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:

![Diagram showing various activities and their connections]
Search vs. planning

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

![Diagram of task sequence]

- Problems:
  - *Enormous* search space
  - Actions are complex objects:
    - They have preconditions and they change the world
  - Simple goal test is inadequate
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
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 (= 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 positive facts (ground atomic formulas – like a relational database).
  - 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 positive facts (ground atomic formulas – like a relational database).
  - 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 positive facts (ground atomic formulas – like a relational database).
  - 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 positive facts (ground atomic formulas – like a relational database).
  - 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:
  \{ \text{Block}(a), \text{Block}(b), \ldots, \\
  \text{On}(a, b), \text{OnTable}(b), \ldots, \text{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 \( \text{On}(b, c) \) isn’t in the domain description \( \neg \text{On}(b, c) \) is assumed to hold.

• Constants are assumed to denote distinct individuals, i.e.
  \( a \neq b \).
Action Schema

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

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

  \[
  \text{Action}\left(\text{Fly}\left(p, \text{from}, \text{to}\right)\right)
  \]

  \[
  \text{PRECOND: } \text{At}\left(p, \text{from}\right) \land \text{Flight}\left(p\right) \land \text{Airport}\left(\text{from}\right) \land \text{Airport}\left(\text{to}\right)
  \]

  \[
  \text{EFFECT: } \neg \text{At}\left(p, \text{from}\right) \land \text{At}\left(p, \text{to}\right)
  \]
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.:  Action( Fly(p, from, to) 
  PRECOND: 
  At(p, from) ∧ Flight(p) ∧ Airport(from) ∧ Airport(to) 
  EFFECT: ¬At(p, from) ∧ At(p, to)  )
PDDL Operators

More examples:

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

- **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**: 

- **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: $On(x, z) \land Clear(y) \land \neg On(x, y) \land \neg Clear(z)$
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:** \(On(x, y) \land \neg OnTable(x) \land \neg Clear(y)\)
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: $\text{On}(x, y) \land \neg \text{OnTable}(x) \land \neg \text{Clear}(y)$

- **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
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 $b$:

$$\text{On}(x, b) \land \text{Colour\_of}(x, red).$$
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 $b$:
    
    $$On(x, b) \land Colour\_of(x, red).$$
- 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:
  
  \[
  \text{Action}(\text{Fly}(p, \text{from}, \text{to}))
  \]
  
  PRECOND:
  
  \[
  \text{At}(p, \text{from}) \land \text{Flight}(p) \land \text{Airport}(\text{from}) \land \text{Airport}(\text{to})
  \]
  
  EFFECT: \(\neg \text{At}(p, \text{from}) \land \text{At}(p, \text{to})\)
Planning with PDDL

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

- E.g. recall the schema:

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

This has instance:

\[
\text{Action( Fly(AC118, YVR, YYZ) }
\]
\[
\text{PRECOND: } \quad \text{At(AC118, YVR)} \land \text{Flight(AC118)} \land \text{Airport(YVR)} \land \text{Airport(YYZ)}
\]
\[
\text{EFFECT: } \neg \text{At(AC118, YVR)} \land \text{At(AC118, YYZ)}
\]
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:
  $$\text{PRECOND}(a) \subseteq s \cup \{\neg p \mid p \not\in s\}.$$
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
  \[ s_i = RESULT(a, s_{i-1}) \text{ for } i = 1, \ldots, n. \]

- The action sequence succeeds if every individual action succeeds.
- The action sequence 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 $Plan = ProgPlan(S', Goal)$
            if $Plan \neq fail$ then return $Plan \cdot Act$
        }
    return $fail$
Goal: Get some box into the office
Initial world DB:

\[ \text{Box}(box1), \text{Box}(box2), \]

\[ \text{InRoom}(box1, supplies), \text{InRoom}(box2, closet), \]

\[ \text{InRoom}(robot, office), \]

\[ \text{Connected}(office, supplies), \text{Connected}(supplies, office), \]

\[ \text{Connected}(closet, supplies), \text{Connected}(supplies, closet) \]
Example

Action schema:

\texttt{goThru}(r1, r2)

- PRECOND: \texttt{InRoom(robot, r1)}, \texttt{Connected(r1, r2)}
- EFFECT: \texttt{InRoom(robot, r2)}, \neg \texttt{InRoom(robot, r1)}, \neg \texttt{InRoom(x, r1)}

\texttt{pushThru}(x, r1, r2)

- PRECOND: \texttt{InRoom(robot, r1)}, \texttt{InRoom(x, r1)}, \texttt{Connected(r1, r2)}
- EFFECT: \texttt{InRoom(robot, r2)}, \texttt{InRoom(x, r2)}, \neg \texttt{InRoom(robot, r1)}, \neg \texttt{InRoom(x, r1)}
Progressive Planning Example

With $goThru(office, supplies)$, obtain first progressed DB:

- $Box(box1), Box(box2),$
- $InRoom(box1, supplies), InRoom(box2, closet),$
  $InRoom(robot, supplies),$
- $Connected(office, supplies), Connected(supplies, office),$
  $Connected(closet, supplies), Connected(supplies, closet)\$

With $pushThru(box1, supplies, office)$, obtain the DB:

- $Box(box1), Box(box2),$
- $InRoom(box1, office), InRoom(box2, closet),$
  $InRoom(robot, office),$
- $Connected(office, supplies), Connected(supplies, office),$
  $Connected(closet, supplies), Connected(supplies, closet)$
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 $Init$ and a goal formula $Goal$

Output: A plan or fail

$RegrPlan(Init, Goal)$

if $Goal \subseteq Init$ then return empty plan

for each operator instance $\langle Act, Pre, Add, Del \rangle$
such that $Del \cap Goal = \emptyset$ {

let $Goal' = (Goal \cup Pre) \setminus Add$

let $Plan = RegrPlan(Init, Goal')$

if $Plan \neq fail$ then return $Plan \cdot Act$

}

return fail
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
  \[ \text{pushThru}(\text{box}1, \text{supplies}, \text{office}) \]
  has a delete list that does not intersect with the goal.
- Get regressed subgoal:
  \[ \text{Box}(\text{box}1), \text{InRoom}(\text{robot, supplies}), \text{InRoom}(\text{box}1, \text{supplies}), \text{Connected}(\text{supplies, office}) \]
- The action instance: \[ \text{goThru}(\text{office, supplies}) \]
yields the regressed goal:
  \[ \text{Box}(\text{box}1), \text{InRoom}(\text{robot, office}), \text{InRoom}(\text{box}1, \text{supplies}), \text{Connected}(\text{supplies, office}), \text{Connected}(\text{office, supplies}) \]
- This is satisfied in the initial state.
Regressive Planning: Another Example

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

- Neither forward nor backward search is efficient without a good heuristic.
- Recall: finding an \textit{admissible heuristic} via defining a \textit{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
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
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 simplicity, limited representation, single agent, offline, no concurrency.
PDDL: Summary

- Very successful, mainly because it is simple (basically STRIPS).
- Very limited representation language:
  1. All information must be specified.

  - 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.
  - We can't reason about actions.
  - Single agent. No exogenous actions.
  - Offline. No sensing.
  - 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`.
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).