A Comparison of Abstraction Heuristics for Rubik's Cube

Anonymized

Abstract

Since its invention in the year 1974, Rubik's Cube fascinates people of all ages. Its rules are simple: the player gets a scrambled cube and rotates the six faces until each face contains only stickers of one colors. Nevertheless, finding a short sequence of rotations to solve the cube is hard. We present the first ever model of Rubik's Cube for general problem solvers. To obtain a concise model, we require a special kind of conditional effects which we call *factored effects*. We secondly extend Cartesian CEGAR to support factored effects. Finally, we evaluate how modern abstraction heuristics compare against pattern database (PDB) heuristics, the state-of-the-art for solving Rubik's Cube. We find that PDBs still outperform the more general methods of Cartesian CEGAR and merge-and-shrink. However, in contrast to PDBs, Cartesian CEGAR provides perfect heuristics up to a certain problem difficulty. These findings raise interesting questions for future investigations.

Introduction

In *classical planning*, we solve deterministic search problems. In general, this means that we search for a sequence of operators leading from the initial state of the problem to a state which satisfies some goal condition. A difficult classical planning problem which also enjoys prominence in the general public is *Rubik's Cube*. The traditional Rubik's Cube consists of 26 small, colored cubes called *cubies*. The cubies are stacked in the shape of a larger $3 \times 3 \times 3$ cube such that initially all sides of the larger cube show the same color (cf. Figure 1, right-hand side). The cube has in each dimension (horizontal, vertical, and depth) 3 layers, which can be rotated independently. Rotating a layer changes the colors on the faces of Rubik's Cube. Given an arbitrarily rotated Rubik's Cube, the goal is to find a sequence of rotations which returns the cube to its initial state.

Rubik's Cube has approximately $4.3 \cdot 10^{19}$ reachable states. Those are too many for finding solutions using blind search, which is also the case for many other real world problems. A successfully applied method in classical planning is *heuristic search* (e.g., Hoffmann and Nebel 2001; Helmert and Domshlak 2009; Richter and Westphal 2010; Helmert et al. 2014; Domshlak, Hoffmann, and Katz 2015). A *heuristic* is a function that estimates for a state the minimal cost to the closest goal state. In the case of Rubik's Cube, the cost to the goal is the number of rotations required to reach the goal. The guidance of a good heuristic helps the search to consider only states which are likely to occur on a solution path.

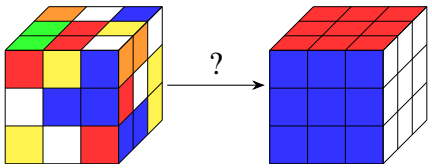

Figure 1: The classic $3 \times 3 \times 3$ Rubik's Cube, scrambled on the left and solved on the right.

One family of heuristics are *abstraction heuristics*. An abstraction is an equivalence relation between states. The states of the original problem within the same equivalence class denote one abstract state. Each transition in the original problem induces a transition between the corresponding abstract states in the abstraction. Since the abstract problem has in general significantly fewer states and preserves transitions between concrete states, it is easier to solve. Abstraction heuristics use the goal distance in the abstract problem to estimate the goal distance in the original problem.

The state-of-the-art for finding optimal solutions for Rubik's Cube are *pattern database* (PDB) heuristics (Korf 1997; Culberson and Schaeffer 1998). PDBs are abstraction heuristics and were developed in the 90's. Since then, *domain abstractions* (Hernádvölgyi and Holte 2000), *Cartesian abstractions* (Seipp and Helmert 2018), and *merge-and-shrink (M&S) abstractions* (Sievers and Helmert 2021) have been introduced to classical planning. These methods differ in the way they map states to abstract states and can be ordered by their degree of freedom: M&S abstractions are more general than Cartesian abstractions, which are in turn more general than domain abstractions, which are in turn more general than PDBs. This raises the question how the more general abstractions compare against the less general PDBs on the difficult Rubik's Cube.

Our contribution is three fold. First, we model Rubik's Cube in finite-domain representation (Helmert 2009). Because of the enormous number of ground operators in this problem, our model requires *factored effects*, a special kind of conditional effects. This enables us for the first time to

use common general problem solvers on Rubik's Cube. Second, we show how Cartesian abstractions can be generated for problems with factored effects. Third, we compare the different abstraction heuristics against one another.

Our experimental evaluation reveals that PDBs keep their spot as the most successful technique for finding optimal solutions for Rubik's Cube, even though, we are unable to use Korf's (1997) patterns. Our adaption of his patterns, still solves the most problems from our newly generated Rubik's Cube benchmarks among all compared heuristics. The heuristic accuracy, however, is trumped by our adaption of the CEGAR algorithm for generating Cartesian abstractions (Seipp and Helmert 2018). M&S abstractions are the least successful on this problem according to our findings.

Background

Classical Planning We consider planning using finite-domain representation (Helmert 2009), an extension of the SAS$^+$ formalism (Bäckström and Nebel 1995) with *conditional effects*. A *planning task* is a 4-tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_I, s_\star \rangle$. $\mathcal{V}$ is a finite set of variables. Each variable $V \in \mathcal{V}$ has a finite domain $\mathrm{dom}(V)$. We call a variable-value assignment $V \mapsto v$ with $V \in \mathcal{V}$ and $v \in \mathrm{dom}(V)$ an *atom*. A *partial state* s is associated with a set of variables $\mathit{vars}(s) \subseteq \mathcal{V}$ and assigns each variable $V \in \mathit{vars}(s)$ one value $v \in \mathrm{dom}(V)$. A partial state s where $\mathit{vars}(s) = \mathcal{V}$ is a *state*. $S(\Pi)$ is the set of all states of Π. For a partial state s, we define $s[V] = v$ if $(V \mapsto v) \in s$. $\mathcal{O}$ is a finite set of *operators*. Each operator $o \in \mathcal{O}$ has a *precondition* $\mathit{pre}(a)$ which is a partial state, a finite set of *effects* $\mathit{eff}(a)$, and a *cost* $\mathit{cost}(a) \in \mathbb{R}_0^+$. Each effect is a triple $\langle c, V, v \rangle$ where c is a partial state called the *effect condition*, $V \in \mathcal{V}$ is a variable, and $v \in \mathrm{dom}(V)$ is the new value for V. An operator o is *applicable* in state s iff $\mathit{pre}(o) \subseteq s$. Applying an applicable operator o in state s, written as $s[\![o]\!]$, leads to a *successor state* s'. For each variable $V \in \mathcal{V}$, $s'[V]$ is defined as:

$$s'[V] = s[\![o]\!][V] = \begin{cases} v & \text{if } \exists \langle c, V, v \rangle \in \mathit{eff}(o) : c \subseteq s \\ s[V] & \text{otherwise} \end{cases}$$

This definition assumes that effects are *conflict-free*, i.e., it is never ambiguous which value is assigned to a variable. More formally, for each state s and each applicable operator a, there are no tuples $\langle c_1, V, v_1 \rangle, \langle c_2, V, v_2 \rangle \in \mathit{eff}(a)$ with $c_1 \subseteq s$ and $c_2 \subseteq s$ and $v_1 \neq v_2$. $s_I \in S(\Pi)$ is the *initial state* and $s_\star$ is a partial state called the *goal*.

An *operator sequence* $\pi = \langle o_1, \dots, o_n \rangle$ is applicable in a state s_0 if every operator o_i is applicable in s_{i-1} and $s_i = s_{i-1}[\![o_i]\!]$ in turn. We write $s[\![\pi]\!]$ to denote the final state s_n that results from applying an applicable π of length n in s_0. The *cost* of an operator sequence π is the sum of costs of its operators, i.e., $\mathit{cost}(\pi) = \sum_{i=1}^{n} \mathit{cost}(o_i)$. We call an operator sequence π a *plan* if it is applicable in s_I and $s_I[\![\pi]\!] \supseteq s_\star$. A plan π is *optimal* if its cost is minimal among all plans.

A *transition system* $\mathcal{T} = \langle S, \mathcal{L}, T, s_I, S_\star \rangle$ is an edge-labeled, directed graph with nodes S, transition labels $\mathcal{L}$, transitions $T \subseteq S \times \mathcal{L} \times S$, initial node $s_I \in S$, and goal nodes $S_\star \subseteq S$. Every planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_I, s_\star \rangle$ induces a transition system $\mathcal{T}(\Pi) = \langle S(\Pi), \mathcal{O}, T(\Pi), s_I, \{s \in S(\Pi) \mid s_\star \subseteq s\} \rangle$ where $T(\Pi) = \{\langle s, o, s' \rangle \mid s \in S(\Pi), o \in \mathcal{O}, \mathit{pre}(o) \subseteq s, s' = s[\![o]\!]\}$.

A (optimal) plan for Π corresponds to a (shortest) path from s_I to its closest goal node in $\mathcal{T}(\Pi)$ and vice versa. A common approach to find (optimal) plans is to find (shortest) paths on $\mathcal{T}(\Pi)$ with *heuristic search*. Heuristic search consists of two elements, a search algorithm and a heuristic. A search algorithm such as A* (Hart, Nilsson, and Raphael 1968) explores systematically the transition system and uses a *heuristic function* which estimates for each node the cost to reach the goal. A* in particular comes with the guarantee that if the heuristic is *admissible*, i.e., it never overestimates the cost to the goal, then the path it finds is optimal.

Abstractions Let $\mathcal{T} = \langle S, \mathcal{L}, T, s_I, S_\star \rangle$ be a transition system. An *abstraction* $\alpha : S \to S^\alpha$ maps the states of $\mathcal{T}$ to a set of *abstract states* S^α. The induced transition system is $\mathcal{T}^\alpha = \langle S^\alpha, \mathcal{L}, T^\alpha, \alpha(s_I), \{\alpha(s) \mid s \in S_\star\} \rangle$ where $T^\alpha = \{\langle \alpha(s), o, \alpha(s') \rangle \mid \langle s, o, s' \rangle \in T\}$. By construction, every path in $\mathcal{T}$ is a path in $\mathcal{T}^\alpha$. Consequently, the length of a shortest path between state $\alpha(s)$ and $\alpha(s')$ in $\mathcal{T}^\alpha$ is a lower bound to the length of a shortest path between state s and s' in $\mathcal{T}$. Thus, the abstract goal distance for a given state is an admissible estimate of the true goal distance.

There are three common abstraction heuristics: pattern databases (PDBs), merge-and-shrink (M&S) heuristics, and Cartesian abstraction heuristics.

PDBs are based on projections. The projection $\phi_P(s)$ of a (partial) state s onto a set of variables P (called pattern) is $\{V \mapsto v \mid (V \mapsto v) \in s, V \in P\}$. ϕ_P is an abstraction and induces an abstract transition system $\mathcal{T}^{\phi_P}$. For every abstract state of $\mathcal{T}^{\phi_P}$, a PDB calculates and stores its distance to the goal. When queried as heuristic on a state s, the PDB returns the stored value for $\phi_P(s)$.

Cartesian abstractions are a generalization of PDBs. Let $\mathcal{V} = \{V_1, \dots, V_n\}$ be the set of variables and $D_i \subseteq \mathrm{dom}(V_i)$ be subsets of the domains of the variables. Then $A = D_1 \times \cdots \times D_n$ is a Cartesian set. It represents a collection of states from the original task which collectively are an abstract state. An abstraction is called Cartesian if all its abstract states describe Cartesian sets of states.

Cartesian abstractions can be obtained by using counterexample-guided Cartesian abstraction refinement (CEGAR, Seipp and Helmert 2018). CEGAR starts with an abstract transition system $\mathcal{T}^\alpha$ with a single abstract state. It iteratively generates a plan π^α for $\mathcal{T}^\alpha$, executes π^α on the original task until it finds a flaw, and then refines $\mathcal{T}^\alpha$ by splitting the abstract state where π^α failed such that the flaw does not occur in future iterations. This repeats until some other stopping criterion (e.g., timeout) is reached. The final abstract transition system is used as heuristic like in the PDB setting. If at any point no plan exists, then $\mathcal{T}^\alpha$ is unsolvable and so is the original task. If at any point there is no flaw, then the plan is also a solution to the original task. For a transition $\langle A, o, B \rangle \in \pi^\alpha$ and the concrete state s, there are two kinds of flaws: (1) o is not applicable in s and (2) $s[\![o]\!] \notin B$. To refine the latter flaw, we need to know the regression of an operator and an abstract state, i.e., a set

of abstract states $regr(B, o)$ which reach B via the operator o. For more information regarding the refinement, we direct the interested reader to Seipp and Helmert (2018)

M&S (Sievers and Helmert 2021) produces even more general abstractions. The algorithm starts with a pool of abstract transition systems. Initially, this pool contains for every variable $V \in \mathcal{V}$ the abstract transition system $\mathcal{T}^{\pi_{\{V\}}}$ of the projection $\pi_{\{V\}}$. M&S iteratively merges two transition systems $\mathcal{T}^1$ and $\mathcal{T}^2$ from the pool by replacing them with their cross product $\mathcal{T}^{\otimes}$. If necessary, $\mathcal{T}^{\otimes}$ is shrunk by combining some of its states to reduce the memory footprint. These steps repeat until exactly one abstract transition system is left in the pool. As for PDBs, the perfect heuristic is calculated and stored for every state s of the final abstract transition system and used as heuristic for the state of the original transition system.

Cartesian Abstraction Refinement for Factored Effect Tasks

Seipp and Helmert (2018) introduce counterexample-guided Cartesian abstraction refinement (CEGAR) to classical planning. Their theory is limited to planning tasks without conditional effects, i.e., for all operators $o \in \mathcal{O}$ and all effects $\langle c, V, v\rangle \in eff(o)$, we have $c = \emptyset$. In this section, we extend the theory to a specific kind of planning tasks that we call *factored effect tasks*.

Factored Effect Tasks

The structure of our planning task model for Rubik's Cube (see next section) has a distinctive characteristic: every effect is conditioned only on the variable it changes. Besides Rubik's Cube, this characteristic is useful to model other permutation problems such as the *15 puzzle*, the *pancake problem*, *genome rearrangement*, and more. We define factored effect tasks to describe arbitrary planning tasks with this characteristic as follows.

Definition 1. ***Factored Effect Operator***
An operator $o \in \mathcal{O}$ is a factored effect operator *if every effect $e \in eff(o)$ is a triple $e = \langle\{V \mapsto v\}, V, v'\rangle$. We abbreviate the notation to $e = \langle V, v, v'\rangle$.*

Definition 2. ***Factored Effect Task***
A factored effect task *is a planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_I, s_\star\rangle$ such that all $o \in \mathcal{O}$ are factored effect operators.*

For an operator $o \in \mathcal{O}$, we define $vars(eff(o)) = \{V \mid \langle V, v, v'\rangle \in eff(o)\}$, i.e., the variables *affected* by o. Furthermore, let $eff(o, V) = \{e \in eff(o) \mid e = \langle V, v, v'\rangle\}$ be the set of effects of operator o that affect variable $V \in \mathcal{V}$. To ensure that a factored effect operator is conflict-free, we require that $v_1 \neq v_2$ for all pairs $\langle V, v_1, v_1'\rangle, \langle V, v_2, v_2'\rangle \in eff(o, V)$ and for all $V \in \mathcal{V}$.

Cartesian CEGAR

There are two obstacles for Cartesian CEGAR (Seipp and Helmert 2018) in the presence of conditional effects: (1) The regression of a Cartesian set with respect to an operator with conditional effects is not necessarily Cartesian (violation of Seipp and Helmert's Property 4); (2) It is impossible to infer which facts hold after applying an operator without a state as context (required for Seipp and Helmert's Algorithm 5). We overcome these obstacles for factored effect tasks.

Algorithm 1: Transition check. Return true iff factored effect operator o induces at least one transition between abstract states A and B. Changes compared to Seipp and Helmert (2018) are highlighted.

```
function CheckTransition(A, o, B)
  foreach V_i ∈ 𝒱 do
    if V_i ∈ vars(pre(o)) and pre(o)[V] ∉ A_i then
      return false
    if V_i ∈ vars(eff(o)) and
      possible(V_i, A, o) ∩ B_i = ∅ then
      return false
    if V_i ∉ vars(eff(o)) and A_i ∩ B_i = ∅ then
      return false
  return true
```

Theorem 1. *The regression of a Cartesian set with respect to a factored effect operator o is Cartesian.*

Proof. Let $B = B_1 \times \cdots \times B_n$ be a Cartesian set representing an abstract state over variables $\mathcal{V} = \{V_1, \ldots, V_n\}$ where $B_i \subseteq \mathrm{dom}(V_i)$. A factored effect operator o changes the value of a variable independent of the other variables. Hence, we can look at all variables individually to derive the regression of an abstract state.

Let us consider the regression of a single variable V_i. There are two cases why $b \in B_i$ after applying o in an abstract state $A = A_1 \times \cdots \times A_n$: either $b \in A_i$ and there is no effect $\langle V_i, b, x\rangle \in eff(o)$ or $\langle V_i, a, b\rangle \in eff(o)$. For regression, we reason over the possible values A_i of V_i preceding B as follows.

$$A_i = \{a \in \mathrm{dom}(V_i) \mid \exists\langle V_i, a, b\rangle \in eff(o) : b \in B_i\} \cup \{b \in B_i \mid \nexists\langle V_i, b, x\rangle \in eff(o)\}$$

Then, the regression of Cartesian set B with respect to factored effect operator $o \in \mathcal{O}$ is $regr(B, o) = A_1 \times \cdots \times A_n$ where $A_i \subseteq \mathrm{dom}(V_i)$. Clearly, $regr(B, o)$ is Cartesian. □

To address (2), we use a similar argument as in the proof of Theorem 1. We define the set of possible values for a variable V_i in the successor of an abstract state A through factored effect operator o. Recall that operators are conflict-free and hence at most one effect $\langle V, v, v'\rangle$ exists for each atom $V \mapsto v$.

$$possible(V_i, A, o) = \bigcup_{a \in A_i} \begin{cases} b & \text{if } \langle V_i, a, b\rangle \in eff(o) \\ a & \text{otherwise} \end{cases}$$

We can then use this to adapt Algorithm 5 of Seipp and Helmert (2018). To check whether a transition between two abstract states A and B exists for operator o, each variable V_i must have a *possible* successor in B_i (cf. Algorithm 1; the changes compared to Seipp and Helmert are highlighted in red).

Rubik's Cube Model

Rubik's Cube is a permutation puzzle invented by Ernő Rubik in 1974 (cf. Figure 1). It is a 3-dimensional cube with 6 *faces*. The cube is sliced into 3 *layers* in each dimension, resulting in 27 smaller cubes called *cubies*. The faces of the cubies are called *facelets* and each cubie is colored differently on all its facelets. The puzzle is solved when all facelets on the same face of the cube have the same color. Each layer can be rotated by multiples of 90° around the cubie in its center. Rotating a layer rearranges the cubies and changes the colors on the faces. In an initial configuration of the puzzle, the cube is scrambled by an arbitrary, unknown sequence of rotations. The goal is to find a sequence of rotations which leads to the solved state where all faces show a single color only. Simple algorithms exist which find such sequences (Kociemba 1992), but finding the shortest such sequence is challenging.

We describe some properties of Rubik's Cube in the following list:

(1) Rotating a layer changes the position and orientation of its cubies, but never reveals new facelets. The number and colors of visible facelets never changes for any cubie.

(2) There are 4 kinds of cubies differing by the number of visible facelets: 1 *inner* cubie not visible at all; 6 *center* cubies showing 1 facelet; 12 *edge* cubies showing 2 facelets; and 8 *corner* cubies showing 3 facelets.

(3) Rotating the middle layer of any dimension is equivalent to rotating the other layers of the same dimension in the opposite direction. Hence, we forbid rotating the middle layer without changing the possible configurations of Rubik's Cube. As a consequence, the center cubies never change their position and we can use them to identify the cube faces.

(4) There are 12 positions for the edge cubies and 8 positions for the corner cubies. Each position can be described using the faces it touches.

(5) At each position, edge cubies can occur in 2 possible orientations and corner cubies can occur in 3 possible orientations.

Based on these observations, we model Rubik's Cube as a family of factored effect tasks $\Pi_i = \langle \mathcal{V}_e \cup \mathcal{V}_c, \mathcal{O}, s_i, s_\star \rangle$. We address the faces as front (F), back (B), left (L), right (R), upper (U), and downward (D). $\mathcal{F}_i$ denotes the faces which lie on the i-th dimension of Rubik's Cube: $\mathcal{F}_1 = \{F, B\}$, $\mathcal{F}_2 = \{L, R\}$, $\mathcal{F}_3 = \{U, D\}$. Furthermore, $\mathcal{F} = \bigcup_i \mathcal{F}_i$ is the set of all faces and the faces *adjacent* to some $f \in \mathcal{F}_i$ is $\mathcal{A}_f = \mathcal{F} \setminus \mathcal{F}_i$. We denote the face that follows face f' in clockwise turn order from the perspective of face f as $\mathcal{A}_f[f']$, e.g., $\mathcal{A}_F[U] = R$.

$\mathcal{V}_c$ contains 8 variables representing the corner cubies. According to observations (4) and (5) they can be described by their position using the three faces they touch and by one of three possible orientations. Hence, the domain of a corner cubie $V_c \in \mathcal{V}_c$ is $\mathrm{dom}(V_c) = \{\langle \{f_1, f_2, f_3\}, o \rangle \mid f_1 \in \mathcal{F}_1, f_2 \in \mathcal{F}_2, f_3 \in \mathcal{F}_3, o \in \{1, 2, 3\}\}$. Similarly, $\mathcal{V}_e$ contains 12 variables representing the edge cubies. To describe the edge cubies, we require their position described by the two faces they touch and their orientation. Hence, the domain of an edge cubie $V_e \in \mathcal{V}_e$ is $\mathrm{dom}(V_e) = \{\langle \{f_1, f_2\}, o \rangle \mid f_1 \in \mathcal{F}_i, f_2 \in \mathcal{F}_j, i \neq j, o \in \{1, 2\}\}$.

$\mathcal{O}$ has one operator *rotate*$(f, angle)$ for each face $f \in \mathcal{F}$ and for the angles 90° (clockwise), 180° (half-turn), and 270° (counter-clockwise). All operators $o \in \mathcal{O}$ are always applicable, i.e. *pre*$(o) = \emptyset$. An operator *rotate*$(f, angle)$ changes the position and rotation of any cubie positioned on f.

Let's start with describing the effect of *rotate*$(f_1, 90)$. We first model how the cubies on face f_1 change their position. A corner cubie at position $\{f_1, f_2, f_3\}$ moves to $\{f_1, \mathcal{A}_{f_1}[f_2], \mathcal{A}_{f_1}[f_3]\}$. More specifically, the faces f_2 and f_3 are updated to the next face in clockwise direction (from the perspective of f_1). Similarly, an edge cubie at position $\{f_1, f_2\}$ moves to position $\{f_1, \mathcal{A}_{f_1}[f_2]\}$.

To easily model how the orientation changes, we use a trick. We represent the orientation of a cubie as triple. More specifically, we represent the orientation of a corner cubie as a permutation of $\{1, 2, 3\}$, and the orientation of an edge cubie as a permutation of $\{1, 2, \#\}$. Where # represents a *blank* symbol. The first non-blank value identifies the orientation of the cubie. If we now rotate any cubie with an orientation $\langle o_1, o_2, o_3 \rangle$ around the dimension $d \in \{1, 2, 3\}$, then the new orientation of the cubie is constructed by exchanging the elements of the triple which are not at position d, e.g. a rotation around $d = 2$ leads to the new triple $\langle o_3, o_2, o_1 \rangle$. This is possible, because we observe that for any corner cubie, there are only three different triples reachable at every position and for any edge cubie there are only 2 triples reachable at any position. Thus, given the cubie, its position, and its orientation, we can construct the correct triple, modify it, and extract the new orientation value. We call the function described in this paragraph $next_orientation$.

Putting this together, the effects of a 90° rotation of f_1 are:

$$
\begin{aligned}
&\textit{eff}(\textit{rotate}(f_1, 90)) = \{ \\
&\langle V_c, \langle \{f_1, f_2, f_3\}, o \rangle, \langle \{f_1, \mathcal{A}_{f_1}[f_2], \mathcal{A}_{f_1}[f_3]\}, o' \rangle \rangle \mid \\
&\qquad V_c \in \mathcal{V}_c, f_1 \in \mathcal{F}_d, \\
&\qquad o' = \textit{next_orientation}(V_c, \{f_1, f_2, f_3\}, o, d) \\
&\} \cup \{ \\
&\langle V_e, \langle \{f_1, f_2\}, o \rangle, \langle \{f_1, \mathcal{A}_{f_1}[f_2]\}, o' \rangle \rangle \mid \\
&\qquad V_e \in \mathcal{V}_e, f_1 \in \mathcal{F}_d \\
&\qquad o' = \textit{next_orientation}(V_c, \{f_1, f_2\}, o, d) \\
&\}
\end{aligned}
$$

Rotating a face for 180° or 270° is modeled by replacing every occurence of $\mathcal{A}_{f_1}[f_2]$ by two respectively 3 calls to the same function and doing the same for $next_orientation$.

The goal state $s_\star$ has in each position one cubie and all cubies have an orientation of 1. Any state reachable by applying a sequence of operators can be the initial state s_i.

Experiments

We evaluate the performance of heuristics based on PDBs, Cartesian abstractions, and M&S abstractions on the prob-

lem of Rubik's Cube using Fast Downward (Helmert 2006). We add support for factored effect tasks in the generation of Cartesian abstractions using CEGAR. In this section, we describe the tasks we use for our evaluation, give an overview of the compared planner configurations (i.e., heuristics), and discuss their performance. All our code, data, and models will be available online.

Benchmarks

We implement a problem generator for Rubik's Cube tasks according to our model. To avoid issues with the translator component of Fast Downward (Helmert 2009), our generator writes the tasks in the input format for the search component. Our generator starts from the solved configuration and applies n arbitrary rotations where n is a user specified value. We ensure that two consecutive rotations concern different faces, because two consecutive rotations of the same face can be combined to one rotation. Nevertheless, some sequences of rotations can cancel each other out. Thus, n is just an upper bound on the minimum plan length.

We generate 10 tasks for each $n \in \{1, \ldots, 20\}$ to obtain a benchmark set with diverse difficulty levels. We choose 20 as the upper limit because it has been shown that the most difficult Rubik's Cube instances require 20 moves to solve (Rokicki et al. 2014). The benchmark set is ensured to contain no duplicates, which leaves us with a total of 200 distinct Rubik's Cube instances for our evaluation.

Evaluated Heuristics

While we compare the performance of 3 abstraction classes, there is a total of 5 configurations that we include in our evaluation. We briefly describe them here.

Blind Heuristic We use a blind search denoted as blind as baseline for our investigations. It corresponds to a breadth-first exploration of the transition system.

Pattern Database Heuristics For PDBs, the crucial decisions include how to choose the patterns and how to combine their individual heuristic values. Korf (1997) split the variables in two patterns, one for the edge and one for the corner cubies. Korf used a specialized solver explicitly designed for Rubik's Cube. In contrast, Fast Downward is a domain-independent planner without optimizations for Rubik's Cube. Thus, it cannot fit the PDBs of Korf into memory. Hence, we are not able to compare to this state-of-the-art approach within this experiment. Instead, we use the largest patterns which fit into memory for our Rubik's Cube model. Those consist of 4 variables. We use the following two configurations for PDBs:

PDB-man Inspired by Korf's approach, we have two patterns containing the corner cubies: one for the cubies on the front face and one for the cubies on the back face. Additionally, we have three patterns for the edge cubies: one for the cubies on the front face, one for the cubies on the back face, and one for the cubies in the middle.

PDB-sys This configuration systematically generates all (interesting) patterns up to a certain size (Pommerening, Röger, and Helmert 2013). For Rubik's Cube, this strategy turned out to be feasible for patterns of size up to 3. The heuristic is the maximum over all these patterns.

Cartesian Abstraction Heuristic By CEGAR we denote the Cartesian CEGAR technique as discussed in this paper. We do not limit the number of states in the abstraction, but terminate the abstraction refinement after at most 900 seconds.

Merge and Shrink Heuristic The last configuration we consider is M&S. It uses bisimulation as shrinking strategy (Nissim, Hoffmann, and Helmert 2011), strongly connected components as merging strategy (Sievers, Wehrle, and Helmert 2016), and exact label reduction (Sievers, Wehrle, and Helmert 2014). We limit the abstraction to at most 50'000 states.

Setup

We implement our extension in Fast Downward Scorpion (Seipp 2018), an extension of Fast Downward. Fast Downward Scorpion supports already PDBs, M&S, as well as the original Cartesian CEGAR. We use Lab (Seipp et al. 2017) to manage our experiment. We execute A* searches with each of the five heuristics from above on all 200 benchmark tasks. The experiment is conducted on Intel Xeon-Silver 4114 processors running on 2.2 GHz with a time limit of 30 minutes and a memory limit of 3.5 GB.

Results

The most successful heuristic in terms of solved tasks is PDB-man which solves 128 tasks. PDB-sys are almost as successful with 123 solved tasks, closely followed by CEGAR which solves 120. M&S is behind with only 96 solved tasks and blind solves, as expected, the fewest tasks, namely 68. For PDB-sys, all unsolved tasks are due to the time limit. For all other configurations, the unsolved tasks are due to the memory limit.

None of the heuristics is able to solve problems where the optimal solution requires more than 12 rotations. According to Rokicki et al. (2014), this restriction allows us to solve only 0.0001% out of the $4.3 \cdot 10^{19}$ possible states of Rubik's Cube.

Figure 2 plots the number of states expanded before the last f-layer of A*. For each search, we ignore the expansions in the last f-layer, because all states in this layer, including the goal state, have the same priority for A* and it is due to random tie-breaking in which order they are expanded. We plot against PDB-man which is the heuristic that solves the most problems. It reveals that both PDB variants are similar in terms of heuristic guidance, as they expand similarly many states. M&S generally expands more states. This indicates that M&S abstractions (using our parameters) capture less of the essence of Rubik's Cube than the simpler projections on manually chosen variables. This is surprising given that M&S is the more general class of abstractions. In contrast, CEGAR seems to produce very fine-grained abstractions up to a certain size; the data points on the bottom line indicate that we obtain perfect heuristic values (within the

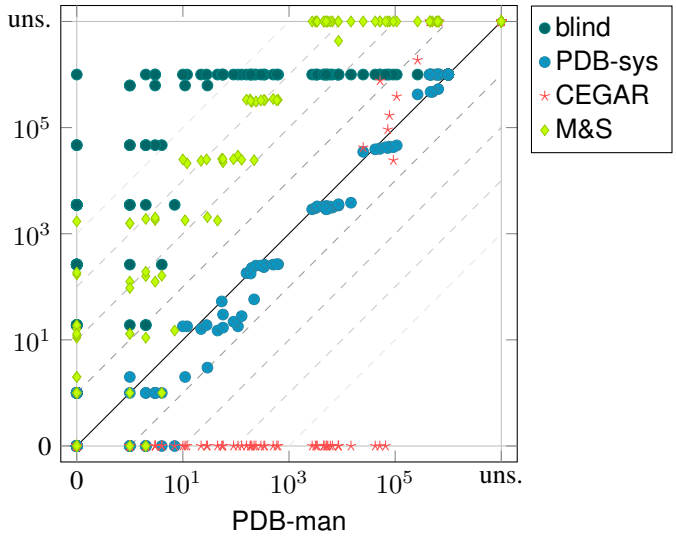

Figure 2: Number of state expansions before the last f-layer.

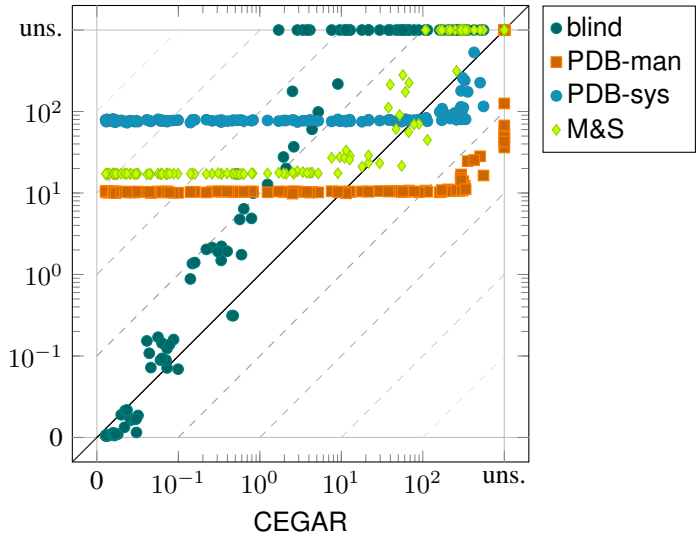

Figure 3: Runtime in seconds.

search space) during the search with our Cartesian abstractions. However, this changes abruptly when the problem instances become harder. At some point, the guidance quickly deteriorates and becomes worse than the guidance of PDB-man.

Looking at the runtimes reveals more information about the techniques for generating abstractions in the first place. Figure 3 plots all configurations against CEGAR because it has the most evenly distributed runtimes for the problems of different difficulties. blind is faster on the trivial tasks but is quickly outperformed by all other heuristics. For PDB-man, PDB-sys, and M&S, the preprocessing step constructing the abstractions and precomputing the heuristic, dominates the required time on most instances. Only after a certain difficulty, the search time starts to show up in Figure 3, namely when the constant line starts to ascend. Meanwhile, CEGAR allows to terminate the construction of the abstraction early if it finds a solution during the refinement step. This is an advantage that pays off for the simpler problems but is later outmatched by the PDB based techniques.

Interestingly, CEGAR yields constructs regularly the perfect heuristic. This happens mostly in the instances where it finds a solution during the abstraction refinement, but not always.

Discussion

Overall, the performance of CEGAR looks promising and competitive until we reach the point where the heuristic quality deteriorates and many state expansions are required. We are currently not able to explain why exactly this happens. Given that the state space of Rubik's Cube is huge, we assume that the abstraction is fine-grained locally around the initial state, but becomes almost blind when reaching the borders of that area. This is a contrast to the pattern databases, which ignore parts of the information evenly among the entire state space.

The poor performance of M&S is another surprising result. The initial heuristic values for M&S are the worst among all considered heuristics and they never surpass the value 4. Potentially, the chosen size limit of at most 50'000 abstract states is too restrictive for Rubik's Cube.

Conclusion

We analyze the performance of modern abstraction heuristics on the problem of Rubik's Cube. First, we present a model for Rubik's Cube as finite-domain planning task which allows us to use general problem solvers on this task. Our model relies on a special kind of conditional effect where the effect condition is concerned exactly with the variable changed by the effect; when rotating a face of Rubik's Cube, the cubies change location and orientation based on their own previous location and orientation. We introduce these problems as factored effect tasks.

While conditional effects are supported in PDBs and M&S abstractions, this is not the case for Cartesian abstractions created using Counterexample-guided abstraction refinement. Hence, we secondly extend this theory for factored effect tasks.

Finally, we evaluate the performance of heuristic search based on the aforementioned abstraction classes on a newly generated benchmark set of Rubik's Cube tasks. The experiment reveals that PDBs remain untouched as the state-of-the-art for this domain, at least not in terms of overall coverage of problem instances. However, CEGAR yields promising heuristics for problem instances up to a certain difficulty. Future work investigates further what exactly is the limitation for Cartesian CEGAR.

A missing piece in the hierarchy of abstractions are *domain abstractions*. We are not aware of any work that generates domain abstractions for classical planning problems in a principled way. Filling this gap would be a major accomplishment and could shed more light in the analysis of

abstraction heuristics for Rubik's Cube.

Rubik's Cube is one domain in a family of permutation puzzles with many similarities. Other domains like sliding tiles, the pancake problem, genome rearrangement, and many more can be modeled as factored effect tasks as well. It would be interesting to see whether the abstraction classes compare similarly for all of these domains.

References

Bäckström, C.; and Nebel, B. 1995. Complexity Results for SAS$^+$ Planning. *Computational Intelligence*, 11(4): 625–655.

Culberson, J. C.; and Schaeffer, J. 1998. Pattern Databases. *Computational Intelligence*, 14(3): 318–334.

Domshlak, C.; Hoffmann, J.; and Katz, M. 2015. Red-black planning: A New Systematic Approach to Partial Delete Relaxation. *Artificial Intelligence*, 221: 73–114.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *Artificial Intelligence*, 173: 503–535.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *Journal of the ACM*, 61(3): 16:1–63.

Hernádvölgyi, I. T.; and Holte, R. C. 2000. Experiments with Automatically Created Memory-Based Heuristics. In Choueiry, B. Y.; and Walsh, T., eds., *Proceedings of the 4th International Symposium on Abstraction, Reformulation and Approximation (SARA 2000)*, volume 1864 of *Lecture Notes in Artificial Intelligence*, 281–290. Springer-Verlag.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Kociemba, H. 1992. Close to God's algorithm. *Cubism for Fun*, 28: 10–13.

Korf, R. E. 1997. Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI 1997)*, 700–705. AAAI Press.

Nissim, R.; Hoffmann, J.; and Helmert, M. 2011. Computing Perfect Heuristics in Polynomial Time: On Bisimulation and Merge-and-Shrink Abstraction in Optimal Planning. In Walsh, T., ed., *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, 1983–1990. AAAI Press.

Pommerening, F.; Röger, G.; and Helmert, M. 2013. Getting the Most Out of Pattern Databases for Classical Planning. In Rossi, F., ed., *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, 2357–2364. AAAI Press.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, 39: 127–177.

Rokicki, T.; Kociemba, H.; Davidson, M.; and Dethridge, J. 2014. The Diameter of the Rubik's Cube Group is Twenty. *SIAM Review*, 56(4): 645–670.

Seipp, J. 2018. Fast Downward Scorpion. In *Ninth International Planning Competition (IPC-9): Planner Abstracts*, 77–79.

Seipp, J.; and Helmert, M. 2018. Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning. *Journal of Artificial Intelligence Research*, 62: 535–577.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

Sievers, S.; and Helmert, M. 2021. Merge-and-Shrink: A Compositional Theory of Transformations of Factored Transition Systems. *Journal of Artificial Intelligence Research*, 71: 781–883.

Sievers, S.; Wehrle, M.; and Helmert, M. 2014. Generalized Label Reduction for Merge-and-Shrink Heuristics. In Brodley, C. E.; and Stone, P., eds., *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2014)*, 2358–2366. AAAI Press.

Sievers, S.; Wehrle, M.; and Helmert, M. 2016. An Analysis of Merge Strategies for Merge-and-Shrink Heuristics. In Coles, A.; Coles, A.; Edelkamp, S.; Magazzeni, D.; and Sanner, S., eds., *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2016)*, 294–298. AAAI Press.