On Solving the Rubik's Cube with Domain-Independent Planners Using Standard Representations

Bharath Muppasani, Vishal Pallagani, Biplav Srivastava, Forest Agostinelli

AI Institute, University of South Carolina, Columbia, South Carolina, USA
{bharath@email., vishalp@email., biplav.s@, foresta@cse.}sc.edu

Abstract

Rubik's Cube (RC) is a well-known and computationally challenging puzzle that has motivated AI researchers to explore efficient alternative representations and problem-solving methods. The ideal situation for planning here is that a problem be solved optimally and efficiently represented in a standard notation using a general-purpose solver and heuristics. The fastest solver today for RC is DeepCubeA with a custom representation, and another approach is with Scorpion planner with State-Action-Space+ (SAS+) representation. In this paper, we present the first RC representation in the popular PDDL language so that the domain becomes more accessible to PDDL planners, competitions, and knowledge engineering tools, and is more human-readable. We then bridge across existing approaches and compare performance. We find that in one comparable experiment, DeepCubeA[1] solves all problems with varying complexities, albeit only 18% are optimal plans. For the same problem set, Scorpion with SAS+ representation and pattern database heuristics solves 61.50% problems, while FastDownward with PDDL representation and FF heuristic solves 56.50% problems, out of which all the plans generated were optimal. Our study provides valuable insights into the trade-offs between representational choice and plan optimality that can help researchers design future strategies for challenging domains combining general-purpose solving methods (planning, reinforcement learning), heuristics, and representations (standard or custom).

Introduction

The Rubik's Cube is a 3D puzzle game that has been widely popular since its invention in 1974. It has been a subject of interest for researchers in Artificial Intelligence (AI) due to its computational complexity and potential for developing efficient problem-solving algorithms. RC has motivated researchers to explore alternative representations that simplify the problem while preserving its complexity. Efficient algorithms have been developed to solve RC in the least number of moves, and they have been used in various applications, including robot manipulation, game theory, and machine learning. Therefore, in this paper, we aim to explore the different representations and algorithms to solve RC and evaluate their performance and effectiveness in solving this challenging puzzle.

Various solution approaches have been proposed RC including Reinforcement Learning (RL) and search. For instance, DeepCubeA (Agostinelli et al. 2019a) uses RL to learn policies for solving RC, where the cube state is represented by an array of numerical features. Although DeepCubeA is a domain-independent puzzle solver, it employs a custom representation for RC. On the other hand, Büchner et al. (2022) utilized SAS+ representation to model the RC problem in a finite domain representation, which enables standard general-purpose solvers like Scorpion to be used on the RC problem. Despite the success of these approaches, no prior work has explored the use of Planning Domain Definition Language (PDDL) to encode a 3x3x3 RC problem. While a previous study[2] has encoded a 2x2x2 RC problem using PDDL and solved it with a Fast-Forward planner, there exists no PDDL encoding for a 3x3x3 RC problem.

In this paper, we introduce a novel approach for representing RC in PDDL. We encode the initial state and goal state using a set of predicates, each of which specifies the color of a sticker on a particular cube piece or edge piece. We then define the actions that can be taken to manipulate the cube pieces and edges. Our PDDL representation enables us to model RC as a classical planning problem, which can be solved using off-the-shelf planning tools. To the best of our knowledge, this is the first attempt to represent RC formally using PDDL. We also evaluate the effectiveness of our approach by comparing it with other state-of-the-art representations in terms of the efficiency and effectiveness of problem-solving. Our major contributions are:

- We develop the first PDDL formulation for the 3x3x3 Rubik's Cube, which is a novel and significant contribution to the existing literature. This PDDL formulation will enable the use of standard PDDL planners for solving Rubik's Cube problems, which was not previously possible.
- We bridge across hither-to incomparable RC solving approaches, compare their performance and draw insights from results to facilitate new research.

[1]DeepCubeA trained with 12 RC actions

[2]https://wu-kan.cn/2019/11/21/Planning-and-Uncertainty/

Notations	Description
RC	Rubik's Cube
PDDL	Planning Domain Description Language (Fox and Long 2003)
SAS+	State-Action-Space+ (Fikes and Nilsson 1971a)
Custom	RC representation in DeepCubeA (Agostinelli et al. 2019a)
Blind	FastDownward with Blind[3]
GC	FastDownward with Goal count[3]
CG	FastDownward with Causal Graph[3]
CEA	FastDownward with Context-enhanced Additive[3]
LM-Cost	FastDownward with LM-Cost Partitioning[3]
FF	FastDownward with FF[3]
M&S	Scorpion with Merge & Shrink[4]
PDB-Man	Scorpion with Max Manual PDB[4]
PDB-Sys	Scorpion with Max Systematic PDB[4]
d1	Dataset of 200 problems generated considering 12 RC actions
d2	Dataset of 200 problems generated considering 18 RC actions
m1	PDDL model with 12 RC actions
m2	PDDL model with 18 RC actions

Table 1: Notations or abbreviations and their descriptions.

- We perform a comparative analysis of two formal languages, SAS+ and PDDL, and custom one in DeepCubeA, a RL approach for solving RC on a set of common benchmark RC problems. This comparative analysis is important as it provides insights into the strengths and weaknesses of these different approaches, and helps to identify which method may be most appropriate for a given problem setting.

The paper is organized as follows: we begin with giving an overview of Rubik's Cube solving ecosystem, including the RC problem, domain-independent planners and heuristics, and learning-based RC solvers. Then, we present a comparison of three different representations for RC: DeepCubeA, SAS+, and PDDL. Next, we outline the experiments conducted, including the heuristics considered and the experimental setup, followed by results. We compare RC solvers and heuristics for the number of problems solved and plan optimality. Finally, the paper concludes with a discussion of the findings and their implications for future research in solving larger RC problems.

The RC Solving Ecosystem

In this section, we describe the RC problem, planners, and heuristics that are used for our study. Table 1 summarizes the notations used and Figure 1 shows the entire functionality of our designed ecosystem.

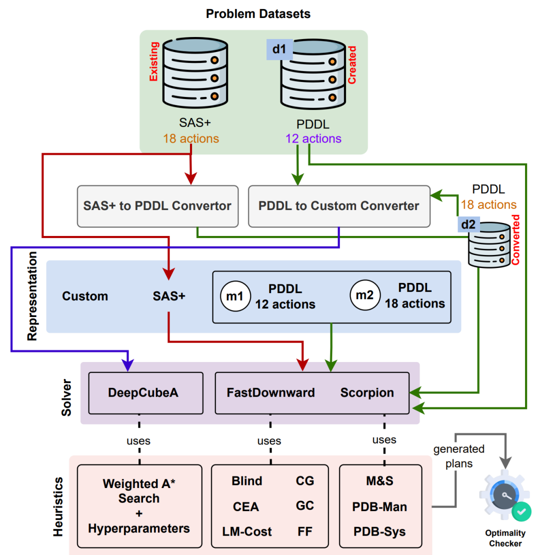


Figure 1: Setup for comparing RC representation, solvers and heuristics. Converters crucially bridge representations.

RC Problem

The Rubik's Cube is a 3-D combination puzzle with colored faces made up of 26 smaller colored pieces linked to a central spindle, with the goal of rotating the blocks until each face of the cube is a single color. To solve the puzzle, one can perform certain actions that correspond to the different faces of the cube. The major actions of a Rubik's cube are Up(U), Down(D), Right(R), Left(L), Front(F), and Back(B), which define a rotation of 90 degrees in a clockwise direction of the respective face per action. The inverse of these actions corresponds to a 90-degree rotation in the anti-clockwise direction (suffix 'rev'). The cube is initially rotated by a random sequence of rotations in the puzzle's initial configuration. The goal is to find a series of rotations that results in the solved state, which has all faces displaying the same color. One can solve the RC from a scrambled state to the original solved configuration by performing a set of the above-mentioned actions.

Domain-Independent Planners and Heuristics

Classical Planning Formalism Consider F to be a set of propositional variables or *fluents*. A *state* $s \subseteq F$ is a subset of fluents that are true, while all fluents in $F \setminus s$ are implicitly assumed to be false. A subset of fluents $F' \subseteq F$ holds in a state s if and only if $F' \subseteq s$. A classical planning instance is a tuple $P = \langle F, A, I, G \rangle$, where F is a set of fluents, A is a set of actions, $I \subseteq F$ an initial state, and $G \subseteq F$ a goal condition. Each action $a \in A$ has precondition pre$(a) \subseteq F$, add effect add$(a) \subseteq F$, and delete effect del$(a) \subseteq F$, each a subset of fluents. Action a is applicable in state $s \subseteq F$ if and only if pre(a) holds in s, and applying a in s results in a new state $s \oplus a = (s \setminus$ del$(a)) \cup$ add(a). A *plan* for P is a sequence of actions $\prod = \langle a_1, ..., a_n \rangle$ such that a_1 is

[3]https://www.fast-downward.org/Doc/Evaluator
[4]https://jendrikseipp.github.io/scorpion/Evaluator/

applicable in I and, for each $2 \leq i \leq n, a_i$ is applicable in $I \oplus a_1 \oplus ... \oplus a_{i-1}$. The plan $\prod$ solves P if G holds after applying $a_1, ..., a_n$, i.e. $G \subseteq I \oplus a_1 \oplus ... \oplus a_n$.

Abstractions Let $\mathcal{T} = \langle S, \mathcal{L}, T, s_I, S_* \rangle$ be a transition system. An *abstraction* $\alpha : S \rightarrow S^\alpha$ maps the states of $\mathcal{T}$ to a set of *abstract states* S^α. The induced transition system is $\mathcal{T}^\alpha = \langle S_\alpha, \mathcal{L}, T^\alpha, \alpha(s_I), \{\alpha(s) | s \in S_*\} \rangle$ where $T^\alpha = \{\langle \alpha(s), o, \alpha(s') \rangle | \langle s, o, s' \rangle \in T\}$. By construction, every path in $\mathcal{T}$ is a path in $\mathcal{T}^\alpha$. Consequently, the length of the shortest path between state $\alpha(s)$ and $\alpha(s')$ in $\mathcal{T}^\alpha$ is a lower bound on the length of the 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 (Büchner et al. 2022). In the later section of the paper, we mention the abstraction heuristics used for our work.

PDDL The field of planning has seen many representations. For example, in classical planning, there was STRIPS (Fikes and Nilsson 1971b), Action Description Language (ADL) (Pednault 1994) and SAS+ (Bäckström 1995) before Planning Domain Description Language (PDDL) (McDermott et al. 1998; Fox and Long 2003) standardized the notations. Nowadays, planners routinely use PDDL for problem specification even if they may convert to other representations later for solving efficiency (Helmert 2009). PDDL envisages two files, a domain description file which specifies information independent of a problem like predicates and actions, and a problem description file which specifies the initial and goal states. A problem is characterized by an initial state, together with a goal state that the agent wants to transition to, both states specified as configurations of objects. A planner takes as input the domain and problem file to generate a plan, which can be verified using a plan validator, VAL (Howey and Long 2003).

Learning-based RC Solver

There exist specialized solvers for solving the Rubik's Cube, which can be classified as either domain-dependent or domain-independent. DeepCubeA is an example of a domain-independent solver that employs custom representation encoding for RC, as proposed in (McAleer et al. 2018; Agostinelli et al. 2019b). The solver employs a weighted A* search algorithm and learns a domain-dependent heuristic via deep-learning, resulting in state-of-the-art performance. However, interpreting the solutions provided by DeepCubeA remains a challenge (McAleer et al. 2019), as does comparing its performance with that of other solvers *on identical problem instances*.

Comparision of RC representations

In this section, we describe and provide a comparative analysis of different RC representations comprising of RL and Planning formal languages.

DeepCubeA

The DeepCubeA algorithm adopts a unidimensional array as a representation of the Rubik's Cube (RC) state. Specifically, this array encompasses 54 elements, each of which corresponds to a unique sticker color present on a cube piece of the RC. While this array-based modeling offers computational advantages, it is limited by its inability to fully encapsulate the spatial orientation of Rubik's Cube. Furthermore, the usage of a hard-coded representation and implicit assumptions concerning the position of cubelets poses a challenge to novice users seeking to comprehend the array-based representation.

Listing 1: Action L of Rubik's Cube modeled in PDDL

```
(:action L
:effect (and
;for corner cubelets
(forall(?x ?y ?z)(when (cube1 ?x ?y ?z)
  (and (cube2 ?y ?x ?z))))
(forall(?x ?y ?z)(when (cube3 ?x ?y ?z)
  (and (cube1 ?y ?x ?z))))
(forall(?x ?y ?z)(when (cube4 ?x ?y ?z)
  (and (cube3 ?y ?x ?z))))
(forall(?x ?y ?z)(when (cube2 ?x ?y ?z)
  (and (cube4 ?y ?x ?z))))
;for edge cubelets
(forall(?x ?z)(when (edge13 ?x ?z)
  (and (edge12 ?x ?z))))
(forall(?y ?z)(when (edge34 ?y ?z)
  (and (edge13 ?y ?z))))
(forall(?x ?z)(when (edge24 ?x ?z)
  (and (edge34 ?x ?z))))
(forall(?y ?z)(when (edge12 ?y ?z)
  (and (edge24 ?y ?z))))))
```

SAS+

In Büchner et al. 2022, Rubik's Cube is modeled with 18 actions in SAS+ representation as a factored effect task, with each face labeled as F, B, L, R, U, or D. The orientation of each cube piece is represented as a triple of values, and for corner cube pieces, the orientation is a permutation of {1, 2, 3}, while for edge cube pieces, it is a permutation of {1, 2, #} (where # represents a blank symbol). The rotation of the cube in 3D space is captured as the permutation of the respective triple for each cube piece. The SAS+ model has 20 variables, 480 fact pairs and bytes required for representing each state is 16 bytes.

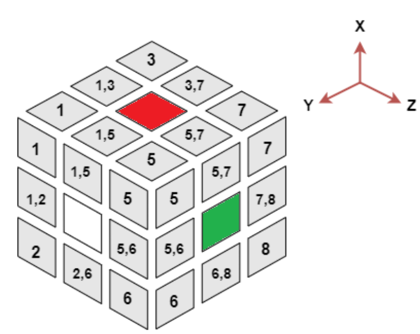


Figure 2: Rubik's cube description to define the domain encoding.

PDDL

In the PDDL domain, the Rubik's cube problem environment has been defined by assuming the cube pieces are in a fixed position and are named accordingly, as defined in Figure 2. These fixed cube pieces are modeled as predicates in the RC domain and the colors they possess in the three-dimensional space as parameters of these predicates. With the help of conditional effects, each action in the RC environment is defined as the change of colors on these fixed cube pieces. The 3D axis of the cube is considered as three separate parameters *X*, *Y*, and *Z* that specify the position of the colors on the cube's pieces. One of these axes can be connected to each face of the cube. According to the representation shown in Figure 2, the respective faces on each axis are: $F_X = \langle U, D\rangle$; $F_Y = \langle R, L\rangle$; $F_Z = \langle F, B\rangle$. These different faces of the cube can be identified by the color of the middle cube piece. We considered White, Red, and Green colors as the colors on the front(F), up(U) and right(R) faces respectively (similarly, the counter colors on the counter faces).

The following conventions regarding the RC cube pieces are considered to model the RC domain actions in the PDDL:

1. The corner cube pieces of the RC are modeled as a three-color cubelet and are specified as a predicate with three parameters: *x*, *y*, and *z*, which indicate the piece's colors on three separate axes. There are 8 corner pieces in RC.
2. The edge cube pieces, which are in between corner cube pieces, are modeled as two-color cubelet and is specified as a predicate with two parameters denoting the piece's colors on the two axes. There are 12 edge pieces in RC.
3. We do not consider the rotations performed on the middle layer, as this can be resolved into rotation of right and left faces in the opposite direction. As a result, the middle cube piece of a face is unaltered.

The predicate names define the fixed position of the cubelets that are defined with respect to the different faces of the cube. The representation considered for the cube positions is shown in Figure 2. One of the actions, action 'L', of RC designed in PDDL from the description provided is shown in Listing 1. In this, we refer to corner cube pieces as *cubeP* and edge cube pieces as *edgePQ* where *P* and *Q* are the numbers for the cube pieces as stated in Figure 2. When the move L is applied to the RC, for example, the left face is rotated clockwise. This may be regarded as a 90-degree clockwise translation of colors from the left-face corner and edge cube pieces. Considering the RC representation shown in Figure 2, the colors on the pieces: cube1, cube2, cube4, and cube3, are circularly shifted towards the right. The same applies to the edge pieces. As the left face falls in the Z-plane, only the X-axis and Y-axis colors on the cube pieces are affected.

During execution of a problem, FastDownward first translates the domain into a SAS model. The resulting SAS version of the RC PDDL domain model has 480 variables and 960 fact pairs, with each state requiring 60 bytes for representation.

Experiments

In the following section, we will discuss the heuristics considered in our evaluation and the experimental setup, which includes the datasets, problem representations, and details about the planner.

Heuristics Considered

Blind heuristic Blind heuristic refers to a decision-making strategy that does not incorporate any specific information regarding the problem domain. It relies solely on the present state of the problem and employs a trial-and-error method to find a solution.

Causal Graph heuristic This heuristic is predicated on a causal graph that delineates the causal relationships among the diverse variables in the problem. It is frequently employed in planning problems where actions have intricate preconditions.

Context-enhanced additive heuristic Additive heuristic functions combine multiple heuristics to get a more accurate evaluation of a problem's solution space. Context-enhanced additive heuristics improve on this by incorporating the problem context into the heuristic estimates, leading to even more accurate evaluations.

Goal Count heuristic The goal count heuristic estimates the number of unsatisfied goals in a state, prioritizing states with fewer unsatisfied goals. This method is useful in problems with multiple goals, such as game playing and planning, and can improve the efficiency of finding a solution.

LM cost partitioning heuristic The partitioning heuristic estimates the cost of a plan by dividing the problem into subproblems and computing their costs separately. It's helpful when the goal can be broken down into subgoals.

FF heuristic The FF heuristic is a popular heuristic function in classical planning problems. It eliminates the preconditions of the actions in the problem, making it useful for finding feasible solutions.

Merge and Shrink In the Merge and Shrink (M&S) heuristic we use 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 abstractions to 50,000 states.

Pattern Database Heuristics The key step in using Pattern Database (PDBs) heuristics is selecting appropriate patterns for the problem at hand. Korf (1997) specified two sets of patterns for solving the Rubik's Cube. We evaluate two settings of PDBs:

Max Manual PDB: Inspired by Korf's patterns, Büchner et al. (2022) have considered 2 patterns for the corner cube pieces and 3 patterns for the edge cube pieces resulting in 4 variables for each pattern. We have considered these patterns for the evaluation of PDDL and SAS+ models.

Max Systematic PDB: This configuration systematically generates all interesting patterns up to a certain size (Pommerening, Röger, and Helmert 2013). A pattern size of 3 has

Planner with Heuristic	**d1**		**d2**		
	m1	**m2**	**m1**	**m2**	**SAS+[*]**
FastDownward with Blind	78 *(96.15%)*	67 *(100%)*	56 *(39.39%)*	65 *(100%)*	66 *(100%)*
FastDownward with Causal Graph	96 *(83.33%)*	76 *(100%)*	72 *(37.50%)*	75 *(100%)*	77 *(100%)*
FastDownward with Context-enhanced Additive	99 *(87.87%)*	71 *(100%)*	68 *(48.53%)*	75 *(100%)*	77 *(100%)*
FastDownward with Goal count	103 *(89.32%)*	88 *(100%)*	75 *(29.33%)*	85 *(100%)*	87 *(100%)*
FastDownward with LM-Cost Partitioning	103 *(92.23%)*	97 *(100%)*	75 *(28%)*	86 *(100%)*	87 *(100%)*
FastDownward with FF	**137** *(88.32%)*	**135** *(100%)*	**104** *(21.15%)*	**113** *(100%)*	**123** *(100%)*
Scorpion with Merge & Shrink	114 *(88.60%)*	105 *(100%)*	82 *(25.61%)*	95 *(100%)*	90 *(100%)*
Scorpion with Max Manual PDB	95 *(91.58%)*	83 *(100%)*	64 *(35.94%)*	78 *(100%)*	123 *(100%)*
Scorpion with Max Systematic PDB	88 *(92.05%)*	78 *(100%)*	63 *(33.33%)*	73 *(100%)*	120 *(100%)*
DeepCubeA	200 *(78.50%)*		200 *(18%)*		

Table 2: Comparison of planner configurations based on the total number of solved problems and the percentage of optimal plans for different Rubik's Cube models. ([*]SAS+ dataset presented by Büchner et al. (2022))

been considered for this evaluation in the interest of memory constraints.

Experimental Setup

To compare the performance of our RC PDDL model with the existing literature work, we have used the benchmark problem test set presented by Büchner et al. (2022). In the benchmark test set, the problem tasks have been generated using 18 actions of RC - 12 actions correspond to 90-degree rotations of each face in clockwise and anti-clockwise directions, and the additional six actions are 180-degree rotation (suffix '2') on each face. The problem test set consists of 200 problems of varying difficulties. We have considered the scramble sequences provided to generate the respective PDDL versions of the problems. Additionally, we have generated our own test set of 200 RC problems considering only 12 actions. The problem generator starts from the goal state of RC and applies n arbitrary actions from the list of 12 available actions. For every value of n, ten unique random problem states are generated. The value of n is between 1 and 20. The upper limit of 20 is chosen because the authors in (Rokicki 2008) state that all the RC problem instances can be solved with at-most 20 moves. It has been considered that every consecutive rotation corresponds to a different face of the RC, as such rotations can not be combined into a single rotation.

The main difference between the two datasets *d1* and *d2* is that a 180-degree turn (half-turn) is considered as two actions in generating dataset *d1*, while it has been considered as a single action in generating dataset *d2*. The reason for evaluating two different datasets is that we wanted to capture the performance difference between the two PDDL models *m1* (12 actions) and *m2* (18 actions) in accordance with the difference in the branching factor. The PDDL model *m2* and SAS+ model have similar branching factors.

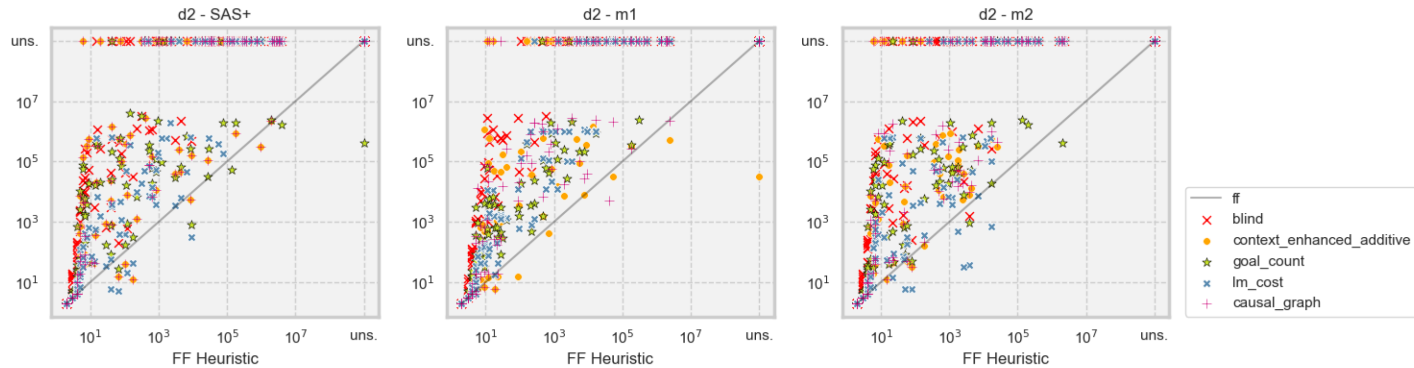
To evaluate the RC PDDL model, we have used Scorpion planner (Seipp, Keller, and Helmert 2020), which is an extension of Fast-Downward planner (Helmert 2006). Scorpion planner contains the implementation for PDBs that support conditional effects modeled in the domain file. We perform A* searches with each heuristic mentioned above on the test sets and the two PDDL models. We bound the A* search with an overall time limit of 30 minutes and a memory limit of 3.5GB. This constraint is the same for the abstraction heuristics as well, despite the fact that these heuristics require significant time for preprocessing and generating abstractions prior to the start of the search.

Result Analysis

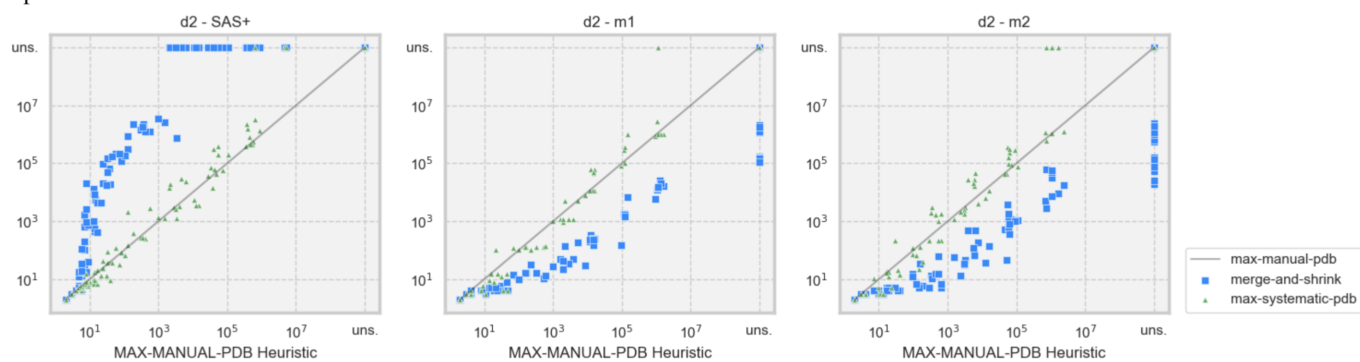
Comparision of Heuristics

We conducted an empirical evaluation of the performance of two different PDDL models (m1 and m2), each with varying numbers of modeled actions, on two test datasets (d1 and d2). Furthermore, we compared the efficacy of various heuristics on the SAS+ dataset provided by Büchner et al. (2022). We also evaluated the test datasets using DeepCubeA (Agostinelli et al. 2019a), a state-of-the-art domain-independent RC solver that leverages a combination of deep reinforcement learning and search algorithms. Our results show that DeepCubeA was able to solve all the problems in both datasets, albeit with a lower percentage of optimal plans. We provide a detailed explanation of plan optimality in the subsequent section. Table 2 presents the experimental results, including the total number of problems solved and the percentage of optimal plans generated for each configuration tested. Our findings offer valuable insights into the efficiency and effectiveness of the models, heuristics, and representations employed for solving RC problems.

1. It has been observed that abstraction heuristics are sensitive to problem representation and exhibit poorer performance in PDDL compared to SAS+ representation.
2. Interestingly, the FF heuristic, which is a non-abstraction heuristic, has been found to perform equally (in SAS+) or better than the state-of-the-art PDB heuristic with Korf's patterns in the case of PDDL representation.
3. The CG and CEA heuristics may not be effective for solving Rubik's Cube, a puzzle-solving domain with no modeled preconditions. The complex nature of the domain and large branching factor makes it challenging to construct an accurate causal graph for the CG heuristic, and the lack of contextual information renders CEA heuristic ineffective.
4. Modeling a domain with all possible actions leads to an



(a) Non-abstraction heuristics comparison evaluated using FastDownward Planner. FF heuristic has the least state expansion trend across the representations.



(b) Abstraction heuristics comparison evaluated using Scorpion Planner. M&S heuristic has lesser state expansion trend in PDDL than SAS+.

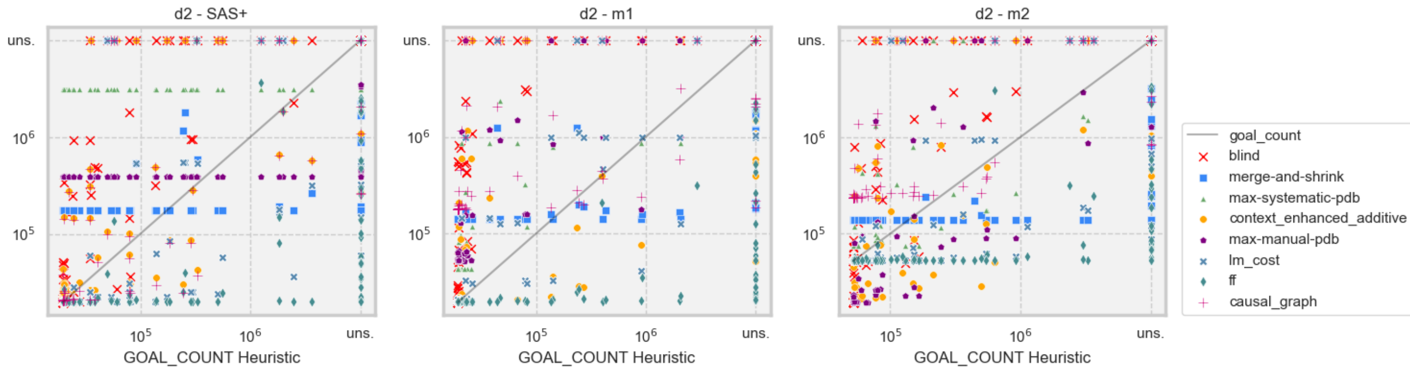
Figure 3: Comparison of the number of states expanded for dataset *d2*.

increase in the number of optimal plans.

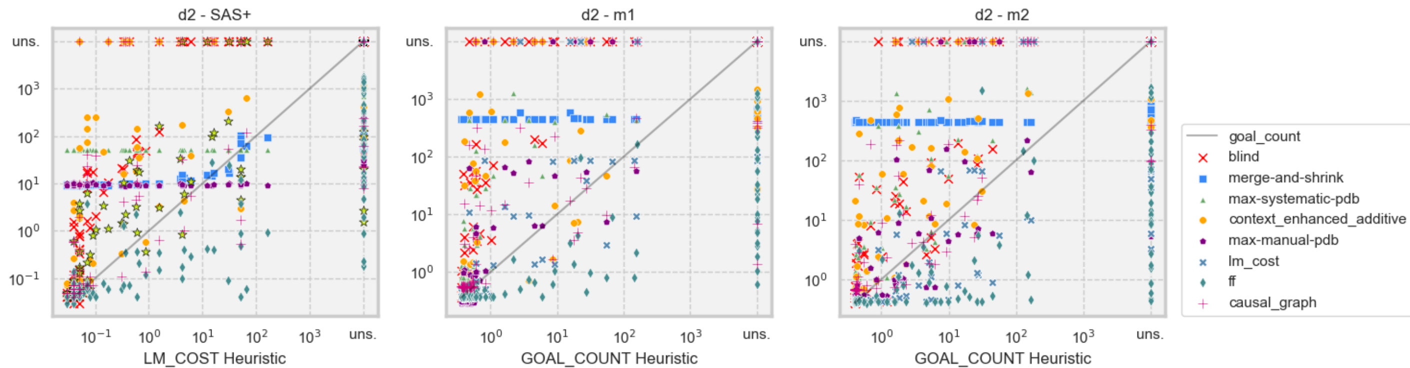
In this paper, we provide plots that compare the states expanded, runtime, and memory usage for the dataset *d2* using both PDDL and SAS+ representations. We have chosen to focus on the performance of dataset *d2* because it can be compared with the SAS+ dataset provided in the study by Büchner et al. 2022 as they have similar branching factor. Comparable figures for dataset *d1* are available in the supplementary material, and the conclusions reached in the paper are consistent with those results. Additionally, we include supplementary plots that depict the number of states expanded, memory usage, and runtime comparisons against all other heuristics for both datasets and all models.

When assessing the efficacy of planning-based solvers in terms of their heuristics and representations, our findings indicate that no planner configuration was able to solve problems with optimal plan lengths exceeding 13 steps. However, DeepCubeA was capable of solving problems up to 26 steps in length. In terms of the number of problems solved, the FF heuristic is the best performing across all models in both datasets. The M&S abstraction heuristic is the second-best performing heuristic in PDDL representation, but this is not the case for SAS+ representation. In fact, PDBs performed much better in the SAS+ representation than in the PDDL representation and were equally as effective as the FF heuristic. This can also be inferred from the states expansion trend of the abstraction heuristics shown in Figure 3(b). The reason why pattern databases performed better in SAS+ representation than in PDDL representation and were equally effective as the FF heuristic is due to the greater expressiveness of SAS+ models, which allow for more efficient and compact representations of problems. Pattern databases are better able to capture the structure and relationships of problems in SAS+ representation, leading to more accurate and effective heuristics. However, preprocessing time for pattern databases can be longer in SAS+ representation because the language is more explicit and requires the computation of more states to generate the pattern. This is evident from the runtime comparison plot shown in Figure 4(b).

Comparison based on states expanded: Figure 3 illustrates the number of states expanded in the A* search algorithm. Specifically, Figure 3(a) compares the performance of various non-abstract heuristics against the FF heuristic, which was found to be the best-performing heuristic for solving the given set of problems. The diagonal line in the plot represents the performance of the heuristics if they were to perform equally well as the FF heuristic. Heuristics that perform better than the FF heuristic would appear below the diagonal, while heuristics that perform worse would appear above it. On the plot in Figure 3(a), the unsolved points on the *y*-axis represent the set of problems that were unable to be solved by the other heuristics, while the FF heuristic



(a) Memory usage comparison (unit in KB). See text for implications.



(b) Run time comparison (unit in seconds). See text for implications.

Figure 4: Comparison of memory usage and runtime for different heuristics and models for dataset *d2* evaluated using Fast-Downward Planning system.

was able to solve them. Conversely, the unsolved points on the *x*-axis represent the set of problems that the FF heuristic was unable to solve, while the other heuristics were able to solve them. This applies to all other plots provided in the paper. As seen in the plot, all non-abstract heuristics performed worse than the FF heuristic as they lie on the left side of the diagonal. This indicates that the number of states expanded by these heuristics is higher than that of the FF heuristic. This finding provides an explanation as to why the other heuristics performed poorly within the given time and memory constraints when compared to the FF heuristic.

Figure 3(b) displays the trend of state expansion for abstraction heuristics compared to PDB-Man. The reason for selecting PDB-Man was to allow for an interesting comparison between Merge-and-Shrink (M&S) and Pattern Database (PDBs) heuristics across different problem representations. It is observed that the state expansion trend for PDB-Man and PDB-Sys is identical, while M&S performance varies depending on the representation used. Specifically, M&S is found to expand more states than PDBs in the case of the SAS+ model, while in PDDL models, this is not the case. In fact, M&S performs better in PDDL models, where it expands fewer states than PDBs. These results suggest that the choice of abstraction heuristic can have a significant impact on search algorithm performance, depending on the problem representation. Different abstraction heuristics may be better suited for different problem representations, emphasizing the need for careful evaluation to determine the most effective heuristic for each representation.

Runtime and memory usage comparison: Figure 4 presents a comparison of the runtime and memory usage of all considered heuristics. Figure 4(a) presents a comparison of the memory usage for all heuristics, plotted against the GC heuristic, which exhibits an evenly distributed memory usage pattern for problems with different difficulties among the considered heuristics. Similarly, Figure 4(b) displays the runtime comparison for the considered heuristics, plotted against the LM-Cost heuristic for the SAS+ model and GC heuristic for PDDL models as they exhibit a comparatively better distribution of runtime across the problems of varying difficulty in the respective representations.

The following observations have been made from the comparison of runtime and memory usage of different heuristics and problem representations shown in Figure 4.

1. In the case of SAS+ models, the preprocessing time of pattern database (PDB) heuristics is higher and remains constant even for trivial tasks. However, this is not the case for PDDL models, where the preprocessing time does not exhibit a constant trend and takes significantly less time for trivial tasks. This observation highlights the impact of the problem representation on the preprocessing time of PDB

heuristics.

2. The M&S heuristic exhibits a higher runtime in PDDL models compared to SAS+ models, but the memory usage pattern remains similar across the representations. This is intriguing given that the bytes required to represent a single state are higher in PDDL models than in SAS+ models. The state expansion plots provided in Figure 3 further support this observation, showing that M&S heuristic expands more states in SAS+ representation than in PDDL.

3. In SAS+ representation, the preprocessing time of both M&S and PDB-Man heuristics is similar. However, as the problem complexity increases, M&S heuristic exhibits a higher search time compared to PDB-Man. This trend is evident in Figure 4(b), where the constant line starts to ascend earlier for M&S than for PDB-Man. These results suggest that while both heuristics may be suitable for simple problem instances, PDB-Man may offer better performance for more complex instances in SAS+ representation.

4. The runtime performance of the PDB-Sys heuristic is comparable to the blind heuristic, as both exhibit poor performance. This is supported by the runtime comparison plotted against PDB-Sys in the supplementary material.

5. In the case of PDDL, as the number of actions increases, the memory usage pattern for trivial tasks is found to be lesser for the PDB-Man heuristic. This suggests that PDB-Man is able to use its precomputed pattern database more effectively as the size of the planning problem increases. In contrast, for the FF heuristic, the memory usage pattern is found to be the reverse, indicating that as the size of the problem increases we find that there is an increase in the memory usage pattern.

6. FF heuristic is the most efficient heuristic comparatively in terms of both runtime and memory usage across the representations. This explains the fact that the FF heuristic was able to solve the highest number of problems within the given time and memory budget.

7. For both SAS+ and PDDL representations, the runtime pattern of the causal graph and context-enhanced additive heuristics are similar.

Plan Optimality Analysis

Table 2 presents the performance of different planner configurations with various problem representations, including the number of problems solved and the percentage of optimal plans generated. Algorithm 1 shows the methodology employed to evaluate the optimality of a given RC plan. This algorithm checks the actions sequences in the generated plan iteratively to determine whether any of the actions sequences are not optimal. In doing so, it checks if any consecutive actions in the current sequence can be replaced with a single action. The table also includes the performance of DeepCubeA on the considered datasets. It was found that DeepCubeA was able to solve all the problems in both datasets. However, the percentage of optimal plans generated by DeepCubeA are *78.5%* and *18%* for datasets *d1* and *d2*, respectively. The poor performance of DeepCubeA on dataset *d2* can be attributed to the fact that DeepCubeA was trained with only 12 RC actions in its modeling[5], whereas the dataset has problem states generated considering 18 RC actions. Furthermore, the actions sequence generated by DeepCubeA for a problem instance in dataset *d2* can be further simplified when considering the 18 RC actions set. For instance, if the RC is shuffled with action *F2*, which is a 180 degree turn on the front face, DeepCubeA generates a plan of sequence (*F*, *F*), where these two actions can be combined into a single action *F2*. This also explains why DeepCubeA has a higher percentage of optimal plans for dataset *d1*. This is the same reason for the lower percentage of optimal plans in the case of dataset *d2* tested with PDDL model *m1*. We find that the percentage of optimal plans for RC increases as the number of actions modeled in the PDDL increases.

Algorithm 1: Check Action Sequence Optimality

Input : current action face : cf, last action face : lf, second to last action face : slf
Output: True: *not optimal*, **False**: *optimal*
if *cf* = *lf* **then**
  return True
end
if *cf* = *slf* **then**
  if $((cf \in \{F, B\})$ ***and*** $(lf \in \{F, B\}))$ ***or***
    $((cf \in \{R, L\})$ ***and*** $(lf \in \{R, L\}))$ ***or***
    $((cf \in \{U, D\})$ ***and*** $(lf \in \{U, D\}))$ **then**
    return True
  end
end
return False

Discussion and Conclusion

In this study, we conducted an extensive comparison of planning-based and learning-based approaches to solve a complex combinatorial problem: the 3x3x3 RC. We evaluated the effectiveness of existing SAS+ and custom representations for RC, and introduced the first PDDL representation. We examined the capabilities of different heuristics for various representational configurations in solving RC. Our results indicate that a symbolic planner can benefit from using SAS+ representation, which offers a more compact state representation that is approximately $\sim$75% more memory efficient than PDDL. However, a specific planner configuration could only solve 61.50% of RC problems with 100% optimality. In contrast, the DeepCubeA learning-based approach was able to solve 100% of the problems sub-optimally (worst-case 18%). However, the current custom representation used by DeepCubeA may have an impact on the optimal plan generation as it lacks any semantics representing RC. Based on our experimental insights, we note that using SAS+ representation to encode RC problems for DeepCubeA and learning PDBs instead of its current weighted A* search, may improve plan optimality ratio. We also note that while traditional planners generate higher

[5]The performance of DeepCubeA to generate optimal plans may increase if 18 RC actions were considered in the training phase. We followed the documentation and data provided in (Agostinelli 2019) while training the model which has only 12 RC actions modeled.

optimal plans, they are limited to solving only 0.0001% of the $4.3x10^{19}$ states of RC (Rokicki et al. 2014). However, DeepCubeA is able to solve for all possible states of 3x3x3 RC. Our study highlights the potential of both automated and learning-based planners, and suggests a unified approach that can generalize to higher-dimensional RC configurations while preserving the solving capabilities of a learned approach and the optimality of a traditional planner.

References

Agostinelli, F. 2019. Code of DeepCubeA. In *https://github.com/forestagostinelli/DeepCubeA*.

Agostinelli, F.; McAleer, S.; Shmakov, A.; and Baldi, P. 2019a. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8): 356–363.

Agostinelli, F.; McAleer, S.; Shmakov, A.; and Baldi, P. 2019b. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Mach. Intell.*, 1(8): 356–363.

Büchner, C.; Ferber, P.; Seipp, J.; and Helmert, M. 2022. A Comparison of Abstraction Heuristics for Rubik's Cube. In *ICAPS 2022 Workshop on Heuristics and Search for Domain-independent Planning*.

Bäckström, C. 1995. Expressive equivalence of planning formalisms. *Artificial Intelligence*, 76(1): 17–34. Planning and Scheduling.

Fikes, R. E.; and Nilsson, N. J. 1971a. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4): 189–208.

Fikes, R. E.; and Nilsson, N. J. 1971b. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, IJCAI'71, 608–620. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20: 61–124.

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. *Artif. Intell.*, 173(5–6): 503–535.

Howey, R.; and Long, D. 2003. VAL's Progress: The Automatic Validation Tool for PDDL2.1 used in the International Planning Competition. In *ICAPS 2003 workshop on "The Competition: Impact, Organization, Evaluation, Benchmarks", Trento, Italy*.

Korf, R. E. 1997. Finding optimal solutions to Rubik's Cube using pattern databases. In *AAAI/IAAI*, 700–705.

McAleer, S.; Agostinelli, F.; Shmakov, A.; and Baldi, P. 2018. Solving the Rubik's Cube with Approximate Policy Iteration. In *International Conference on Learning Representations*.

McAleer, S.; Agostinelli, F.; Shmakov, A.; and Baldi, P. 2019. Solving the Rubik's Cube Without Human Knowledge. In *ICLR*.

McDermott, D.; Ghallab, M.; Knoblock, C.; Wilkins, D.; Barrett, A.; Christianson, D.; Friedman, M.; Kwok, C.; Golden, K.; Penberthy, S.; Smith, D.; Sun, Y.; and Weld, D. 1998. PDDL - The Planning Domain Definition Language. Technical report, Technical Report.

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 *Twenty-Second International Joint Conference on Artificial Intelligence*.

Pednault, E. P. D. 1994. ADL and the State-Transition Model of Action. *Journal of Logic and Computation*, 4(5): 467–512.

Pommerening, F.; Röger, G.; and Helmert, M. 2013. Getting the most out of pattern databases for classical planning.

Rokicki, T. 2008. Twenty-five moves suffice for Rubik's cube. *arXiv preprint arXiv:0803.3435*.

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.; Keller, T.; and Helmert, M. 2020. Saturated cost partitioning for optimal classical planning. *Journal of Artificial Intelligence Research*, 67: 129–167.

Sievers, S.; Wehrle, M.; and Helmert, M. 2014. Generalized label reduction for merge-and-shrink heuristics. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28.

Sievers, S.; Wehrle, M.; and Helmert, M. 2016. An analysis of merge strategies for merge-and-shrink heuristics. In *Twenty-Sixth International Conference on Automated Planning and Scheduling*.