Principles of AI Planning
13. Planning as search: Partial-Order Reduction

Motivation

- **Worst case**: Heuristic search may explore exponentially more states than necessary, even if heuristic is almost perfect (Helmert and Röger, 2008).
- **Example**: A* search in Gripper domain explores all permutations of ball transportations if heuristic is off only by a small constant.
- **Idea**: Complement heuristic search with orthogonal technique(s) to reduce size of explored state space.
- **Desired properties of this technique**: preservation of completeness and, if possible, optimality.

Partial-Order Reduction

**Idea:**
- Enforce particular ordering among operators.
- Ignore all other orderings.

**Example**

```
put-on-left-shoe
put-on-right-shoe
```

```
put-on-right-shoe
```

```
put-on-left-shoe
```
Preliminaries

Basic Definitions

Definition (Operators)
Let \( \Pi = (V, I, O, \gamma) \) be a SAS\(^+\) planning task and \( o \in O \) an operator. Then

- \( \text{prevars}(o) := \text{vars}(\text{pre}(o)) \) are the variables that occur in the precondition of \( o \).
- \( \text{effvars}(o) := \text{vars}(\text{eff}(o)) \) are the variables that occur in the effect of \( o \).
- \( o \) reads \( v \in V \) iff \( v \in \text{prevars}(o) \).
- \( o \) modifies \( v \in V \) iff \( v \in \text{effvars}(o) \).

Variable \( v \in V \) is goal-related iff \( v \in \text{vars}(\gamma) \).

Assumption: \( \text{effvars}(o) \neq \emptyset \) for all \( o \in O \).

Operator Dependencies

Definition (Operator dependencies)
Let \( \Pi = (V, I, O, \gamma) \) be a planning task and \( o, o' \in O \).

1. \( o \) disables \( o' \) iff there exists \( v \in \text{effvars}(o) \cap \text{prevars}(o') \) such that \( \text{eff}(o)(v) \neq \text{pre}(o')(v) \).
2. \( o \) enables \( o' \) iff there exists \( v \in \text{effvars}(o) \cap \text{prevars}(o') \) such that \( \text{eff}(o)(v) = \text{pre}(o')(v) \).
3. \( o \) and \( o' \) conflict iff there is \( v \in \text{effvars}(o) \cap \text{effvars}(o') \) such that \( \text{eff}(o)(v) \neq \text{eff}(o')(v) \).
4. \( o \) and \( o' \) interfere iff \( o \) disables \( o' \), or \( o' \) disables \( o \), or \( o \) and \( o' \) conflict.
5. \( o \) and \( o' \) are commutative iff \( o \) and \( o' \) do not interfere, and neither \( o \) enables \( o' \), nor \( o' \) enables \( o \).
Motivation

Preliminaries

Setting

Operator Dependencies

Necessary Enabling Sets and Disjunctive Action Landmarks

Stubborn Sets

Conclusion

Operator Dependencies

Example

\[
\begin{align*}
\text{put-on-left} &= \langle \text{pos} = \text{home} \land \text{left} = f, \text{left} := t \rangle \\
\text{put-on-right} &= \langle \text{pos} = \text{home} \land \text{right} = f, \text{right} := t \rangle \\
\text{go-to-uni} &= \langle \text{left} = t \land \text{right} = t, \text{pos} := \text{uni} \rangle \\
\text{go-to-gym} &= \langle \text{left} = t \land \text{right} = t, \text{pos} := \text{gym} \rangle
\end{align*}
\]

Then:

- \text{go-to-uni} and \text{go-to-gym} disable \text{put-on-left} and \text{put-on-right}.
- \text{put-on-left} and \text{put-on-right} enable \text{go-to-uni} and \text{go-to-gym}.
- \text{go-to-uni} and \text{go-to-gym} conflict.
- \text{put-on-left} and \text{put-on-right} are commutative.

Necessary Enabling Sets and Disjunctive Action Landmarks

Definition (Necessary enabling set)

Let \( \Pi = \langle V, I, O, \gamma \rangle \) be a planning task, \( s \) a state, and \( o \in O \) an operator that is not applicable in \( s \). A set \( N \) of operators is a necessary enabling set (NES) for \( o \) in \( s \) if all operator sequences that lead from \( s \) to a goal state and include \( o \) contain an operator in \( N \) before the first occurrence of \( o \).

Note: NESs not uniquely determined for given \( o \) and \( s \). (E.g., supersets of NESs are still NESs.)

Proof

Let \( L \) be such a disjunctive action landmark.

Then each operator sequence that leads from \( s \) to a state satisfying \( \text{pre}(o) \) contains some operator in \( L \).

Thus, each operator sequence that leads from \( s \) to a goal state and includes \( o \) contains an operator in \( L \) before the first occurrence of \( o \).

Therefore, \( L \) is an NES for \( o \) in \( s \).
Stubborn Sets

Back to the motivation:
If, in state \( s \), some set of operators can be applied in any order and the order does not matter, we want to commit to one such order and ignore all other orders.

Idea:
Identify operators that can be postponed since they are independent of all operators that are not postponed.

E.g., put-on-right could be postponed, since it is independent of put-on-left (that is not postponed).

Idea (more precisely): Identify operators that should not be postponed, and postpone the rest.

Question: When should an operator \( o \) not be postponed?

Answer:
1. **Base case**: If \( o \) may be immediately relevant to reaching (part of) the goal, or
2. **Inductive case I**: If \( o \) may be immediately relevant to contributing to making another operator applicable that should not be postponed, or
3. **Inductive case II**: If \( o \) might not be applicable any more if we postponed it, or if its effect might conflict with the effect of another operator that should not be postponed (\( \approx o \) interferes with such an operator).

Let’s formalize the above answer:

**Definition (Strong stubborn set)**
Let \( \Pi = (V, I, O, \gamma) \) be a planning task and \( s \) a state. A set \( T_s \subseteq O \) is a **strong stubborn set** in \( s \) if

1. \( T_s \) contains a disjunctive action landmark in \( s \), and
2. for all \( o \in T_s \) that are not applicable in \( s \), \( T_s \) contains a necessary enabling set for \( o \) and \( s \), and
3. for all \( o \in T_s \) that are applicable in \( s \), \( T_s \) contains all operators that interfere with \( o \).

Instead of applying all applicable operators in \( s \) only apply those that are applicable and contained in \( T_s \).
Strong Stubborn Sets

Example

\[ \Pi = (V, I, O, \gamma) \]

\[ I = \{ \text{pos} \mapsto \text{home}, \text{left} \mapsto f, \text{right} \mapsto f \}, \quad \gamma = \{ \text{pos} \mapsto \text{uni} \} \]

- put-on-left = (pos = home \land left = f, left := t)
- put-on-right = (pos = home \land right = f, right := t)
- go-to-uni = (left = t \land right = t, pos := uni)

- Step 1: DAL in \( I \) is \( \{ \text{go-to-uni} \} \) \implies \( T_s := \{ \text{go-to-uni} \} \).
- Step 2: go-to-uni not applicable in \( I \). One possible NES for go-to-uni in \( I \) is \( \{ \text{put-on-left} \} \) \implies \( T_s := T_s \cup \{ \text{put-on-left} \} \).
- Step 3: put-on-left is applicable in \( I \). The only operator that interferes with it, go-to-uni, is already in \( T_s \).
- Hence, \( T_s = \{ \text{go-to-uni}, \text{put-on-left} \} \), and \( T_s \) restricted to the applicable operators is \( \{ \text{put-on-left} \} \). During search, only apply put-on-left (not put-on-right).

Question: Can we do better than that in this example?

Domain Transition Graphs

Definition (Domain transition graph)

Let \( \Pi = (V, I, O, \gamma) \) be a SAS* planning task and \( v \in V \). The domain transition graph for \( v \) is the directed graph \( DTG(v) = (\mathcal{D}_v, E) \) where \((d, d') \in E\) iff there is an operator \( o \in O \) with

- \( \text{eff}(o)(v) = d' \), and
- \( v \notin \text{prevars}(o) \) or \( \text{pre}(o)(v) = d \).

**Motivation**

**Preliminaries**

**Stubborn Sets**

**Strong Stubborn Sets**

**Active Operators**

**Properties of Stubborn Sets**

**Some Experiments**

**Conclusion**
Active Operators

Definition (Active operators)

Let \( \Pi = \langle V, I, O, \gamma \rangle \) be a planning task and let \( s \) be a state. The set of active operators \( \text{Act}(s) \subseteq O \) in \( s \) is defined as the set of operators such that for all \( o \in \text{Act}(s) \):

- For every variable \( v \in \text{prevars}(o) \), there is a path in \( DTG(v) \) from \( s(v) \) to \( \text{pre}(o)(v) \). If \( v \) is goal-related, then there is also a path from \( \text{pre}(o)(v) \) to the goal value \( \gamma(v) \).
- For every goal-related variable \( v \in \text{effvars}(o) \), there is a path in \( DTG(v) \) from \( \text{eff}(o)(v) \) to the goal value \( \gamma(v) \).

Remark 1: Even when excluding inactive operators, this preserves completeness and even optimality of a search algorithm (see proof below).

Remark 2: Excluding inactive operators can “cascade” in the sense that additional active operators need not be considered.

Active Operators

Proposition

1. \( \text{Act}(s) \) can be identified efficiently for a given state \( s \) by considering paths in the projection of \( \Pi \) onto \( v \).
2. Operators not in \( \text{Act}(s) \) can be treated as nonexistent when reasoning about \( s \) because they are not applicable in all states reachable from \( s \), or they lead to a dead-end from \( s \).

Proof

1. Homework: Specify efficient algorithm for identification of \( \text{Act}(s) \).
2. Obvious.

Strong Stubborn Sets

Definition (Strong stubborn set with active operator pruning)

Let \( \Pi = \langle V, I, O, \gamma \rangle \) be a planning task and \( s \) a state. A set \( T_s \subseteq O \) is a strong stubborn set in \( s \) if

1. \( T_s \) contains a disjunctive action landmark in \( s \), and
2. for all \( o \in T_s \) that are not applicable in \( s \), \( T_s \) contains a necessary enabling set for \( o \) and \( s \), and
3. for all \( o \in T_s \) that are applicable in \( s \), \( T_s \) contains all operators that are active in \( s \) and interfere with \( o \).

Instead of applying all applicable operators in \( s \) only apply those that are applicable and contained in \( T_s \).
Recall the previous example where strong stubborn sets without active operator pruning were useless.

**Example**
- \( I = \{ u_1 \mapsto 0, u_2 \mapsto 0, v \mapsto 0, w \mapsto 0 \} \),
- \( \gamma = \{ v \mapsto 0, u_1 \mapsto 1, u_2 \mapsto 1 \} \)
- \( o_1 = \{ u_1 = 0, u_1 = 1 \land w = 2 \} \)
- \( o_2 = \{ u_2 = 0, u_2 = 1 \land w = 2 \} \)
- \( o_3 = \{ u_1 = 0 \land u_2 = 0, v = 1 \land w = 1 \} \)

Now, with active operator pruning:
- Step 1: Include \( o_1 \) (or \( o_2 \)) in \( T_s \) as DAL.
- Step 2: Operator \( o_3 \) is not active in any reachable state.
  \( \leadsto o_3 \) not in \( T_s \), although it interferes with \( o_1 \) (or \( o_2 \)).

**Example (Example, ctd.)**

Now, with active operator pruning:
- Step 1: Include \( o_1 \) (or \( o_2 \)) in \( T_s \) as DAL.
- Step 2: Operator \( o_3 \) is not active in any reachable state.
  \( \leadsto o_3 \) not in \( T_s \), although it interferes with \( o_1 \) (or \( o_2 \)).

**Weak Stubborn Sets**

With **weak** stubborn sets, some operators that disable an operator in \( T_s \) need not be included in \( T_s \).

Therefore, weak stubborn sets potentially allow more pruning than strong stubborn sets.

**Definition (Weak stubborn set)**

Let \( \Pi = \langle V, I, O, \gamma \rangle \) be a planning task and \( s \) a state. A set \( T_s \subseteq O \) is a **weak stubborn set in** \( s \) if
1. \( T_s \) contains a disjunctive action landmark in \( s \), and
2. for all \( o \in T_s \) that are not applicable in \( s \), \( T_s \) contains a necessary enabling set for \( o \) and \( s \), and
3. for all \( o \in T_s \) that are applicable in \( s \), \( T_s \) contains the active operators in \( s \) that have conflicting effects with \( o \) or that are disabled by \( o \).

**Theorem**

In the best case, weak stubborn sets admit exponentially more pruning than strong stubborn sets.

**Proof**

Homework.

**Strong Stubborn Sets**

Why operator activity matters

For weak stubborn sets, it suffices to include active operators \( o' \) that are disabled or conflict with applicable operators \( o \subseteq T_s \).

However, \( o' \) does not need to be included if \( o' \) disables an applicable operator \( o \subseteq T_s \).

No computational overhead of computing weak stubborn sets over computing strong stubborn sets.

**Example (Example, ctd.)**

Now, with active operator pruning:
- Step 1: Include \( o_1 \) (or \( o_2 \)) in \( T_s \) as DAL.
- Step 2: Operator \( o_3 \) is not active in any reachable state.
  \( \leadsto o_3 \) not in \( T_s \), although it interferes with \( o_1 \) (or \( o_2 \)).
- Hence, e.g., \( T_s = \{ o_1 \} \) strong stubborn set (with active operator pruning) in \( I \).
- Even active operator \( o_2 \) is not included in \( T_s = \{ o_1 \} \).
  \( \leadsto \) some pruning occurs.

**Weak Stubborn Sets**

With **weak** stubborn sets, some operators that disable an operator in \( T_s \) need not be included in \( T_s \).

Therefore, weak stubborn sets potentially allow more pruning than strong stubborn sets.

**Definition (Weak stubborn set)**

Let \( \Pi = \langle V, I, O, \gamma \rangle \) be a planning task and \( s \) a state. A set \( T_s \subseteq O \) is a **weak stubborn set in** \( s \) if
1. \( T_s \) contains a disjunctive action landmark in \( s \), and
2. for all \( o \in T_s \) that are not applicable in \( s \), \( T_s \) contains a necessary enabling set for \( o \) and \( s \), and
3. for all \( o \in T_s \) that are applicable in \( s \), \( T_s \) contains the active operators in \( s \) that have conflicting effects with \( o \) or that are disabled by \( o \).
**Motivation**

**Preliminaries**

**Stubborn Sets**

**Strong Stubborn Sets**

**Active Operators**

**Weak Stubborn Sets**

**Algorithms**

**Properties of Stubborn Sets**

**Some Experiments**

**Conclusion**

---

**Algorithms**

**compute-DAL:** Compute a disjunctive action landmark.

**Precedure compute-DAL**

```python
def compute-DAL(γ):
    select v ∈ vars(γ) with s(v) ̸= γ(v)
    L ← {o’ ∈ Act(s) | eff(o')(v) = γ(v)}
    return L
```

Selection of v ∈ vars(γ) arbitrary. Any variable will do. Selection heuristics?

---

**Algorithms**

**compute-NES:** Compute a necessary enabling set.

**Precedure compute-NES**

```python
def compute-NES(o, s):
    select v ∈ prevars(o) with s(v) ̸= pre(o)(v)
    N ← {o’ ∈ Act(s) | eff(o')(v) = pre(o)(v)}
    return N
```

Selection of v ∈ prevars(o) arbitrary. Any variable will do. Selection heuristics?

---

**Algorithms**

**compute-interfering-operators:** Compute interfering operators.

**Precedure compute-interfering-operators (for strong SS)**

```python
def compute-interfering-operators(o):
    disablers ← {o’ ∈ O | o’ disables o}
    disablees ← {o’ ∈ O | o disables o’}
    conflicting ← {o’ ∈ O | o and o’ conflict}
    return disablers ∪ disablees ∪ conflicting
```

**Precedure compute-interfering-operators (for weak SS)**

```python
def compute-interfering-operators(o):
    disablees ← {o’ ∈ O | o disables o’}
    conflicting ← {o’ ∈ O | o and o’ conflict}
    return disablees ∪ conflicting
```

---

**Algorithms**

Computing (strong and weak) stubborn sets for planning can be achieved with a fixpoint iteration until the constraints of Ts are satisfied:

**compute-stubborn-set:** Compute (strong or weak) stubborn set.

**Precedure compute-stubborn-set**

```python
def compute-stubborn-set(s):
    Tₛ ← compute-DAL(γ)
    while no fixed-point of Tₛ reached do
        for o ∈ Tₛ applicable in s:
            Tₛ ← Tₛ ∪ compute-interfering-operators(o)
        for o ∈ Tₛ not applicable in s:
            Tₛ ← Tₛ ∪ compute-NES(o, s)
    end while
    return Tₛ
```

---

January 8th, 2020 B. Nebel, R. Mattmüller – AI Planning
**Motivation**

**Observation:** stubborn sets are state-dependent, but not path-dependent.

This allows filtering the applicable operators in $s$ in graph search algorithms like A*$ that perform duplicate detection, too.

Instead of applying all applicable operators $\text{app}(s)$ in $s$, only apply operators in $T_{\text{app}(s)} = T_s \cap \text{app}(s)$.

**Preservation of Completeness and Optimality**

**Theorem**

*Weak stubborn sets are completeness and optimality preserving.*

**Proof**

Let $T_{\text{app}(s)} := T_s \cap \text{app}(s)$ for a weak stubborn set $T_s$.

We show that for all states $s$ from which an optimal plan consisting of $n > 0$ operators exists, $T_{\text{app}(s)}$ contains an operator that starts such a plan.

We show by induction that A*$ restricting successor generation to $T_{\text{app}(s)}$ is optimal.

Let $T_s$ be a weak stubborn set and $\pi = o_1, \ldots, o_n$ be an optimal plan that starts in $s$.

...
**Preservation of Completeness and Optimality**

**Remark:** The argument to move \( o_k \) to the front also holds for strong stubborn sets: in this case, \( o_k \) is not even disabled by any of \( o_1, \ldots, o_{k-1} \) (and hence, \( o_k \) is independent of \( o_1, \ldots, o_{k-1} \)), which is a stronger property than needed in the proof.

**Corollary**

*Strong stubborn sets are completeness and optimality preserving.*

---

**Some Experiments: Overview**

Optimal Planning, A* with LM-cut Heuristic, Selected Domains

<table>
<thead>
<tr>
<th>Domain (problems)</th>
<th>Coverage</th>
<th>Nodes generated</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A + SSS</td>
<td>A + SSS</td>
</tr>
<tr>
<td><strong>PARCPRINTER-08 (30)</strong></td>
<td>18</td>
<td>2455181 &lt;1%</td>
</tr>
<tr>
<td><strong>PARCPRINTER-OPT1 (20)</strong></td>
<td>13</td>
<td>2454533 &lt;1%</td>
</tr>
<tr>
<td><strong>WOODWORKING-OPT08 (30)</strong></td>
<td>17</td>
<td>26796212 &lt;1%</td>
</tr>
<tr>
<td><strong>WOODWORKING-OPT11 (20)</strong></td>
<td>12</td>
<td>26795517 &lt;1%</td>
</tr>
<tr>
<td><strong>SATELLITE (36)</strong></td>
<td>7</td>
<td>5116312 2%</td>
</tr>
<tr>
<td><strong>rovers (40)</strong></td>
<td>7</td>
<td>1900691 22%</td>
</tr>
<tr>
<td><strong>airport (50)</strong></td>
<td>28</td>
<td>545072 93%</td>
</tr>
<tr>
<td><strong>OPENSTACKS-OPT08 (30)</strong></td>
<td>19</td>
<td>56684063 51%</td>
</tr>
<tr>
<td><strong>OPENSTACKS-OPT11 (20)</strong></td>
<td>14</td>
<td>56459699 51%</td>
</tr>
<tr>
<td><strong>DRIVERLOG (20)</strong></td>
<td>13</td>
<td>3679376 82%</td>
</tr>
<tr>
<td><strong>SCANALYZER-OPT08 (30)</strong></td>
<td>15</td>
<td>14203012 100%</td>
</tr>
<tr>
<td><strong>SCANALYZER-OPT11 (20)</strong></td>
<td>12</td>
<td>14202884 100%</td>
</tr>
<tr>
<td><strong>parking-OPT1 (20)</strong></td>
<td>11</td>
<td>1991169 100%</td>
</tr>
</tbody>
</table>

**Remaining domains (980)**

<table>
<thead>
<tr>
<th></th>
<th>Coverage</th>
<th>Nodes generated</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A + SSS</td>
<td>A + SSS</td>
</tr>
<tr>
<td><strong>satellite</strong> (36)</td>
<td>12</td>
<td>70299721 92.804%</td>
</tr>
</tbody>
</table>

⇒ In practice (or, at least, in the standard benchmark problems) there is no significant difference between weak and strong stubborn sets.
Need for techniques orthogonal to heuristic search, complementing heuristics.

One idea: Commit to one order of operators if they are independent. Prune other orders.

Class of such techniques: partial-order reduction (POR)

One such technique: strong/weak stubborn sets

Can lead to substantial pruning compared to plain A*.

Many other POR techniques exist.

Other pruning techniques exist as well, e.g., symmetry reduction.