the schema,
- a precondition, and
- an effect.

**Precondition:** Specifies those conditions that must be met before the operator can be applied.

Given as a conjunction of literals

**Effect:** Defines the effect of the action.

Given as a conjunction of literals

E.g.:

Action (Fly (p, from, to))

**PRECOND:**
At (p, from) \(\land\) Flight (p) \(\land\) Airport (from) \(\land\) Airport (to)

**EFFECT:**
\(\neg\) At (p, from) \(\land\) At (p, to)
Action Schema

- An action schema consists of
  - the action name,
  - a list of variables used in the schema,
  - a precondition, and
  - an effect.

- **Precondition**: Specifies those conditions that must be met before the operator can be applied.
  - Given as a conjunction of literals

E.g.:

Action

(Fly(\(p\), from, to))

PRECOND:

At\((p,\text{from})\) \&\& Flight\((p)\) \&\& Airport\((\text{from})\) \&\& Airport\((\text{to})\)

EFFECT:

¬ At\((p,\text{from})\) \&\& At\((p,\text{to})\)
Action Schema

- An action schema consists of
  - the action name,
  - a list of variables used in the schema,
  - a precondition, and
  - an effect.

- **Precondition**: Specifies those conditions that must be met before the operator can be applied.
  - Given as a conjunction of literals

- **Effect**: Defines the effect of the action.
  - Given as a conjunction of literals

E.g.:

Action

(Fly (p, from, to))

PRECOND:

At (p, from) ∧ Flight (p) ∧ Airport (from) ∧ Airport (to)

EFFECT:

¬ At (p, from) ∧ At (p, to)
Action Schema

- An action schema consists of
  - the action name,
  - a list of variables used in the schema,
  - a precondition, and
  - an effect.

- **Precondition**: Specifies those conditions that must be met before the operator can be applied.
  - Given as a conjunction of literals

- **Effect**: Defines the effect of the action.
  - Given as a conjunction of literals

- E.g.: \( \text{Action}(\text{Fly}(p, \text{from}, \text{to})) \)
  
  **PRECOND:**
  \[
  \text{At}(p, \text{from}) \land \text{Flight}(p) \land \text{Airport}(\text{from}) \land \text{Airport}(\text{to})
  \]

  **EFFECT:** \( \neg \text{At}(p, \text{from}) \land \text{At}(p, \text{to}) \)
PDDL Operators

More examples:

- \textit{Move}(x, y, z): Move \(x\) from being on \(y\) to being on \(z\).
  - PRECOND:

- \textit{Stack}(x, y): Move \(x\) from being on the table to being on \(y\).
  - PRECOND:
  - EFFECT:

- \textit{Unstack}: (Exercise)
More examples:

- $Move(x, y, z)$: Move $x$ from being on $y$ to being on $z$.
  - PRECOND: $On(x, y) \land Clear(x) \land Clear(z)$
  - EFFECT:
More examples:

- $\textit{Move}(x, y, z)$: Move $x$ from being on $y$ to being on $z$.
  - PRECOND: $\text{On}(x, y) \land \text{Clear}(x) \land \text{Clear}(z)$
  - EFFECT: $\text{On}(x, z) \land \text{Clear}(y) \land \neg \text{On}(x, y) \land \neg \text{Clear}(z)$
PDDL Operators

More examples:

- **Move**\((x, y, z)\): Move \(x\) from being on \(y\) to being on \(z\).
  - **PRECOND:** \(\text{On}(x, y) \land \text{Clear}(x) \land \text{Clear}(z)\)
  - **EFFECT:** \(\text{On}(x, z) \land \text{Clear}(y) \land \neg \text{On}(x, y) \land \neg \text{Clear}(z)\)

- **Stack**\((x, y)\): Move \(x\) from being on the table to being on \(y\).
  - **PRECOND:** \(\text{OnTable}(x) \land \text{Clear}(x) \land \text{Clear}(y) \land x \neq y\)
  - **EFFECT:** \(\text{On}(x, y) \land \neg \text{OnTable}(x) \land \neg \text{Clear}(y)\)

(Note: **Unstack** was not included due to it being an exercise.)
PDDL Operators

More examples:

- **Move**(\(x, y, z\)): Move \(x\) from being on \(y\) to being on \(z\).
  - **PRECOND**: \(On(x, y) \land Clear(x) \land Clear(z)\)
  - **EFFECT**: \(On(x, z) \land Clear(y) \land \neg On(x, y) \land \neg Clear(z)\)

- **Stack**(\(x, y\)): Move \(x\) from being on the table to being on \(y\).
  - **PRECOND**: \(OnTable(x) \land Clear(x) \land Clear(y) \land x \neq y\)
  - **EFFECT**: \(On(x, y) \land \neg OnTable(x) \land \neg Clear(y)\)

- **Unstack**: (Exercise)
Planning with PDDL

- The initial state is completely specified
  - i.e. all facts initially true are given.
  - recall: A fact not mentioned is assumed to be false

- For example, put a red block on b:
  - On(x, b) \land \text{Colour}(x, \text{red})
Planning with PDDL

• The initial state is completely specified
  • i.e. all facts initially true are given.
  • recall: A fact not mentioned is assumed to be false
• There is also a \textit{goal} to be achieved.
  • For example, put a red block on \( b \):

\[ \text{On}(x, b) \land \text{Colour\_of}(x, \text{red}). \]
Planning with PDDL

- The initial state is completely specified
  - i.e. all facts initially true are given.
  - recall: A fact not mentioned is assumed to be false
- There is also a *goal* to be achieved.
  - For example, put a red block on $b$:
    \[
    \text{On}(x, b) \land \text{Colour\_of}(x, \text{red}).
    \]
- To establish a goal, a sequence of action instances needs to be found that leads from the initial state to the goal.
Planning with PDDL

- An *action instance* $a$ is an action along with bindings for its free variables.

- E.g. recall the schema:

  $Action(\ Fly(p, \ from, \ to)\ )$

  **PRECOND:**
  
  \[\ At(p, \ from) \land Flight(p) \land Airport(from) \land Airport(to)\]

  **EFFECT:** $\neg At(p, \ from) \land At(p, \ to)$


Planning with PDDL

• An *action instance* \( a \) is an action along with bindings for its free variables.

• E.g. recall the schema:

\[
\text{Action( } \text{Fly}(p, \text{from, to}) \\
\text{PRECOND:} \\
\quad \text{At}(p, \text{from}) \land \text{Flight}(p) \land \text{Airport}(\text{from}) \land \text{Airport}(\text{to}) \\
\text{EFFECT: } \neg \text{At}(p, \text{from}) \land \text{At}(p, \text{to}) \)
\]

This has instance:

\[
\text{Action( } \text{Fly}(AC118, \text{YVR, YYZ}) \\
\text{PRECOND:} \\
\quad \text{At}(AC118, \text{YVR}) \land \text{Flight}(AC118) \land \text{Airport}(\text{YVR}) \land \text{Airport}(\text{YYZ}) \\
\text{EFFECT: } \neg \text{At}(AC118, \text{YVR}) \land \text{At}(AC118, \text{YYZ}) \)
\]
Planning with PDDL

- An action instance $a$ is *possible* in state $s$ iff every precondition in $\text{PRECOND}(a)$ holds in $s$.
- If we describe $s$ by listing those atoms that hold in $s$, then this can be expressed as
  - $\text{PRECOND}^+(a) \subseteq s$
  - $\text{PRECOND}^-(a) \cap s = \emptyset$

  where
  - $\text{PRECOND}^+(a)$ is the set of positive literals and
  - $\text{PRECOND}^-(a)$ is the set of negated literals in the precondition.
- Equivalently, we can write:

  $$\text{PRECOND}(a) \subseteq s \cup \{\neg p \mid p \notin s\}.$$
Planning with PDDL

- Let
  - $ADD(a)$ be the set of positive literals in $EFFECT(a)$ and
  - $DEL(a)$ be the set of atoms given by the negative literals in $EFFECT(a)$.
- The result of executing an action instance $a$ that is possible in $s$ is the state:

$$RESULT(a, s) = (s - DEL(a)) \cup ADD(a).$$
Planning with PDDL

• Given an instantiated action sequence \(a_1, \ldots, a_n\), and a situation \(s\), we set

\[
S_0 = s
\]

and

\[
s_i = \text{RESULT}(a, s_{i-1}) \quad \text{for} \quad i = 1, \ldots, n.
\]

• The action sequence \textit{succeeds} if every individual action succeeds.

• The action sequence \textit{achieves} the goal \(G\) if \(s_n\) entails \(G\).
Planning with PDDL

- Planning can be done in either a “forward” or “backward” manner.
- Known as *progressive* and *regressive* planning respectively.
- Originally regressive planners were most used, due to their focus on the goal.
- With better heuristics and increased computational power, progressive planners have come to dominate.
Progressive Planning in PDDL

- The most intuitive way to try to obtain a plan is to:
  - begin at the initial state and
  - find a sequence of actions that lead to the goal.

- This is called a *progressive planner* since it progresses the initial state forward until a state satisfying the goal is found.
Progressive Planning

Depth-First Progressive Planner:

Input: A world description $S$ and goal formula $Goal$
Output: A plan or $fail$

ProgPlan($S$, $Goal$)
  if $Goal \subseteq S$ then return empty plan
  for each operator instance $\langle Act, Pre, Add, Del \rangle$
    such that $S$ satisfies $Pre$ do {
      let $S' = (S \setminus Del) \cup Add$
      let $Plan = \text{ProgPlan}(S', Goal)$
      if $Plan \neq fail$ then return $Plan \cdot Act$
    }
  return fail
Goal: Get some box into the office
Example

Initial world DB:

\[ \text{Box}(\text{box1}), \text{Box}(\text{box2}), \]
\[ \text{InRoom}(\text{box1, supplies}), \text{InRoom}(\text{box2, closet}), \]
\[ \text{InRoom}(\text{robot, office}), \]
\[ \text{Connected}(\text{office, supplies}), \text{Connected}(\text{supplies, office}), \]
\[ \text{Connected}(\text{closet, supplies}), \text{Connected}(\text{supplies, closet}) \]
Example

Action schema:

\[ \text{goThru}(r_1, r_2) \]
- **PRECOND**: \( \text{InRoom}(\text{robot}, r_1), \text{Connected}(r_1, r_2) \)
- **EFFECT**: \( \text{InRoom}(\text{robot}, r_2), \neg\text{InRoom}(\text{robot}, r_1) \),

\[ \text{pushThru}(x, r_1, r_2) \]
- **PRECOND**: \( \text{InRoom}(\text{robot}, r_1), \text{InRoom}(x, r_1), \text{Connected}(r_1, r_2) \)
- **EFFECT**: \( \text{InRoom}(\text{robot}, r_2), \text{InRoom}(x, r_2), \neg\text{InRoom}(\text{robot}, r_1), \neg\text{InRoom}(x, r_1) \)
Progressive Planning Example

With \textit{goThru}(office, supplies), obtain first progressed DB:

\textit{Box}(box1), \textit{Box}(box2),
\textit{InRoom}(box1, supplies), \textit{InRoom}(box2, closet),
\textit{InRoom}(robot, supplies),
\textit{Connected}(office, supplies), \textit{Connected}(supplies, office),
\textit{Connected}(closet, supplies), \textit{Connected}(supplies, closet)

With \textit{pushThru}(box1, supplies, office), obtain the DB:

\textit{Box}(box1), \textit{Box}(box2),
\textit{InRoom}(box1, office), \textit{InRoom}(box2, closet),
\textit{InRoom}(robot, office),
\textit{Connected}(office, supplies), \textit{Connected}(supplies, office),
\textit{Connected}(closet, supplies), \textit{Connected}(supplies, closet)
Regressive Planning with PDDL

- **Idea**: Begin with the goal state, and work backwards to try to get to the initial state.
- The *search space* can be defined in a “backwards chaining” fashion:
  - **Idea**: Work backwards, repeatedly simplifying the goal until we get a goal satisfied in the initial state.
  - Called *goal regression*
Regressive Planning

Depth-First Regressive Planner:

Input: The initial world description \( \text{Init} \) and a goal formula \( \text{Goal} \)
Output: A plan or \( \text{fail} \)

\[
\text{RegrPlan}(\text{Init}, \text{Goal})
\]
\[
\begin{align*}
\text{if } \text{Goal} & \subseteq \text{Init} \text{ then return empty plan} \\
\text{for each } & \text{ operator instance } \langle \text{Act}, \text{Pre}, \text{Add}, \text{Del} \rangle \\
& \text{ such that } \text{Del} \cap \text{Goal} = \emptyset \{ \\
& \text{let } \text{Goal}' = (\text{Goal} \cup \text{Pre}) \setminus \text{Add} \\
& \text{let } \text{Plan} = \text{RegrPlan}(\text{Init}, \text{Goal}') \\
& \text{if Plan } \neq \text{ fail then return Plan} \cdot \text{Act} \\
\} \\
\text{return fail}
\end{align*}
\]
Regressive Planning Example

- Planner is called with the initial world DB and the goal:
  \[ \text{Box}(x), \text{InRoom}(x, \text{office}) \]
- The goal is not satisfied by the initial world DB.
- The action instance
  \[ \text{pushThru}(\text{box1}, \text{supplies}, \text{office}) \]
  has a delete list that does not intersect with the goal.
- Get regressed subgoal:
  \[ \text{Box(\text{box1})}, \text{InRoom(\text{robot}, \text{supplies})}, \text{InRoom(\text{box1}, \text{supplies})}, \]
  \[ \text{Connected(\text{supplies}, \text{office})} \]
- The action instance:
  \[ \text{goThru(office, supplies)} \]
  yields the regressed goal:
  \[ \text{Box(\text{box1})}, \text{InRoom(\text{robot, office})}, \text{InRoom(\text{box1}, \text{supplies})}, \]
  \[ \text{Connected(\text{supplies}, \text{office})}, \text{Connected(office, supplies)} \]
- This is satisfied in the initial state.
Regressive Planning: Another Example

- A, B, and C are on the table.
- The goal is $\text{On}(A, B)$ and $\text{On}(B, C)$.
- Initial state:
  \[
  \text{Init} = \{ \text{OnTable}(A), \text{OnTable}(B), \text{OnTable}(C), \\
  \text{Clear}(A), \text{Clear}(B), \text{Clear}(C) \}\]
Regressive Planning: Another Example

- $A$, $B$, and $C$ are on the table.
- The goal is $On(A, B)$ and $On(B, C)$.
- Initial state:
  
  $Init = \{OnTable(A), OnTable(B), OnTable(C), Clear(A), Clear(B), Clear(C)\}$

- Initial call: $RegrPlan(Init, \{On(A, B), On(B, C)\})$
Regressive Planning: Another Example

- $A$, $B$, and $C$ are on the table.
- The goal is $On(A, B)$ and $On(B, C)$.
- Initial state:
  \[ \text{Init} = \{ \text{OnTable}(A), \text{OnTable}(B), \text{OnTable}(C), \text{Clear}(A), \text{Clear}(B), \text{Clear}(C) \} \]
- Initial call: $\text{RegrPlan(Init, \{ On(A, B), On(B, C) \})}$
- Action instance: $\text{Stack}(A, B)$
Regressive Planning: Another Example

- $A$, $B$, and $C$ are on the table.
- The goal is $\text{On}(A, B)$ and $\text{On}(B, C)$.
- Initial state:
  \[
  \text{Init} = \{ \text{OnTable}(A), \text{OnTable}(B), \text{OnTable}(C), \\
  \text{Clear}(A), \text{Clear}(B), \text{Clear}(C) \} 
  \]
- Initial call: $\text{RegrPlan}(\text{Init}, \{ \text{On}(A, B), \text{On}(B, C) \})$
- Action instance: $\text{Stack}(A, B)$
- Call:
  \[
  \text{RegrPlan}(\text{Init}, \{ \text{OnTable}(A), \text{Clear}(A), \text{Clear}(B), \text{On}(B, C) \}) 
  \]
Regressive Planning: Another Example

- $A$, $B$, and $C$ are on the table.
- The goal is $On(A, B)$ and $On(B, C)$.
- Initial state:
  
  $Init = \{ OnTable(A), OnTable(B), OnTable(C),
  \text{Clear}(A), \text{Clear}(B), \text{Clear}(C) \}$

- Initial call: $RegrPlan(Init, \{On(A, B), On(B, C)\})$

- Action instance: $Stack(A, B)$

- Call:
  
  $RegrPlan(Init, \{ OnTable(A), \text{Clear}(A), \text{Clear}(B), On(B, C) \})$

- Action instance: $Stack(B, C)$
Regressive Planning: Another Example

- A, B, and C are on the table.
- The goal is $On(A, B)$ and $On(B, C)$.
- Initial state:
  
  $Init = \{OnTable(A), OnTable(B), OnTable(C),
  \quad Clear(A), Clear(B), Clear(C)\}$

- Initial call: $RegrPlan(Init, \{On(A, B), On(B, C)\})$

- Action instance: $Stack(A, B)$

- Call:
  
  $RegrPlan(Init, \{OnTable(A), Clear(A), Clear(B), On(B, C)\})$

- Action instance: $Stack(B, C)$

- Call: $RegrPlan(Init, \{OnTable(A), Clear(A), Clear(B), OnTable(B), Clear(C)\})$
Regressive Planning: Another Example

• A, B, and C are on the table.
• The goal is $On(A, B)$ and $On(B, C)$.
• Initial state:
  $$Init = \{ OnTable(A), OnTable(B), OnTable(C), \]
  $$ \quad \quad \quad Clear(A), Clear(B), Clear(C) \}$$
• Initial call: $RegrPlan(Init, \{ On(A, B), On(B, C) \})$
• Action instance: $Stack(A, B)$
• Call:
  $$RegrPlan(Init, \{ OnTable(A), Clear(A), Clear(B), On(B, C) \})$$
• Action instance: $Stack(B, C)$
• Call: $RegrPlan(Init, \{ OnTable(A), Clear(A), Clear(B), OnTable(B), Clear(C) \})$
  which is satisfied in the initial state.
Heuristics for Planning

- Neither forward nor backward search is efficient without a good heuristic.
- Recall: finding an admissable heuristic via defining a relaxed problem.
Heuristics for Planning

- Neither forward nor backward search is efficient without a good heuristic.
- Recall: finding an admissible heuristic via defining a relaxed problem.
- Heuristics:
  - Ignore some or all of the preconditions
  - Ignore the delete list

Problem:
- The simplified planning problem is still NP-hard
- Resolve by using a greedy algorithm
- Also: domain-specific heuristics
Heuristics for Planning

- Neither forward nor backward search is efficient without a good heuristic.
- Recall: finding an *admissable heuristic* via defining a *relaxed problem*.
- Heuristics:
  - Ignore some or all of the preconditions
  - Ignore the delete list
- Problem:
  - The simplified planning problem is still NP-hard
  - Resolve by using a greedy algorithm
Heuristics for Planning

- Neither forward nor backward search is efficient without a good heuristic.
- Recall: finding an admissible heuristic via defining a relaxed problem.
- Heuristics:
  - Ignore some or all of the preconditions
  - Ignore the delete list
- Problem:
  - The simplified planning problem is still NP-hard
  - Resolve by using a greedy algorithm
- Also: domain-specific heuristics
Heuristics for Planning

Other possibilities:

- **State abstraction**: Combine states by ignoring some fluents
- **Problem decomposition**:
  - Divide a problem into parts;
  - solve each part independently;
  - combine the parts

Other types of planners:

- Partial-order planners
- GRAPHPLAN
Heuristics for Planning

Other possibilities:

- State abstraction: Combine states by ignoring some fluents
- Problem decomposition:
  - Divide a problem into parts;
  - solve each part independently;
  - combine the parts

Other types of planners:

- Partial-order planners
- GRAPHPLAN
PDDL: Summary

- Very successful, mainly because it is simple (basically STRIPS).

- Very limited representation language:
  1. All information must be specified.
  2. Actions with state-dependent effects must be split. E.g., a move doesn't change the colour of an object usually, but it does if an object moves into the path of a spray gun.
  3. We can't reason about actions.
  5. Offline. No sensing.
  6. No concurrency, non-determinism.

- General planning comment: Things get tricky very quickly. E.g: On(B, table), On(C, A), Goal: On(A, B), On(B, C).
PDDL: Summary

- Very successful, mainly because it is simple (basically STRIPS).
- Very limited representation language:
  1. All information must be specified.
  2. Actions with state-dependent effects must be split.
    - E.g., a move doesn’t change the colour of an object usually, but it does if an object moves into the path of a spray gun.
    - In PDDL need actions move object into path of spray gun and move object elsewhere.
  3. We can’t reason about actions.
  5. Offline. No sensing.
  6. No concurrency, non-determinism.
- General planning comment: Things get tricky very quickly.
  - E.g: On (B, table), On (C, A), Goal: On (A, B), On (B, C).
PDDL: Summary

- Very successful, mainly because it is simple (basically STRIPS).
- Very limited representation language:
  1. All information must be specified.
  2. Actions with state-dependent effects must be split.
     - E.g., a *move* doesn’t change the colour of an object usually, but it does if an object moves into the path of a spray gun.
     - In PDDL need actions \textit{move\_object\_into\_path\_of\_spray\_gun} and \textit{move\_object\_elsewhere}.
PDDL: Summary

- Very successful, mainly because it is simple (basically STRIPS).
- Very limited representation language:
  1. All information must be specified.
  2. Actions with state-dependent effects must be split.
     - E.g., a *move* doesn’t change the colour of an object usually, but it does if an object moves into the path of a spray gun.
     - In PDDL need actions *move_object_into_path_of_spray_gun* and *move_object_elsewhere*.
  3. We can’t reason about actions.
PDDL: Summary

• Very successful, mainly because it is simple (basically STRIPS).

• Very limited representation language:
  1. All information must be specified.
  2. Actions with state-dependent effects must be split.
      • E.g., a move doesn’t change the colour of an object usually, but it does if an object moves into the path of a spray gun.
      • In PDDL need actions move_object_into_path_of_spray_gun and move_object_elsewhere.
  3. We can’t reason about actions.
PDDL: Summary

- Very successful, mainly because it is simple (basically STRIPS).
- Very limited representation language:
  1. All information must be specified.
  2. Actions with state-dependent effects must be split.
     - E.g., a *move* doesn’t change the colour of an object usually, but it does if an object moves into the path of a spray gun.
     - In PDDL need actions *move_object_into_path_of_spray_gun* and *move_object_elsewhere*.
  3. We can’t reason *about* actions.
  5. Offline. No sensing.
PDDL: Summary

• Very successful, mainly because it is simple (basically STRIPS).

• Very limited representation language:
  1. All information must be specified.
  2. Actions with state-dependent effects must be split.
     • E.g., a move doesn’t change the colour of an object usually, but it does if an object moves into the path of a spray gun.
     • In PDDL need actions move_object_into_path_of_spray_gun and move_object_elsewhere.
  3. We can’t reason about actions.
  5. Offline. No sensing.
  6. No concurrency, non-determinism.
PDDL: Summary

• Very successful, mainly because it is simple (basically STRIPS).

• Very limited representation language:
  1. All information must be specified.
  2. Actions with state-dependent effects must be split.
     • E.g., a *move* doesn’t change the colour of an object usually, but it does if an object moves into the path of a spray gun.
     • In PDDL need actions *move_object_into_path_of_spray-gun* and *move_object_elsewhere*.
  3. We can’t reason *about* actions.
  5. Offline. No sensing.
  6. No concurrency, non-determinism.

• General planning comment: Things get tricky very quickly.
  • E.g: *On(B, table), On(C, A)*, Goal: *On(A, B), On(B, C).*