What are State-Dependent Action Costs?

- In classical planning: actions have unit costs.
  - Each action $a$ costs 1.

- Simple extension: actions have constant costs.
  - Each action $a$ costs some $\text{cost}_a \in \mathbb{N}$.
  - Example: Flying between two cities costs amount proportional to distance.
  - Still easy to handle algorithmically, e.g. when computing $g$ and $h$ values.

- Further extension: actions have state-dependent costs.
  - Each action $a$ has cost function $\text{cost}_a : S \rightarrow \mathbb{N}$.
  - Example: Flying to a destination city costs amount proportional to distance, depending on the current city.
Why Study State-Dependent Action Costs?

- **Human perspective:**
  - "natural", "elegant", and "higher-level"
  - modeler-friendly --- less error-prone?
- **Machine perspective:**
  - more structured --- exploit structure in algorithms?
  - fewer redundancies, exponentially more compact
- **Language support:**
  - numeric PDDL, PDDL 3
  - RDDL, MDPs (state-dependent rewards!)
- **Applications:**
  - modeling preferences and soft goals
  - application domains such as PSR

(Expiration: SDAC = state-dependent action costs)


Handling State-Dependent Action Costs

- **Good news:**
  - Computing $g$ values in forward search still easy. 
  - (When expanding state $s$ with action $a$, we know $cost_a(s)$.)
- **Challenge:**
  - But what about SDAC-aware $h$ values
    - (relaxation heuristics, abstraction heuristics)?
  - Or can we simply compile SDAC away?
- **This chapter:**
  - Proposed answers to these challenges.


State-Dependent Action Costs

- **Definition**
  - A SAS* planning task with state-dependent action costs or SDAC planning task is a tuple $\Pi = (V, I, O, \gamma, (cost_a)_{a \in O})$ where $(V, I, O, \gamma)$ is a (regular) SAS* planning task with state set $S$ and $cost_a : S \rightarrow \mathbb{N}$ is the cost function of $a$ for all $a \in O$.

- **Assumption:** For each $a \in O$, the set of variables occurring in the precondition of $a$ is disjoint from the set of variables on which the cost function $cost_a$ depends.

  - (Question: Why is this assumption unproblematic?)

- Definitions of plans etc. stay as before. A plan is optimal if it minimizes the sum of action costs from start to goal.

  - For the rest of this chapter, we consider the following running example.

February 1st, 2019 B. Nebel, R. Mattmüller – AI Planning 8 / 79
State-Dependent Action Costs

Running Example

Example (Household domain)

Actions:

- vacuumFloor = \langle \top, \text{floorClean} \rangle
- washDishes = \langle \top, \text{dishesClean} \rangle
- doHousework = \langle \top, \text{floorClean} \land \text{dishesClean} \rangle

Cost functions:

- \text{cost}_{\text{vacuumFloor}} = [\neg \text{floorClean}] \cdot 2
- \text{cost}_{\text{washDishes}} = [\neg \text{dishesClean}] \cdot (1 + 2 \cdot [\neg \text{haveDishwasher}])
- \text{cost}_{\text{doHousework}} = \text{cost}_{\text{vacuumFloor}} + \text{cost}_{\text{washDishes}}

(Question: How much can applying action washDishes cost?)
State-Dependent Action Costs
Compilation II: “Purely Sequential Action Decomposition”

Example
Assume we own a dishwasher:

\[
\text{cost}_{\text{doHousework}} = 2 \cdot [\neg \text{floorClean}] + [\neg \text{dishesClean}]
\]

- \text{floorClean}: 0
- \text{dishesClean}: 0

\[
\text{doHousework}_1(\text{fC}) = \{ \text{fC}, \text{fC} \}, \quad \text{cost} = 0
\]

\[
\text{doHousework}_1(\neg \text{fC}) = \{ \neg \text{fC}, \text{fC} \}, \quad \text{cost} = 2
\]

\[
\text{doHousework}_2(\text{dC}) = \{ \text{dC}, \text{dC} \}, \quad \text{cost} = 0
\]

\[
\text{doHousework}_2(\neg \text{dC}) = \{ \neg \text{dC}, \text{dC} \}, \quad \text{cost} = 1
\]

Properties:
- **✓** only linear blow-up
- **✗** not always possible
- **●** plan lengths not preserved
  E.g., in a state where \(\neg \text{fC}\) and \(\neg \text{dC}\) hold, an application of \text{doHousework}

in the SDAC setting is replaced by an application of the action sequence

\[\text{doHousework}_1(\neg \text{fC}), \text{doHousework}_2(\neg \text{dC})\]

in the compiled setting.

Properties (ctd.):
- plan costs preserved
- blow-up in search space
  E.g., in a state where \(\neg \text{fC}\) and \(\neg \text{dC}\) hold, should we apply \text{doHousework}_1(\neg \text{fC}) or \text{doHousework}_2(\neg \text{dC}) first?

\[\implies\] impose action ordering!

- attention: we should apply all partial effects at end!
  Otherwise, an effect of an earlier action in the compilation might affect the cost of a later action in the compilation.

Question: Can this always work (kind of)? \[\implies\] Compilation III
State-Dependent Action Costs
Compilation III: “EVMDD-Based Action Decomposition”

Example

\[
\text{cost}_{\text{doHousework}} = \neg \text{floorClean} \cdot 2 + \neg \text{dishesClean} \cdot (1 + 2 \cdot \neg \text{haveDishwasher})
\]

Simplify right-hand part of diagram:

- Branch over single variable at a time.
- Exploit: \text{haveDishwasher} irrelevant if \text{dishesClean} is true.

Later:

- Compiled actions
- Auxiliary variables to enforce action ordering

Compilatlon III

- exploit as much additive separability as possible
- multiply out variable domains where inevitable
- Technicalities:
  - fix variable ordering
  - perform Shannon and isomorphism reduction (cf. theory of BDDs)

Properties:

- always possible
- worst-case exponential blow-up, but as good as it gets
- as with Compilation II: plan lengths not preserved, plan costs preserved
- as with Compilation II: action ordering, all effects at end!

Compilation III provides optimal combination of sequential and parallel action decomposition, given fixed variable ordering.

Question: How to find such decompositions automatically?

Answer: Figure for Compilation III basically a reduced ordered edge-valued multi-valued decision diagram (EVMDD)!

[Lai et al., 1996; Ciardo and Siminiceanu, 2002]
EVMDDs
Edge-Valued Multi-Valued Decision Diagrams

EVMDDs:
- Decision diagrams for arithmetic functions
- Decision nodes with associated decision variables
- Edge weights: partial costs contributed by facts
- Size of EVMDD compact in many “typical”, well-behaved cases (Question: For example?)

Properties:
✓ satisfy all requirements for Compilation III, even (almost) uniquely determined by them
✓ already have well-established theory and tool support
✓ detect and exhibit additive structure in arithmetic functions

Consequence:
- represent cost functions as EVMDDs
- exploit additive structure exhibited by them
- draw on theory and tool support for EVMDDs

Two perspectives on EVMDDs:
- graphs specifying how to decompose action costs
- data structures encoding action costs (used independently from compilations)

Properties of EVMDDs:
✓ Existence for finitely many finite-domain variables
✓ Uniqueness/canonicity if reduced and ordered
✓ Basic arithmetic operations supported

(Lai et al., 1996; Ciardo and Siminiceanu, 2002)
EVMDDs

Arithmetic operations on EVMDDs

Given arithmetic operator \( \otimes \in \{+, -, \ldots\} \), EVMDDs \( \mathcal{E}_1, \mathcal{E}_2 \).

Compute EVMDD \( \mathcal{E} = \mathcal{E}_1 \otimes \mathcal{E}_2 \).

Implementation: procedure `apply(\( \otimes \), \( \mathcal{E}_1 \), \( \mathcal{E}_2 \))`:

- **Base case**: single-node EVMDDs encoding constants
- **Inductive case**: apply \( \otimes \) recursively:
  - push down edge weights
  - recursively apply \( \otimes \) to corresponding children
  - pull up excess edge weights from children

Time complexity [Lai et al., 1996]:

- **additive operations**: product of input EVMDD sizes
- **in general**: exponential

EVMDD-Based Action Compilation

Idea: each edge in the EVMDD becomes a new micro action with constant cost corresponding to the edge constraint, precondition that we are currently at its start EVMDD node, and effect that we are currently at its target EVMDD node.

Example (EVMDD-based action compilation)

Let \( a = (\chi, e) \), \( \text{cost}_a = xy^2 + z + 2 \).

Auxiliary variables:

- One semaphore variable \( \sigma \) with \( \mathcal{D}_\sigma = \{0, 1\} \) for entire planning task.
- One auxiliary variable \( \alpha = \alpha_a \) with \( \mathcal{D}_\alpha_a = \{0, 1, 2, 3, 4\} \) for action \( a \).

Replace \( a \) by new auxiliary actions (similarly for other actions).
EVMDD-Based Action Compilation

Definition (EVMDD-based action compilation)
Let \( \Pi = (V, I, O, \gamma, (cost_a)_{a \in O}) \) be an SDAC planning task, and for each action \( a \in O \), let \( d_a \) be an EVMDD that encodes the cost function \( cost_a \).

Let \( EAC(a) \) be the set of actions created from \( a \) using \( d_a \) similar to the previous example. Then the EVMDD-based action compilation of \( \Pi \) using \( d_a \), \( a \in O \), is the task \( \Pi' = EAC(\Pi) = (V', I', O', \gamma') \), where
- \( V' = V \cup \{ \sigma \} \cup \{ \alpha_a | a \in O \} \),
- \( I' = I \cup \{ \sigma \not\rightarrow 0 \} \cup \{ \alpha_a \not\rightarrow 0 | a \in O \} \),
- \( O' = \bigcup_{a \in O} EAC(a) \), and
- \( \gamma' = \gamma \land (\sigma = 0) \land \bigwedge_{a \in O} (\alpha_a = 0) \).

Proof.
By construction.

Proposition
\( \Pi' \) has only state-independent costs.

Proof.
By construction.

EVMDD-Based Action Compilation

Let \( \Pi \) be an SDAC task and \( \Pi' = EAC(\Pi) \) its EVMDD-based action compilation (for appropriate EVMDDs \( d_a \)).

Proposition
\( \Pi \) and \( \Pi' \) admit the same plans (up to replacement of actions by action sequences). Optimal plan costs are preserved.

Proof.
Let \( \pi = a_1, \ldots, a_n \) be a plan for \( \Pi \), and let \( s_0, \ldots, s_n \) be the corresponding state sequence such that \( a_i \) is applicable in \( s_{i-1} \) and leads to \( s_i \) for all \( i = 1, \ldots, n \).

For each \( i = 1, \ldots, n \), let \( d_{a_i} \) be the EVMDD used to compile \( a_i \). State \( s_{i-1} \) determines a unique path through the EVMDD \( d_{a_i} \), which uniquely corresponds to an action sequence \( a_i^0, \ldots, a_i^{k_i} \) (for some \( k_i \in \mathbb{N} \); including \( a_i^0 \) and \( a_i^{k_i} \)).

Therefore, by induction, \( \pi' = a_1^0, \ldots, a_n^0, \ldots, a_n^{k_n} \) is applicable in \( s_0 \cup \{ \sigma \not\rightarrow 0 \} \cup \{ \alpha_a \not\rightarrow 0 | a \in O \} \) (and leads to a goal state). Moreover,
\[
\text{cost}(\pi') = \text{cost}(a_1^0) + \cdots + \text{cost}(a_i^0) + \cdots + \text{cost}(a_n^0) + \cdots + \text{cost}(a_n^{k_n}) = \text{cost}_{a_1}(s_0) + \cdots + \text{cost}_{a_n}(s_{n-1}) = \text{cost}(\pi).
\]

Still to show: \( \Pi' \) admits no other plans. It suffices to see that the semaphore \( \sigma \) prohibits interleaving more than one EVMDD evaluation, and that each \( \alpha_a \) makes sure that the EVMDD for \( a \) is traversed in the unique correct order.

Proof (ctd.)
By construction, \( \text{cost}(a_1^0) + \cdots + \text{cost}(a_i^0) = \text{cost}_{a_i}(s_{i-1}) \).

Moreover, the sequence \( a_1^0, \ldots, a_i^0 \) is applicable in \( s_{i-1} \cup \{ \sigma \not\rightarrow 0 \} \cup \{ \alpha_a \not\rightarrow 0 | a \in O \} \) and leads to \( s_i \cup \{ \sigma \not\rightarrow 0 \} \cup \{ \alpha_a \not\rightarrow 0 | a \in O \} \).

Therefore, by induction, \( \pi' = a_1^0, \ldots, a_n^0, \ldots, a_n^{k_n} \) is applicable in \( s_0 \cup \{ \sigma \not\rightarrow 0 \} \cup \{ \alpha_a \not\rightarrow 0 | a \in O \} \) (and leads to a goal state). Moreover,
\[
\text{cost}(\pi') = \text{cost}(a_1^0) + \cdots + \text{cost}(a_i^0) + \cdots + \text{cost}(a_n^0) + \cdots + \text{cost}(a_n^{k_n}) = \text{cost}_{a_1}(s_0) + \cdots + \text{cost}_{a_n}(s_{n-1}) = \text{cost}(\pi).
\]

Still to show: \( \Pi' \) admits no other plans. It suffices to see that the semaphore \( \sigma \) prohibits interleaving more than one EVMDD evaluation, and that each \( \alpha_a \) makes sure that the EVMDD for \( a \) is traversed in the unique correct order.
Let $\Pi = \langle V, I, O, \gamma \rangle$ with $V = \{x, y, z, u\}$, $D_x = D_z = \{0, 1\}$, $D_y = D_u = \{0, 1, 2\}$, $I = \{x \mapsto 1, y \mapsto 2, z \mapsto 0, u \mapsto 0\}$, $O = \{a, b\}$, and $\gamma = (u = 2)$ with

\[
\begin{align*}
  a = (u = 0, u := 1), & \quad \text{cost}_a = xy^2 + z + 2, \\
  b = (u = 1, u := 2), & \quad \text{cost}_b = z + 1.
\end{align*}
\]

Optimal plan for $\Pi$:

\[\pi = a, b \text{ with } \text{cost}(\pi) = 6 + 1 = 7.\]
Relaxation Heuristics

We know: Delete-relaxation heuristics informative in classical planning.

Question: Are they also informative in SDAC planning?

Relaxed SAS+ Tasks

Delete relaxation in SAS+ tasks works as follows:

- Operators are already in effect normal form.
- We do not need to impose a positive normal form, because all conditions are conjunctions of facts, and facts are just variable-value pairs and hence always positive.
- Hence $a^+ = a$ for any operator $a$, and $\Pi^+ = \Pi$.
- For simplicity, we identify relaxed states $s^*$ with their on-sets $\text{on}(s^*)$.
- Then, a relaxed state $s^*$ is a set of facts $(v, d)$ with $v \in V$ and $d \in D_v$ including at least one fact $(v, d)$ for each $v \in V$ (but possibly more than one, which is what makes it a relaxed state).

Assume we want to compute the additive heuristic $h^{\text{add}}$ in a task with state-dependent action costs.

But what does an action $a$ cost in a relaxed state $s^*$?

And how to compute that cost?
Relaxed SAS⁺ Tasks

- A relaxed operator $a$ is applicable in a relaxed state $s^*$ if all precondition facts of $a$ are contained in $s^*$.
- Relaxed states accumulate facts reached so far.
- Applying a relaxed operator $a$ to a relaxed state $s^*$ adds to $s^*$ those facts made true by $a$.

Example

Relaxed operator $a^* = (x = 2, y := 1, z := 0)$ is applicable in relaxed state $s^* = \{ (x, 0), (x, 2), (y, 0), (z, 1) \}$, because precondition $(x, 2) \in s^*$, and leads to successor $(s^*)' = s^* \cup \{ (y, 1), (z, 0) \}$.

Relaxed plans, dominance, monotonicity etc. as before. The above definition generalizes the one for propositional tasks.

Action Costs in Relaxed States

Example

Assume $s^*$ is the relaxed state with

$$s^* = \{ (x, 0), (x, 1), (y, 1), (y, 2), (z, 0) \}.$$ 

What should action $a$ with $cost_a = xy^2 + z + 2$ cost in $s^*$?

Idea: We should assume the cheapest way of applying $a^*$ in $s^*$ to guarantee admissibility of $h^*$. (Allow at least the behavior of the unrelaxed setting at no higher cost.)

Example

$$x = 0, y = 1, z = 0 \rightarrow a \ [2]$$

$$x = 0, y = 2, z = 0 \rightarrow a \ [2]$$

$$x = 1, y = 1, z = 0 \rightarrow a \ [3]$$

$$x = 1, y = 2, z = 0 \rightarrow a \ [6]$$

Idea: We should assume the cheapest way of applying $a^*$ in $s^*$ to guarantee admissibility of $h^*$. (Allow at least the behavior of the unrelaxed setting at no higher cost.)

Example

$$s^* \rightarrow a^* \ [2] \rightarrow t^*$$
**Definition**

Let $V$ be a set of FDR variables, $s : V \rightarrow \bigcup_{v \in V} D_v$ an unrelaxed state over $V$, and $s^+ \subseteq \{(v, d) \mid v \in V, d \in D_v\}$ a relaxed state over $V$. We call $s$ consistent with $s^+$ if $\{(v, s(v)) \mid v \in V\} \subseteq s^+$.

**Definition**

Let $a \in O$ be an action with cost function $\text{cost}_a$, and $s^+$ a relaxed state. Then the relaxed cost of $a$ in $s^+$ is defined as

$$\text{cost}_a(s^+) = \min_{s \in S \text{ consistent with } s^+} \text{cost}_a(s).$$

(Question: How many states $s$ are consistent with $s^+$?)

**Problem with this definition:** There are generally exponentially many states $s$ consistent with $s^+$ to minimize over.

**Central question:** Can we still do this minimization efficiently?

**Answer:** Yes, at least efficiently in the size of an EVMDD encoding $\text{cost}_a$.

**Example**

Relaxed state $s^+ = \{(x, 0), (x, 1), (y, 1), (y, 2), (z, 0)\}$.

- Computing $\text{cost}_a(s^*) = \min_{s \in S \text{ consistent with } s^+} \text{cost}_a(s)$ for all $s$ consistent with $s^*$.
- Observation: Minimization over exponentially many paths can be replaced by top-sort traversal of EVMDD, minimizing over incoming arcs consistent with $s^*$ at all nodes!
Cost Computation for Relaxed States

**Theorem**
A top-sort traversal of the EVMDD for $\text{cost}_a$, adding edge weights and minimizing over incoming arcs consistent with $s^+$ at all nodes, computes $\text{cost}_a(s^+)$ and takes time in the order of the size of the EVMDD.

**Proof.**

Cost Computation for Relaxed States

Relaxation Heuristics

The following definition is equivalent to the RPG-based one.

**Definition (Classical additive heuristic $h^{\text{add}}$)**

$$h^{\text{add}}_s(s) = h^{\text{add}}_s(\text{GoalFacts})$$

$$h^{\text{add}}_s(\text{Facts}) = \sum_{\text{fact} \in \text{Facts}} h^{\text{add}}_s(\text{fact})$$

$$h^{\text{add}}_s(\text{fact}) = \begin{cases} 
0 & \text{if } \text{fact} \in s \\
\min_{a \text{ of fact}} \left[h^{\text{add}}_s(\text{pre}(a)) + \text{cost}_a\right] & \text{otherwise}
\end{cases}$$

**Question:** How to generalize $h^{\text{add}}$ to SDAC?

Relaxations with SDAC

**Example**

$$a = \langle \top, x=1 \rangle \quad \text{cost}_a = 2 - 2y$$

$$b = \langle \top, y=1 \rangle \quad \text{cost}_b = 1$$

$$s = \{x \mapsto 0, y \mapsto 0\}$$

$$h^{\text{add}}_s(y = 1) = 1$$

$$h^{\text{add}}_s(x = 1) = ?$$

Relaxations with SDAC

(Here, we need the assumption that no variable occurs both in the cost function and the precondition of the same action):

**Definition (Additive heuristic $h^{\text{add}}$ for SDAC)**

$$h^{\text{add}}_s(\text{fact}) = \begin{cases} 
0 & \text{if } \text{fact} \in s \\
\min_{a \text{ of fact}} \left[h^{\text{add}}_s(\text{pre}(a)) + \text{cost}_a\right] & \text{otherwise}
\end{cases}$$

$$\text{Cost}_s = \min_{\hat{s} \in S_a} [\text{cost}_a(\hat{s}) + h^{\text{add}}_s(\hat{s})]$$

$S_a$: set of partial states over variables in cost function

$|S_a|$: exponential in number of variables in cost function
Relaxations with SDAC

**Theorem**

Let $\Pi$ be an SDAC planning task, let $\Gamma$ be an EVMDD-based action compilation of $\Pi$, and let $s$ be a state of $\Pi$. Then the classical $h^{\text{add}}$ heuristic in $\Gamma$ gives the same value for $s \cup \{ \sigma \mapsto 0 \} \cup \{ \alpha_a \mapsto 0 | a \in O \}$ as the generalization of $h^{\text{add}}$ to SDAC tasks defined above gives for $s$ in $\Pi$.

Computing $h^{\text{add}}$ for SDAC:

- **Option 1:** Compute classical $h^{\text{add}}$ on compiled task.
- **Option 2:** Compute $\text{Cost}_a^s$ directly. How?
  - Plug EVMDDs as subgraphs into RPG
  - Efficient computation of $h^{\text{add}}$

Option 2: RPG Compilation

**Option 2: RPG Compilation**

**Computing $\text{Cost}_a^s$**

Evaluate nodes:

- $\text{Cost}_a = xy^2 + z + 2$
- Variable nodes become $\lor$-nodes
- Weights become $\land$-nodes
- Augment with input nodes
- Ensure complete evaluation
- Insert $h^{\text{add}}$ values
- $\land: \sum(\text{parents}) + \text{weight}$
- $\lor: \min(\text{parents})$
- $\text{Cost}_a^s = xy^2 + z + 2$

RPG Compilation

**Remark:** We can use EVMDDs to compute $C_a^s$ and hence the generalized additive heuristic directly, by embedding them into the relaxed planning task.

We just briefly show the example, without going into too much detail.

**Idea:** Augment EVMDD with input nodes representing $h^{\text{add}}$ values from the previous RPG layer.

- Use augmented diagrams as RPG subgraphs.
- Allows efficient computation of $h^{\text{add}}$.

Additive Heuristic

- Use above construction as subgraph of RPG in each layer, for each action (as operator subgraphs).
- Add AND nodes conjoining these subgraphs with operator precondition graphs.
- Link EVMDD outputs to next proposition layer.

**Theorem**

Let $\Pi$ be an SDAC planning task. Then the classical additive RPG evaluation of the RPG constructed using EVMDDs as above computes the generalized additive heuristic $h^{\text{add}}$ defined before.
Abstractions

Abstraction Heuristics for SDAC

Question: Why consider abstraction heuristics?
Answer:
- admissibility
- \(\rightarrow\) optimality

Abstraction Heuristics for SDAC

Question: What are the abstract action costs?
Answer: For admissibility, abstract cost of action should be:

\[
\text{cost}_a(s^{\text{abs}}) = \min_{s \text{ concrete state s abstracted to } s^{\text{abs}}} \text{cost}_a(s).
\]

Problem: exponentially many states in minimization
Aim: Compute \(\text{cost}_a(s^{\text{abs}})\) efficiently (given EVMDD for \(\text{cost}_a(s)\)).

Cartesian Abstractions

We will see: possible if the abstraction is Cartesian or coarser.
(Includes projections and domain abstractions.)

Definition (Cartesian abstraction)
A set of states \(s^{\text{abs}}\) is Cartesian if it is of the form

\[
D_1 \times \cdots \times D_n,
\]

where \(D_i \subseteq \mathcal{S}\) for all \(i = 1, \ldots, n\).
An abstraction is Cartesian if all abstract states are Cartesian sets.

[Seipp and Helmert, 2013]

Intuition: Variables are abstracted independently.
\(\rightarrow\) exploit independence when computing abstract costs!
Why does the topsort EVMDD traversal (cheapest path computation) correctly compute $\text{cost}_{s_{\text{abs}}}(s_{\text{abs}})$?

Short answer: The exact same thing as with relaxed states, because relaxed states are Cartesian sets!

Longer answer:
1. For each Cartesian state $s_{\text{abs}}$ and each variable $v$, each value $d \in D_v$ is either consistent with $s_{\text{abs}}$ or not.
2. This implies: at all decision nodes associated with variable $v$, some outgoing edges are enabled, others are disabled.
   This is independent from all other decision nodes.
3. This allows local minimizations over linearly many edges instead of global minimization over exponentially many paths in the EVMDD when computing minimum costs.
   $\Rightarrow$ polynomial in EVMDD size!

Counterexample-Guided Abstraction Refinement

Wanted: principled way of computing Cartesian abstractions.

$\Rightarrow$ Counterexample-Guided Abstraction Refinement (CEGAR)
[Clarke et al., 2000] [Seipp and Helmert, 2013]
CEGAR and Cartesian Abstractions

Assume the following:

- **Initial abstraction** is one-state abstraction with single abstract state $D_1 \times \cdots \times D_n$.
  - $\Rightarrow$ Cartesian abstraction
- Each refinement step takes one abstract state $s_{abs} = D_1 \times \cdots \times D_n$, one variable $v_i$, and splits $s_{abs}$ into
  - $D_1 \times \cdots \times D_{i-1} \times D'_i \times D_{i+1} \times \cdots \times D_n$
  - $D_1 \times \cdots \times D_{i-1} \times D''_i \times D_{i+1} \times \cdots \times D_n$
  such that $D'_i \cap D''_i = \emptyset$ and $D'_i \cup D''_i = D_i$.
  - $\Rightarrow$ still a Cartesian abstraction

So, inductively:

- Initial abstraction is Cartesian.
- Each refinement step preserves being Cartesian.
  - $\Rightarrow$ All generated abstractions are Cartesian.

Some questions:

- **Q:** When to split abstract states?
  - A: When first flaw is identified. (Details below.)
- **Q:** How to split abstract states?
  - A: So as to resolve that flaw. (Details below.)

CEGAR by Example

Example (one package, one truck)

Consider the following FDR planning task $\langle V, I, O, \gamma \rangle$:

- $V = \{t, p\}$ with
  - $\mathcal{D}_t = \{L, R\}$
  - $\mathcal{D}_p = \{L, T, R\}$
  - $\mathcal{I} = \{t \mapsto L, p \mapsto L\}$
- $O = \{\text{pick-in}_i \mid i \in \{L, R\}\}$
  - $\cup \{\text{drop-in}_i \mid i \in \{L, R\}\}$
  - $\cup \{\text{move}_{ij} \mid i, j \in \{L, R\}, i \neq j\}$, where
  - $\text{pick-in}_i = \{t = i \land p = i, p := T\}$
  - $\text{drop-in}_i = \{t = i \land p = T, p := i\}$
  - $\text{move}_{ij} = \{t = i, t := j\}$
- $\gamma = (p = R)$.
Before we look at CEGAR applied to this task, here is the concrete transition system (just for reference):

Refinement step 0 (initial abstraction):

Refinement step 1:

Refinement step 2:

Refinement step 3:

Refinement step 4:

Abstract plan:

$\pi_0 = \langle \rangle$

$\pi_1 = \langle \text{drop-in}_R \rangle$

$\pi_2 = \langle \text{move}_{L,R}, \text{drop-in}_R \rangle$

$\pi_3 = \langle \text{move}_{L,R}, \text{drop-in}_R \rangle$

$\pi_4 = \langle \text{pick-in}_L, \text{move}_{L,R}, \text{drop-in}_R \rangle$

Flaw: $s_0 = \text{LL}$ is not a goal state.

Preconditions ($t = \text{R}$) and ($p = \text{T}$) of $\text{drop-in}_R$ not satisfied in $s_0 = \text{LL}$. Precondition ($p = \text{T}$) of $\text{drop-in}_R$ not satisfied in $s_1 = \text{RL}$.

Flaw 1: Abstract plan terminates in concrete non-goal state.

Resolution: Split abstraction of last state $s_n$ of concrete trace into (a) part containing $s_n$, but containing no concrete goal state, and (b) rest.
Flaw 2: Abstract plan fails because some operator precondition is violated.

Resolution: Split abstraction of state $s_{i-1}$ of concrete trace, where operator precondition $\chi$ is violated, into (a) part containing $s_{i-1}$, but no concrete state in which precondition $\chi$ is satisfied, and (b) rest.

Flaw 3: Concrete and abstract paths diverge.

Resolution: Split abstraction of state $s_{i-1}$ of concrete trace, after which paths diverge when applying operator $o$, into (a) part containing $s_{i-1}$ where applying $o$ always leads to the "wrong" abstract successor state, and (b), rest.

Flaw 4: Action is more costly in concrete state than in abstract state.

Resolution: Split abstraction of violating concrete state into two parts that differ on the value of a variable that is relevant to the cost function of the operator in question, such that we have different cost values in the two parts.

Remark: In tasks with state-dependent action costs, there is a fourth type of flaws, so-called cost-mismatch flaws.

Example (Cost-mismatch flaw)

\[
\begin{align*}
a &= \langle \top, x \land y \rangle, & \text{cost}_a &= 2x + 1 \\
b &= \langle \top, \neg x \land y \rangle, & \text{cost}_b &= 1
\end{align*}
\]

\[
s_0 = 10, & \quad s_* = x \lor y
\]

Optimal abstract plan: $\langle a \rangle$ (abstract cost 1)

This is also a concrete plan (concrete cost $3 \neq 1$)

\[
\rightarrow \text{split } \{0,1\} \times \{0\}
\]

Cf. optimal concrete plan: $\langle b, a \rangle$ (conc. and abstr. cost 2)
Summary: State-dependent actions costs practically relevant. EVMDDs exhibit and exploit structure in cost functions. Graph-based representations of arithmetic functions. Edge values express partial cost contributed by facts. Size of EVMDD is compact in many “typical” cases. Can be used to compile tasks with state-dependent costs to tasks with state-independent costs. Alternatively, can be embedded into the RPG to compute forward-cost heuristics directly. For $h^{add}$, both approaches give the same heuristic values. Abstraction heuristics can also be generalized to state-dependent action costs.

Future Work and Work in Progress:
- Investigation of other delete-relaxation heuristics for tasks with state-dependent action costs.
- Investigation of static and dynamic EVMDD variable orders.
- Application to cost partitioning, to planning with preferences, …
- Better integration of SDAC in PDDL.
- Tool support.
- Benchmarks.

References
