Count-Based Novelty Exploration in Classical Planning

by

Giacomo Rosa

ORCID:0009-0009-6403-6356

Under the Supervision of

Professor Nir Lipovetzky

A thesis submitted in partial fulfillment for the
degree of Master of Computer Science

in the
Department of Engineering and Information Technology
School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

June 2024

THE UNIVERSITY OF MELBOURNE

Abstract

Department of Engineering and Information Technology
School of Computing and Information Systems

Master of Computer Science

by **Giacomo Rosa**

ORCID:0009-0009-6403-6356

Under the Supervision of

Professor Nir Lipovetzky

Count-based exploration methods are widely employed to improve the exploratory behavior of learning agents over sequential decision problems. Meanwhile, Novelty search has achieved success in Classical Planning through recording of the first, but not successive, occurrences of tuples. In order to structure the exploration, however, the number of tuples considered needs to grow exponentially as the search progresses. We propose a new novelty technique, classical count-based novelty, which aims to explore the state space with a constant number of tuples, by leveraging the frequency of each tuple's appearance in a search tree. We then justify the mechanisms through which lower tuple counts lead the search towards novel tuples. We also introduce algorithmic contributions in the form of a trimmed open list that maintains a constant size by pruning nodes with bad novelty values. These techniques are shown to complement existing novelty heuristics when integrated in a classical solver, achieving competitive results in challenging benchmarks from recent International Planning Competitions. Moreover, adapting our solver as the frontend planner in dual configurations that utilize both memory and time thresholds demonstrates a significant increase in instance coverage, surpassing current state-of-the-art solvers, while also maintaining competitive planning time performance. Finally, we introduce two solvers implementing alternative count-based heuristics and provide promising results for future developments of the ideas presented in this study.

Declaration of Authorship

I, Giacomo Rosa, declare that this thesis titled, 'Count-Based Novelty Exploration in Classical Planning' and the work presented in it are my own. I confirm that:

- The thesis comprises only my original work towards the Master of Computer Science except where indicated in the preface;
- due acknowledgement has been made in the text to all other material used; and
- the thesis is fewer than the maximum word limit in length, exclusive of tables, maps, bibliographies and appendices as approved by the Research Higher Degrees Committee.

Giacomo Rosa
June 2024

Preface

The majority of the work and contributions included in Chapters 3, 4, 5, 6, and 7 have been submitted for publication to the *27th European Conference on Artificial Intelligence* (ECAI) on April 25th, 2024, in collaboration with Nir Lipovetzky, and are under review at the time of writing. Chapter 8.2.1 and results in Section 3.6 constitute unpublished material that has not been submitted for publication.

Modifications to the IPC problem files for domains ‘Storage’, ‘Tidybot’ and ‘GED’, required to make them amenable for execution with the ‘Approximate-BFWS’ solver adopting the ‘Tarski’ grounder, were provided by Anubhav Singh.

Information and commands for running the “First” variant of the ‘Scorpion-Maidu’ solver as described in Section 6.2 were provided by Augusto B. Corrêa.

This research was supported by the Melbourne Research Cloud.

Acknowledgements

I extend my sincere appreciation to my supervisor, Professor Nir Lipovetzky, for sharing his knowledge and enthusiasm for the subject, and for supporting my ideas from the very start. Your guidance helped me build the confidence to tackle this project.

My most heartfelt gratitude goes to my mother, Anna, father, Marco, and brother, Tommaso, for their unwavering support throughout all the decisions in my life that have led me to this point. As I grow, I learn to be humbled by your strength and wisdom.

A special acknowledgement goes to my friends, who keep welcoming me back no matter how bad I may be at keeping in touch. I am blessed to have you in my life.

Contents

Abstract i

Declaration of Authorship ii

Preface iii

Acknowledgements iv

List of Figures viii

List of Tables x

Abbreviations xii

1 Introduction 1

1.1 Classical Planning . . . 1
1.2 Novelty Search in Classical Planning . . . 2
1.3 Summary of Contributions . . . 3
1.4 Thesis Structure . . . 4

2 Background and Related Work 6

2.1 Classical Planning . . . 6
2.1.1 The Classical Planning Model . . . 6
2.1.2 Classical Search Algorithms . . . 7
2.2 Complexity of Planning Domains . . . 9
2.3 Concepts of Width in Classical Planning . . . 10
2.3.1 Hamming Width and Variants . . . 11
2.3.2 Novelty Width . . . 12
2.4 Novelty . . . 14
2.4.1 SIW and BFS(f) . . . 15
2.4.2 Analysis and Explanations of Novelty Performance . . . 16
2.5 Novelty Limitations and Variants . . . 17
2.5.1 Partitioned Novelty . . . 18
2.5.2 Approximate Novelty . . . 21
2.5.3 Novelty as Heuristic Search . . . 23
2.5.4 Novelty Forgetting and Open List Reset . . . 24

2.5.5 Sketches . . . 25
2.6 Count-based Exploration . . . 26
2.6.1 The Exploration-Exploitation Dilemma . . . 27
2.6.2 Epsilon-Greedy Exploration . . . 28
2.6.3 The UCB1 Algorithm in the Multi-Armed Bandit Problem . . . 29
2.6.4 Count-based Exploration in Reinforcement Learning . . . 30
2.6.5 Relation to Novelty in Planning . . . 34

3 Count-Based Novelty in Classical Planning 36
3.1 Motivation . . . 36
3.2 Preliminaries . . . 40
3.3 Classical Count-Based Novelty . . . 40
3.4 Theoretical Analysis . . . 43
3.4.1 Parent-Child Average Distance Bounds . . . 45
3.4.2 Estimating Novel k-Tuples . . . 49
3.5 Comparison with Novelty in Planning . . . 51
3.6 Connections to Count-Based Exploration . . . 53
3.6.1 RL Setting Notation . . . 53
3.6.2 Minimum Empirical Count Distribution . . . 54
3.6.3 Connection to Pseudocounts . . . 54

4 Trimmed Open List 56
4.1 Motivation . . . 56
4.2 Trimmed Open List . . . 59
4.2.1 Single Trimmed Open List . . . 59
4.2.2 Double Trimmed Open List . . . 59
4.3 Trimmed Open List Analysis . . . 61

5 Count-Based Search Algorithms 63
5.1 Count-Based Solvers . . . 63
5.2 Hybrid Solvers . . . 66
5.2.1 Memory Thresholds . . . 66
5.2.2 Hybrid Count-Based Solvers . . . 67

6 Implementation Details 69
6.1 Implementation of Count-Based Solvers . . . 69
6.1.1 Planning Engine . . . 70
6.1.2 State Nodes . . . 71
6.1.3 Count-Based Novelty Heuristic . . . 71
6.1.4 Trimmed Open List . . . 72
6.2 Implementation of Hybrid Configurations . . . 72
6.2.1 LAMA-First . . . 73
6.2.2 "First" Variant of IPC2023 Scorpion-Maidu . . . 73
6.3 Experiment Implementation . . . 74

7 Experimental Evaluation 76
7.1 Experimental Setup . . . 76
7.2 Count-Based Solvers . . . 77

7.2.1 Coverage Results . . . 77
7.2.2 Search Efficiency . . . 79
7.2.3 Search Time . . . 80
7.2.4 Plan Length . . . 83
7.3 Hybrid Solvers . . . 84
7.3.1 Coverage Results . . . 84
7.3.2 Frontend Failure Analysis . . . 86
7.3.3 Backend Contribution and Threshold Analysis . . . 87

8 Conclusion and Future Work **91**
8.1 Conclusion . . . 91
8.2 Future Directions . . . 93
8.2.1 Alternative Count-Based Novelty Heuristics . . . 93
8.2.2 Other Future Directions . . . 99

A Atomic Width of Domains **101**

B Extended Proofs **102**

C Extended Tables of Results **107**

Bibliography **111**

List of Figures

2.1 Classical planning search algorithms. Image from Lipovetzky [1]. 8

2.2 Comparison of states expanded and pruned by *IW(1)* and *IW(2)*. Fluents belong to set $F = \{A, B, C, D\}$, *IW* visits from left to right. For instances N_1 and N_2 with initial state $I = \{A\}$ and goal formulas $g_1 = \{B, C\}$ and $g_2 = \{D\}$ respectively, then N_1 has width 1, and N_2 has width 2. 14

2.3 Example of partitioned novelty. H represents the heuristic value of states, the tables are the fluent history for a width of 1. Fluent B in Figure 2.3c is not novel in the partition of novelty reserved for states with H=2, thus the node is pruned. 19

3.1 Each circle represents a visited nodes. Circles contain variables (left) and the node's count-based novelty over size-1 tuples (right), for a BrFS expansion ordering from left to right. 41

3.2 Parent-child average Hamming distance example scenarios. 48

3.3 $\mathbb{E}[\#$ novel k-tuple$]$ according to Theorem 3.11. In (A) N is a variable, and in (B) $\alpha(s)$ is a variable. Otherwise, parameters are set as $L = 100$, $t = 50000$, $N = 5$, $\alpha(s) = 0.3$. A realistic $\alpha(s)$ value was determined through simulation[1]. 50

5.1 Instance coverage vs. (A) memory usage, and (B) time, for solver BFNoS$_t$ $(f_5(C_1), f_5(W_2))$. The curves show the diminishing returns on instances covered with respect to time and memory. Measurements were conducted over the set of benchmark planning problems described in Section 7.1. . . 66

7.1 Number of nodes expanded to solve instances by BFNoS and BFWS$_t(f_5)$. Blue crosses represent instances not solved by at least one planner. 80

7.2 Time (*sec*) to solve instances by BFNoS and BFWS$_t(f_5)$. Blue crosses represent instances not solved by at least one planner. 81

7.3 Plan cost over instances solved by BFNoS and BFWS$_t(f_5)$. Blue crosses represent instances not solved by at least one planner. 83

7.4 Coverage analysis of proposed solvers. Lines indicate the % of IPC benchmark domains where the % of domain instances solved is $\geq$ a given value. 85

7.5 Cumulative % of all failures attributed to memory failures vs. time of failure (*sec*), for BFNoS solvers with 6 *GB* and 8 *GB* memory and 1800 *sec* time limits. 87

7.6 Instance coverage vs. time (*sec*). Comparison of BFNoS with the three presented hybrid configurations. The solvers are run in an identical configuration to previous sections, with BFNoS using the full 8 *GB* memory allowance when run on its own, and a 6 *GB* memory threshold when run as the frontend of hybrid configurations. The vertical lines signal the 1400 *sec* (green) and 1600 *sec* (red) time thresholds. 88

8.1 Instance coverage vs. time (*sec*). Comparison of BFNoS variants discussed in this section, and the respective BFNoS-Maidu-h^2 configurations. BFNoS-V1 is BFNoS$_t(f_5(C_1), f_5(W_2))$, BFNoS-V2 is BFNoS$_t(f_5(C_1), f_5 (LFC))_{8:1}$, and BFNoS-V3 is BFNoS$_t(f_5(AC_{1,2}), f_5(LFC))_{8:1}$. All BFNoS variants use the full 8 *GB* memory allowance when ran on their own in Figure (A), and 6 *GB* memory threshold and 1400 *sec* time thresholds when ran as the frontend solver in the in BFNoS-Maidu-h^2 configurations in Figure (B). The red vertical line in Figure (B) signals the time threshold. 97

A.1 Atomic width of solving serialized single atom subgoals over a set of benchmark planning domains. I is the number of resulting instances. Other columns denote the number of solved atomic instances using iterative width with width w_e. From Lipovetzky [2]. 101

List of Tables

2.1 Limitations of novelty search. 18

3.1 Size-1 tuple counts after visiting all nodes in Fig. 3.1. 41

7.1 % score and coverage comparison of proposed variants. % score is the average of the % of instances solved in each individual domain, calculated over all benchmark domains. The coverage is provided over the full set of benchmark domains, as well as the subsets of domains corresponding to those used in recent IPC 2023 and IPC 2018 competitions. Values for solvers that use a trimmed open list show the mean and standard deviation across 5 measurements. 78

7.2 % of instances across all IPC satisficing benchmarks where a node with count $\geq N$ was recorded across generated and expanded nodes by a BFCS$_t(f_5)$ planner. This includes unsolved instances. 79

7.3 Comparative coverage analysis across benchmark domains. *% score* is the average of the % of instances solved in each domain. *BFNoS % coverage share* refers to the % of covered instances solved by the BFNoS frontend, when run in a dual configuration. Domains which are fully solved by all planners are omitted. Values for BFNoS variants and Approximate-BFWS represent the mean and include the standard deviation across 5 measurements. The complete table of results is included in Appendix C. . 84

7.4 Comparative coverage analysis of tested and benchmark solvers across IPC 2023 and IPC 2018 instance sets. 86

7.5 Coverage and % score comparison of hybrid planner variants adopting only a memory threshold, and the usual configuration with both memory and time thresholds. All memory thresholds are set at 6 *GB* for both configurations of each planner. The BFNoS-Maidu planner is a variant of BFNoS-Maidu-h^2 that does not use the h^2 preprocessor with its Scorpion-Maidu backend. Values represent the mean and the standard deviation across 5 measurements. 89

8.1 Coverage comparison of BFNoS and BFNoS-Maidu-h^2 variants. % score is the average of the % of instances solved in each individual domain, and is provided for results calculated over the full set of benchmark domains. The best results for each solver class are highlighted in bold. 97

C.1 Comparative performance analysis across the full set of benchmark domains. *% score* is the average of the % of instances solved in each domain. *Front-end % coverage share* refers to the % of covered instances solved by the BFNoS front-end. Values for BFNoS variants and Approximate-BFWS represent the mean and include the standard deviation across 5 measurements. 108
C.2 Comparative performance analysis across the full set of benchmark domains of 'memory-only' threshold dual-configuration BFNoS variants. *% score* is the average of the % of instances solved in each domain. *Front-end % coverage share* refers to the % of covered instances solved by the BFNoS front-end. Coverage values represent the mean and include the standard deviation across 5 measurements. 109
C.3 Comparative performance analysis across the full set of benchmark domains of BFNoS variants. BFNoS-V2 is BFNoS$_t(f_5(C_1), f_5\ (LFC))_{8:1}$, and BFNoS-V3 is BFNoS$_t(f_5(AC_{1,2}), f_5(LFC))_{8:1}$. *% score* is the average of the % of instances solved in each domain. *Front-end % coverage share* refers to the % of covered instances solved by the BFNoS front-end. Coverage values represent the mean and include the standard deviation across 3 measurements. 110

Abbreviations

BFCS	**B**est **F**irst **C**ount **S**earch
BFNoS	**B**est **F**irst **No**velty **S**earch
BFS	**B**est **F**irst **S**earch
BFWS	**B**est **F**irst **W**idth **S**earch
IPC	**I**nternational **P**lanning **C**ompetition
IW	**I**terative **W**idth
LAPKT	**L**ightweight **A**utomated **P**lanning Tool**K**i**T**
MDP	**M**arkov **D**ecision **P**rocess
PAC	**P**robably **A**pproximately **C**orrect
RL	**R**einforcement **L**earning

Chapter 1

Introduction

1.1 Classical Planning

AI Planning is the field of Artificial Intelligence which explores the design of algorithms to solve problems that are represented by large transition systems, where solutions consist of *plans*: sequences of actions that map an initial configuration to a desirable outcome. The *Classical* planning problem further restricts the problem to a discrete state-space, where actions have deterministic effects and the initial configuration specifies a single initial state. States are defined by a discrete set of variables, or *atoms*, which are either *true* or *false* in a given state. An action is applicable in a state if the set of *precondition* fluents is *true*, and modifies the state according to its *effects*.

A key characteristic which differentiates it from other fields of AI is the aim of employing a single algorithm to solve all problem domains. Finding a valid solution, a *plan*, in most domains is worst-case NP-hard [3], and yet modern planners can tractably handle many such instances [4]. The prevailing technique has been that of guiding the search of solutions through the use of *heuristics*, estimates of distance to the goal derived automatically from the problem's description.

1.2 Novelty Search in Classical Planning

An alternative successful approach in recent years has seen the adoption of *width-based search methods* [5], which prioritize an efficient exploration of the state-space over greedily following legacy heuristics. The efficient exploration is achieved through *Novelty* metrics that compare a state's information content with previously visited states, prioritizing states with more novel information. The related concept of *width* characterizes the difficulty of solving problems using these methods. For a given problem width, width-based search methods achieve polynomial plan generation complexity without the need for additional heuristics. Research on these methods has significantly impacted planning by introducing algorithms that use novelty heuristics for efficient state-space exploration, leading to the development of multiple state-of-the-art algorithms [6]. The performance of these algorithms is often explained in terms of balancing exploration and exploitation, where novelty drives exploration and traditional heuristics direct exploitation.

This does not come without limitations, as Lipovetzky and Geffner [5, 7] show that the complexity of computing novelty metrics needed to solve planning problems is exponential in their cardinality – roughly speaking, the dimensionality of such heuristics. In practice, this causes novelty metrics of cardinality greater than 2 to be computationally unfeasible, limiting the technique's effectiveness in domains that would benefit from a higher cardinality. The cardinality is connected to a hardness measure for Classical Planning known as *classical planning atomic width*.

Multiple contributions have sought to address this limitation. Lipovetzky and Geffner [8] introduce *partition functions*, which subdivide planning problems into smaller subproblems through the use of *partitioning heuristics* to control the direction of search and increase the number of novel nodes. Katz et al. [9] provide a definition of novelty of a state with respect to its heuristic estimate, providing multiple novelty measures which quantify the novelty degree of a state in terms of the number of novel and non-novel state facts. More recently, Singh et al. [10] introduce *approximate novelty*, which uses an approximate measurement of state novelty that is more time and memory efficient, proving capable of estimating novelty values of cardinality greater than 2 in practical scenarios. Analysing Novelty and problem width through connections to other concepts, such as *dominance pruning*, also constitutes an active area of research [11, 12].

1.3 Summary of Contributions

Existing novelty paradigms limit themselves to the original idea of measuring state information content through the first occurrence of tuples in the search history. Instead, we propose a count-based measure of state novelty, *classical count-based novelty*, which seeks to induce efficient exploration of the state space by making use of the additional information contained in the count of occurrences of tuples in the search history. This addresses shortcomings of the current Novelty framework (see [6]), which we refer to as *width-novelty* to distinguish from our contributions in this thesis.

Our first aim was to study what additional information is contained in tuple counts and develop an extension of novelty which incorporates this knowledge. This is addressed in our first research question:

What theoretical and empirical insights may be found by incorporating tuple occurrence knowledge into existing novelty theory in the domain of classical planning?

Our second aim was to study implementations of novelty search algorithms using the metrics we derived in the theoretical component of our project, alongside complementary techniques which improve their effectiveness, and analyze their performance on satisficing classical planning tasks. Our second research question is focused on this aspect:

How can the performance of novelty search algorithms be improved by leveraging count-based novelty metrics?

Our proposed count-based metric is not limited by width-novelty's binary classification of novel information, providing a more fine-tuned separation of the degree of novelty of a state and maintaining its informedness without the risk of exhausting novel nodes. A key motivation behind our study is thus to obtain a more general novelty framework that can maintain its efficacy across diverse sets of problems in Classical Planning, such as domains that require higher atomic widths.

In this regard, we note that count-based exploration techniques are well studied in relation to the exploration-exploitation problem in Multi-Arm Bandits and Reinforcement Learning (RL) settings. Such algorithms record state visitation to obtain an *exploration bonus* used to guide the agent towards a more efficient exploration of the state-space, where algorithms such as MBIE-EB [13] achieve theoretical bounds on sample complexity in tabular settings. The focus of our research diverges from these methods, as we aim to discover a heuristic to control the order of state exploration in a Classical Planning context. Instead of state counts, we base our approach on the frequency of tuple events, inspired by work on width-novelty in the field of Classical Planning [5, 7, 14]. Still, our contributions provide a useful basis to connect count-based exploration across the two fields.

We also introduce algorithmic contributions in the form of a simple memory-efficient *trimmed open list*. Polynomial width-based planning algorithms prune nodes whose novelty cardinality is worse than a given bound to achieve a more efficient search [14]. Inspired on this idea, our contribution allows us to prune nodes with bad novelty values with a gradual and self-balancing cutoff without maintaining an explicit threshold value.

Finally, we demonstrate the effectiveness of our proposed planning algorithms as fast but memory-intensive *frontend* solvers through an effective use of a *memory threshold*, which allows us to relate the progress of search directly to the amount of information we store from the history of the search. Many successful solvers such as FF, Probe or Dual-BFWS rely on a dual strategy [8, 15, 16], with the frontend of such solvers playing a key role in their performance. The performance of our proposed frontend planner could improve such solvers even further.

1.4 Thesis Structure

Our contributions include the development of a new novelty technique, accompanied by theoretical analysis that connects it to existing width-novelty measures. Additionally, we introduce the trimmed open list method and a planner called $BFNoS$, which integrates these techniques. Finally, we propose a procedure to adapt $BFNoS$ as an effective frontend planner in a dual strategy. We structure our thesis as follows. In Chapter 3 we introduce classical count-based novelty metrics for Classical Planning and provide related theoretical findings. We then propose a novel open list implementation to exploit classical count-based novelty more efficiently in Chapter 4. Chapters 5 and 6 introduce all our proposed count-based classical search algorithms, providing explanations and implementation details for design choices. Here we also present the concept of memory thresholds, central to our hybrid planner contributions. Chapter 7 is divided into two components: we first compare the performance of solvers incorporating our proposed count-based novelty techniques, and then show the impact of our frontend solver when used in conjunction with different time-and-memory thresholds and backend solvers, achieving state-of-the-art performance. Lastly, in Section 8.2.1 we introduce preliminary results for two alternative count-based heuristics, one of which diverges more markedly from existing work on novelty than our other contributions. Results provided in this chapter offer further insights into the potential of count-based novelty, and suggest promising directions for future developments of the ideas presented in this thesis.

Chapter 2

Background and Related Work

In this chapter we characterize the foundational knowledge on which our study is based. We begin by providing an introduction to the classical planning model and to classical planners. We then provide a review of work characterizing domain complexity in Classical Planning, with special focus on recent results on *Width* and *Novelty*, their variants and the successful solvers which rely on these techniques. Finally, we discuss the broader subject of *count-based exploration* and its relevance to addressing the exploration/exploitation dilemma in related Multi-Arm Bandit and Reinforcement Learning problems.

2.1 Classical Planning

2.1.1 The Classical Planning Model

The classical planning model is defined as $S = \langle S, s_0, S_G, A, f \rangle$, where S is a discrete finite state space, s_0 is the the initial state, S_G is the set of goal states, and $A(s)$ denotes the set of actions $a \in A$ that deterministically map one state s into another $s' = f(a, s)$, where $A(s)$ is the set of actions applicable in s. We adopt a notation whereby, in a classical planning problem, a state is visited (generated) sequentially at each time-step t. Let $s_t \in S$ denote the t^{th} visited (generated) state in a search problem. We use $s_{0:t}$ to denote the sequence of $t+1$ states generated at time-steps $0, 1, ..., t$. A solution to a classical planning model is given by a *plan*, a sequence of actions $a_0, ..., a_{x_m}$ that induces a state sequence $s_{0:x_{m+1}}$ such that $a_{x_i} \in A(s_{x_i})$, $s_{x_{i+1}} = f(a_{x_i}, s_{x_i})$, and $s_{x_{m+1}} \in S_G$. Each action in a plan has a positive action cost, and summing the action costs for all actions in a plan yields the plan's cost.

A classical planning problem P defines a classical state model $\mathcal{S}(P)$ through a set of variables in a planning language. We use the STRIPS planning language [17] to define a classical planning problem $P = \langle F, O, I, G \rangle$, where F denotes the set of boolean variables, O denotes the set of operators, $I \subseteq F$ is the set of variables that fully describe the initial state, and $G \subseteq F$ is the set of variables that are *true* in the goal state. The *preconditions* of an operator $o \in O$ specify the subset of boolean variables which have to be *true* in a state s in order for operator o to be applicable in that state. The *effects* of operator o are then the state variables that o changes in the process of mapping state s to its successor s' when taken. We say that variables are *added*, when the boolean variable value changes $false \rightarrow true$, or *deleted* when the value changes $true \rightarrow false$. A plan for a classical planning problem is said to be *optimal* when there is no other plan that can solve the same problem with lower cost. In absence of explicit cost information, it is assumed all action costs are 1, and that the plan cost is thus equivalent to the plan length, that is, the total number of actions in the plan. An optimal plan thus consists of the shortest possible solution to a given problem P. In this research, we look at *satisficing* planners, that is, planners which are not constrained to searching for optimal plans, but rather aim for computing good-quality plans successfully and fast, if a plan exists.

2.1.2 Classical Search Algorithms

Within the context of Classical Planning, *search algorithms* (also referred to as *planners* or *solvers*), are algorithms which, provided a classical planning model description $\mathcal{S}(P)$, automatically compute a solution to $\mathcal{S}(P)$ in the form of a valid plan. This is achieved by exploiting the existing correspondence between $\mathcal{S}(P)$ and directed graphs, where the nodes of the graph represent the states s in the model, and, for the considered case of unitary costs, directed edges (s, s') capture the transitions induced by operators applicable in state s. Beginning from a root node corresponding to the starting state, a search algorithm can then recursively generate, or visit, successor nodes by following the directed edges, seeking to find a node corresponding to a valid goal. The plan can then be obtained by extracting in order the operators corresponding to each edge in the path from the root node to the goal node.

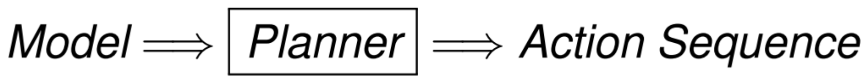


FIGURE 2.1: Classical planning search algorithms. Image from Lipovetzky [1].

Of particular importance is the distinction between *blind search* and *heuristic* (or *informed*) search:

- A **blind search algorithm** searches the state space by only using the basic information obtained during a search, such as the overall cost of reaching a state s, but without using nor requiring any additional information to determine whether exploring certain states or actions is more useful to finding a solution. Well-known examples include *best-first search*, *depth-first search*, as well as *Dijkstra's algorithm* [18].
- A **heuristic search algorithm** exploits *heuristics*, estimates of the distance, or remaining cost, between a state s and the goal, in order to obtain information on whether certain states or actions may be more useful towards finding a solution. Such heuristics seek to estimate the *optimal heuristic* h^*, which represents the true distance of the node to the goal. Using heuristics allows the search algorithm to alter the order of visitation of nodes compared to a blind search, in order to prioritize nodes which are expected to lead the search closer to the goal.

A search algorithm progresses by selecting a node from an *open list*, a set of nodes visited in the search which are candidates for *expansion*. A node is said to be *expanded* when it is selected and removed from the open list, such that all operators whose preconditions are satisfied may be applied to *generate* a set of successor nodes, which are inserted into the open list. After being expanded, nodes are then inserted into a *closed list*, allowing a search algorithm to avoid expanding duplicate nodes, as well as being useful in practice for the implementation of *lazy* node evaluation, which allows a node's state to be evaluated as late as possible to save time and memory.

Of particular interest to our research are the families of *best-first search* (BFS) and *greedy best-first search* (GBFS) planners. In heuristic search algorithms, heuristics for successor nodes tend to be evaluated at node generation time, before being inserted into the open list, which is often implemented as a priority queue that prioritizes nodes with a lower heuristic value. A BFS algorithm is a planner that weights nodes through a combination of cost, corresponding to the length of the path from the starting node when adopting unit costs, and heuristic value of the node. When these quantities are added, the algorithm estimates the overall length of the plan. GBFS algorithms, on the other hand, constitute a subset of BFS algorithms that only weigh the heuristic value of a given node, thus prioritizing nodes purely based on their estimated vicinity to the goal. GBFS algorithms provide the basis for multiple modern planners, including well-known state-of-the-art solvers [8, 19–22], which further extend the GBFS paradigm by ordering nodes in an open list through a hierarchy of heuristics, where heuristic values are used in order to break ties among higher-ranked heuristics.

2.2 Complexity of Planning Domains

Multiple studies have sought to formalize properties of *planning domains* - families of problems with shared structures, actions, and rules, but varying initial states and goals - which contribute to the underlying complexity of domain-independent planning, with a focus on learning what separates tractable and intractable problems [23].

Bylander [3] found that the task of determining whether a planning instance admits any solution is PSPACE-complete when placing restrictions on type of formulas, pre- and post-conditions (effects) in *propositional STRIPS*, a foundational language for defining planning domains [24]. Planning problems where variables extend over an infinite domain were found to be undecidable [25, 26]. NP-Complete and even polynomial time results have been found for more constrained problem classes; however, these only hold under specific conditions and do not apply to most practical problems, placing them at odds with the overall goal of tractable planning across domains [3, 23, 27].

An alternative line of research has explored the impact that limiting heuristic complexity has on heuristic quality across domains, helping to distinguish “easy” from “difficult” domains [28–30]. Studies show that most domains are amenable to search with the optimal delete relaxed $h+$ heuristic [28], and many are polynomial-time solvable using $h+$ with the *FF* search algorithm [15]. The $h+$ heuristic is also found to approximate

h^*, the true goal distance, by a constant factor in multiple benchmarks [29]. Finding the *optimal delete relaxed plan*, however, remains an NP-complete problem, thus precluding tractability over arbitrary domains.

Other heuristic classes including pattern databases [31], the h^k heuristic [32] and potential heuristics [33] share the property of being capable of converging to the optimal heuristic h^* at the expense of computational complexity. The h^k and pattern database heuristics however perform poorly when the size of their representation is bounded, in the worst case failing completely to converge to h^* in all studied domains. Additive pattern database heuristics, a class of heuristics obtained by assigning to each state a sum of multiple pattern values, fare better, achieving worst case constant factor approximation to the h^* heuristic in all but one studied domain [29], *Blocksworld*.

Efficient *potential heuristics* based on 2-dimensional linear combinations of state features have been found to induce state-spaces with no local minima in many benchmark domains. This simplifies search by preventing an algorithm from getting stuck in states whose value is lower than its surroundings, thus always pointing a way forward to the global minimum heuristic value. The dimensionality of such a potential heuristic required to guarantee a solution with no local minima for a given planning domain induces the concept of *correlation complexity*, which quantifies the degree of interrelatedness of state facts and acts as a measure to separate simple from complex planning domains [30]. Furthermore, several benchmark domains have been evaluated and found to have a low correlation complexity, guaranteeing efficient evaluation. However, the study relies on manually-created potential heuristics for the analysis of planning benchmarks. As such, it does not provide techniques to find valid features and weights for potential heuristics automatically, nor does it address the existence of domain-independent feature classes, limiting the techniques applicability in domain-independent scenarios.

2.3 Concepts of Width in Classical Planning

An alternative measure of the complexity of a planning domain, based on the interrelation of state fluents, is found in the (distinct) concepts of *width* introduced by Chen and Giménez [34], and Lipovetzky and Geffner [5]. Both interpretations of problem width seek to capture domain-independent structural conditions that define domain complexity and ensure practical tractability in various benchmark domains. Unlike previous work, these width measures do not rely on heuristic-related information to classify domain tractability. Their practical use has led to new search algorithms with strong theoretical backing.

2.3.1 Hamming Width and Variants

Chen and Giménez [34] developed a basic notion of width alongside three related extensions which bound the complexity of the domain. The width of an instance bounds the number of variables that may be changed in a plan from every state s to another state comprised of all satisfied goal variables in s plus an additional goal variable. It is said to have *width k* if either there is no solution, or every state reachable from the initial state can bring each one of its unsatisfied variables to a state where it is satisfied by changing up to k variables. *Persistent width*, *Hamming width*, and *persistent Hamming width* relax the initial notion of width by either requiring respectively that there exist one unsatisfied variable, as opposed to every unsatisfied variable, which may be satisfied; using Hamming width as a measure of distance; or a combination of the definitions of persistent width and Hamming width (persistent Hamming width).

For all definitions of width, the plan generation problem for instances displaying bounded width k is solvable in polynomial time using the *width-k algorithm.* Multiple planning benchmarks demonstrate bounded width, thus achieving polynomial-time tractability over multiple domains. A major limitation arises for domain instances containing *dead-ends* - states from which no sequence of actions can reach a goal - because if such a state is reachable, then it implies the existence of a reachable non-goal state where all unsatisfied goal variables cannot be brought to their goals. Thus, k remains undefined over such instances, precluding tractability of the underlying domains. Furthermore, the width-k algorithm scales exponentially on the value of k, and in practice anything but low polynomials are impractical to compute for non-trivial domain instances.

2.3.2 Novelty Width

2.3.2.1 Definition

Lipovetzky and Geffner [5] proposed a definition of problem width which, at a high level, quantifies instance and domain complexity with respect to the size of variable conjunctions required to achieve goals. This definition bounds the complexity of the shortest, or *optimal*, plan search in many benchmark planning domains to a practical low-polynomial complexity when considering goals comprised of size-1 variable conjunctions (*atomic* goals).

This definition of width and the related notion of *Novelty* are central to our study as they provide the foundational theory which we seek to extend through the incorporation of information pertaining to atom occurrences. From here on, when not explicitly mentioned, we will be referring to Lipovetzky and Geffner's [5] definition of width.

Width relies upon the concept of *tuple graphs*, where a *tuple* of size n refers to a conjunction of n state variables, or *atoms*, and is *true* in a state s if said conjunction is present among atoms that are *true* in s. Tuple graphs then define a reachability relation over tuples t of bounded size.

Definition 2.1. For $P = \langle F, O, I, G \rangle$, $\mathcal{G}^i$ is the graph with vertices from Q^i, where Q^i stands for the collection of tuples from P of size no greater than i, defined inductively as follows:

1. q is a root vertex in $\mathcal{G}^i$ iff q is true in I,
2. $q \rightarrow q'$ is a directed edge in $\mathcal{G}^i$ and for every optimal plan π for $P(q)$ there is an action $a \in O$ such that π followed by a is an optimal plan for $P(q')$ [5].

In other words, given an integer i, the tuple graph $\mathcal{G}^i$ refers to a graph where nodes are tuples of size no larger than i. Tuples that are *true* in the initial state represent the root nodes of the graph, and tuple q' is a successor of tuple q if all optimal plans for q can be extended into optimal plans for q' by adding one action.

Furthermore, we have that, if all the optimal plans to a *goal formula* g_1 are also optimal plans to a goal formula g_2, $\{g_1, g_2\} \subseteq F$, then g_1 is said to *imply* g_2. This allows us to then define the width of a planning problem:

Definition 2.2. For a formula ϕ over the fluents in P that is not true in the initial situation I, the width of ϕ relative to P is the min w such that $\mathcal{G}^w$ contains a tuple that implies ϕ. If ϕ is true in I, its width is 0 [5].

Definition 2.3. The width of a planning problem P, $w(P)$, is the width of its goal G relative to P [5].

2.3.2.2 The Complexity of Problem Width

The value of characterizing the "hardness" of a planning problem by its problem width then lies in the capability to limit the complexity of resolving the problem based on its width:

Theorem 2.4. *If $w(P) = i$, P can be solved optimally in time that is exponential in i* [5].

Of course, the meaningfulness of Theorem 2.4 then lies in whether actual planning problems exhibit low and tractable problem widths i. In this regard, Lipovetzky and Geffner [5] find that most domain benchmarks do exhibit a small problem width insofar as the goal G is limited to a single atom (Appendix A), further proving the result on domains *Blocksworld*, *Logistics*, and *n-puzzle*. We say that these domains have low *atomic width*.

In such domains, the complexity of a problem is thus derived mainly from its *goal structure*, that is, the challenge posed by conjunctively achieving single atoms subgoals. In other words, if it is easy to achieve single atoms, but difficult to solve full problems, the difficulty must then come from being able to simultaneously achieve all the single atoms in the goal. Such goals are referred to as *conjunctive goals*. An intuitive example can be found in the popular puzzle 'Rubik's Cube': we may manage to solve a full face of the cube, but anyone who has tried to solve a cube knows that the challenge of the puzzle is derived from the fact that further progress cannot then be achieved without impacting the progress achieved thus far, as the currently completed face must be partially undone in the process of solving the rest of the cube.

2.4 Novelty

In practical settings, the notion of problem width lays the groundwork for enhancing search performance via the principle of Novelty:

Definition 2.5. The novelty $w(s)$ of a newly generates state s is i iff there is a tuple of i atoms and no smaller tuple, that is true in s and false in all states s' generated before s. Such tuple is said to be *novel*.

We can then perform a width-w search by performing a breadth-first search which evaluates the novelty value of each newly generated state, and *prunes* the state, not inserting it into the open list for later expansion, if the novelty value of that state is greater than w. This idea is the basis for the *iterative width* algorithm (IW(w))[5], which, starting from $w = 1$, iteratively performs searches of greater width until it finds a solution or exceeds the maximum width (Alg. 1).

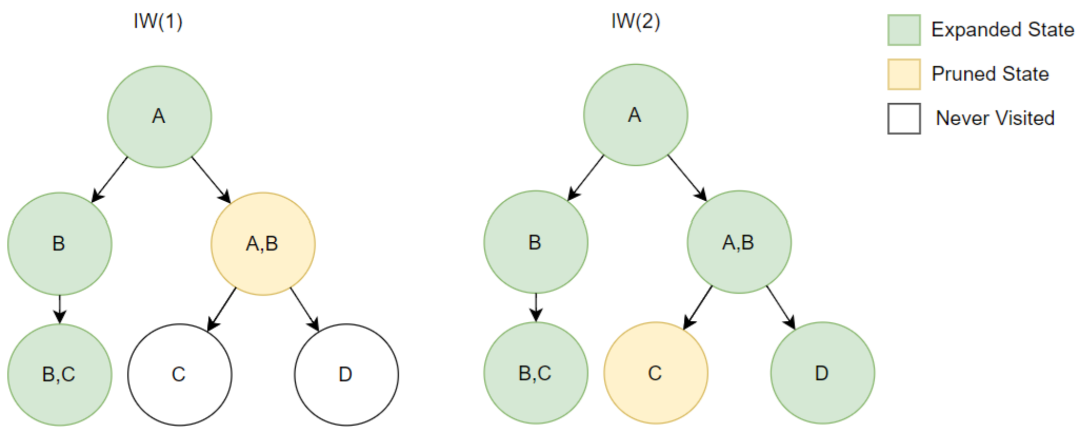


FIGURE 2.2: Comparison of states expanded and pruned by *IW(1)* and *IW(2)*. Fluents belong to set $F = \{A, B, C, D\}$, *IW* visits from left to right. For instances N_1 and N_2 with initial state $I = \{A\}$ and goal formulas $g_1 = \{B, C\}$ and $g_2 = \{D\}$ respectively, then N_1 has width 1, and N_2 has width 2.

The novelty value of a node can be computed in time polynomial in the number of variables F in problem P, and exponential in the maximum width w measured, that is, the maximum arity of tuples considered. Similarly, for a given maximum width w, the memory requirements to store the record of previously seen tuples is polynomial in w. From Theorem 2.4, we can thus obtain polynomial search algorithms where both the novelty metric and the overall width-w search can be evaluated in time polynomial in problem size, and that therefore find a solution or fail also in polynomial time.

Algorithm 1 Iterative Width w algorithm

```
Require: $w \geq 1$
  procedure ITERATIVE WIDTH W(search problem $P$, source node $s$, max width $w$)
      $i \leftarrow 1$
      while $i \leq w$ do
          if WIDTH-W SEARCH($P$, $s$, $i$) finds a valid solution $\pi$ to $P$ then
              End search and return solution $\pi$
          else
              Increment $i$

  function WIDTH-W SEARCH(search problem $P$, source node $s$, max width $w$)
      Initialize queue $Q$
      Initialize tuple record $R$ for all tuples of size $\leq w$
      Evaluate novelty of $r$ and update $R$ with novel tuples
      Mark $r$ as visited
      Enqueue $r$ in $Q$
      while Q is not empty do
          $n \leftarrow$ dequeue Q
          if $n$ is a goal then
              End search and extract plan
          $S \leftarrow$ set of successors of $n$ according to problem $P$
          for successor node $s \in S$ do
              if $s$ is not visited then
                  $v \leftarrow$ evaluate novelty of $s$ and update $R$ with novel tuples
                  if $v > w$ then
                      Prune $s$
                  else
                      Mark $s$ as visited
                      Enqueue $s$ in $Q$
```

2.4.1 SIW and BFS(f)

Two seminal planners which first demonstrated the benefits of adopting a novelty metric in practical scenarios are *serialized iterative width* (SIW) and *BFS(f)*.

SIW is an enhancement of IW(w) which improves its ability to solve more complex planning problems. The solver *serializes* the goal, subdividing the goal formula into distinct atomic goals, and then performs a sequence of calls to the IW algorithm. Each time, the width search restarts from the most recent *partial goal*, a state where only a subset of the goal state atoms are satisfied, seeking a new partial goal containing one additional atom, until a goal state containing all atoms in the goal formula is found, or the search runs out of nodes of novelty $\leq$ its maximum width. Given the linear nature of solving partial goals, the algorithm is also polynomial. The importance of SIW derives from it being the first blind planning algorithm capable of achieving good performance across challenging benchmark domains, thus demonstrating the feasibility of subdividing planning problems by serializing the goals and adopting a novelty planner to iteratively solve each subproblem [5].

BFS(f) is the first novelty planner to demonstrate state-of-the-art performance. The planner is a greedy best first search solver guided by the $novelha(n)$ evaluation function. This is a measure that combines novelty and *helpful actions* [15], further breaking ties in the open list using a subgoal-counting heuristic alongside the h_{add} heuristic [35]. In addition to demonstrating an equivalent planning performance to IPC-winning planner LAMA'11 [19], an ablation study shows novelty as being the component that contributes the most to its overall planning performance, further highlighting the meaningfulness of the metric.

2.4.2 Analysis and Explanations of Novelty Performance

Multiple studies have sought to understand and explain why novelty metrics prove so effective at improving the performance of search algorithms while still being *goal-agnostic*, meaning that they do not provide additional information or estimate on the location of a valid goal state in the search state space.

An intuitive justification of the performance derived from novelty search is that it induces an efficient search with respect to the objective of minimizing (and bounding) the number of states expanded to find (optimal) plans from any start state to all reachable size-1 tuples in a domain instance. This intuition is directly tied to Theorem 2.4 from Lipovetzky and Geffner [5]: for an instance of bounded single-goal width w, theoretical results ensure that $IW(w)$ is going to find a path to the goal, and the number of expanded nodes is guaranteed to be $O(n^w)$, where n is the number of variables in the instance. In this respect, nodes that do not have any novel tuple of size $\leq w$, and are therefore pruned, can be judged to contain redundant information that is less useful to solving the search task, pointing at a connection to an information theoretical view where we may want to maximize the amount of useful information in the nodes we visit. Such usefulness is not strictly defined, but clear connections may be identified with potentially useful events, such as the search for novel preconditions and actions, or, as Groß et al. [11] find, its relation to dominance pruning.

Groß et al. [11] interpret novelty pruning as an unsafe approximation of dominance. Dominance pruning [36, 37] is a technique whereby states which are provably suboptimal to, or dominated by, other states are pruned. More specifically, a state is said to dominate another state if it leads to an equal or better outcome in all possible scenarios, and is strictly better in at least one scenario. Such an event then allows the safe pruning of dominated states, meaning that it guarantees that pruned states are not part of an optimal solution path due to the existence of strictly preferred alternatives. Novelty can then be related as an unsafe approximation of dominance, where states containing novel tuples enable novel paths to the goal and are therefore less likely to be dominated. This view thus reframes novelty in terms of the outgoing action paths from a given state, suggesting that exploring novel states is preferable as they lead to new paths to the goal that were not possible from previously visited states.

Still, existing contributions do not explain why so many problems demonstrate a low atomic width, nor do they identify general domain characteristics that may determine this low atomic width, which has proved pivotal to the success of the technique.

2.5 Novelty Limitations and Variants

As introduced in Section 2.4, a major result over domains with low width over single atom goals is that domain complexity can then only arise through the *goal structure* of the problem, thus laying blame on conjunctive goals for causing hard-to-solve benchmark domains. Such domains remain one of the major challenges for novelty search, and planning in general [6]. The other main limitation for Classical Planning domains remains the exponential increase in complexity of $IW(i)$ as i increases. In practice, most state-of-the-art width-based algorithms only ever perform a width w search of up to 2 [6], with recent results partially increasing this value [10].

Multiple studies in recent years have sought to extend Lipovetzky and Geffner [5]'s concepts of width and novelty to address these limitations [8, 10, 14, 20, 38], further outlined in Table 2.1, as well as adapting the ideas of novelty search to improve heuristic search [9, 21, 22] and preferred operators [39, 40] techniques. This has led to promising developments and practical implementations which outperform previous state-of-the-art novelty algorithms in sets of commonly tested planning domains. Such results justify the direction of our proposed research, addressing questions on the potential relevance of our study towards improving novelty-based planning techniques as well as contributing to the field's understanding of the topic.

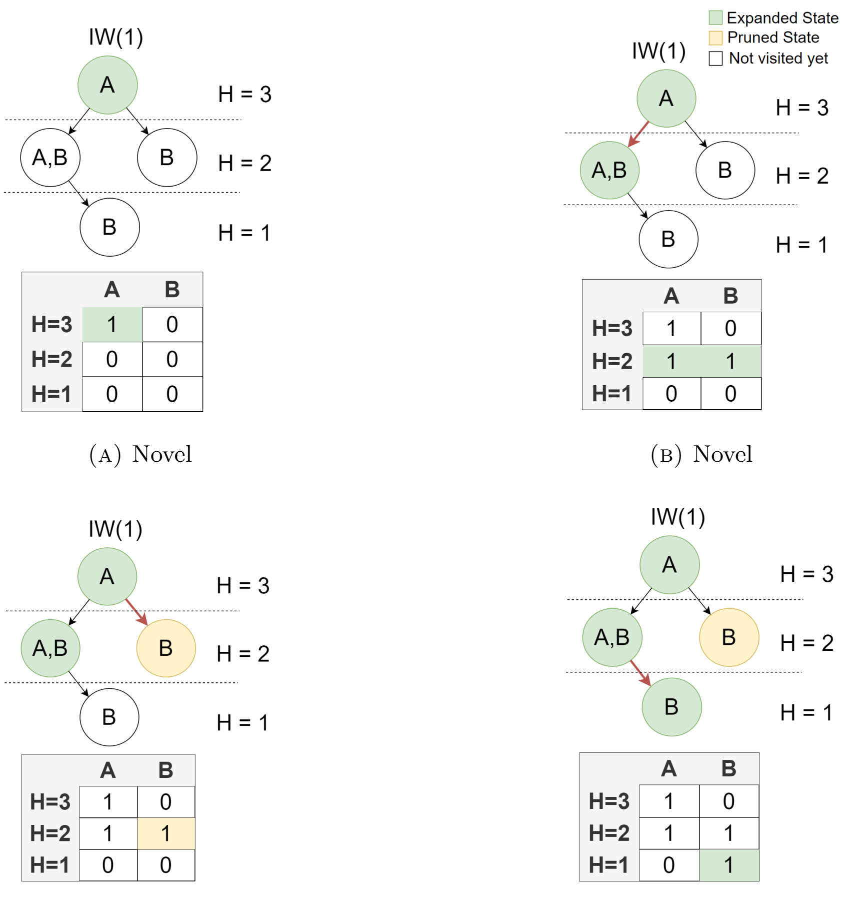
Limitation	Description
High atomic width	Atomic goals with high width quickly become computationally intractable. This manifests both in terms of the exponential increase in time and memory requirements of generating and keeping track of all possible tuple conjunctions when tuple sizes increase, as well as the exponentially larger number of states expanded during search. In practice, non-approximate forms of novelty are bounded to a width of up to 2 when subject to common IPC time and memory constraints [6].
Hard-to-serialize conjunctive goals	Certain domains display *conjunctive goals* - subgoals defined through formulas whose size is greater than one - that depend on other partial goals to be satisfied in advance. This makes the instance hard to serialize as valid plans must thus solve sub-problems in a specific order. Greedy approaches such as SIW then may fail to find a correct sub-goal ordering, precluding the achievement of a valid plan [7].
Discrete representation	Traditional novelty metrics [5] limit themselves to assigning a discrete novelty value based on the smallest novel tuple in a state, grouping states among few discrete categories. This poses a challenge for methods which require prioritization of preferred states, such as best-first search, as well as large state-spaces, requiring additional techniques to further partition the groups [7, 8, 14], break ties [8, 14] or quantify novelty value [9, 39].

Table 2.1: Limitations of novelty search.

2.5.1 Partitioned Novelty

Lipovetzky and Geffner [8] extend the concept of novelty (Definition 2.5) with the introduction of *partition functions*, whereby selected functions are used subdivide the collection of observed tuples into partitions, and the novelty of a state is then only calculated relative to tuples previously observed in states belonging to the same partition (Fig. 2.3).

Definition 2.6. The novelty $w(s)$ of a newly generated state s given the functions $h_1, ..., h_m$ is i iff there is a tuple (conjunction) of i atoms and no smaller tuple, that is *true* in s and *false* in all states s' generated before s with the same function values $h_1(s') = h_1(s), ..., h_m(s') = h_m(s)$ [8].



	A	B
H=3	1	0
H=2	0	0
H=1	0	0

(A) Novel

	A	B
H=3	1	0
H=2	1	1
H=1	0	0

(B) Novel

	A	B
H=3	1	0
H=2	1	1
H=1	0	0

(C) Not novel

	A	B
H=3	1	0
H=2	1	1
H=1	0	1

(D) Novel

FIGURE 2.3: Example of partitioned novelty. H represents the heuristic value of states, the tables are the fluent history for a width of 1. Fluent B in Figure 2.3c is not novel in the partition of novelty reserved for states with H=2, thus the node is pruned.

Augmenting the basic definition of novelty through the use of partitions achieves two main beneficial effects on the search. Firstly, it extends the availability of novel nodes before the "novel" search space is exhausted. This effect arises because the novelty of a node, when evaluated relative to only a subset of previously visited nodes, must inherently be greater than or equal to its novelty when assessed against the entire set of visited nodes. Consequently, partitioned novelty enhances the efficacy of novelty search algorithms in problems characterized by a high atomic width. It allows polynomial novelty planners, which prune non-novel nodes, to achieve greater coverage of the search space, and *complete*[1] novelty planners, which prioritize novel nodes and revert to blind search or a secondary heuristic when novel nodes run out, to extend the informedness of their metrics.

[1]A complete search algorithm is one that will eventually visit all reachable nodes, thereby guaranteeing to find a solution if one exists. In practice, time and resource constraints can still limit the exploration of the entire state space.

Secondly, the use of appropriate partition functions guides the search towards the goal. This guidance is achieved indirectly, not by prioritizing the expansion of nodes depending on their partition – as is typical with heuristic functions – but by generating new partitions in areas of the state space where progress is detected by the partition functions. In this way, the exploration process is given more freedom while still being directed towards areas deemed more promising without tampering directly with the order of expansion of nodes.

Furthermore, this approach also partially addresses the issue of conjunctive goals. The greater number of novel nodes in newly discovered partitions naturally prioritizes the search in those areas, as compared to previous thoroughly-explored partitions. However, the strategy does not de-prioritize the expansion of novel nodes found in earlier partitions, thereby still enabling the discovery of alternative paths from these older partitions if progress stalls in the newly discovered areas. This represents an improvement over previous methodologies such as Serialized Iterative Width (SIW), which adheres more rigidly to specific subgoal orderings in its search strategy.

2.5.1.1 Best First Width Search

Partitioned novelty acts as the core component of the Best First Width Search (BFWS) planners [8]. BFWS represents a family of novelty-driven BFS solvers with the peculiarity of ordering node expansion primarily through a partitioned novelty measure, using goal-directed heuristics only for tie-breaking. This distinguishes these solvers from most previous contributions, which relied heavily on goal-oriented heuristics as the primary method of guiding the search. Prioritizing novelty exploration over heuristic exploitation then enables the search to lessen the impact of uninformed heuristic measures leading search algorithms into undesirable local minima of the state space topology. This has also allowed the implementation of polynomial variants of BFWS planners, pruning non-novel nodes, that achieve comperable problem coverage to previous state-of-the-art planners [14].

Crucially, BFWS solvers have demonstrated competitive performance when used in conjunction with less-informed but cheaply-computable partition functions, allowing for an alternative solution compared to state-of-the-art planners such as LAMA [19], which typically adopt highly-informed but expensive heuristics such as h_{ff} [15], that can instead rapidly explore a greater number of nodes per unit time, albeit at a higher memory cost (as nodes placed in the closed list consume memory). This trait of BFWS planners proves crucial to fostering our own planner contributions in Chapters 5 and 8.2.1.

Two key solvers of the BFWS family are BFWS(f_5), and Dual-BFWS [8]. BFWS(f_5) is a family of greedy best first search planners adopting the $f_5 = \langle w, \#g \rangle$ evaluation function, where w is the partitioned novelty metric, and $\#g$ refers to the goal-counting heuristic, that counts the number of atomic subgoals not-yet achieved in a given state. The partitioned novelty metric adopted is $w = w_{\#g,\#r}$ – with subscript referring to the partition functions – where $\#g(s)$ is still the goal-counting heuristic computed at state s, and the $\#r(s)$ heuristic refers to the number of atoms in the last computed relaxed plan which were made *true* between when the relaxed plan was computed and s. Such relaxed plans are computed only for states that improve their $\#g(s)$ heuristic compared to their predecessor, allowing the time-consuming relaxed plans to be computed more sparsely, and therefore speeding up the search algorithm. k-BFWS solvers represent polynomial variants of BFWS(f_5), where nodes with novelty greater than k are pruned.

Dual-BFWS is a *dual configuration* solver, meaning that it runs a first solver, referred to as the *frontend*, which is designed to seek a solution to the planning task, or fail under certain conditions. When the frontend fails, the dual configuration then falls back to a *backend* solver. Dual-BFWS adopts a fast polynomial 1-BFWS frontend solver, a BFWS(f_5) solver that prunes all nodes with a novelty $w(s) > 1$, quickly failing if all novel nodes run out before it finds a solution to the problem. The backend is a slower BFWS variant which takes inspiration from the $BFS(f)$ solver, using h_L [41] and h_{ff} [15] partition functions in width-2 partitioned novelty metrics, alongside h_L, h_{ff}, and helpful actions [15], for tie breaking.

2.5.2 Approximate Novelty

Singh et al. [10] proposed *Approximate Novelty*, which introduces the use of Bloom filters and state feature sampling to keep track of observed tuples in novelty and partitioned novelty search, demonstrating a beneficial impact on time and memory complexity, and allowing for an increase in the width of tractably computable novelty measurements. Bloom filters enable a more efficient storage of tuple records compared to the exponential blowup in the number of tuples with respect to the tuple size, albeit by allowing false positives when testing for the containment of an object in a set. Sampling state features instead reduces the set of tuples considered for novelty comparison, speeding up the computation of the measure. The study also formally describes the error probability introduced by the technique, which turns out to be low in practice. To our knowledge, no other work has addressed the idea of statistical sampling of preferred state features in novelty search nor planning in general. Extraction and use of state features [30, 38] also focuses on selection of relevant state features, but such features are obtained through general or domain dependent analysis as opposed to online statistical information. The greater width achievable through use of approximate novelty aids performance over domains with high atomic width, where use of partitioned novelty alone may not prove sufficient.

Another key contribution related to approximate novelty is the *open list control* algorithm for selecting nodes to insert into the open list when based on their approximate novelty metric value. Open list control is an adaptive policy which controls the rate of growth of states in the open list, restricting its size to avoid the exponential blowup in number of expanded states at higher problem widths. Singh et al. [10] model the search as a discrete-time dynamical system subject to perturbation, and formulate an optimal control problem [42] where optimal policies ensure that the branching factor b of $\mathcal{S}(P)$ − the average number of successors of each state − upper bounds the rate of growth sustained by the open list, deriving such an optimal policy analytically. In practice, this policy efficiently limits the number of nodes of each width value w observed which make their way into the open list, limiting its overall size and, thus, the rate of growth of memory usage.

Approximate novelty is adopted by the Approximate-BFWS planner [10]. This is a planner which iteratively runs a polynomial k-BFWS adopting approximate novelty, increasing the max-width k at each iteration. The use of approximate novelty in conjunction with open list control then controls the rate of growth of states in the open list, restricting its size to reduce the exponential blowup in number of expanded states at higher problem widths. The improvements to time and memory complexity brought by these modifications enable Approximate-BFWS to achieve state-of-the-art performance, improving the coverage of previous novelty planners on IPC planning benchmarks and achieving competitive running times in finding solutions.

2.5.3 Novelty as Heuristic Search

Katz et al. [9] introduced *novelty heuristics*, a notion of state novelty with respect to the value of underlying heuristic estimates [9, 39]. This novelty metric is based on individual state *facts*, where a fact f is a pair $\langle v, x \rangle$ where v is a state variable, and x represents a valuation for variable v out of a finite domain of possible valuations for that variable.

Definition 2.7. Given a heuristic function $h : \mathcal{S} \mapsto \mathbb{R}^{0+}$ and a set of states seen so far S, the novelty score of a *fact* (variable value) f is defined as

$$N(f, S, h) = \begin{cases} \min_{s \in S, f \in s} h(s) & \text{if } f \in s \text{ for some } s \in S, \\ \infty & \text{otherwise.} \end{cases}$$

Given a state s, the novelty score of a fact f in state s is then defined as $N(f, s, S, h) = N(f, S, h) - h(s)$ if $f \in s$ [9].

Unlike early contributions from Lipovetzky and Geffner [5, 7] – limited to recording occurrence of a state atom to determine a binary novelty value – the *heuristic novelty* measures the *novelty score* of individual facts in a state by calculating the difference between the minimum heuristic value among all states in which a fact was previously observed, and the heuristic value of the current state. In this way, it can quantify a fact novelty score that is not binary, but rather can account for how much better or worse the current state heuristic is compared to other states in which we already observed each individual fact. Crucially, it enables the detection of facts which are novel by the traditional novelty framework: if a fact was never observed previously in the search, thus corresponding to a novel atom in the original novelty framework [5], the best observed heuristic value of the fact is set to ∞, thus yielding a novelty score of ∞. However, when a fact was previously observed, it still enables the quantification of novelty or non-novelty relative to the underlying state heuristic value.

Multiple metrics are designed to aggregate individual novelty scores. One set of such metrics uses the novelty score to assign a binary value to each state fact, corresponding to its novelty or non-novelty, depending on whether the novelty score is greater than or less than 0. In this respect, these metrics conceptually share some characteristics with partitioned novelty, which also enables the evaluation of novel tuples in partitions – determined by underlying heuristic values – where said tuple was never observed. This includes partitions which improve the best heuristic value for a given fact, corresponding to a positive novelty score. Another set of metrics further extends the dichotomy between novel and non-novel facts by quantifying the value of individual facts through a range of values induced by the fact's underlying novelty score. To our knowledge, this technique represents the only instance thus far of a novelty metric over state variables which manages to quantify the contribution of novel and non-novel individual facts to the state novelty, offering a more refined state ranking compared to discrete partitioning based on novelty value. Such metrics have been adopted in multiple state-of-the-art planners [21, 22, 40].

2.5.4 Novelty Forgetting and Open List Reset

An alternative body of work that jointly targets the underlying novelty metric of a solver in conjunction with its open list, as does approximate novelty [10], is provided by Corrêa et al. [20] through *open list reset* and novelty *forgetting*.

The authors introduce two alternative open list reset techniques, where at key points in the search process the open list prioritizes newly generated states, biasing the search towards newly acquired information. Unlike the *open list control* policy [10] which prunes novel states to control the complexity of the search, *reset* deprioritizes old information, favoring fresh data. The first variant simply clears the entire open list once the search makes progress, resulting in a more aggressive approach which abruptly cuts off old information. The second variant, instead, implements a bucket-based queue, where the open list prioritizes the most recent queue, and once the search makes progress, pushes that queue back one bucket. "Progress" is generally tracked in terms of some underlying heuristic or event in the search, such as the improvement of the best heuristic value or the discovery of a new subgoal.

The technique employs the concept of forgetting, which broadens the search scope by "cancelling" the entire history of tracked tuples each time the search makes progress. This approach to forgetting the full history is less restrictive in some respects than previous methods that depend on partition functions [8, 10, 14]. By resetting the record of novel tuples – rather than partitioning them – it potentially allows a greater number of nodes to be expanded shortly after search progress is achieved, thereby aiding in solving subproblems with higher width. Conversely, there may be scenarios where partition functions provide the algorithm more leeway to explore different segments of the state space, particularly when there has been no progress for an extended period of time, and previously less-explored partitions may still offer more novel search directions compared to a single partition with a more recent history. In such situations, a single partition may be more prone to getting "stuck" and exhausting novel tuples.

The adoption of novelty with forgetting has been adapted for use in the *Scorpion-Maidu* solver [20], which incorporates a novelty heuristic into one of several open lists [43], instead of relying solely on novelty as the primary heuristic in its search, as is the case with BFWS solvers [8]. Consequently, the issue of exhausting novel nodes may not be as significant, since other heuristics in alternative open lists can potentially find different solutions to achieve progress in those areas of the state space. Meanwhile, forgetting novelty history offers the benefit of requiring only a single partition, which is recycled each time the algorithm makes new progress. This effectively addresses the significant issue of high memory costs associated with tracking tuple occurrences across all novelty partitions in scenarios with a large number of partitions.

In practice, Scorpion Maidu has demonstrated state-of-the-art performance, with a recent variant of the planner that additionally includes the h^2 preprocessor [44] winning the satisficing track of the most recent 2023 International Planning Competition [45]. This success underlines the effectiveness of its approach, offering a compelling alternative to the BFWS family of planners in blending novelty and heuristic search.

2.5.5 Sketches

The concept of *sketches* presented by Bonet and Geffner [46] provides an alternative extension to the principles of width-based planning. As outlined earlier in this section, the effectiveness of width-based search methods is heavily reliant on two main factors. Firstly, it depends on the feasibility of solving a given planning problem by decomposing it into subproblems, typically achieved by serializing the goal statement into conjunctions of atomic subgoals. Secondly, it relies on the low atomic width of these subproblems to tractably solve them using width-based algorithms. Work on sketches takes the approach of tackling these limitations directly by fundamentally altering the problem decomposition in order to make the underlying subproblems more tractable, guaranteeing low atomic widths and feasible solutions over serialized problem representations. The flip-side of the coin is that these decompositions end up not being domain-independent, thus requiring domain knowledge and manual definition of the underlying rules, although more recent contributions have also achieved progress in obtaining domain-independent planners which automatically define such subproblems [38].

Policy sketches are general problem decompositions which are represented as a set of *sketch rules* over a set of binary and numerical features. A feature here represents a function of the state, for example whether a condition is satisfied − *is a yellow block on the table?* − or a count − *how many yellow blocks are on the table?* Individual sketch rules then take the form $C \mapsto E$. C represents boolean conditions, in the form c or $\neg c$ for boolean features, and $c = 0$ or $c > 0$ for non-boolean features. E represent feature effects, a condition which describes a given change in the underlying feature, with features not mentioned in E expected to remain unchanged. In states where the conditions C are satisfied, sketch rules then allow the definition of sets of subgoal states representing the solution to a subproblem. The width of the sketch is given by the width of possible subproblems defined by the sketch rules in the policy sketch. Suitably crafted sketch policies can then bound the complexity of the problem, and guarantee tractability of the problem decomposition. Drexler et al. [47] provide policy sketches for multiple benchmark planning domains, including challenging problems for domain-independent planners such as *Childsnack*, proving a low sketch-width and demonstrating that SIW$_\mathrm{R}$, a variant of the SIW blind novelty algorithm with the modification of decomposing problems according to predefined sketches, fully solves instances for such domains in time polynomial in the underlying number of problem atoms.

2.6 Count-based Exploration

Learning agents in online decision making settings generally seek to learn a best possible policy, that is, which course of action to take in order to achieve the best possible outcome. In a Multi-armed Bandit setting, the goal is generally to minimize the *regret*, the difference in the sum of rewards obtained by following a given policy compared to the sum of rewards obtained by following the optimal policy − the best possible expected outcome. In a Reinforcement Learning setting, agents instead may seek to maximize the rewards obtained in a longer sequence of decisions. These pursuits introduce a fundamental trade-off − the exploration-exploitation dilemma − between following courses of action which the agent "knows" are good, and seeking new possible courses of action which have the potential to be better, but may also be worse. Effectively managing this dilemma is key to developing intelligent systems that can learn optimal behaviors in uncertain and dynamic environments.

2.6.1 The Exploration-Exploitation Dilemma

The exploration-exploitation dilemma encapsulates the trade-off that learning agents must navigate:

- Exploration involves trying out different actions to gather more information about the environment. This is crucial because without sufficient exploration, the agent may never discover the best actions to take. Exploration helps the agent to build a more accurate model of the environment's reward structure, but it can come at the cost of obtaining lower immediate rewards since the agent might choose suboptimal actions in the process of learning.
- Exploitation means using the known information to make decisions that maximize the immediate reward based on current knowledge. This is effective when the agent has a reliable understanding of the environment, as it focuses on optimizing returns with the learned strategies. However, excessive exploitation can lead to suboptimal long-term results if the agent overlooks better options that were less explored.

In the **Multi-Armed Bandit** (MAB) problem, the agent faces a set of actions (arm pulls) each associated with an unknown probability distribution of rewards. The challenge is to determine which arm to pull to maximize the cumulative reward over time. The regret in this context is quantified as the difference between the rewards obtained by always pulling the best arm (in hindsight, after infinite trials) and the rewards actually gathered by the agent's strategy.

In **Reinforcement Learning** (RL), the agent interacts with a more complex environment where actions not only yield immediate rewards but also affect future states and subsequent rewards. The environment is typically a tabular setting or a *Markov*

Decision Process (MDP), a framework for modeling the agent's decision-making situations consisting of states, actions, rewards and transition probabilities. This setting extends the dilemma by adding the dimension of state transitions, which influence the optimal policy in a dynamic manner. Agents must learn a policy that maximizes cumulative future rewards, often discounted over time to prioritize more immediate returns. This requires balancing exploration to discover profitable long-term strategies against exploiting short-term known gains.

2.6.2 Epsilon-Greedy Exploration

A naïve solution to the exploration-explitation dilemma is given by the *Epsilon-Greedy* policy, a simple method where the agent chooses the best-known action most of the time (exploitation) but also randomly selects any available action with a small probability ϵ (exploration). This allows for continual learning while mostly focusing on high-reward actions. Modifications of the technique further have the ϵ probability decay as the learning progresses, promoting a behaviour that is more exploratory at the beginning, and that gradually shifts towards exploiting as the model accumulates more information about the environment. This adjustment reflects the understanding that exploration is more meaningful when information is scarce, and exploitation is more effective when substantial information has been gathered.

This approach is considered naïve because it employs an uninformed method of exploration − necessitating empirical testing of multiple ϵ values and decay strategies to optimize performance − as it does not tailor exploration to the agent's experiences and knowledge of the environment. For instance, an agent might have non-uniform knowledge of the environment, with certain actions or areas of the state space visited more frequently than others, resulting in more accurate estimates for those areas compared to less visited regions. It is desirable for a more informed exploration policy to account for this imbalance, strategically targeting less explored areas to refine the estimates which may be less accurate due to insufficient data, and thereby ensuring that exploration efforts are concentrated where they are most needed.

To address these shortcomings, count-based exploration methods have been introduced. These techniques adjust the exploration incentive based on a record of the frequency of occurrence of a given course of action − how many times an action has been chosen or a state has been visited. For example, actions that have been explored less frequently are considered more uncertain and thus are given a higher priority for future exploration. This method helps to systematically reduce uncertainty about the less-explored alternatives.

2.6.3 The UCB1 Algorithm in the Multi-Armed Bandit Problem

As mentioned in the previous section, count-based exploration methods are used to obtain a more informed exploration policy that, in the MAB setting, seeks to explore bandit arms whose expected rewards are more uncertain. Still, the exploration-exploitation dilemma also dictates that we must balance such exploratory behaviour with our desire to obtain the best possible cumulative rewards. Selecting a bandit arm with a low expected payoff a few times may be necessary to confirm its distribution of rewards, but ideally the agent would also stop selecting that arm as soon as possible.

Such balance in MAB and RL settings is most commonly achieved through an *exploration bonus* whose value is added to a reward, providing an additional incentive for the agent to explore options with low visitation counts, while still accounting for the underlying expected reward or the state and/or action. This formulation introduces the concept of "*optimism in the face of uncertainty*", a key principle adopted by many exploration techniques across multiple fields. This idea embodies maintaining an optimistic outlook on untested options, encouraging the exploration of lesser-known paths that may yield greater rewards.

This tactic is exemplified by the well-known *UCB1* exploration algorithm [48]. The algorithm optimally balances exploration and exploitation by selecting actions, referred to as the arms of a bandit, which maximize an *upper confidence bound*:

$$Q_t(i) + \sqrt{\frac{2\log N}{N(i)}}$$

− the sum of the empirical average rewards $Q_t(i)$ of selecting arm i, and a confidence interval term (the exploration bonus) $\sqrt{\frac{2\log N}{N(i)}}$, where $N(i)$ is the count of pulls of arm i, and N is count of total arm pulls.

The exploration bonus is constructed from the *Chernoff-Hoeffding inequality*:

$$P(X + a < \mu) \leq e^{-2na^2}$$

Where $X = \frac{1}{n}\sum_{i=1}^{n} X_i$ is the sample mean of observations, μ is the true mean of the distribution, a the one-sided upper confidence bound, and n is the number of observations. The formula bounds the probability that the sample mean X, augmented by the upper confidence bound a, is less than the true mean μ, showing how the probability decreases exponentially with increasing observations n.

This formula embodies the underlying intuition behind the UCB1 algorithm's strategy. In a scenario in which an arm is pulled twice, and two low values are observed, the sample mean for that arm is going to be low; however, it is still possible that the arm actually has a high true mean, and that the observed low values were simply due to low-probability events where the samples were outliers. In order to obtain an empirical estimate which does not underestimate the true mean with a high degree of confidence, the UCB1 exploration bonus, then, is crafted such that adding it to the sample mean of an arm produces an upper confidence bound, representing its largest plausible mean with high probability. Auer et al. [48] critically prove that the inclusion of this exploration bonus in UCB1 guarantees an upper bound on cumulative regret which grows logarithmically in the total count of arm pulls N, achieving near-optimal performance in a polynomial number of time steps (arm pulls).

2.6.4 Count-based Exploration in Reinforcement Learning

The theoretical and empirical results obtained by UCB1 are foundational to highlighting the potential of count-based exploration techniques, and how a suitable use and analysis of the empirical counts of events − a relatively simple measure on its own − can give rise to breakthrough and non-trivial results. These achievements have promoted the development of count-based algorithms that prioritize exploration across various fields including reinforcement learning and decision-making processes.

2.6.4.1 Provably Efficient Count-Based RL

Strehl and Littman [13] introduce two variants of the MBIE (Model Based Interval Estimation) [49] reinforcement learning algorithm that achieve efficient exploratory behavior. The two algorithms only differ in the way in which action-value estimates (the *Q-function*) – estimates maintained for each state-action pair and used by the algorithm to select actions – are computed. Both algorithms achieve their exploratory behavior by computing confidence intervals which rely on the empirical counts of occurrences of state-action pairs, thus following the “optimism in the face of uncertainty” principle. The MBIE-EB variant (MBIE-Exploration Bonus) in particular manages to achieve efficient exploratory behavior by only adding an exploration bonus component $\frac{\beta}{\sqrt{n(s,a)}}$ to the base Q-function, where β is a constant and $n(s, a)$ is the empirical count of state-action visitations.

Such efficient behavior is defined in terms of the concepts of *sample complexity* and *PAC-MDP* (Probably Approximately Correct in Markov Decision Processes) [50]. Sample complexity refers to the number of timesteps over the course of any run in which an algorithm is not executing a near-optimal policy, referred to as ϵ-optimal where ϵ is the allowed degree of error, from its current state. Considering the degree of error with respect to its current state allows for the an evaluation based on the trajectories or areas of the state space the algorithm has visited thus far, as opposed to a comparison of its policies to the global optimal policy, thus evaluating the learning algorithm itself in terms of the “mistakes” it is expected to make from its current position. An algorithm is then PAC-MDP if it ensures a sample complexity which is polynomial in quantities $(|S|, |A|, \frac{1}{\epsilon}, \frac{1}{\delta}, \frac{1}{1-\gamma})$ with probability $1 - \delta$, where $|S|$ and $|A|$ are the number of states and actions, ϵ is the aforementioned degree of “acceptable” error, and γ is the future reward discount factor. In other words, it seeks to bound polynomially the number of timesteps where it does not act near-optimally. For an algorithm to be PAC-MDP, it must also maintain polynomial per-step computational complexity in these quantities, to keep in check the computational cost of the algorithm. Strehl and Littman [13] prove that both their proposed variants of MBIE are PAC-MDP, with the simpler exploration bonus of MBIE-EB achieving so with a lower computational cost.

2.6.4.2 Count-Based Exploration and MCTS

The count-based UCB1 exploration strategy has also been adapted to Monte-Carlo Tree Search algorithms with the introduction of the UCT algorithm [51]. Monte Carlo Tree Search (MCTS) represents a family of algorithms that explore a tree of possible options by systematically expanding the most promising nodes based on the results of randomized simulations from the current state. It progressively builds a search tree, balancing the exploration of new paths with the exploitation of paths that have previously shown success. The UCT algorithm applies the UCB1 exploration bonus to the empirical estimates of state or state-action value in the search tree, seeking a similar exploratory behavior where the additional exploration bonus allows the algorithm to not underestimate the true mean of rewards of a state or state-action, incentivizing the exploration of paths which may feasibly lead to better discounted rewards.

A similar exploration bonus is also adopted in the landmark AlphaGo MCTS algorithm [52]. AlphaGo augments the base MCTS algorithm using function approximation to estimate a policy network, which outputs a probability distribution over legal moves, and a value network, which outputs a scalar estimate of the value of a state. Crucially, the action value of a state-action pair is added to a bonus which is proportional to the prior probability given by the policy network, but also decays proportionally to the empirical count of visitation of state-action pairs to promote the exploration of less visited options.

2.6.4.3 Count-Based Exploration in Large State Spaces and Continuous RL

Challenges arise in very large tabular settings and in continuous MDP settings, primarily because most states or actions are never actually visited, rendering count-based techniques less effective. In expansive state spaces, like those AlphaGo navigates[2], most states are not visited within a feasible amount of training time. To manage this, AlphaGo utilizes its policy network and exploration bonuses to limit the number of states it explores, aiming to make the vast state space more manageable. Even then, the approach still demanded significant computational resources, and extensive pre-training on existing domain knowledge. In MDPs with continuous features, applying count-based

[2]The game of Go is notoriously “hard”, as it admits roughly 250^{150} possible sequences of moves [52].

strategies becomes fundamentally impractical as the set of possible states becomes uncountably infinite in theory. Even attempts to approximate these spaces, reducing the expressiveness of the domain, can still result in the large state-space issue, even for relatively simple problems.

This issue was addressed by the introduction of *pseudocounts* [53], a state visitation measure which relies on a density function to estimate visitation history, enabling the count-based exploration framework to generalize across large and continuous state-spaces. This is done by defining two unkowns: a *pseudocount function* at time n, $\hat{N}_n$, and a *pseudo-count total* $\hat{n}$. These are related through the following constraints:

$$\rho_n(x) = \frac{\hat{N}_n}{\hat{n}} \qquad\qquad \rho'_n(x) = \frac{\hat{N}_n + 1}{\hat{n} + 1}$$

where ρ is the density function, and $\rho'_n(x)$ refers the updated density function after state visitation at timestep n. These constraints are used to model the behavior of pseudocounts in terms of that of discrete empirical counts $-$ the discrete empirical count density for a state given its count N and the total count n (equivalent to the number of timesteps) is $\frac{N}{n}$, and this density is updated by adding 1 to both N and n components when visiting the state. The pseudocount is derived from the constraints:

$$\hat{N}_n(x) = \frac{\rho_n(x)(1 - \rho'_n(x))}{\rho'_n(x) - \rho_n(x)} = \hat{n}\rho_n(x) \tag{2.1}$$

The pseudocount function $\hat{N}$ can thus be obtained by only using the prior and posterior values of density function ρ, whereas the pseudocount total $\hat{n}$ is implicit, and does not need to be calculated. The use of this pseudocount function then allows the obtainment of count estimates derived from powerful density models which are capable of generalizing across large state spaces and continuous features, addressing the pain points of previous count-based solutions. Furthermore, such pseudocounts are shown to demonstrate useful properties, allowing new states to have non-zero counts while maintaining roughly zero counts for novel events, as well as asymptotically converging to a constant ratio of the empirical count measure N under given assumptions of the density function. Multiple techniques have been proposed as density models that generalize across large and continuous state spaces such as pixel-level representations, including Context Tree Switching [53], PixelCNN [54], and a linear function approximation of feature space density [55].

2.6.5 Relation to Novelty in Planning

The goal-agnostic nature of novelty heuristics has allowed novelty planners to be adapted to the task of planning over simulators, enabling its adaptation to RL problems such as the Arcade Learning Environment [56], augmenting the exploratory capabilities of online agents [57, 58], as well as RL agents leveraging learnt policies [59]. These results underscore the potential for Novelty to foster knowledge transfer across the subfields of exploration in RL and Planning, which typically display limited overlap in ideas and techniques [60]. However, gaps still remain in linking the theoretical analysis of width-based exploration methods in Planning with exploration methods in RL. In this regard, multiple conceptual links can be identified between width-based exploration methods and count-based exploration.

A superficial connection may be identified in underlying definitions, where Novelty focuses on whether a piece of information has occurred, corresponding to the dichotomy between an empirical count of 0 and a count ≥ 0. Such simplification allows for multiple techniques to improve the time and memory complexity of novelty techniques, at the cost of ignoring the additional information included in empirical counts > 0 that count-based exploration methods leverage to obtain their results. A major difference in this respect is that novelty focuses on the occurrence of tuples, as opposed to the majority of discrete tabular count-based exploration methods which record state visitation count. This provides novelty exploration with an intrinsic means of generalizing feature information across states, which proves essential in Classical Planning, where states are generally not visited twice.

Novelty metrics in classical planning are also fundamentally optimist in the face of uncertainty. They prioritize visitation of new states which have a certain degree of unseen information, and in a more extreme case of what seen in RL, even achieve good result by precluding exploration all-together of states that do not contribute to the search with novel information. Use of partition functions or novelty heuristics go one step further, seeking an automatic balance between exploration of novel nodes and exploitation of beneficial heuristic values, suggestive of exploration bonuses added to rewards in RL.

Ultimately both techniques seek to induce a sort of efficiency in the exploration process through an optimistic outlook on unseen information, in order to explore various alternatives available, but also to uncover the best options and minimize the effort spent exploring options which are unlikely to yield useful search progress or rewards. Both novelty planners and count-based exploration agents can achieve some degree of theoretical polynomial guarantees in doing so, through tuple-graph analysis of low-width planning domains and PAC-MDP analysis in RL respectively. It is possible that common theoretical foundations to both techniques may be responsible for observed efficient-exploration results across the fields.

Another framework which closely describes width-based exploration methods applied to reinforcement learning is that of *intrinsic motivation*. Intrinsic motivation seeks to provide agent guidance through an intrinsic reward brought by the exploration of that which "surprises" them [61]. This is commonly achieved through a concept of *learning progress*, whereby the difference or divergence between some underlying prior and posterior value or distribution before and after observing a new piece of information is measured, favouring the exploration of larger such divergences, which corresponds to a greater degree of "surprise" in the observed information.

The intrinsic motivation argument can serve as a rationale for novelty, where the prior on observed information is represented by a record over binary tuples [5, 8, 10] or feature values [57–59]. Agents seek states that can update this prior distribution with new information by observing novel tuples or feature values, greedily maximizing the acquisition of new information. Bellemare et al. [53] establish a theoretical link in RL between intrinsic motivation and their proposed pseudocounts, showing that exploration bonuses derived from pseudocounts effectively upper-bound those obtained through *information gain*, a commonly used quantity to quantify intrinsic reward. The common connection with intrinsic motivation, then, further strengthens the conceptual links between novelty and pseudocount measures. Moreover, the focus on tuple-based occurrences in width-based exploration can be directly integrated into the density models used in the pseudocount formula, bridging another key difference between novelty and count-based exploration frameworks.

These results underscore the substantial conceptual links between width-based and count-based techniques, hinting at a significant knowledge gap for stronger theoretical connections between the two successful efficient-exploration techniques.

Chapter 3

Count-Based Novelty in Classical Planning

This chapter introduces our proposed count-based extension to novelty frameworks in classical planning. We first highlight the main limitations of existing techniques that a count-based novelty solution addresses in Section 3.1, and describe in detail the notation that will be used throughout this thesis in Section 3.2. *Classical count-based novelty* and *partitioned classical count-based novelty* are introduced in Section 3.3, followed by the main theoretical analysis in Section 3.4. Sections 3.5 and 3.6 further discuss the technique and theoretical results, comparing them to previous work on novelty in classical planning and providing initial results to bridge the gap between count-based exploration techniques in classical planning and reinforcement learning.

3.1 Motivation

Sections 2.4 and 2.5 introduce the concept of Novelty and variations of the metric which are currently adopted in several state-of-the-art classical planning algorithms. All mentioned techniques limit themselves to the original idea of measuring state information content through the occurrence of tuples in the search history. Our main contribution consists of a count-based measure of state novelty, *classical count-based novelty*, which seeks to induce efficient exploration of the state space by making use of the additional information contained in the count of occurrences of tuples in the search history. This addresses shortcomings of the current Novelty framework and techniques presented in Chapter 2. We refer to previous width-based Novelty methods as *width-novelty*, to distinguish them more clearly from our contributions in this thesis.

There are several main limitations of width-novelty frameworks that our contribution addresses:

1. Width-novelty metrics are defined and implemented with respect to a maximum width w, which determines what set of tuples the novelty metric considers, limiting the maximum size of such variable conjunctions. This bounds the width-w search that the metric allows the search algorithm to perform or approximate, according to the implicit tuple graph (Definitions 2.1 and 2.3) generated by the search. The exponential time and memory complexity with respect to w (Theorem 2.4) turns out to be one of the main limitations of practical implementations of novelty heuristics, forcing a trade-off between the maximum width selected and the tractability of the heuristic computation. This trade-off also makes search algorithm performance highly susceptible to the selected maximum width, which is generally either selected empirically as a hyperparameter of the model [8, 20, 21], or increased iteratively in polynomial novelty planners when the previous iteration fails [5, 7, 10].

2. At the opposite end of the novelty-informedness/tractability trade-off is the issue of width-novelty metrics losing informedness once they run out of novel nodes, and thus the "range" of their informedness being inevitably tied to the maximum width of the adopted novelty heuristic. The ability to prune non-novel nodes is central to the polynomial performance of novelty-based polynomial planners, however running out of novel nodes forces the algorithm to fail without finding a solution. For complete, non-polynomial implementations, running out of novel nodes precludes the novelty heuristic from performing an efficient-exploration, until new novel nodes are found. This issue is partially alleviated through the use of secondary tie-breaking heuristics or alternative open lists, but these are forced solutions to the underlying limitations of width-novelty heuristics. Increasing the width w iteratively has also demonstrated promise in balancing such issue with the underlying complexity of computation of the heuristic. However, this solution proves to be unfeasible for adaptation to search algorithms that are not designed

around it, for example Scorpion-Maidu [20], which adopts a novelty heuristic in just one of several open lists used for search. It also inevitably requires redundant computations to be performed at each new search iteration, which limits its applicability to planners with high per-node computational requirements and forces hard trade-offs in the selection of heuristics, such as restricting the use of highly informed – but slower – heuristics like h_{ff}.

3. Ignoring the intractable computational cost of high-width novelty heuristics, the exponential increase in the number of feasible novel nodes introduces an issue regarding the degree of separation of nodes with higher novelty values. Separating novel from non-novel nodes provides a meaningful improvement to the efficiency of the search, however there already are multiple domains where a width-2 BFWS(f_5) solver only ever expands novelty-2 nodes, and yet still takes a long time or even fails to find a valid plan. In these situations, a more granular, yet still informed, degree of separation of beneficial and non-beneficial nodes would help further direct the search towards valid solutions. Novelty heuristics [9] explicitly address this issue, providing formulations which account for the degree of novelty and non-novelty of underlying state variables (Section 2.5.3). However, this formulation is with respect to the value of underlying heuristics, which plays against the effectiveness of novelty in achieving an efficient search to fill the gaps created by the uninformedness of goal-aware heuristics used in modern planning algorithms. If such heuristics were consistently very accurate in estimating the optimal heuristic h^*, there would arguably rarely be any need for novelty in the first place, and planning would be a quasi-convex optimization problem.

4. Width-novelty exploration frameworks, while adaptable to a broader set of problems outside of Classical Planning, including planning with simulators and reinforcement learning, still remain theoretically independent to other exploration techniques. Furthermore, current theoretical results mainly revolve around the behavior of novelty in Classical Planning, and find little common ground with respect to existing theoretical analysis of alternative exploration techniques in other fields. This is also true for novelty techniques which modify the basic novelty definition to target problems not covered by the discrete atomic definition of novelty (Definition 2.5), such as numerical features [62]. As with the connections presented in Section 2.6.5, these solutions demonstrate clear conceptual links to novelty, but fail

to achieve explicit theoretical connections. Results which bridge such a gap could provide beneficial knowledge transfer, and more general and explicit theoretical justifications of novelty's exploratory behavior in non-discrete and non-planning domains.

Count-based novelty is not constrained to width-novelty's binary classification of novel information, and provides new solutions addressing the limitations in existing novelty literature listed above.

Extracting the full set of information included in tuple counts allows count-based novelty to estimate and solve higher atomic width problems through the evaluation of smaller-sized tuples. This study focuses on the analysis and evaluation of a variant of our proposed metric which only evaluates sets of size-1 tuples, corresponding to the underlying instance features, thus achieving a novelty metric which only ever operates over a linear number of such tuples. Analysis in Section 3.4 and our experimental results demonstrate that this formulation is sufficient to estimate the presence of larger novel tuples in visited states, approximating higher-width searches without the previously discussed exponential blowup in number of evaluated tuples.

The substitution of width-novelty's dichotomic notion of tuple occurrence with a countable record over tuples also provides a more granular metric that does not run out of "useful" information, precluding the exhaustion of novel nodes. The novelty heuristic this way remains informative even when observing high tuple occurrence counts, enabling the separation and ranking of states in well-explored areas of the state space. This is done through a blind pure-novelty solution, true to the original definition of novelty (Section 2.4) and that does not rely on the use of goal-aware heuristics.

Finally, we provide an explicit connection between our contribution and a general RL count-based exploration framework, demonstrating that the count-based novelty value of a state can be represented as a pseudocount [53] through the use of a discrete density function that selects the minimum empirical count of underlying tuples of feature values. Through this result, and the connections to existing novelty theory in Classical Planning provided our theoretical analysis, we show that count-based exploration bridges the gap between exploration techniques and theoretical analysis in Planning and RL, providing the basis for further knowledge transfer.

3.2 Preliminaries

We briefly reiterate the main aspects of the notation used to describe the classical planning model presented in Section 2.1.1:

- The classical planning model is defined as $S = \langle S, s_0, S_G, A, f \rangle$, where S is a discrete finite state space, s_0 is the the initial state, S_G is the set of goal states, and A is the set of actions.
- $A(s)$ denotes the set of actions $a \in A$ that deterministically map one state s into another $s' = f(a, s)$, where $A(s)$ is the set of actions applicable in s.
- $s_t \in S$ denotes the t^{th} visited (generated) state.
- $s_{0:t}$ denotes the sequence of $t+1$ states generated at time-steps $0, 1, ..., t$.
- The STRIPS planning language defines a problem $P = \langle F, O, I, G \rangle$, where F denotes the set of boolean variables, O denotes the set of operators, $I \subseteq F$ is the set of variables that fully describe the initial state, and $G \subseteq F$ is the set of variables that are *true* in the goal state.

3.3 Classical Count-Based Novelty

Classical count-based novelty operates over states that assign a value to a finite number of variables $v \in V$ over finite and discrete domains. In problems defined via STRIPS, without loss of generality, $V = F$ are boolean variables. Let V be the set of all variables, and $U^{(k)} = \{X \subseteq V \mid |X| = k\}$ the set of all k-element variable conjunctions. A tuple $u \in U^{(k)}$, specifically $u = \{v_1, v_2, \ldots, v_k\}$, represents a conjunction of k variables. Given a state s that assigns a boolean value to each variable in V, the value of the tuple u in state s, denoted $s(u)$, is defined as the conjunction of the values of the k variables in u, $s(u) = s(v_1) \wedge s(v_2) \wedge \cdots \wedge s(v_k)$, where $s(v_i)$ is the value of variable v_i in state s. We say $s(u)$ is *true* if all $v \in s(u)$ are *true*, and tuple u is *true* in s if $s(u)$ is *true*. Let $s_{0:t}(u)$ denote the sequence of values of tuple $u \in U^{(k)}$ in state sequence $s_{0:t}$, and let $U^{+(k)}(s) \subseteq U^{(k)}$ denote the set of tuples u in state s where $s(u)$ is *true*.

Definition 3.1 (Classical count-based novelty)**.** The count-based novelty $c^U(s)$ of a newly generated state s at time-step $t+1$ given a history of generated states $s_{0:t}$ and set of variable conjunctions $U = U^{(k)}$ for some tuple size k is:

$$c^U(s_{t+1}) := \min_{u \in U^+(s_{t+1})} (N_t^u(s_{t+1}))$$

where $N_t^u(s_{t+1})$ counts the number of states $s_i \in s_{0:t}$ where $s_i(u) = s_{t+1}(u)$.

That is, for each tuple u that is *true* in s_{t+1}, we count the number of states $s_i \in s_{0:t}$ where $s_i(u)$ is *true*, and we select the minimum out of those counts.

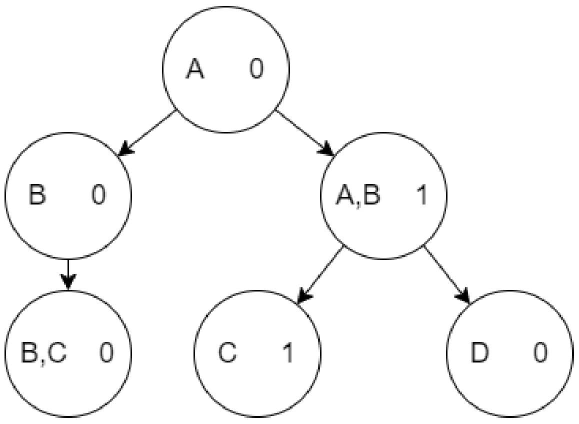


FIGURE 3.1: Each circle represents a visited nodes. Circles contain variables (left) and the node's count-based novelty over size-1 tuples (right), for a BrFS expansion ordering from left to right.

Tuple	Count
A	2
B	3
C	2
D	1

TABLE 3.1: Size-1 tuple counts after visiting all nodes in Fig. 3.1.

Example 3.1. *We refer to Figure 3.1 to represent a planning problem where nodes are visited and expanded in a breadth-first search ordering, visiting children of expanded nodes from left to right. The variables that make up the underlying state for each node are shown in the left portion of each node circle. The value on the right side of each node represents its classical count-based novelty value, according to Definition 3.1 over the set of size-1 tuples $U^{(1)}$, given the aforementioned order of state visitation. Table 3.1 represents the record of tuple counts after all nodes in Fig. 3.1 have been visited. Note that for each state, we first evaluate the minimum count of its tuples according to the history of previously visited states, and then update the counts of tuples that occur in the state. Comparison with the IW(1) (left) graph in Fig. 2.2 also highlights the equivalence between novelty-1 nodes according to a width-novelty heuristic (Definition 2.5) and a size-1 tuple count-based novelty value of 0, as both cases represent variables which have* *never been observed before. Also note that in this example, the count-based novelty value of nodes is not used to affect the order of expansion of nodes.*

Following prior work on Novelty [8], we also define a version of count-based novelty which uses *partition functions* to separate the search space into distinct sub-spaces.

Definition 3.2 (Partitioned classical count-based novelty)**.** The partitioned count-based novelty $c^U(s)$ of a newly generated state s at time-step $t+1$ given partition functions $h_1, ..., h_m$ is:

$$c^U_{h_1,...,h_m}(s_{t+1}) := \min_{u \in U^+(s_{t+1})} (N^u_{t;h_1,...,h_m}(s_{t+1}))$$

where $N^u_{t;h_1,...,h_m}(s_{t+1})$ counts the number of states $s_i \in \{s_{0:t} \mid h_1(s_i) = h_1(s_{t+1}) \wedge ... \wedge h_m(s_i) = h_m(s_{t+1})\}$ where $s_i(u) = s_{t+1}(u)$.

In other terms, we are obtaining tuple counts relative to the partition of previously generated states where $h_j(s_i) = h_j(s_{t+1})$ for all $1 \leq j \leq m$, as opposed to the full state history $s_{0:t}$. It trivially follows that $N^u_{t;h_1,...,h_m}(s_{t+1}) \leq N^u_t(s_{t+1})$ and $c^U_{h_1,...,h_m}(s_{t+1}) \leq c^U(s_{t+1})$.

Partitioned novelty does not run out of novel nodes, and thus does not require partitions to expand the set of novel nodes as do partitioned width-novelty heuristics (Definition 2.6). However, it still benefits from the other main function of partitions, that is guiding the direction of exploration towards areas of the state space that are estimated to be closer to the goal according to the underlying partition functions. Resetting the tuple counts in newly created partitions still encourages exploration of newly discovered areas, compared to previously discovered partitions where tuple counts are likely to be higher. Not running out of novel tuples in more thoroughly explored areas of the state space, however, helps maintain a novelty-led efficient exploration in older partitions even after lower width-based novelty heuristics would revert to blind search. This additional trait of the heuristic can help prevent a search algorithm from getting "stuck" in undesirable search directions, potentially aiding it in finding alternative paths to the goal from earlier in the search.

3.4 Theoretical Analysis

Average Hamming distance measures how "new" the information in a state, represented using binary features, is compared to all previously visited states based on a normalized Manhattan distance, helping define lesser-seen areas of the state space. Theorems 3.6 to 3.9 detail the extent to which information on size-1-tuple counts can direct a search towards such areas. Theorems 3.10 and 3.11 then show the mechanisms through which prioritizing nodes with low counts and greater average Hamming distances enables the discovery of more previously-unseen tuples in the search, thus uncovering more novel information.

In this section, we define node $n_i = n_i(s_i)$ as referring to a state s_i, where the sequence $n_{0:t}$ corresponds to sequence $s_{0:t}$. The distinction between a node n_i and its corresponding state s_i lies in the equality operator: $n_i = n_j$ iff $i = j$, implying that $s_i = s_j$, whereas $s_i = s_j$ denotes the equality of all underlying variable values v in s_i and s_j. Crucially, throughout the entire section we assume that $U = U^{(1)} = V$, that is, we are only looking at counts over single-variable tuples. We thus simplify the tuple notation by denoting $s^i = s(v_i)$.

Let $L = |s| = |V|$, and *Hamming distance*:

$$H(n, n_j) = H(n(s), n_j(s_j)) = \sum_{i=0}^{L-1} 1_{s^i \neq s^i_j}$$

We then define *normalized Hamming distance* as:

$$\delta(n, n_j) = \frac{1}{L}(H(n, n_j)) = \frac{1}{L}\sum_{i=0}^{L-1} 1_{s^i \neq s^i_j}$$

We define the *average normalized Hamming distance* of a node n with respect to all nodes in $n_{0:t}$ as:

$$\alpha_{0:t}(n) = \frac{1}{W} \sum_{i=0; n_i \neq n}^{t} \delta(n, n_i)$$

where $W = t$ if $n \in n_{0:t}$ or $W = t + 1$ otherwise, noting that in the first case we are skipping a node's comparison with itself.

Let the *tuple count distribution* and *minimum empirical count distribution* be

$$\mu_t^i(s) = \frac{N_t^{v_i}(s)}{t+1} \qquad \text{and} \qquad \mu_t^{min}(s) = \min_{i \in V}(\mu_t^i(s)) \tag{3.1}$$

noting that the minimum is over the entire set of variables $V = U^{(1)}$ rather than the set of *true* variables $U^{+(1)}$ in a state s used in Definition 3.1 and 3.2, and $0 \leq \mu_t^i(s) \leq 1$.

We provide a justification of this change through Lemmas 1 and 2, demonstrating a correspondence between empirical counts and Hamming distances over $U^{(1)}$ in binary vectors, and the same metrics over the set $U^{+(1)}$ in binary vectors that include negated variables. This allows us to align our results with a STRIPS representation that includes negated variables. For the set of L variables V, we define s_{neg} for states s over $V_{neg} = V \cup \{\neg v \mid v \in V\}$ such that $s_{\text{neg}}(v) = s(v)$, $s_{\text{neg}}(\neg v) = \neg s(v)$ for all $v_i \in V$.

Lemma 3.3. *The Hamming distance $H(s, s')$ between states s and s' equals the true Hamming distance $H_{true}(s_{neg}, s'_{neg})$, considering only variables in s_{neg} that are true.*

Proof. Since $H(s, s') = |\{i \mid s(v_i) \neq s'(v_i)\}|$, we define $H_{\text{true}}(s, s') = |\{i \mid s(v_i) \neq s'(v_i), s(v_i) = 1\}|$ and $H_{\text{false}}(s, s') = |\{i \mid s(v_i) \neq s'(v_i), s(v_i) = 0\}|$, noting that $H(s, s') = H_{\text{true}}(s, s') + H_{\text{false}}(s, s')$. Since for each variable $s(v_i) = 1 \in s$ there are two variables $s_{neg}(v_i) = 1 \in s_{neg}$ and $s_{neg}(\neg v_i) = 0$, and for each variable $s(v_i) = 0 \in s$ there are two variables $s_{neg}(\neg v_i) = 1 \in s_{neg}$ and $s_{neg}(v_i) = 0$, follows that $H_{\text{true}}(s_{neg}, s'_{neg}) = H_{\text{true}}(s, s') + H_{\text{false}}(s, s') = H(s, s')$ □

Lemma 3.4. *The empirical count $N_t^{v_i}(s)$ for any value $s(v_i)$ corresponds to the empirical count $N_t^{v_x}(s_{neg})$, where $x = i$ if $s(v_i) = 1$ and $x = j | s_{neg}(v_j) \equiv s_{neg}(\neg v_i)$ if $s(v_i) = 0$.*

Proof. From the definition of s_{neg}, for every variable $s_j(v_i) = 0 \in s_{0:t}(v_i)$ we have that $s_{neg;j}(v_i) = 0$ and $s_{neg;j}(\neg v_i) = 1$. The proof for $s_j(v_i) = 1$ case is symmetrical. Lemma 3.4 follows. □

We can thus show that the minimum of counts over *true* variables in s_{neg} − its count-based novelty − corresponds to the minimum of counts over all variables in s.

Proposition 3.5. $c^{V_{neg}}(s_{neg;t+1}) = \min_{v \in V}(N_t^v(s_{t+1}))$

Proof. Proof follows from Lemma 3.4 and Definition 3.1, by noting that to each feature in the left-hand side equation, corresponds a feature in the right-hand side, and viceversa. Thus, a feature in the left-hand side has the minimum count over $v \in V^{+}_{neg}$ iff a corresponding feature in the right-hand side has a minimum count over $v \in V$, and these values must be equivalent. □

Theorem 3.6. *The average normalized Hamming distance $\alpha_{0:t}(s)$ of a state s to the $t+1$ states in history $s_{0:t}$ is upper bounded by:*

$$\alpha_{0:t}(s) \leq 1 - \mu_t^{min}(s)$$

Proof. The average Hamming distance of $s(v_i)$ with respect to the value of variable i in all states $s_j \in s_{0:t}$ is equivalent to

$$\frac{1}{t+1}\sum_{j=0}^{t} 1_{s_j^i \neq s^i} = 1 - \frac{1}{t+1}\sum_{j=0}^{t} 1_{s_j^i = s^i} = 1 - \frac{1}{t+1} N_t^{v_i}(s) = 1 - \mu_t^i(s)$$

Thus we have

$$\begin{aligned} \alpha_{0:t}(s) &= \frac{1}{t+1}\sum_{j=0}^{t}\frac{1}{L}\sum_{i=0}^{L-1} 1_{s_j^i \neq s^i} = \frac{1}{L}\sum_{i=0}^{L-1}\frac{1}{t+1}\sum_{j=0}^{t} 1_{s_j^i \neq s^i} \\ &= \frac{1}{L}\sum_{i=0}^{L-1}(1-\mu_t^i(s)) \leq \frac{1}{L}\sum_{i=0}^{L-1}(1-\mu_t^{min}(s)) \\ &= 1 - \mu_t^{min}(s) \end{aligned} \tag{3.2}$$

□

3.4.1 Parent-Child Average Distance Bounds

We provide a set of results on the average normalized Hamming distances of a child node with respect to its parent node, and the impact that the count-based novelty of a node has on this bound. Let n^c and n^p be the child and parent node respectively, where an action $a \in A$ is performed on n^p to flip the binary value of e variables, which we refer to as the *effects*. Let n^c be the newly generated node n_t.

Theorem 3.7. *Lower and upper bounds for $\alpha_{0:t}(n^c)$ are given by:*

$$\alpha_{0:t}(n^p) - \frac{t-1}{t}\frac{e}{L} \leq \alpha_{0:t}(n^c) \leq \alpha_{0:t}(n^p) + \frac{t-1}{t}\frac{e}{L}$$

Proof. In the lower bound, all e effect variables change their corresponding valuation to match with all states in history except for the parent node, reducing Hamming distance to each state by 1 for each effect e. Parent and child states share all variable valuations except for the e effects, which change valuation from parent to child node. This yields, for all cases where $n' \in n_{0:t}$, $n' \neq n^c$ and $n' \neq n^p$

$$\delta(n^c, n') \geq \frac{1}{L}\Big(H(n^p, n') - e\Big) = \delta(n^p, n') - \frac{e}{L} \tag{3.3}$$

$$\delta(n^c, n^p) = \frac{1}{L}\Big(H(n^p, n^p) + e\Big) = \frac{e}{L} \tag{3.4}$$

We can redefine the average $\alpha_{0:t}(n^c)$ as

$$\alpha_{0:t}(n^c) = \frac{1}{t}\left[\sum_{i=0; n_i \notin \{n^p, n^c\}}^{t} \Big(\delta(n^c, n_i)\Big) + \delta(n^c, n^p)\right] \tag{3.5}$$

Since we define that $n^c = n_t$:

$$\alpha_{0:t}(n^p) = \frac{1}{t}\left[\sum_{i=0; n_i \notin n^p}^{t-1} \Big(\delta(n^p, n_i)\Big) + \delta(n^c, n^p)\right] \tag{3.6}$$

Substituting (3.3) and (3.4) into (3.5), noting that $\sum_{i=0; n_i \notin \{n^p, n^c\}}^{t}(\frac{e}{L}) = (t-1)\frac{e}{L}$, and then substituting (3.6) yields Theorem 3.7.

For the upper bound, we note that it is symmetrical in that in the upper bound all effects e are novel, that is, their variable valuation in n^c has never been observed in $n_{0:t-1}$. Thus we get $\delta(n^c, n') \leq \delta(n^p, n') + \frac{e}{L}$. Following the same procedure yields the upper bound. An extended version of the proof is presented in Appendix B. □

Theorem 3.8. *Given a minimum empirical count distribution $\mu = \mu_{t-1}^{min}(n^c)$, the upper bound $\alpha_{0:t}(n^c)$ with respect to μ is given by:*

$$\alpha_{0:t}(n^c) \leq \alpha_{0:t}(n^p) + \frac{t-1}{t}\frac{e(1-2\mu)}{L}$$

Proof. Since μ is the minimum feature occurrence, acting as a constraint, upper bound occurs when all effects e have occurrence equal to μ, that is, the minimum possible occurrence they are allowed to have. Thus, for $t-1$ nodes $n' \in n_{0:t-1}$, $n' \neq n^p$, we have that

$$\delta(n^c, n') = \frac{1}{L}\Big(H(n^p, n') + e\Big) \text{ a total of } (1-\mu)\cdot(t-1) \text{ times, and}$$

$$\delta(n^c, n') = \frac{1}{L}\Big(H(n^p, n') - e\Big) \text{ a total of } \mu\cdot(t-1) \text{ times.}$$

Proof follows from the steps outlined in the derivation of Theorem 3.7. An extended version of the proof is presented in Appendix B. □

Theorem 3.9. *Lower bound for $\alpha_{0:t}(n^c)$ when $\mu^{min}_{t-1}(n^c) = 0$ is given by:*

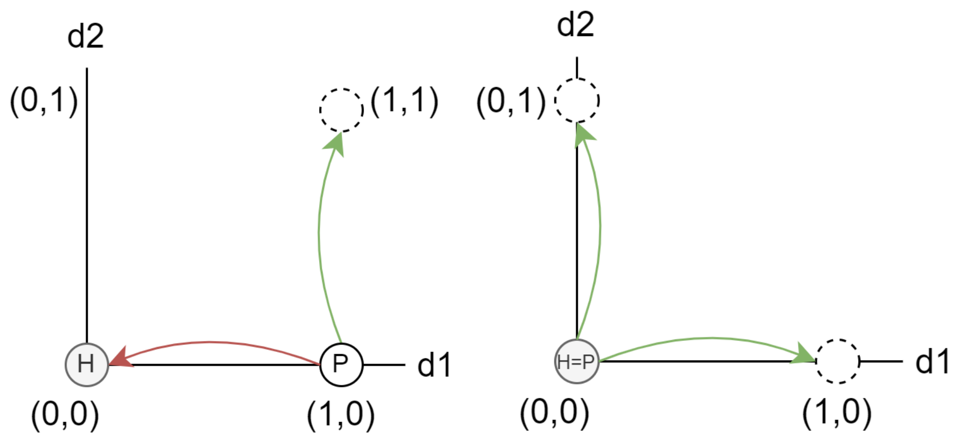
$$\alpha_{0:t}(n^c) \geq \alpha_{0:t}(n^p) - \frac{t-1}{t}\frac{e-2}{L}$$

Proof. In the lower bound, one effect is novel, and $e-1$ effects match all previous history except n^p. Thus,

$$\delta(n^c, n') = \frac{1}{L}\Big(H(n^p, n') - (e-1) + 1\Big)$$

Proof follows from the steps outlined in the derivation of Theorem 3.7. An extended version of the proof is presented in Appendix B. □

A comparison of the bounds in Theorem 3.7 with those in Theorems 3.8 and 3.9 demonstrates the relation between novelty count and Hamming distance through improved bounds, in terms of changes in the average distances from parent to child node, for nodes with a low empirical count N, which acts through the empirical count distribution μ. The upper bound in Theorem 3.8 details the main improvement, signalling greater potential of the child node being located in newer areas of the state space. Theorem 3.9 is a notable special case for novel variable valuations never encountered before, which guarantees an improvement of the lower bound through the novel information that could not have already been observed in the parent node. Through the recursive nature of Theorems 3.7 to 3.9, we also conclude that paths consisting of low count-based novelty nodes are more likely to exhibit rapidly increasing average Hamming distances, thus facilitating a quicker exploration of novel state spaces. We cannot establish an upper bound in relation to the parent state because the least common variable (the one corresponding to the minimum state count) might not be an effect, however modifying count-based novelty metrics to consider effect occurrences could overcome this limitation.



(A) Case 1: Novel parent node. (B) Case 2: Parent node = H.

FIGURE 3.2: Parent-child average Hamming distance example scenarios.

Example 3.2. *We provide a simplified example in Figure 3.2 of the intuition behind our recursive definition of average Hamming distance with respect to a node's parent. H represents the set of $t-1$ nodes in history, except for the parent node P. In these examples, we assume the history is comprised of a set of identical states. In Figure 3.2b, we further assume that the parent node P also has the same underlying state as all nodes in H, as represented by $H = P$ in the figure. The problem is represented as a 2-dimensional state space over binary variables d1 and d2. Available actions from a parent node are represented by the arrows, and in all cases only alter 1 effect variable. The state of child nodes is represented by the destination of the arrows. We assume that a considered child node is the $t+1^{th}$ node generated through selection of one of the actions (arrows) from the parent node (thus the other child node is not generated, and does not contribute to the average distance).*

- *In (A), the average distance of the parent node to all other nodes, including the child node, at time-step $t+1$ is $\frac{1}{t} \times t = 1$. This is apparent from the image. The upper bound of its children's average distance to all nodes is given by $\frac{1}{t}(t+(t-1)) = 1 + \frac{t-1}{t} =$ parent avg. distance $+ \frac{t-1}{t}$. The other child node's distance to all nodes is given by $\frac{1}{t}(t-(t-1)) =$ parent avg. distance $- \frac{t-1}{t} = \frac{1}{t}$. Children nodes in (A) also correspond to the general upper (green arrow) and lower (red arrow) bounds for actions with 1 effect according to Theorem 3.7.*
- *In (B), the parent node's average distance to all nodes is $\frac{1}{t}$, as it only has a Hamming distance of 1 from the only child node. Both possible children have identical*

average Hamming distances of $\frac{1}{t} \times t = 1 =$ parent avg. distance $+ \frac{t-1}{t}$. They both correspond to the general upper bound from Theorem 3.7 (parent avg. distance $+ \frac{t-1}{t}$). However, as its parent in (B) is not novel and has a lower average Hamming distance to nodes in history H, its average Hamming distance of 1 is less than that of the green-arrow child in (A), which is $1 + \frac{t-1}{t}$.

Still, greater Hamming distances alone do not quantify the impact of count-based novelty on search progress. We provide Theorems 3.10 and 3.11 to tie our results to prior theoretical contributions on novelty-based search (see [7, 11, 12]) through an analysis of the expected count of novel tuples of size k (k-tuples).

3.4.2 Estimating Novel k-Tuples

Let history $s_{0:t}$ represent $t+1$ independent and uniformly distributed binary vectors of size L. A tuple is novel if its valuation in $s = s_{t+1}$ was not observed in any state in history $s_{0:t}$.

Theorem 3.10. *The expected number of novel tuples of size k found in $s = s_{t+1}$ given search history $s_{0:t}$ is given by:*

$$\binom{L}{k}\left[1 - (1 - \alpha_{0:t}(s))^k\right]^{t+1} \tag{3.7}$$

Proof. From equation (3.2) we can obtain

$$\begin{aligned} \alpha_{0:t}(s) &= \frac{1}{(t+1)\cdot L}\sum_{j=0}^{t}\sum_{i=0}^{L-1} 1_{s_j^i \neq s^i} \\ &= \mathbb{E}_{s_j \in s_{0:t}, v_i \in V}[\mathbf{1}_{\{s_j(v_i) \neq s(v_i)\}}] \\ &= P(s_j(v_i) \neq s(v_i) \text{ for some } j, i) \end{aligned} \tag{3.8}$$

Thus, the probability that it has the same value becomes $1 - \alpha_{0:t}(s)$, and for a tuple of size k, the probability that any of its constituent variable values is different in s_j than in s is $1 - (1 - \alpha_{0:t}(s))^k$. Calculating the union for a tuple over the full history and multiplying by the number of possible tuples of size k yields the expectation in (3.7). □

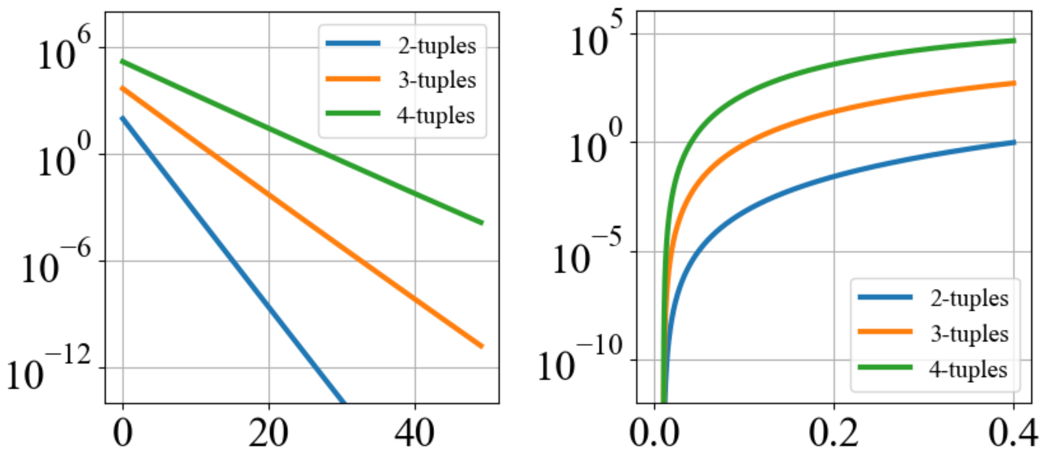
Theorem 3.11. *The expected number of novel tuples of size k found in state $s = s_{t+1}$ given information on occurrence count $N = N_t^v(s)$ for some variable $v \in V$ and search* *history $s_{0:t}$ is given by:*

$$\binom{L-1}{k}\left[1-(1-\beta_{0:t}(s))^k\right]^{t+1}+\binom{L-1}{k-1}\left[1-(1-\beta_{0:t}(s))^{k-1}\right]^N$$

where $\beta_{0:t}(s)$ represents the average normalized Hamming distance after discounting the contribution of variable v:

$$\beta_{0:t}(s)=\frac{\alpha_{0:t}(s)\cdot L-(1-\frac{N}{t+1})}{L-1}$$

Proof. The left-hand side component of the addition is given by equation (3.7) taken over tuples deriving from variables except for the variable v whose empirical count N we observe. The right-hand side component is given by the probability $1-(1-\beta_{0:t}(s))^{k-1}$ that, for some variable x other than v in a k-tuple containing v and in a state s_j where $s_j(v)=s(v)$, $s_j(x)\neq s(x)$. Thus, the tuple's valuation in s_j is different than in s. Taking a union over N states with matching v valuation and multiplying by the total number of tuples in s containing v yields the right-hand side component. Summing the two expectations proves the theorem. $\square$



(A) $\mathbb{E}[\#$ novel k-tuple$]$ vs. N (B) $\mathbb{E}[\#$ novel k-tuple$]$ vs. $\alpha(s)$

FIGURE 3.3: $\mathbb{E}[\#$ novel k-tuple$]$ according to Theorem 3.11. In (A) N is a variable, and in (B) $\alpha(s)$ is a variable. Otherwise, parameters are set as $L=100$, $t=50000$, $N=5$, $\alpha(s)=0.3$. A realistic $\alpha(s)$ value was determined through simulation[1].

[1]We remove a randomly generated root binary vector of size L=100 from the front of an open list, generate 4 children nodes uniformly at random flipping 3 binary variables each, and append each into the open list. We repeated the process to create 10000 nodes, and measured the average Hamming distance of the last 100 nodes, yielding an average value of ≈ 35. The value of α is then obtained by dividing the average distance to other states by the number of variables in each state, $L=100$. We round the value of α down to 0.3 to account for a marginally more pessimistic scenario.

Theorem 3.10 reveals that, without count information, the expectation decreases exponentially with increasing t, rendering the measure effective only for small history sizes. Conversely, Theorem 3.11 introduces a component independent of t and exponential in count number N, emphasizing the crucial role of the minimum count function in identifying states likely to contain novel tuples, necessary to fulfill new action preconditions.

We thus highlight the double role played by count-based novelty in inducing novel tuples, as shown in Figure 3.3: directly, through the greater probability that a novel tuple may contain a variable with low occurrence N, as well as indirectly, through the effect that a low count N induces a greater average Hamming distance compared to history $s_{0:t}$ (Theorems 3.6 to 3.9), which in turn increases the expected number of novel tuples.

The estimation model we provide in Theorem 3.11 is domain-independent, in that it does not account for the differences in structure present in each domain such as correlation among feature values and effects. Still, it provides a valid baseline estimation, and thus valuable guidance for an ordering of nodes in the open list of a search algorithm that promotes an efficient exploratory behavior. We also believe that such structure is more likely to induce a higher correlation among visited variables, and consequently increase the expected probability for novel tuples.

3.5 Comparison with Novelty in Planning

The results in the previous section enable us to relate the exploratory behavior of count-based exploration to that of width-novelty through a common theoretical link pertaining to the discovery of novel tuples. The differences in such tuple discovery process justify the claims made in Section 3.1.

Width-novelty directly distinguishes whether or not a tuple has been observed in its relevant history, but for a width w, is not capable of inferring any information on whether non-novel nodes are more or less likely to contain novel tuples of size $> w$. Our theoretical analysis concludes that a count-based novelty heuristic over tuples of size w, on the other hand, is always capable of inferring nodes that are more likely to contain novel tuples of size $> w$, because it implicitly estimates the expected number of such novel tuples. These findings can then be directly linked to the results of Groß et al. [11], whereby novel tuples enable the discovery of new paths to the goal that were not accessible from previously visited states, thus establishing the necessary theoretical foundation to support count-based novelty's beneficial impact to search performance.

The heuristic thus tackles high-atomic width problems by directly seeking larger size novel tuples. In this regard, we note that the magnitudes over the expected number of novel tuples estimated in Figure 3.3a are realistic and useful, especially if we consider that novelty search algorithms are often capable of generating $100,000$s of nodes in seconds. Still, for problems that induce high counts N, the expected probability of encountering novel tuples still decreases exponentially in N.

While this can quickly reduce the probability of encountering novel small tuples $-$ mostly tuples of size 1, 2 or 3 $-$ we note that not only do larger size novel tuples exhibit higher expected probability of occurrence, but also have this probability decay less rapidly as N increases. Encountering novel tuples of greater size can then promote a positive cycle through the greater Hamming distance of the node (Fig. 3.3b, Theorems 3.6-3.9), that increases the expected probability for small-size tuples. We also argue that in such problems, if the minimum count over tuples in states increases rapidly, then available novel tuples of smaller sizes are quickly exhausted, and thus it must be the larger-size feature conjunctions that provide the breakthrough for further search progress. This justifies the continued informedness of count-based novelty even as the count-based heuristic value grows higher.

The effectiveness of a linear-size count-based novelty heuristic, that only records the occurrence of individual features (size-1 tuples) in a state then follows from our analysis. Such a heuristic implies an $O(V)$ time and memory complexity under all scenarios, where $V = F$ is the set of variables in a planning problem. Importantly, this heuristic is then more robust to the underlying characteristics of a planning domain than width-novelty. Through the measure of variable counts, it adapts dynamically to the domain, without the need to set a maximum width w to tackle a problem, or iteratively increase the hyperparameter value to search for the ideal value for w.

We have only provided analysis on the case where we measure occurrence counts over tuples of size 1, which aligns with our experimental evaluation of the technique, however we are confident that the presented conclusions generalize to counts over larger size tuples. These simply increase the maximum tuple size at which the count-based metric switches from being certain of observing novel tuples, to implicitly estimating their expected probability of being novel. As with width-novelty, this comes at the cost of an exponential increase in number of evaluated tuples, and thus time and memory complexity.

3.6 Connections to Count-Based Exploration

3.6.1 RL Setting Notation

We provide an adaptation for use in the reinforcement learning setting we explore in this section of the notation adopted in Sections 3.3 and 3.4, which simplifies comparisons with Bellemare et al. [53]'s work on pseudocounts. In this setting, $s_{1:t} \in S^t$ represents a sequence of t states from a countable state space S where S^* is the set of finite sequences of states – thus extending the binary representation from previous sections. We label the first state in the sequence s_1 to align with RL notation convention, and to differentiate from the notation in previous sections where the root node of a planning problem was represented as s_0. $s_{1:t} \cdot s$ represents the concatenation of sequence $s_{1:t}$ with a new state s. Let $\rho_t(s) := \rho(s; s_{1:t}) \equiv \rho : s^* \times s \rightarrow [0, 1]$ be a *density model* that provides a probability distribution for each sequence $s_{1:t} \in S^t$. The *recoding probability* of a state s is then $\rho'_t(s) := \rho(s; s_{1:t} \cdot s)$, that is, the probability assigned to s after observing its occurrence. The density models used with pseudocounts assume that states are independently distributed.

Let $N_t(s) := N(s, s_{1:t})$ be the number of occurrences of a state s in the sequence $s_{1:t}$. Bellemare et al. [53] define the *empirical distribution* μ_t:

$$\mu_t(s) := \mu(s; s_{1:t}) := \frac{N_t(s)}{t}$$

An important fact from [53] for our analysis is then:

$$\text{If } \rho_t = \mu_t \text{, then the pseudocount } \hat{N}_t = N_t. \tag{3.9}$$

3.6.2 Minimum Empirical Count Distribution

We further reiterate previous notation from Section 3.3 for clarity. We use V to define the set of variables in states $s \in S$, and $U^{(k)} = \{X \subseteq V \mid |X| = k\}$ the set of all k-element variable conjunctions, with tuple $u \in U^{(k)}$ representing a conjunction of k variables. $s(u)$ denotes the valuation of variables in tuple u in state s, and $N_t^u(s_{t+1}) := N^u(s_{t+1}; s_{1:t})$ counts the number of states $s_i \in s_{1:t}$ where $s_i(u) = s_{t+1}(u)$. We can adapt the *tuple empirical count distribution* for tuple u from Equation 3.1 to this setting, and define it as:

$$\mu_t^u(s) = \frac{N_t^u(s)}{t}$$

noting that the denominator changes to t, as it enumerates the sequence of states starting from s_1 rather than s_0.

Given a set of tuples $U = U^{(k)}$ over features V of states in S for some tuple size k, we also adapt the *minimum empirical count distribution*:

$$\mu_t^{min}(s) = \min_{u \in U}(\mu_t^u(s))$$

This definition is a generalization of Equation 3.1 over possible tuple sets $U^{(k)}$, and countable features. It is a valid density model for a pseudocount measure, as it satisfies the assumption that states are independently distributed, as well as being *learning positive*, which is an assumption over density models used in much of Bellemare et al. [53]'s analysis:

Definition 3.12. A density model ρ is learning-positive if for all $s_{1:n} \in S^n$ and all $s \in S$, $\rho'_n(s) \geq \rho_n(s)$ [53].

3.6.3 Connection to Pseudocounts

We can then provide the link between count-based novelty and pseudocounts.

Proposition 3.13. *Given a tuple size k and an underlying set of tuples $U = U^{(k)}$ over features V of states in S, if $\rho_t = \mu_t^{min}(s)$, then the pseudocount $\hat{N}_t = \min_{u \in U}(N_t^u(s_{t+1}))$.*

Proof. Proof of the proposition follows from Fact 3.9 and from the pseudocount function definition in Equation 2.1. An explicit derivation is provided in Appendix B. □

As with the minimum empirical count distribution defined in this section, the pseudocount in Proposition 3.13 represents a more general definition of the count formula in the right-hand side of Proposition 3.5 over possible tuple sets $U^{(k)}$ and countable features, with equivalence to Proposition 3.5 when applied to binary features and $U = V$. In a given binary state space, it thus achieves an explicit link between the analysis on the effect of count-based novelty value of a state on classical planning exploration we provide in Section 3.4, and existing work on the exploratory behavior induced by a pseudocount exploration bonus in tabular and MDP RL settings.

The binary space requirement is an effect of the theoretical analysis in Section 3.4 being performed over a representation of STRIPS Classical Planning problems as binary feature vectors, and is stricter than a countable numerical discrete range, commonly used to define MDP environments. Still, non-trivial environments can also be represented through binary vectors, as for example bit-wise inputs, which are often simplified to produce binary features for white and black pixels in a screen to obtain a more efficient state-space representation. Finite discrete countable features may also be transformed into a binary state-space representation which, while possibly inefficient for practical implementation, may be used to study theoretical links across the two fields.

Additional parallels between Classical Planning and RL strengthen the relevance of the presented connection between planning and RL exploration. The sequences of states visited and transitions performed across multiple episodes of RL training in an MDP or tabular environment with deterministic actions can build a planning search tree containing all visited states, conceptually similar to a Monte Carlo Tree Search over full training episodes. Moreover, useful high-level features in such settings can be defined using tuples of feature values. Bridging knowledge from Novelty and Classical Planning, where domains often have explicit definitions of atoms and goals, and work exists on uncovering favorable atom conjunctions to find a goal, may aid the study of exploration techniques concerning high-level information in less structured MDPs.

Chapter 4

Trimmed Open List

In this chapter, we introduce the *trimmed open list* algorithmic contribution. Section 4.1 explains the need for this solution and why traditional open lists are inadequate for our use case. In Section 4.2, we present the *single trimmed open list* and *double trimmed open list* techniques, which we implement in our solvers as described in Chapter 5. Finally, in Section 4.3, we analyze how these techniques address the limitations of traditional open lists and provide an intuitive explanation of their workings.

4.1 Motivation

Balancing the amount of memory occupied by low-rank nodes is a common strategy which allows for better ranked exploratory nodes to appear further down the search.

A common implementation technique is that of *lazily* generating nodes. This solution involves not storing the description of a node's state and only generating it later at expansion time from its parent node's state representation and action. Doing so prevents memory being occupied by the node's state representation, reducing the overall memory footprint of nodes in the open list. However, even when lazily generated, the large number of nodes that populate the open list can end up consuming a significant portion of a search algorithm's overall memory footprint. Furthermore, lazy generation may require the computation of a node's underlying state twice, once at generation time to evaluate heuristic values, and another time at expansion for the lazy evaluation. This introduces redundant computations, thus trading time for memory. Nonetheless, this time overhead is often relatively small, and lazy evaluation is a common technique adopted by many modern planners, including those presented in this work.

Another effective technique relevant to novelty planners is node pruning. Polynomial width-novelty planners [10, 14] prune nodes with a novelty value greater than a preset threshold, usually the maximum width of the novelty search algorithm, as they are deemed not useful for the search. As discussed in Section 2.4, this is key to achieving the polynomial bounds of such search algorithms, as theoretical limits to the number of novel nodes guarantees that the algorithm either finds a solution, or fails in polynomial time. Another key benefit, relevant to the technique presented in this section, is that pruning a node frees up the memory which it would have otherwise occupied in the open list. This allows the algorithm to insert fewer nodes in the open list, and in many cases significantly reduce its overall memory footprint. Pruning nodes that are deemed to not be useful to the search thus provides a more targeted approach, which balances a lower memory usage with the ability to preserve nodes which are likely to aid the efficient exploration of the state space. Doing so is especially useful for novelty planners, because the low computational overhead of novelty and partition heuristics adopted in algorithms such as BFWS [8] (described in Section 2.5.1.1) generates larger open lists that thus consume a larger portion of the available memory over time. Pruning non-novel nodes can thus help solve instances which would otherwise hit preset memory limits in computational environments with constrained resources.

Open list control [10], briefly introduced in Section 2.5.2, further restricts the set of visited nodes which make their way to the open list by adaptively controlling the rate of growth of states in the open list. This is beneficial to the search, since at higher widths of 3 or 4, even novel nodes end up being too numerous and consuming too much space in the open list. Through the formulation of an optimal control problem, open list control can therefore prune additional nodes of different novelty value, further reducing the memory footprint of the open list compared to a basic polynomial novelty algorithm.

As with non-novel nodes in novelty search algorithms, count-based novelty heuristics generate multiple nodes which may be considered less useful to the search at a given point in time. These are nodes which have very high counts relative to that of the "best" nodes in the open list that are currently being expanded. However, adopting a predefined threshold for pruning nodes as in the width-novelty case is unfeasible due to the granularity of the metric.

This trait is shared by best-first search algorithms in general. In large search problems, where a complete search is unfeasible, heuristic best-first search algorithms visit and evaluate a large number of nodes with a "bad" heuristic value – a high estimate of their distance to the goal – which never get expanded, as better newly generated nodes are prioritized for expansion in the open list. This issue is exacerbated if the algorithm makes meaningful search progress, thus visiting and expanding nodes that are closer to the goal, and continually generating new nodes which are themselves closer to the goal. The result is a large number of "deadweight" nodes that, like stale water, remain stuck at the bottom of a priority heap open list, never moving up as newly generated nodes continually overtake them.

Addressing this issue, however, can be challenging, as defining an exact boundary between nodes which will and will not get expanded is unfeasible without future knowledge. Indeed, the search algorithm may get stuck in local minima of the state space topology induced by a heuristic measure, only generating bad states without finding new states that improve the heuristic value of previously generated nodes. In such a case, old nodes in the open list which were previously "stale" may start moving towards the top, providing new search directions for the algorithm to overcome the local minima.

The *trimmed open list* addresses this challenge by providing an open list which limits its size to a maximum pre-defined value to constrain its maximum memory usage, whilst still allowing nodes with "good" heuristic values to not be pruned. Intuitively, the goal is to "trim" the bottom layers of a binary heap while minimizing the impact that this has on its normal order of node expansion. This is achieved dynamically, without the need to maintain an explicit threshold, but rather by sampling from an implicit distribution induced by the nodes already in the open list, avoiding an increase in the asymptotic time complexity of the insertion procedure. Furthermore, the proposed algorithm can be implemented through a simple modification of a conventional binary heap.

4.2 Trimmed Open List

4.2.1 Single Trimmed Open List

Built on a binary heap, the trimmed open list inserts new nodes like a regular binary heap when its size is below the predefined maximum size threshold L, limiting its growth by pruning less promising nodes when it exceeds L. This pruning process involves randomly selecting a leaf node, comparing its heuristic value with the candidate for insertion n using the open list's comparison function, and then pruning or swapping nodes based on their heuristic values. A unique heapify-up operation is applied to the newly positioned leaf, which, unlike standard heaps, is not required to be the last element. This process is described in detail in Algorithm 2.

This process relies on a binary heap's property of pushing less preferred nodes towards the bottom of the binary tree. At each level, there may still be nodes whose value is less than that of nodes in levels above, but not their parent node; however the mean node value in each level of a *full* complete binary tree − that is, with also the last layer completely filled − must be greater than or equal to that of all levels above (open lists are min-heaps as lower heuristic estimates are preferred). Finding the absolute minimum node value in the heap would require checking among all leaf values − corresponding to half+1 of all nodes in a complete tree − thus requiring an $O(n)$ time complexity for n nodes in the binary heap. By relying on the aforementioned property of binary heaps, we can instead simply add an $O(1)$ time procedure to the insertion process of the binary heap, which allows us to sample a threshold value estimate for insertion of a new node. The insertion procedure also preserves the binary heap property, as a newly inserted node may only substitute a leaf node in the binary heap, with no children, and it then recursively swaps with its parent only if it has a better value.

4.2.2 Double Trimmed Open List

We also developed a *double trimmed open list* for heuristic alternation [43], accommodating dual open lists for node insertion under distinct heuristics and enabling alternate node retrieval. Newly generated nodes are candidates for insertion to each of two open lists, and are separately evaluated for insertion in each open list employing the same

Algorithm 2 Trimmed Open List

```
procedure TRIMMED OPEN LIST(new node $N$, heap $H$, heap size limit $L$)
    $S \leftarrow$ size of $H$
    if $S < L$ then                                        ▷ If heap hasn't reached size limit $L$
        insert $N$ into $H$
        heapify-up($H$)                                    ▷ Reorder last element
    else
        $i \leftarrow$ uniformly random leaf index of $H$
        $O \leftarrow H[i]$                                ▷ Node at random leaf index
        if $N$ has a better heuristic value than $O$ then
            $H[i] \leftarrow N$                            ▷ Replace $O$ with $N$
            heapify-up($H, i$)                             ▷ Reorder element at index $i$
            discard $O$
        else
            discard $N$
```

pruning strategy as the single trimmed open list, but using each open list's heuristic evaluation function for node comparison.

This variant further distinguishes itself by tracking each node's interaction with either open list. An interaction refers to a node being *popped* from the top of the open list − thus that node being expanded − or the node being *trimmed* from the bottom, which occurs when it is pruned right away, or after a subsequent comparison and swap with another node generated at a later time step. A node becomes eligible for deletion when its interaction count equals the number of lists it is associated with, provided it is not in the closed list. This ensures a node is removed only when it is confirmed to be redundant, safeguarding against premature deletion crucial for the lazy expansion of successors. It also removes the need to generate two separate nodes for each state inserted in both open lists, allowing for a total memory footprint of both open lists that is less than or equal to that of two single trimmed open lists.

These techniques are further generalizable to k-trimmed open list variants with minor modifications, where k is the number of open lists. For values of k greater than 2, the insertion procedure is identical, but over k lists rather than just 2. Similarly, nodes may be safely deleted when their interaction count equals k and they are not already in the closed list. While this study does not implement any variant of trimmed open lists with $k > 2$, this trait of the algorithm enhances its versatility, broadening its applicability across a wider range of use cases and search algorithms.

4.3 Trimmed Open List Analysis

The single trimmed open list variant thus achieves a space complexity which is linear in the average size of nodes in the open list, which is multiplied by a constant factor given by the maximum size of the open list L. Similarly, the double trimmed open list variant has a space complexity which is less than or equal to twice that of an equivalent single trimmed open list. The insertion procedure of new nodes simply adds a constant time operation – the comparison and, if necessary, swap and deletion of a node if the open list has reached its maximum size – that therefore enables it to maintain the $O(\log n)$ time complexity of a regular binary heap. Technically, the overhead actually becomes $O(1)$, as the maximum size of the heap limits its maximum depth and, thus, the complexity of insertion; still, the benefit of this is marginal in practice. Finally, the *find-min* procedure also does not encounter any relevant additional overhead in the double trimmed open list compared to the single variant, simply alternating among open lists.

The effectiveness of this solution therefore derives from its ability to obtain valid estimates for a threshold value for insertion of new nodes, without worsening the underlying asymptotic complexity of the binary heap. In this regard, the entire last layer of the heap is used as a "barrier to entry" for newly inserted nodes, to allow good nodes to go in but bad nodes to stay out. The set of all nodes in the leafs at any point in time induces an implicit distribution given by their heuristic values. Due to the binary heap properties, the heuristic value of each node in this set must be less than or equal to that of all its ancestors in the heap. For a trimmed open list with a realistic maximum size of $L = 2^{20}$, this corresponds to 19 ancestors. Furthermore, as mentioned earlier in the section, the mean of heuristic values of nodes in the last layer – the mean of the implicit distribution – must be greater than or equal to the mean of all layers above. Selecting a node at random from the leafs and using its heuristic value then corresponds to sampling from this distribution. These facts thus warrant that we sample from a distribution whose values, in expectation, are greater than that of all other distributions induced by the above layers of the heap, pushing the sampled heuristic values *up*. On the other hand, the process of comparing and swapping leaf nodes with "better" newly generated nodes pushes the sampled heuristic values *down*, improving the overall mean. This balance achieves a threshold distribution which can automatically adapt to the heuristic values of nodes inside the open list, and to that of nodes being generated.

Another important factor resides in the open list's alternation between trimming inserted nodes, when it is full, and performing a "normal" heap insertion when it is not full. Every time the open list expands a node, it frees up one space which will then be filled by the first newly generated node, irrespective of its heuristic value, increasing the chances that it may have a value much lower to that of most nodes in the open list. This lowers the overall mean of the distribution. This event does not constitute a problem, as the main goal of this barrier is to allow good nodes to pass through. As in any min-heap, any node that gets added must still be the best in the heap in order to be expanded. Bad nodes are thus likely to simply linger in the last layer until they get trimmed again later in the search.

A meaningful parallel between trimmed open lists and novelty pruning may be drawn. As with novelty pruning, this solution prunes nodes which are deemed not useful to the search. Unlike novelty pruning, however, it does not rely on a fixed threshold, but rather it dynamically adapts to the values of nodes in upper layers of the open list. If a search algorithm gets "stuck" without finding new low-heuristic-value nodes then, as the average heuristic value of nodes in the open list increases, so does the mean of the implicit distribution in the open list's boundary layer. When the search then progresses, finding a large number of states with very low heuristic values, a higher-heuristic-value state gets trimmed from the bottom every time a low-heuristic-value state gets inserted into the open list, lowering the distribution's mean again.

Still, novelty pruning is also used to induce a width-w polynomial search, whereas adopting a trimmed open list loses such polynomial guarantees. The trimmed open list also produces a search which is technically not complete, as the trimming process does not guarantee that it will eventually visit every possible state in the state space. Overall, however, the significant reduction in memory footprint brought by the constant memory complexity of a trimmed open list with a suitable maximum size L can outweigh these limitations, and improve overall performance significantly in satisficing domains, as backed by experimental results in Section 7.2. Furthermore, we discuss in Section 5.2.1 and demonstrate in Section 7.3 that implementing a memory limit as a cutoff for halting the planner is an effective alternative strategy to a polynomially-bounded search, mitigating the impact of these limitations.

Chapter 5

Count-Based Search Algorithms

This chapter introduces the count-based solvers proposed and evaluated in this thesis, and is divided into two main sections. Section 5.1 presents the BFWS$_t$, BFCS$_t$, and BFNoS$_t$ variants of the BFWS solver family [8], and specific implementations using techniques from Chapters 3 and 4. Section 5.2 motivates our adoption of *memory thresholds*, and explains how we integrate our solvers from Section 5.1 with memory and time thresholds to create hybrid planners employing a *dual strategy* alongside classical planners from previous literature.

5.1 Count-Based Solvers

We define three new planning solvers to evaluate the performance of our proposed trimmed open list and classical partitioned count-based novelty techniques. All our solvers are based on and extend the BFWS(f_5) search algorithm [8] (introduced in Section 2.5.1.1). Reiterating the main features of BFWS(f_5), the Best First Width Search family of solvers represents best-first search algorithms that adopt a novelty heuristic as the primary tie-breaking criterion for nodes in the open list. Consequently, these solvers prioritize exploration through goal-agnostic heuristics over the greedier approach of traditional solvers that use goal-aware primary heuristics. This strategy seeks to overcome the potential uninformedness of goal-aware heuristics at various stages of the search process. f_5 represents the evaluation function $\langle w, \#g \rangle$, where the primary heuristic w is the novelty measure and the secondary heuristic is given by the goal counter $\#g$, which counts the number of atomic goals not true in s. The novelty measure w in BFWS(f_5) is a partitioned novelty heuristic $w_{\langle \#g, \#r \rangle}$, that is computed given partition functions $\#g$ and $\#r(s)$ (described in Section 2.5.1.1).

The solvers proposed in this section also adopt the f_5 function. In this study, we extend the w component to refer to both width-novelty as well as our proposed count-based novelty metric, thus adopting the notation $f_5(X)$ to represent an f_5 function heuristic where X is the chosen novelty measure w. W_x is used to denote the partitioned width-novelty metric $w_{\langle \#g, \#r \rangle}$ with max-width $= x$ [8]. Furthermore, all proposed count-based solvers adopt the C_1 count-based novelty heuristic, where $C_1 = c^V_{\langle \#g, \#r \rangle}$ is the partitioned classical count-based novelty metric over size-1 features $v \in V$ with partition functions $\langle \#g, \#r \rangle$, according to Definition 3.2 in Section 3.3.

In terms of computational complexity, the C_1 heuristic only operates over single-atom tuples, rather than size 1 and 2 tuples checked by W_2. Each partition therefore only needs to store $|V|$ elements, as opposed to $|V|^2$ elements stored by W_2. An important consideration is that W_2 still has an $O(V)$ time complexity, as it can operate by only checking the tuples that have changed from one state to another [5]. C_1, therefore, does not provide an asymptotic improvement in time complexity compared to W_2. On the other hand, it does guarantee a lower space complexity compared to the W_2 heuristic. In practice, the binary checking of W_2 enables it to be implemented through *bit-sets*, which only adopt a single bit of space to check occurrence of a tuple, as opposed to C_1 which relies on integer counts, which may have a size of 16, 32 or 64 bits depending on implementation. Still, the quadratic space blowup of W_2 is often larger than the constant space factor increase of C_1 in satisficing planning problems, with C_1 achieving significant memory usage improvement in problems with a relatively large number of variables $|V|$.

Proposed algorithms also adopt single and double trimmed open list variants (Chapter 4) in place of conventional open lists implemented as min-heaps, to address the blowup in open list memory usage created by the quick rate of state visitation of BFWS variants. Solvers proposed for experimental evaluation include $BFWS_t(f_5(W_2))$, $BFCS_t(f_5(C_1))$, and $BFNoS_t(f_5(C_1), f_5(W_2))$. The trimmed open list is capped at a constant depth $D = 18$ (maximum size $L = 524,287$) determined through empirical testing. We note that empirically, small changes in depth $-$ $D = 17$ or $D = 19$ $-$ did not alter coverage beyond the variance in the results.

- BFWS$_t(f_5(W_2))$: A standard non-polynomial variant of the BFWS(f_5) search algorithm as defined in previous literature [8] with parameter *max-width* $= 2$. The only modification carried out on this solver is represented by the subscript t, which denotes the use of a single trimmed open list instead of a conventional open list. The main purpose of this solver is to provide an analysis of the impact that the trimmed open list alone has on the underlying BFWS(f_5) search algorithm, all else equal, but also to provide a benchmark to asses the performance of count-based solvers $BFCS_t(f_5(C_1))$ and $BFNoS_t(f_5(C_1), f_5(W_2))$.

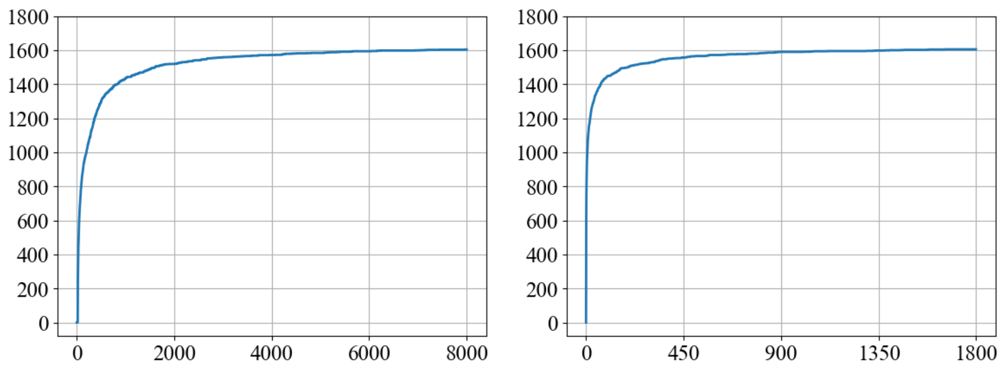
- BFCS$_t(f_5(C_1))$: A BFWS$_t(f_5)$ solver, thus also adopting a single trimmed open list, where the main feature consists of substituting the W_2 width-novelty heuristic with $w = C_1$. The search algorithm allows for a direct comparison of the differences in performance achieved across planning domains by simply substituting the traditional width-novelty heuristic in BFWS(f_5) with the C_1 count-based variant proposed. The name is changed to BFCS (Best First Count Search) to reflect the fact that the count-based heuristic is technically not a width-based heuristic. It may be considered a *width*-1+ search, as it induces a search with a width of at least 1, but it cannot truly perform a width 2 or above search, and it reverts to a simple W_1 heuristic if it performs only a width-1 search, thus it is not classified as such.

- BFNoS$_t(f_5(C_1), f_5(W_2))$: A best first novelty search solver that adopts a double trimmed open list for open list alternation [43]. It employs the evaluation function $f_5(C_1)$, that is with $w = C_1$, for first open list and evaluation function $f_5(W_2)$, with $w = W_2$, for the second open list, alternating node selection between the two lists during the expansion phase. The goal of this solver is to capitalize on the differences between the C_1 and W_2 heuristics. The C_1 heuristic, as discussed in Section 3.4, is effective at estimating states containing tuples of size greater than 2, achieving a consistent level of informedness without running out of "useful" nodes. However, it is incapable of directly measuring the presence of 2-tuples, which W_2 does. Previous work has shown that the step from W_1 to W_2 significantly improves solver coverage across benchmark problems [5, 8, 14], thus suggesting that adding an additional open list which may directly target 2-tuples may complement the effectiveness of the C_1 heuristic. Likewise C_1 can complement W_2 by primarily directing the search when W_2 runs out of nodes with a novelty value of 1 or 2. The solver variant is called BFNoS (Best First Novelty Search), as it blends distinct novelty heuristics – a count-based variant, and a width-based variant. 'BFNoS' is also directly used to abbreviate the $\text{BFNoS}_t(f_5(C_1), f_5(W_2))$ solver in Chapters 5, 6, and 7, as it is the only such variant in these chapters.

5.2 Hybrid Solvers

5.2.1 Memory Thresholds

Solvers tend to exhibit diminishing returns in the number of instances solved with respect to both memory and time (Figs. 5.1a, 5.1b). Thus, as memory usage increases while solving an instance, it is more likely that the instance will not be solvable within the defined memory constraint, indicating that the adopted heuristics are not effective for that particular problem.



(A) Instance coverage vs. memory usage (MB).

(B) Instance coverage vs. time (sec).

FIGURE 5.1: Instance coverage vs. (A) memory usage, and (B) time, for solver BFNoS_t $(f_5(C_1), f_5(W_2))$. The curves show the diminishing returns on instances covered with respect to time and memory. Measurements were conducted over the set of benchmark planning problems described in Section 7.1.

Novelty heuristics W_2 and C_1 provide a quick evaluation of state novelty which, in combination with efficient partition functions, allow solvers to expand considerably more nodes per unit time than counterparts adopting more informed but expensive heuristics such as h_{ff} [15], albeit at a greater memory cost. In other words, they can reach the "flatter" region of the coverage-memory relation more quickly. We leverage this trait to introduce dual configuration planners in which the frontend seeks to prioritize coverage, but also fail as quickly as possible when memory usage reaches those flatter areas, accounting for different domains' characteristics with respect to memory usage. Experimental results in Section 7.3 provide empirical analysis of the technique, validating the benefits of the approach, and justify the selection of count-based novelty solvers presented in Section 5.1 as ideal frontend candidates under this strategy.

5.2.2 Hybrid Count-Based Solvers

Memory thresholds also raise a comparison with polynomial width-novelty planners, which fail in polynomial time if the search exhausts all novel nodes. This trait can be useful to avoid wasting too many resources on a solver whose heuristics may be uninformed in a particular domain or instance, and potentially switch to an alternative solver which may have a better chance at solving the problem. Many successful solvers such as FF, Probe or Dual-BFWS rely on such dual strategy [8, 15, 16], with the frontend of such solvers playing a key role in their performance. Dual-BFWS in particular – introduced in Section 2.5.1.1 – relies on a polynomial width-1 BFWS frontend solver designed to solve the problem or fail very quickly, for the backend to then take over.

While the use of memory thresholds does not provide polynomial time guarantees, it does have benefits compared to the more typical time thresholds adopted in multiple state-of-the-art solvers [20, 21]. In particular, it adapts to the underlying characteristics of the domain with respect to its memory usage, allowing problems with a high memory usage to fail quickly when the memory-coverage relation in Figure 5.1a implies a low probability of solving the instance, but also allocating a significant longer running time for instances showing less memory usage, and thus a higher chance of being solved. This allows it to provide some degree of balance between the time provided to frontend and backend solvers, as opposed to a one-size-fits-all constraint of time thresholds.

We adopt BFNoS$_t(f_5(C_1), f_5(W_2))$ as a frontend solver, capped by a 6 GB memory threshold and a time threshold close to the overall time limit. The additional time threshold is included to enable backend fallback for all unresolved searches. This is because the solver also exhibits diminishing returns to time (Fig. 5.1b), indicating a lower probability of finding a solution when reaching the flatter portion of the the coverage-time relation. Adopting a combination of memory and time thresholds therefore allows the frontend solver to switch whenever the earliest of the two thresholds is hit, maximizing the time provided to the backend solver. The time threshold also ensures that all problems fallback to the backend solver eventually, albeit only reserving a relatively small fraction of the total time allowance for its execution.

We pair the frontend solver with three backend planners from literature: the Dual-BFWS backend component (*BFNoS-Dual*) [8], LAMA-First (*BFNoS-LAMA*) [19], and the "first" version of Scorpion-Maidu (*BFNoS-Maidu-h^2*), in its IPC2023 configuration with the *h^2-preprocessor* [20, 44, 63]. These backend solvers were chosen for their complementary heuristics to our frontend's f_5 partitioning, promoting diverse solution strategies to enhance coverage diversity, and also for the influential status of all selected backend planner in modern planning literature. In the BFNoS-Maidu-h^2, we allow the use of a preprocessor as it is part of a "pre-packaged" backend planner; the dual solver thus adopts a peculiar frontend-preprocessor-backend tactic, that filters the problems that reach the preprocessor, and provides an additional layer of orthogonality between the underlying traits of the frontend and backend solvers.

Frontend time thresholds are set to 1600 *sec* with BFNoS-Dual and BFNoS-LAMA, and 1400 *sec* for BFNoS-Maidu-h^2, to account for up to 180 *sec* of preprocessing allowance determined by its IPC2023 parameter settings. The threshold values were selected empirically and by cross-referencing Figure 5.1. This approach was also influenced by the excessive time and resource requirements of performing a more formal hyperparameter tuning process. As further discussed in Section 7.1, experiments with planners adopting trimmed open lists (the BFNoS frontend) have to be repeated with multiple random seeds to obtain a more accurate estimate of coverage performance. Furthermore, a less thorough hyperparameter search may reduce the risk of overfitting our results to the set of available benchmark domains, as the goal of satisficing solvers is ultimately to perform well across all possible domains. This reasoning also motivates the selection of the maximum depth of trimmed open lists adopted by all proposed frontend models.

Chapter 6

Implementation Details

In this chapter, we provide important details on the implementation of the solvers and techniques presented in previous chapters. Sections 6.1 and 6.2 offer an overview of the implementation of the main components of the count-based solvers and hybrid solvers introduced in Sections 5.1 and 5.2, respectively. These sections also detail the external libraries and solvers from existing literature adopted in our hybrid planners. In Section 6.3, we further provide an overview of the implementation of the experimental portion of our study. The solvers proposed and implemented in this study are accessible through our GitHub repository [64].

6.1 Implementation of Count-Based Solvers

Proposed count-based BFWS variants – BFWS$_t(f_5(W_2))$, BFCS$_t(f_5(C_1))$, and BFNoS$_t(f_5(C_1), f_5(W_2))$ – were fully implemented using the "Devel2.0" branch of the LAPKT planning library [65]. LAPKT (Lightweight Automated Planning Toolkit) is an open-source planning library designed to facilitate implementation, testing and benchmarking of planning algorithms. The framework is designed to be modular and extensible, and already includes multiple implementations for novelty planners and components, as well as planning benchmarks written in PDDL [66]. It uses a C++ backend and Python API.

The implementation of proposed solvers adopted the LAPKT BFWS planning engine as basis for further modifications. The main new modules which have been implemented into LAPKT are the count-based heuristic C_1, and the single and double open lists. The planning engine has been adapted for each variant for integration with its respective components.

6.1.1 Planning Engine

The planning engine represents the core component of the solver, that performs the main planning loop of expanding a node, generating successors, evaluating heuristics and inserting successors into the open list, and checking for goal conditions. $\mathrm{BFWS}_t(f_5(W_2))$ adopts an equivalent planning engine to $\mathrm{BFWS}(f_5(W_2))$, as the main difference is the substitution of the open list, which can be abstracted in its implementation. The $\mathrm{BFCS}_t(f_5(C_1))$ engine is also equivalent, as the C_1 heuristic can also substitute the W_2 heuristic without further modifications.

The $\mathrm{BFNoS}_t(f_5(C_1), f_5(W_2))$ on the other hand requires the addition and evaluation of an additional heuristic, as it adopts both the C_1 and W_2 heuristics. Both open lists adopt a $\#g$ secondary heuristic, allowing it to be utilized for both open lists. The other main modification involves a check for determining which expanded nodes may be deleted. Expanded nodes are inserted into the closed list; however, if the state of the node is already present in the closed list, it is identified as a duplicate. With a single open list, this duplicate node may be deleted. The double trimmed open list complicates this step, as the algorithm must ensure that this node is not present in any of the available open lists. This is done through checks of their interaction count, as described in Section 4.2.2, when the node is normally checked for presence in the closed list. Another subtlety is that the state of these nodes, which is lazily generated, can be deleted if the node is in another closed list and the state is a duplicate of another node already present in the closed list. This saves additional space, and the node which is still present in the closed list does not need its state to be generated as it may be directly marked as a duplicate. It may then be deleted when expanded again without need to spend time lazily re-generating its state.

The measurement of memory usage for comparison with the memory threshold in hybrid solver implementations is also integrated into the BFNoS engine. This measurement is periodically evaluated during the node generation phase, typically once every 10,000 nodes generated. Such an interval is sufficient because each node adds only a small amount of memory, making larger steps suitable while providing an almost negligible time overhead. The operation is performed at node generation since the average number of nodes generated per expanded node – the branching factor of the problem – can vary significantly across different domains. Measuring memory usage at node expansion would therefore result in more inconsistent readings. Maximum memory usage is measured using the maximum resident set size given by the `ru_maxrss` member of the `struct rusage` object from the `sys/resource.h` C++ library.

6.1.2 State Nodes

State nodes represent the actual node objects that contain the state representation of a node, as well as heuristic values of the nodes and additional information. Main modifications to node implementation include the addition of a variable to store the second primary heuristic value in BFNoS solvers, as well as additional information for the double trimmed open list to use to determine whether or not to delete a node and free its memory. This information is composed of a counter variable that records the node's interaction count (mentioned in Section 4.2.2), as well as a duplicate-marking variable that marks nodes expanded by one open list, but still present in another open list.

6.1.3 Count-Based Novelty Heuristic

The C_1 partitioned count-based novelty heuristic implementation is derived from the existing implementation of W_1 partitioned novelty heuristic in LAPKT. The existing implementation already efficiently checks for the presence of size-1 tuples for a given state and partition. The main modification involves replacing binary occurrence checks with the extraction and increment of integer values for the counts. An important implementation detail in this regard is the choice to use a double nested C++ vector for storing partition count information, instead of alternatives like an unordered map. This tradeoff was chosen due to the tight time complexity requirements of the heuristic, which is often evaluated millions of times during the solving of a single satisficing planning instance. The inner vectors represent variable counts for a single partition. New partitions, and thus inner vectors, are added as the search algorithm extends the horizon of "best" reached partitions. With the nested vector implementation, space is pre-allocated for all possible tuples in each partition vector. Although an unordered map could save memory by only allocating space for seen tuples, empirical testing showed that the memory-for-time tradeoff of the nested vector implementation benefited overall planner performance.

6.1.4 Trimmed Open List

Details of the single and double trimmed open list algorithms are described in Section 4.2.1. The implementation in LAPKT modifies its standard min-heap open list, and maintains an explicit vector object, making use of the C++ standard library function `make_heap()` to perform heapify operations. This allows for the implementation of procedures as described in Algorithm 2. The node comparison operation uses the standard `node_comparer` objects in LAPKT. For the double trimmed open list, an additional node comparer is implemented for the secondary open list, which ranks nodes using the second primary heuristic value in the search node instead of the first primary heuristic value.

6.2 Implementation of Hybrid Configurations

We set the memory threshold for the frontend BFNoS solver by performing memory usage measurements directly in LAPKT [65], to then fallback to the backend solver. For the BFNoS-Dual solver, the fallback planner is ran as part of the same process in LAPKT since Dual-BFWS, thus also its backend component, is already implemented in LAPKT. For the BFNoS-LAMA and BFNoS-Maidu-h^2 solvers, a script runs the BFNoS frontend first in LAPKT. The LAMA and IPC2023-Scorpion-Maidu backend solvers are not implemented in LAPKT, but in variants of the Fast Downward planning library [67], and hence must be ran as a separate process. Thus, when a frontend threshold is reached, the LAPKT process is halted sending a suitable signal, such that the backend solver script may then be called by the main process.

6.2.1 LAMA-First

The backend solver of BFNoS-LAMA is the LAMA-First planner. The “first” part of the name refers to the variant of LAMA that halts as soon as it finds the first valid plan from root node to the goal. This is because the full LAMA planner continues running, seeking better (shorter) plans to the goal, until it exceeds the total time threshold. Such an approach is used in the satisficing track of the International Planning Competition, where performance is evaluated based on plan quality. However, our study places less emphasis on plan quality, as performing the plan improvement step on all benchmark instances significantly increases experimental running times. Consequently, it is common practice to evaluate planners without the plan improvement component outside of the International Planning Competition, to streamline testing and obtain results more efficiently.

The LAMA-First backend is ran using the ‘release-23.06’ branch of the fast downward library [67]. An alias for LAMA-First already exists in the library. Thus the planner is easily selected through the command:

```
"--alias", "lama-first"
```

6.2.2 “First” Variant of IPC2023 Scorpion-Maidu

For the backend solver of the BFNoS-Maidu-h^2 solver, we run a “first” version of Scorpion Maidu [20], which halts after finding a solution rather than improving the plan, from the IPC2023 branch of the code base [20] using the following command, as there is no explicit alias to directly run a “first” version of the planner:

```
--evaluator 'hlm=lmcount(
lm_factory=lm_reasonable_orders_hps(lm_rhw()),
transform=adapt_costs(one),pref=false)'
--evaluator 'hff=ff(transform=adapt_costs(one))'
--search 'lazy(alt([single(hff),
single(hff, pref_only=true),
single(hlm), single(hlm, pref_only=true),
type_based([hff, g()]),
novelty_open_list(novelty(width=2,
consider_only_novel_states=true,
reset_after_progress=True),
break_ties_randomly=False,
handle_progress=move)],
boost=1000),preferred=[hff,hlm],
cost_type=one,reopen_closed=false)'
```

This command is just for the planner, and does not include the h^2 preprocessing step, which can be included, together with the preprocessor time limit, using the commands:

```
    "--transform-task", "preprocess-h2",
    "--transform-task-options", "h2_time_limit,180",
```

Where 180 is the preset preprocessor maximum running time, according to its IPC 2023 configuration.

6.3 Experiment Implementation

Planning experiments are implemented using the *LAB* library [68]. LAB is an open-source Python package designed to facilitate running, analysis and comparison of planning experiments on benchmark sets. It supports the running of third-party libraries including LAPKT, controlling the resource availability and offering parallel execution. The library also includes code for parsing results. Using this library enables the implementation of rigorous planning experiments subject to predetermined time and memory constraints, and allocating parallel experiments to a single core each to speed up the obtainment of results.

Experiments for count-based novelty solvers $\text{BFWS}_t(f_5(W_2))$, $\text{BFCS}_t(f_5(C_1))$, and $\text{BFNoS}_t(f_5(C_1), f_5(W_2))$, and the BFNoS-Dual hybrid solver are executed by making the LAB module directly call the LAPKT command for planner execution. This is not possible with BFNoS-LAMA and BFNoS-Maidu-h^2, as the frontend and backend solvers are run using different libraries. As mentioned, the frontend and backend are called by a parent Python script, which catches a suitable error message launched by the frontend solver when it fails, and runs the backend accordingly.

A main consideration of the implementation is how the time and memory limits set by LAB are propagated from parent process, the script that is called, to child processes, the frontend and backend planners: the parent process simply propagates an identical time and memory limit to its child process. This is not a problem for the memory limit, as ending the old process frees its memory, and the new process is then subject to the same memory limit. However, it is an issue for the time limit. If the total time threshold is 1800 *sec*, and the first process fails after 900 *sec*, the second process is supposed to only have 900 *sec* left. However, propagating an identical time limit to the second child process will set another 1800 *sec* limit on it, exceeding the intended overall time allowance. This issue is resolved by directly having the execution time of the frontend child process be measured by the main script, and then having it set a new limit for the backend child process that is equivalent to the difference between the total time limit and the frontend execution time. Furthermore, the wall-clock time of the main python script, recorded by LAB, allows for verification of the total frontend + backend execution time.

Suitable parsers are also implemented using the LAB parser module, to extract relevant information from the planner run logs, in order to obtain the aggregate set of results from all planner runs in an experiment. Analysis of data was then performed separately using Jupyter Notebook, reading the extracted information from the experimental logs.

Chapter 7

Experimental Evaluation

This chapter contains the main experimental results of this thesis. Section 7.1 details the setup of experiments performed to evaluate all solvers and techniques introduced in previous chapters. Section 7.2 evaluates the proposed BFCS$_t$ and BFNoS$_t$ count-based solvers, comparing their performance to BFWS and BFWS$_t$ width-novelty solver variants. Our analysis assesses problem coverage, search time and efficiency, and plan length of the planners. Section 7.3 then provides coverage results for our proposed hybrid solver variants, demonstrating state-of-the-art performance compared to best-in-class benchmark solvers from prior literature. We further highlight the crucial contribution of using BFNoS$_t$ frontend solvers alongside memory thresholds, demonstrating the strength of our solution and providing empirical validation for the analysis of memory thresholds discussed in Section 5.2.

7.1 Experimental Setup

Our experiments were conducted using Downward Lab's experiment module [69], adhering to the IPC satisficing track constraints of 1800 seconds and 8 *GB* memory. Each test was ran on a single core of a cloud instance AMD EPYC 7702 2GHz processor. We implemented all proposed planners in C++, using LAPKT's [65] planning modules. For hybrid experiments, LAMA-First [19] and Scorpion-Maidu [20, 70] served as backend components, employing Fast-Downward [67] and the IPC 2023 code repository [63], respectively. Except for Approximate-BFWS, BFWS variants utilized the FD-grounder for grounding [71], however in problems where the FD grounder produces axioms (unsupported by LAPKT), LAPKT automatically switched to the Tarski grounder [72]. Approximate-BFWS exclusively used the Tarski grounder, following its initial setup and IPC-2023 configuration. We utilized IPC satisficing track benchmarks as in [10], selecting the latest problem sets for recurring domains. We conducted two sets of experiments. The first benchmarked our planners against the base BFWS(f_5) solver, evaluating the degree to which our proposed classical count-based novelty and trimmed open list techniques improve the coverage of BFWS(f_5) and its exploration efficiency, measured as the number of expansions required to find a solution. The second set of experiments compared our hybrid configurations to Dual-BFWS, Approximate-BFWS, LAMA-First, and a "first" version of the IPC-2023 satisficing track winner Scorpion-Maidu, in order to assess the coverage gains obtainable by adopting our proposed frontend solver alongside existing solvers in a dual configuration, relying on memory thresholds alongside more traditional time thresholds to trigger the frontend to fallback.

Multiple measurements were conducted for variants implementing trimmed open lists, as well as for the Approximate-BFWS solver [73]. The measurement runs were repeated 5 times, always using the same set of seeds – {0,1,2,3,4} – to avoid biased seed selection. Figures 7.1, 7.2, 7.3, 7.5, and 7.6 only include one arbitrarily selected run per visualized solver for clarity. The small variance in results among runs does not alter the figures and the discussed trends meaningfully.

7.2 Count-Based Solvers

7.2.1 Coverage Results

The results of the experiment in Table 7.1 show a significant improvement in coverage compared to that of the baseline BFWS(f_5) solver.

The effect that introducing a trimmed open list has on the solver's coverage can be assessed by comparing the results of BFWS($f_5(W_2)$) and BFCS($f_5(C_1)$) to those of BFWS$_t$($f_5(W_2)$) and BFCS$_t$($f_5(C_1)$) respectively, which are identical in all but the open list variant. The trimmed open list's smaller memory footprint substantially boosts the

		Coverage		
Solver	**% Score**	**Total (1831)**	**IPC 2023 (140)**	**IPC 2018 (200)**
BFWS($f_5(W_2)$)	76.76%	1510	67	120
BFCS($f_5(C_1)$)	77.57%	1510	75	129
BFWS$_t$($f_5(W_2)$)	79.78%±0.13	1555±1.64	66±0.45	134±1.82
BFCS$_t$($f_5(C_1)$)	81.53%±0.33	1568±4.76	78±0.89	146±2.79
BFNoS	**83.32%±0.18**	**1600±3.90**	**87±1.10**	**149±1.34**

TABLE 7.1: % score and coverage comparison of proposed variants. % score is the average of the % of instances solved in each individual domain, calculated over all benchmark domains. The coverage is provided over the full set of benchmark domains, as well as the subsets of domains corresponding to those used in recent IPC 2023 and IPC 2018 competitions. Values for solvers that use a trimmed open list show the mean and standard deviation across 5 measurements.

instance coverage of BFCS and BFWS solvers alike, demonstrating the versatility of its node filtering mechanism even when dealing with the W_2 heuristic's narrower range.

BFCS$_t$(f_5) outperforms BFWS$_t$(f_5) in both coverage and normalized score. This advantage is especially evident in problems with high atomic widths, such as *Ricochet-Robots* [74] from IPC 2023 [75], where BFCS$_t$(f_5) consistently solves 19 out of 20 instances compared to the BFWS$_t$(f_5)'s single solve. This underscores count-based novelty's scalability in complex problems, contrasting with W_x metrics which have to revert to secondary heuristics after exhausting novel nodes, and demonstrates the $O(n)$ count-based novelty heuristic variant's capacity to seek novel tuples of size > 1 as predicted by the theoretical analysis in Section 3.4.

However, while C_1 can prioritize states with a higher expected number of 2-tuples, it cannot explicitly detect the presence of 2-tuples like W_2. BFCS$_t$(f_5) does show reduced coverage compared to BFWS$_t$(f_5) in various domains, suggesting that there still are multiple domains where an actual width-2 search can provide meaningful benefits compared to an estimation of the number of 2-tuples. As mentioned in Section 3.4, one of the main limitations of classical count-based novelty's estimation of novel k-tuples is its inability to account for the underlying structure of the domain, and the ways in which this structure impacts the underlying distribution of variables and tuples across states. Specific domains may therefore exhibit systematic differences in such distributions, which causes the ordering of nodes induced by the count-based heuristic to implicitly over- or underestimate the number of novel 2-tuples present in specific states. By directly measuring the number of 2-tuples in each state, W_2 helps address this limitation, suggesting that W_2 and C_1 heuristics offer complementary strengths.

	$N \geq 0$	$N \geq 1$	$N \geq 5$	$N \geq 10$	$N \geq 100$	$N \geq 1000$	$N \geq 10000$
Generated	100%	99.45%	97.68%	95.09%	74.61%	45.35%	28.26%
Expanded	100%	34.88%	27.21%	24.45%	16.34%	10.49%	4.91%

TABLE 7.2: % of instances across all IPC satisficing benchmarks where a node with count $\geq N$ was recorded across generated and expanded nodes by a $\text{BFCS}_t(f_5)$ planner. This includes unsolved instances.

This synergy is exemplified by $\text{BFNoS}_t(f_5(C_1), f_5(W_2))$, which surpasses both in coverage due to its dual-heuristic approach. Notably, it also secures a significant 3.5% gain in normalized scores compared to $\text{BFWS}_t(f_5)$. The % score of the solver is meaningful, as it is less biased by the number of problem instances in a domain, which may vary significantly. The results in Table 7.1 are therefore indicative of BFNoS's enhanced cross-domain generalization. To our knowledge, this is the first instance demonstrating performance gains obtained by combining distinct goal-unaware exploration heuristics in planning, as opposed to combining goal-aware exploitation heuristics as in [43].

Results in Table 7.1 also show a greater variance in coverage and % score for solvers adopting a count-based heuristic, potentially due to its wider range reducing the number of ties resolved by the secondary $\#g$ goal-counting heuristic, and instead increasing the relevance of count-based novelty. Given that the novelty value of a state varies based on the visitation history of the search algorithm, it leads to less consistent search guidance compared to traditional heuristics, whose value given a state is deterministic and does not vary, and accentuates differences in search direction introduced by the trimmed open list's randomized insertion procedure. The variance of results for BFNoS is then generally in the middle.

7.2.2 Search Efficiency

Discounting the significant difference in coverage between $\text{BFWS}_t(f_5)$ and BFNoS and focusing only on problems solved by both, Figure 7.1 shows that integrating the C_1 novelty heuristic improves exploration efficiency in many problems. The number of expansions incurred by a search algorithm to solve a problem is used to define the exploration efficiency.

Both planners achieve a significantly better efficiency than the other – at least an order of magnitude fewer expansions – in multiple instances. However, while $\text{BFWS}_t(f_5)$

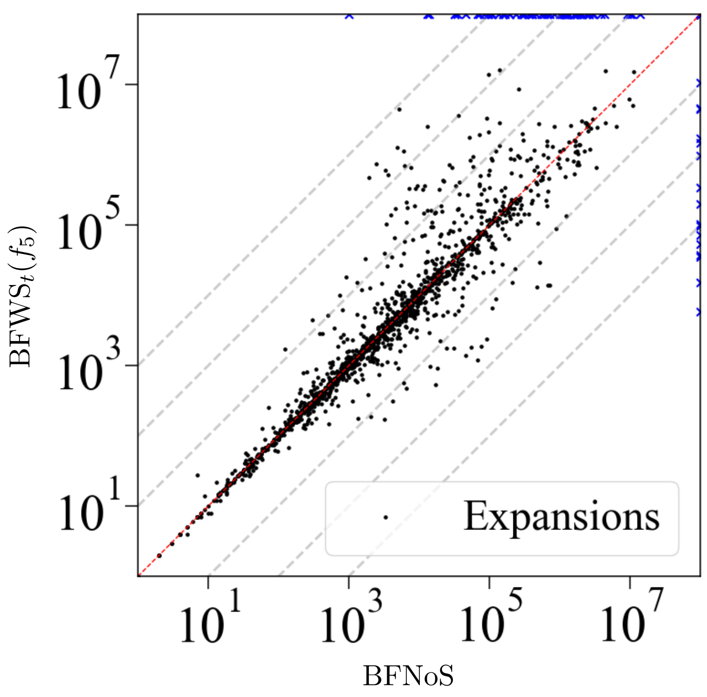


FIGURE 7.1: Number of nodes expanded to solve instances by BFNoS and $\mathrm{BFWS}_t(f_5)$. Blue crosses represent instances not solved by at least one planner.

mainly achieves a small edge across instances requiring relatively fewer expansions on aggregate, denoted by the dots that are closer to the origin, BFNoS demonstrates improved efficiency at tackling “larger” domains that require more expansions on average. This can be observed in the top-right quarter of Figure 7.1, where differences are more markedly in favor of BFNoS (more black dots are pushed further upwards). This range also includes the vast majority of problems in which $\mathrm{BFWS}_t(f_5)$ fails to find a valid plan before hitting the memory or time limits imposed, shown by the blue crosses in the upper edge of the figure. The count-based heuristic therefore provides a more informed search across this set of instances, on aggregate benefiting the performance of the search algorithm. The problems that $\mathrm{BFWS}_t(f_5)$ solves more efficiently, instead, seem to be particularly suited to a width-2 partitioned novelty search, and thus BFNoS’s double heuristic approach delays the discovery of a solution.

7.2.3 Search Time

Analysis of the time required to find a solution by BFNoS and $\mathrm{BFWS}_t(f_5)$ in Figure 7.2 uncovers two main trends:

1. BFWS$_t(f_5)$ systematically achieves a lower running time to solve a main sequence of planning instances from benchmark problems. On average, this trend is represented by a constant factor reduction in running time, with the constant factor being less than an order of magnitude for the vast majority of instances.

2. A polynomial reduction in running time on behalf of BFNoS to solve a smaller subset of problems, which in several cases exceeds an order of magnitude. This trend is made more apparent by also including the numerous instances not solved by BFWS$_t(f_5)$ (blue crosses on the top edge of the figure), which correspond to instances where BFWS$_t(f_5)$ either ran out of time or memory.

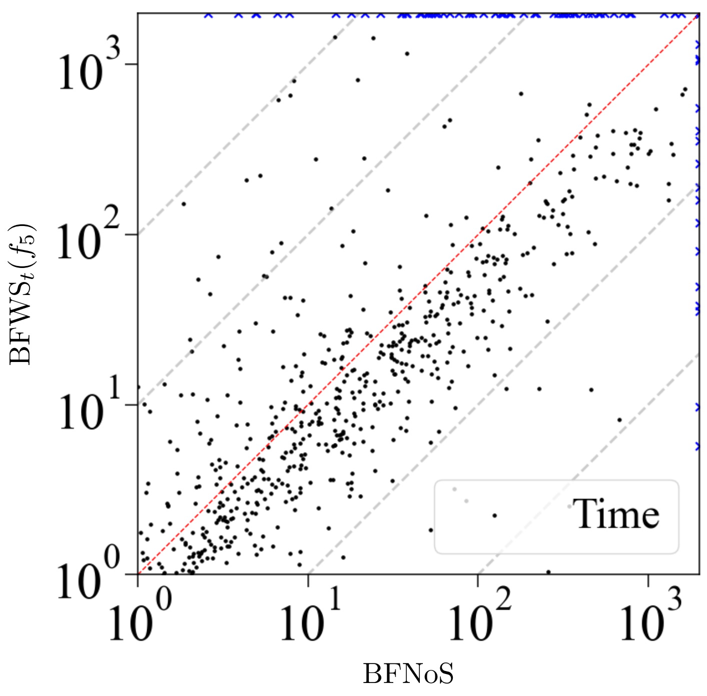


FIGURE 7.2: Time (*sec*) to solve instances by BFNoS and BFWS$_t(f_5)$. Blue crosses represent instances not solved by at least one planner.

Running time is determined by two main factors. The first is the time requirement of expanding a node, generating its successors, and most importantly evaluating the heuristic values for all its successors. In most state-of-the-art search algorithms, the heuristic evaluation is by far the operation responsible for consuming the greatest amount of time. The second main contributor is the number of nodes expanded and generated by the algorithm, as for a given average node expansion time, the number of expansions then determines the algorithms total running time.

Figure 7.1 then helps interpret the results in Figure 7.2. The first trend described in the previous paragraph is determined by the main sequence of instances in Fig. 7.1, where both solvers achieve a very similar expansion count. Since BFNoS corresponds to a BFWS$_t(f_5)$ search algorithm with an additional open list, and additional heuristic evaluation, it is reasonable to expect a constant factor increase in its running time, given an equivalent number of expansions. The second trend seems to correspond to the set of "larger" instances discussed in the upper-right quarter of Figure 7.1 where BFNoS achieves a significantly better efficiency. Similarly, BFNoS achieves a polynomial improvement trend as running time increases in this subset of instances, before BFWS$_t(f_5)$ starts failing altogether to solve a large number of instances between the 10^1 and 10^2 marks on the x-axis.

The constant factor increase in runtime described in the first trend represents one of the main limitations created by the double open list solution, which forces the algorithm to sacrifice some running time to achieve a greater and more robust coverage. In most cases, the increase in running time is not overly substantial, and satisficing planners commonly accept this trade-off to enhance their coverage. A width-1 1-BFWS(f_5) planner, for example, is significantly faster than even a 2-BFWS(f_5) variant, but achieves a markedly lower coverage [8]. It is also worth noting that the single trimmed open list in BFWS$_t(f_5)$ already improves its running time compared to the base BFWS(f_5) algorithm. Firstly, by pruning – often, the majority of – generated nodes which would otherwise require a $\log(n)$ insertion time in the open list for size n of the open list. Secondly, by limiting the number of nodes n in the open list in the first place, which can otherwise grow to much larger values. The solver thus represents a very performant benchmark.

Overall, an on-average small constant-factor increase in runtime over mostly simple problems represents a valid trade-off for the improved coverage, and often running time, of BFNoS over larger instances: in benchmark planning problems, a large number of instances are considered quite simple, and solved by most modern planners. It is the last few hundred instances that truly provide a challenge to modern planners, and that are targeted by new satisficing planners.

This main limitation described in this section may be further addressed through existing solutions, such as a BFNoS variant where the second open list adopts an Approximate Novelty heuristic [10] instead of a basic Novelty-2 heuristic. Approximate Novelty is a width-novelty variant, which may thus benefit from the orthogonality with count-based novelty described in Section 7.2.1, that achieves lower time and memory complexity than the W_x heuristic given maximum width x, improving the running time and memory consumption, and potentially allowing the secondary open list to also target nodes with a higher width of 3.

7.2.4 Plan Length

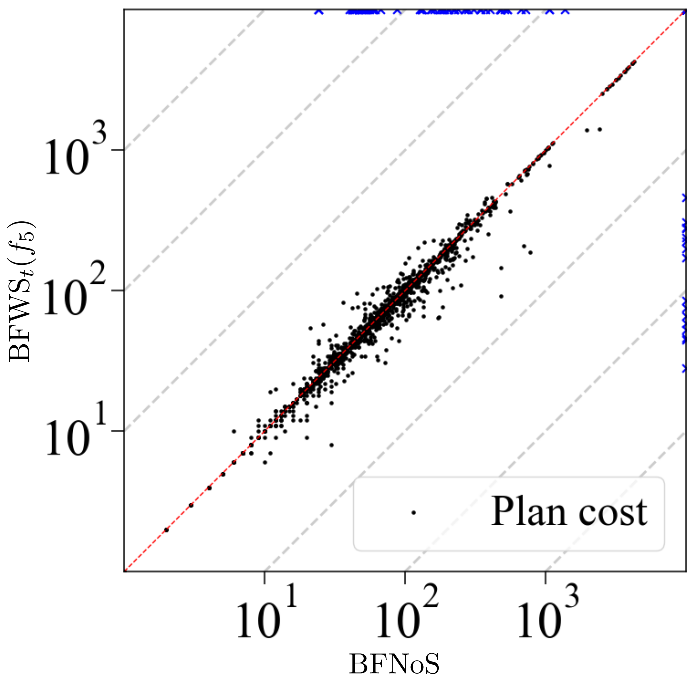


FIGURE 7.3: Plan cost over instances solved by BFNoS and $\mathrm{BFWS}_t(f_5)$. Blue crosses represent instances not solved by at least one planner.

Figure 7.3 compares the plan cost of obtained solutions. In this case, the vast majority of problems show minimal differences in plan quality, however in aggregate $\mathrm{BFWS}_t(f_5)$ maintains an edge compared to BFNoS. Still, the BFWS family of planners achieve competitive plan qualities compared to other state-of-the-art planners [8, 14, 76], and a marginally lower first-plan quality in a few domains is a reasonable trade-off for the better coverage. This may be a negative effect of C_1's greater heuristic granularity. Both $\mathrm{BFWS}_t(f_5)$ and BFNoS adopt node cost as a third tie-breaking heuristic, prioritizing nodes with a lower distance from the root node. In this respect, count-based novelty's granularity creates fewer ties, relying less on node cost and potentially finding alternative paths to the goal from nodes with a lower C_1 heuristic value but a higher plan length compared to $\mathrm{BFWS}_t(f_5)$. This raises a potential area for improvement, however the margins of improvement, as highlighted by Figure 7.3, are limited in most domains. Furthermore, it is common for satisficing planning algorithms to improve their first valid plan by falling back to secondary solvers designed to improve such plans, further alleviating this issue.

7.3 Hybrid Solvers

7.3.1 Coverage Results

Domain	BFNoS	Dual-BFWS	Apx-BFWS (Tarski)	LAMA-First	Maidu	Maidu with h^2	BFNoS-Dual-back	BFNoS-LAMA	BFNoS-Maidu-h^2
agricola-sat18 (20)	15±0.0	12	**18±0.9**	12	12	13	15±0.5	15±0.5	15±0.5
airport (50)	**47±0.6**	46	**47±0.6**	34	38	45	46±0.6	46±0.5	46±0.6
caldera-sat18 (20)	18±0.0	**19**	**19±0.6**	16	16	16	16±0.0	17±0.5	18±0.0
cavediving-14 (20)	**8±0.5**	**8**	**8±0.5**	7	7	7	**8±0.0**	**8±0.0**	**8±0.5**
childsnack-sat14 (20)	1±1.1	**9**	5±0.6	6	6	6	8±0.0	6±0.0	6±0.5
citycar-sat14 (20)	**20±0.0**	**20**	**20±0.0**	5	6	6	**20±0.0**	**20±0.0**	**20±0.0**
data-network-sat18 (20)	17±0.6	13	**19±0.5**	13	16	16	16±0.8	15±1.1	16±0.8
depot (22)	**22±0.0**	**22**	**22±0.0**	20	**22**	**22**	**22±0.0**	**22±0.0**	**22±0.0**
flashfill-sat18 (20)	14±1.3	**17**	15±1.6	14	15	14	**17±0.5**	16±0.6	16±0.9
floortile-sat14 (20)	2±0.5	2	2±0.0	2	2	**20**	2±0.0	2±0.0	**20±0.0**
folding (20)	9±0.0	5	5±0.5	**11**	**11**	**11**	9±0.0	9±0.0	9±0.0
freecell (80)	**80±0.0**	**80**	**80±0.0**	79	**80**	**80**	**80±0.0**	**80±0.0**	**80±0.0**
hiking-sat14 (20)	**20±0.0**	18	**20±0.0**	**20**	**20**	**20**	**20±0.0**	**20±0.0**	**20±0.0**
labyrinth (20)	15±0.5	5	**18±0.5**	1	0	2	15±0.5	15±0.5	15±0.5
maintenance-sat14 (20)	**17±0.0**	**17**	**17±0.0**	11	13	13	**17±0.0**	**17±0.0**	**17±0.0**
mystery (30)	18±0.6	**19**	**19±0.0**	**19**	**19**	**19**	**19±0.0**	**19±0.0**	**19±0.0**
nomystery-sat11 (20)	14±0.8	**19**	14±0.5	11	**19**	18	**19±0.0**	15±0.6	17±0.0
nurikabe-sat18 (20)	16±0.6	14	17±0.5	9	11	16	17±0.6	17±0.0	**18±0.0**
org-synth-split-sat18 (20)	8±0.5	12	8±0.0	**14**	**14**	**14**	11±0.5	**14±0.0**	**14±0.9**
parcprinter-sat11 (20)	9±0.6	16	11±1.3	**20**	**20**	**20**	**20±0.0**	**20±0.0**	**20±0.0**
pathways (30)	26±0.9	**30**	28±1.1	23	25	25	**30±0.0**	27±0.8	27±0.7
pipesworld-nt (50)	**50±0.0**	**50**	**50±0.0**	43	45	45	**50±0.0**	**50±0.0**	**50±0.0**
pipesworld-t (50)	43±1.6	42	**44±0.6**	43	43	43	43±0.8	43±0.5	43±0.6
recharging-robots (20)	**14±0.6**	12	**14±0.8**	13	13	13	**14±0.5**	**14±0.0**	**14±0.5**
ricochet-robots (20)	**20±0.5**	**20**	18±0.6	14	18	18	**20±0.0**	**20±0.0**	**20±0.0**
rubiks-cube (20)	5±0.0	6	5±0.6	**20**	**20**	**20**	5±0.0	**20±0.0**	16±0.6
satellite (36)	34±0.8	33	34±0.5	**36**	**36**	**36**	34±0.6	35±0.0	35±0.0
schedule (150)	149±1.3	**150**	149±1.3	**150**	**150**	**150**	149±0.7	**150±0.0**	**150±0.0**
settlers-sat18 (20)	13±1.5	7	12±0.7	17	**18**	**18**	12±0.5	17±0.0	17±0.5
slitherlink (20)	**5±0.6**	**5**	**5±0.7**	0	0	0	**5±0.5**	3±0.6	4±0.7
snake-sat18 (20)	**20±0.0**	17	**20±0.0**	5	14	14	**20±0.0**	**20±0.0**	**20±0.0**
sokoban-sat11 (20)	15±1.1	17	14±0.9	19	19	**20**	15±0.5	19±0.0	**20±0.0**
spider-sat18 (20)	17±1.3	16	16±1.1	16	16	17	**18±0.0**	**18±0.0**	**18±0.9**
storage (30)	**30±0.5**	29	**30±0.0**	20	25	25	29±0.5	29±0.0	29±0.6
termes-sat18 (20)	10±0.8	10	5±1.5	**16**	14	14	10±0.5	14±0.0	14±0.0
tetris-sat14 (20)	**20±0.0**	17	**20±0.0**	16	17	20	**20±0.0**	**20±0.0**	**20±0.0**
thoughtful-sat14 (20)	**20±0.0**	**20**	**20±0.2**	15	19	19	**20±0.0**	**20±0.0**	**20±0.0**
tidybot-sat11 (20)	**20±0.0**	18	**20±0.2**	17	**20**	**20**	**20±0.0**	**20±0.0**	**20±0.0**
transport-sat14 (20)	**20±0.0**	**20**	**20±0.2**	17	18	16	**20±0.5**	**20±0.0**	**20±0.0**
trucks (30)	8±0.8	19	13±1.5	18	20	**22**	17±0.5	16±0.0	20±0.0
woodworking-sat11 (20)	**20±0.0**	**20**	12±1.1	**20**	**20**	**20**	**20±0.0**	**20±0.0**	**20±0.0**
Coverage (1831)	1600±3.9	1603	1606±3.9	1535	1590	1626	1641±1.9	1662±2.3	**1688±3.3**
% Score (100%)	83.32% ±0.18	83.23%	83.51% ±0.27	79.06%	82.84%	85.31%	86.23% ±0.09	87.87% ±0.17	**89.79% ±0.22**
BFNoS % coverage share	-	-	-	-	-	-	97%	96%	94%

TABLE 7.3: Comparative coverage analysis across benchmark domains. *% score* is the average of the % of instances solved in each domain. *BFNoS % coverage share* refers to the % of covered instances solved by the BFNoS frontend, when run in a dual configuration. Domains which are fully solved by all planners are omitted. Values for BFNoS variants and Approximate-BFWS represent the mean and include the standard deviation across 5 measurements. The complete table of results is included in Appendix C.

Table 7.3 shows BFNoS$_t(f_5(C_1), f_5(W_2))$ alone achieves comparable coverage to all baselines with the exception of the IPC 2023 configuration of Scorpion-Maidu, which is the only baseline solver incorporating a preprocessor. The hybrid solvers adopting a BFNoS$_t(f_5(C_1), f_5(W_2))$ frontend outperform all baselines, denoting a meaningful increase in coverage and normalized % score compared to their respective backends, particularly for the hybrid configuration with LAMA-First backend, which covers 62 and 127 more instances compared to BFNoS and LAMA-First respectively. BFNoS-Maidu-h^2 gains an edge on BFNoS-LAMA mainly thanks to the additional preprocessor in the backend, which only gets used if the frontend solver fails. It is worth noting that the implementations of BFNoS-LAMA and BFNoS-Maidu-h^2 perform the grounding operation a second time for the backend, resulting in redundant computations which may use up significant time in hard-to-ground domains. Addressing this limitation may provide further easily-achievable improvements in coverage and runtime for the two algorithms. BFNoS-Dual-backend performs more poorly than the other hybrid configurations, yet this does not come as a surprise. The Dual-BFWS backend is a Novelty planner which shares many more similarities with our own proposed frontend, most crucially the W_2 primary heuristic, thus causing more overlap in coverage between the frontend and backend. Still, even this version suffices to outperform tested benchmarks.

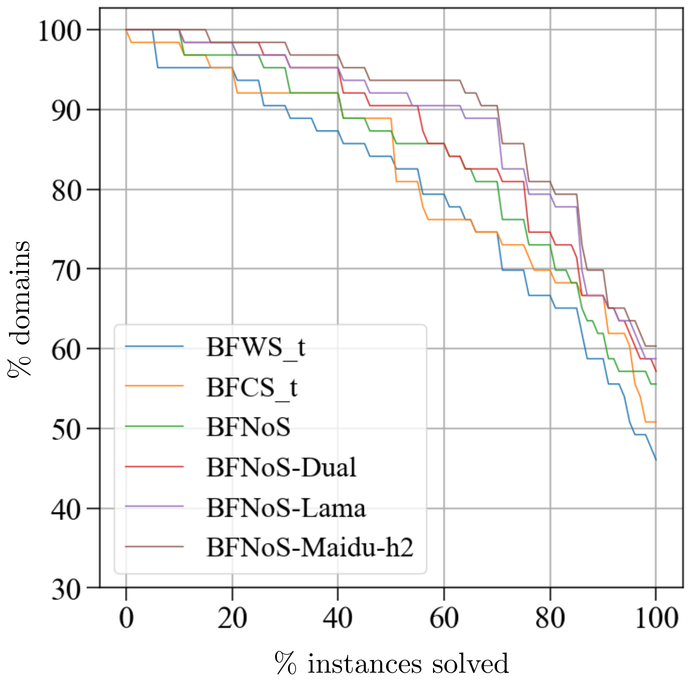


FIGURE 7.4: Coverage analysis of proposed solvers. Lines indicate the % of IPC benchmark domains where the % of domain instances solved is ≥ a given value.

Table 7.4 also underscores a significant increase in coverage of proposed hybrid planners on problem sets from IPC 2023 [45] and IPC 2018 [76], with BFNoS-LAMA covering 101 and 164 instances, and BFNoS-Maidu-h^2 covering 98 and 166 instances on average respectively. In all cases, the implementation of hybrid planner configurations leveraging only two powerful solvers allows us to keep such dual solvers as simple as possible, which we argue helps us minimize the risk of overfitting to the set of available benchmarks. The main improvement in IPC 2023 for BFNoS-LAMA and BFNoS-Maidu-h^2 comes from the domain *Rubiks-Cube*, which BFNoS only solves 5 instances, whereas the LAMA and Maidu-h^2 backends solve all 20 on their own. The limitation of this is that BFNoS fails to solve Rubiks-Cube due to time, and not memory, thresholds. As such, the fallback relies on the time threshold of the hybrid solvers, and is very late in their execution.

The BFNoS solver alone also solves 87 IPC 2023 instances, outperforming all other benchmark solvers. This fact further strengthens the position of count-based novelty among modern Classical Planning heuristics, proving its capability to address the characteristics of certain hard-to-solve domains better than existing alternatives. This makes it a useful component for future planners.

Domain	BFNoS	Dual-BFWS	Apx-BFWS (Tarski)	LAMA-First	Maidu	Maidu with h^2	BFNoS-Dual-back	BFNoS-LAMA	BFNoS-Maidu-h^2
IPC2023 (140)	87±1.10	73	85±0.84	79	82	84	88±0.84	101±0.84	98±0.45
IPC2018 (200)	149±1.34	137	149±2.17	132	146	152	152±0.71	164±1.95	166±2.17

Table 7.4: Comparative coverage analysis of tested and benchmark solvers across IPC 2023 and IPC 2018 instance sets.

7.3.2 Frontend Failure Analysis

In all three proposed hybrid variants in Table 7.3, the BFNoS frontend solver with a 6 *GB* memory threshold is responsible for over 94% of all the solved instances, highlighting its role as the main component of all hybrid planners. Many of these instances are relatively simple planning problems that the backend would have also solved. However, the significant improvement in total coverage compared to that of constituent planners of the hybrid solvers underscores the frontend's ability to also solve a subset of harder problems that state-of-the-art backend planners fail to address. Even for the simpler problems, BFNoS inherits BFWS's favourable runtime and plan-length traits, as discussed in Sections 7.2.3 and 7.2.4.

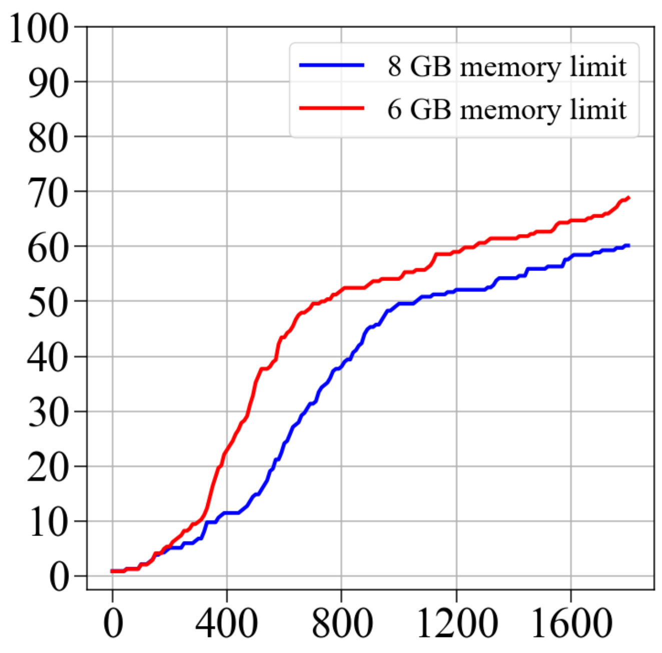


FIGURE 7.5: Cumulative % of all failures attributed to memory failures vs. time of failure (*sec*), for BFNoS solvers with 6 *GB* and 8 *GB* memory and 1800 *sec* time limits.

A better picture of the interplay between the BFNoS frontend solver and memory thresholds in the three hybrid configurations is given not only by assessing the frontend's coverage, but also through the analysis of its failures over time, presented in Figure 7.5. The curves depict a rapid rise in memory failures in the first half of the x-axis, before the 900 *sec* mark. At 8 *GB* and 1800 *sec* limits, BFNoS hits half of its total failures by reaching the memory limit when half of the available time has passed, and this time is reduced to 700 *sec* with a 6 *GB* threshold. Overall, memory failures clearly constitute the majority of total failures.

The trend validates the claim made in Section 5.2.1 that BFNoS planners can reach the "flatter" region of the coverage-memory relation (Figure 5.1) more quickly. This analysis of BFNoS's performance in relation to its memory usage aligns with empirical observations, justifying its seemingly contrasting behavior of achieving state-of-the-art coverage while also failing quickly due to high memory usage in many problems where its heuristics are uninformed.

7.3.3 Backend Contribution and Threshold Analysis

The relative improvement in instance coverage over time of proposed hybrid solvers compared to a base BFNoS solver is further examined in Figure 7.6, visualizing the impact of backend solvers. The figure is particularly useful for visually discerning the relative benefits of time and memory thresholds. The effects of time thresholds are visible at the 1600 *sec* mark for BFNoS-Dual and BFNoS-LAMA, and 1400 *sec* mark for BFNoS-Maidu-h^2.

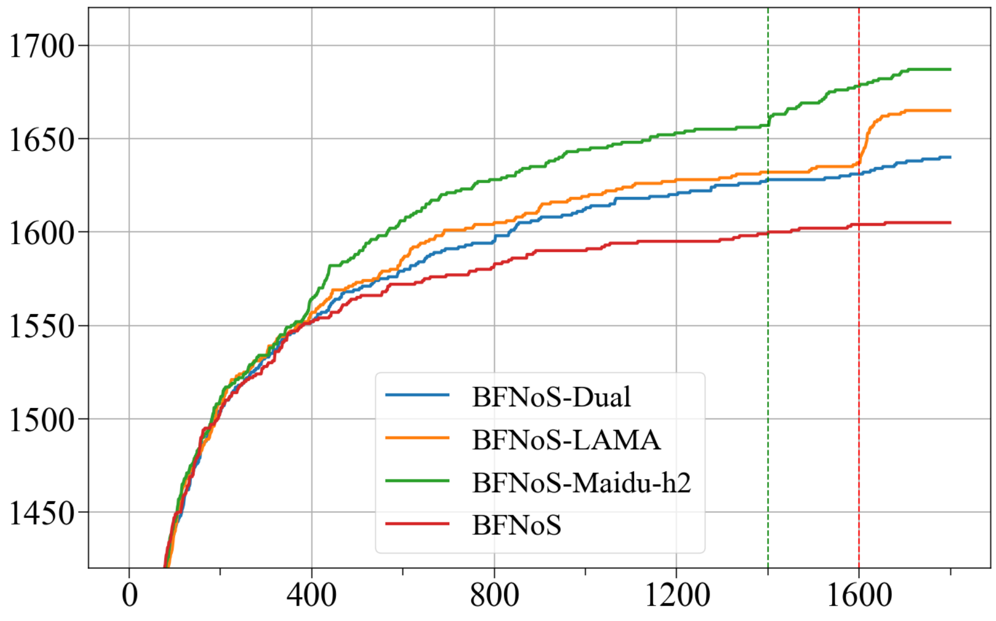


FIGURE 7.6: Instance coverage vs. time (*sec*). Comparison of BFNoS with the three presented hybrid configurations. The solvers are run in an identical configuration to previous sections, with BFNoS using the full 8 *GB* memory allowance when run on its own, and a 6 *GB* memory threshold when run as the frontend of hybrid configurations. The vertical lines signal the 1400 *sec* (green) and 1600 *sec* (red) time thresholds.

A positive takeaway is found in the memory threshold contributing more heavily to the increase in coverage than the time threshold, which is beneficial as it implies an earlier frontend fallback, and lower total runtime to find a solution. The relative contribution of the memory threshold is also outlined in Table 7.5. The BFNoS-Dual solver in particular only covers 5 additional instances on average through the use of a time threshold. denoting a very small contribution.

Figure 7.6 also highlights that the improved performance of BFNoS-LAMA compared to BFNoS-Dual may be deceiving. The solver follows a comparable coverage over time to BFNoS-Dual, and only achieves a jump in coverage thanks to the time threshold at the 1600 *sec* mark. Rubiks-Cube accounts for 15 of the 24 instance difference on average between the 'memory-only' threshold and 'time + memory' threshold in Table 7.5, potentially biasing the improved results of this hybrid solver.

Hybrid Solver	**Only memory TH**	**Time + memory TH**
BFNoS-Dual-Back	1636±3.3 (85.90%±0.14)	1641±1.9 (86.23%±0.09)
BFNoS-LAMA	1638±4.9 (86.13%±0.26)	1662±2.3 (87.87%±0.17)
BFNoS-Maidu	1643±3.7 (86.45%±0.17)	1658±1.6 (87.46%±0.12)
BFNoS-Maidu-h^2	1662±4.7 (88.02%±0.21)	1688±3.3 (89.79%±0.22)

Table 7.5: Coverage and % score comparison of hybrid planner variants adopting only a memory threshold, and the usual configuration with both memory and time thresholds. All memory thresholds are set at 6 *GB* for both configurations of each planner. The BFNoS-Maidu planner is a variant of BFNoS-Maidu-h^2 that does not use the h^2 preprocessor with its Scorpion-Maidu backend. Values represent the mean and the standard deviation across 5 measurements.

While having the backend excel at problems where the frontend under-performs is one of the main goals of a hybrid configuration, a larger contribution from a very late time threshold is a weaker result than earlier memory-threshold failbacks, as it implies a highly sub-optimal near-limit runtime, and leaves a small amount of time for additional tasks such as plan improvement.

Choosing such late time thresholds was a deliberate choice to focus on the contribution of memory thresholds and to separate its effects more clearly, while allowing for the study and comparison of the additional improvement in coverage obtainable with time thresholds. If the focus were on early coverage, the shape of the orange line for BFNoS-LAMA in Figure 7.6 shows that most of the gap compared to the BFNoS base solver attributable to the memory threshold is largely realized by the 1200 *sec* mark. Thus, the hybrid solver could achieve similar performance with a lower time threshold. This analysis also links back to the importance of the steep section of the memory-failures-over-time plots in Figure 7.5, underscoring the significance of selecting BFNoS as the frontend for the presented hybrid solvers.

The BFNoS-Maidu-h^2 solver proves to be preferable to other hybrid variants not only in terms of pure coverage, but also from the shape of the curve in Figure 7.6. It achieves a clear improvement in coverage compared to other hybrid variants starting from the 400 *sec* mark, solving as many instances on average in its 'memory-only' threshold configuration as the second best variant, BFNoS-LAMA, solves in its 'memory + time' threshold variant (Table 7.5). The memory threshold then contributes to almost $\frac{3}{4}$ of total instances solved by the backend solver, solving most of them by the 1200 *sec* mark and thus raising the potential for an earlier time threshold.

Comparing the coverage of BFNoS-Maidu − the same frontend-backend configuration as BFNoS-Maidu-h^2, albeit without the use of the h^2 preprocessor [44] in the backend − in Table 7.5 to that of other hybrid configurations then shows an overall performance in line with that of BFNoS-LAMA. The main contributor to BFNoS-Maidu-h^2's improved performance compared to other hybrid configurations is then the "sandwiched" frontend-preprocessor-backend configuration, whereby the preprocessor only runs when the frontend solver fails. This configuration further enjoys the benefit of not delaying the execution of the frontend planner when solving simple instances where it does not require the preprocessor.

As stated in Chapter 5, the reason BFNoS-Maidu-h^2 is the only configuration adopting the preprocessor is that it is already included in the IPC 2023 configuration of Scorpion-Maidu [20]. The results in this section suggest that further study on the interplay between the presented count-based dual solver configurations and preprocessors could be a promising avenue to extract further performance improvements. This study also does not rule out potential benefits derived from adopting the preprocessor directly with the frontend BFNoS solver, especially since recent work [77] has demonstrated significant improvements in the performance of similar BFWS solvers.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this thesis we introduce the concept of count-based novelty as an alternative novelty exploration framework in Classical Planning, showing that the arity-1 variant of our proposed metric is capable of effectively predicting states with novel k-tuples only using a constant number of tuples that is linear in the number of features of a problem. We introduce the use of counts and Hamming distances to relate the exploratory behavior of count-based novelty to the existing body of knowledge on Novelty. This enables it to act as a more granular and versatile alternative to previous width-novelty heuristics, performing effectively in low-atomic-width and high-atomic-width domains alike. Through the adoption of a partitioned classical count-based novelty heuristic, we extend the BFWS family of planners. The proposed solvers are shown to outperform width-novelty counterparts, with count-based novelty and width-novelty demonstrating complementary strengths. Our techniques used in the BFNoS solver then demonstrate the effectiveness of combining distinct novelty metrics, achieving competitive coverage compared to state-of-the-art planners.

We also propose single and double Trimmed Open List variants which allow us to upper bound the open list size by pruning nodes unlikely to be expanded. This contribution provides an effective solution to the need to reduce the memory impact of nodes in the open list, while minimizing the amount of useful search nodes that are not inserted into the open list. Multiple features make this contribution versatile and relevant to the field of classical planning at large. A trimmed open list only requires setting the maximum open list size, and automatically determines the insertion of new nodes based on an implicit distribution derived from the heuristic values of nodes already in the open list. This allows it to automatically adapt to an algorithms search progress, and across different search problems. Furthermore, it can be implemented through a simple modification to a min-heap data structure, and we detail how our implementation naturally supports the extension to multiple open list variants that go beyond the double open list we adopt. We show that its effectiveness is maintained when using width-novelty heuristics instead of count-novelty heuristics, and argue that the characteristics of trimmed open lists make them ideal for use with best-first search planners, independently of the adopted heuristic.

Finally, we detail a dual-configuration strategy adopting a BFNoS frontend solver and introducing memory thresholds alongside customary time thresholds, justify the suitability of our proposed strategy, and demonstrate improved coverage performance compared to state-of-the-art planners. Memory thresholds enable our hybrid solvers to reserve a variable amount of time for the frontend solver that is correlated to its probability of solving the problem at any point during the search, as opposed to the one-size-fits-all nature of customary time thresholds. This solution not only enhances the overall coverage of the solvers, but crucially achieves significantly improved planning-time performance and characteristics that alleviate the limitations of time-threshold-only dual-configuration planners.

Our work provides foundational knowledge on count-based novelty through basic solutions that mirror our theoretical analysis. The ideas presented may act as a starting point for more advanced solutions leveraging counts. Future directions may include alternative count-based variants over tuples or extracted features to guide exploration in general and domain-specific planners, as well as adaptation to the existing body of work blending Novelty and heuristic estimates. In the following section, we focus on work that introduces promising preliminary results in this direction. We provide an overview and experimental analysis of alternative count-based heuristic variants that demonstrate promising performance, and potential for expanding the scope of count-based techniques in classical planning. Our contributions in this thesis also provide a basis to bridge Classical Planning with the broader paradigm of count-based exploration, benefiting knowledge transfer with related areas such as Reinforcement Learning. Trimmed open lists and memory thresholds also proved to be simple yet effective solutions, pointing at the potential for a deeper study of similar ideas in Classical Planning.

8.2 Future Directions

8.2.1 Alternative Count-Based Novelty Heuristics

Chapter 3 presents classical count-based novelty and partitioned classical count-based novelty as extensions to previous Novelty frameworks, in particular those of Lipovetzky and Geffner [5, 8]. Our empirical implementation and study in Chapters 5, 6 and 7 adopt the C_1 heuristic we propose, which directly implements a partitioned count-based novelty over size-1 tuples, closely aligning with the theoretical portion of our study. This section diverges from the main body of work presented thus far, introducing two alternative count-based-novelty-derived heuristics whose definitions modify the count-based framework from Chapter 3 to some extent. It presents preliminary results that provide a few interesting talking points, complementing the study of count-based novelty and offering promising avenues for future developments. These new heuristics aim to enhance the existing framework by exploring different aspects of count-based novelty, thus contributing to its overall understanding and potential applications.

8.2.1.1 Added Count-Based Novelty

A necessary but not sufficient condition for a tuple to be novel in a newly generated state is that at least one of its constituent atoms must be an *added* atom, meaning that the atom was added as an effect of the action that generated the state. Otherwise, all atoms in the tuple would have already been present in the state's predecessor, implying a previous occurrence of the tuple. Thus, tuples that are known to not be novel don't need to be checked. Width-novelty heuristics [5, 8] make use of this trait, only checking occurrence of tuples where at least one of the atoms is an added atom. This enables a reduction in the computational complexity of a width-w heuristic from $O(n^w)$ to $O(n^{w-1})$, allowing the computation of the W_2 heuristic (Section 5) in linear time. This greatly improves the speed and performance of width-based search algorithms.

Adopting the same trick to reduce the heuristic's computational overhead is not possible with the definitions of count-based novelty provided in Chapter 3: given that it records the occurrence count of a tuple, and not merely the first occurrence, the counts for all tuples in a state need to be incremented regardless of whether or not they are novel. This does not mean that such a heuristic cannot be implemented by modifying the definition of count-based novelty. This is exactly what is done with the heuristic AC_x: it is an added-tuple variant of a count novelty heuristic over size-x tuples C_x, where only the counts of added tuples in a state are measured and incremented, and the lowest count of such tuples is taken as the heuristic value, improving asymptotic computational complexity.

Example 8.1. *State s_1 has atoms $\{a, b, d\}$. Through an action o whose effect is to add an atom c, the search algorithm generates state s_2 with atoms $\{a, b, c, d\}$. The C_2 heuristic measures the occurrence and increments the values of all 6 possible size-2 atom combinations from the set $\{a, b, c, d\}$. The AC_2 heuristic instead only does so with added tuples, which contain the added atom c in this case. The set of considered tuples thus becomes $\{(a, c), (b, c), (d, c)\}$.*

While the analysis in Section 3.4 does not cover this heuristic variation, we are confident that a similar approach to the existing theoretical analysis can be used to show that this technique also helps predict novel tuples of greater sizes, as all novel tuples must contain an effect atom, and are thus considered by the heuristic. Compared to the classical count-based novelty metric proposed in Chapter 3, this definition implicitly prioritizes action-state pairs that produce effect tuples with low counts, de-prioritizing states which would normally have a low count as a result of their parent also having the same tuple with a low count. Thus, variations of count-based novelty frameworks can still partially modify the heuristic's behavior, while achieving the same overall goal, providing an interesting parallel for future study.

The actual heuristic adopted is defined as $AC_{1_3,2}$, thus looking at both tuples of size 1 and size 2, with the only subtlety of limiting maximum value of size-1 counts: size-1 tuple counts are measured only up to a count of 3, giving a heuristic value equal to the minimum count. The heuristic value derived from size-2 tuples with minimum occurrence count n is then $n+4$. Size-1 tuple counts up to 3 are thus prioritized. This is done to distinguish states with a novelty-1 value, and provide tie-breaking for novelty-2 nodes. The heuristic is also partitioned according to the $\#g$ and $\#r$ partition functions, as was the case with C_1 and W_2. For clarity, we refer to this heuristic as $AC_{1,2}$ in this chapter.

8.2.1.2 Lifted Feature Extraction Count-Based Ranking

In classical planning, there exists a distinction between *grounded variables* and *lifted variables*. Grounded variables refer to specific instances of variables that have been instantiated with concrete values or objects, and correspond to the atoms used represent planning states in this study. Lifted variables, on the other hand, are abstract and not yet instantiated; they represent general placeholders that can be bound to specific values during the planning process. For example, in a classical planning problem, there may be a lifted variable, or predicate, $OnChair(T, C)$, where T and C are placeholders, which then defines a set of possible grounded variables $\{OnTable(Table1, Cat1), `OnTable(Table1, Cat2), ...\}$, for a set of available objects $\{Table1, Table2, ..., Cat1, Cat2, ...\}$.

Given a state s, a set of features is extracted from the set of atoms in s; such set consists of an enumeration for each lifted variable in the problem of the number of grounded variables which correspond to the set defined by that lifted variable. The ordering of the count corresponding to each lifted predicate is not important, but must be consistent.

Example 8.2. *For a STRIPS planning problem with predicates* $\{OnTable(T, C), OnFloor(C), InBox(B, C)\}$*, with* T*,* C *and* B *being placeholders for 'Table', 'Cat' and 'Box' objects, and a state* s' *defined through grounded variables* $s' = \{OnTable(Table1, Cat1), OnFloor(Cat2), OnFloor(Cat3)\}$*, the set of extracted lifted features is* $[1, 2, 0]$*, corresponding to 1 '*$OnTable$*' variable, 2 '*$OnFloor$*' variables, and 0 '*$InBox$*' variables in* s'*.*

The extracted lifted features share conceptual similarities with sketch rules defined for various domains in Drexler et al. [47], as both approaches count the number of lifted variables of a particular type present in a state. However, they differ in that the extracted lifted features are automatically generated from any state without specifying a starting state or goal. This method therefore does not define a direction of search but partitions the state-space into an alternative feature space.

A count-based heuristic for *lifted feature counts* LFC is then constructed by counting the occurrence of the lifted feature representation of a state, and states whose lifted feature representation has a lower count are prioritized. That is, for state s' with lifted features $[1, 2, 0]$, it counts the number of times the same set of lifted features $[1, 2, 0]$ was observed in previous states. Thus, unlike all other count-based heuristics defined in this work, the counts are not over occurrence of tuples of underlying variables, but rather over the full extracted state representation. The LFC heuristic also separates the counts in partitions, as previous W_2 and C_1 heuristics, adopting the same $\#g$ and $\#r$ partition functions.

Since the same lifted features values often repeat in different states, counting the occurrence of the full set of features is feasible and empirically more effective than counting tuple occurrences over extracted features. This characteristic creates a count-based metric that more closely resembles discrete count-based exploration bonuses in RL that count the occurrence of states, such as UCB1 [48] and MBIE-EB [13]. Intuitively, the heuristic ranks higher the states with a greater exploration bonus with respect to the extracted lifted features, albeit also adopting partition functions to indirectly influence the direction of exploration.

8.2.1.3 BFNoS Variants

We implement the $AC_{1,2}$ and LFC heuristics into variants of the BFNoS solver presented in Chapter 5, proposing two new BFNoS variants.

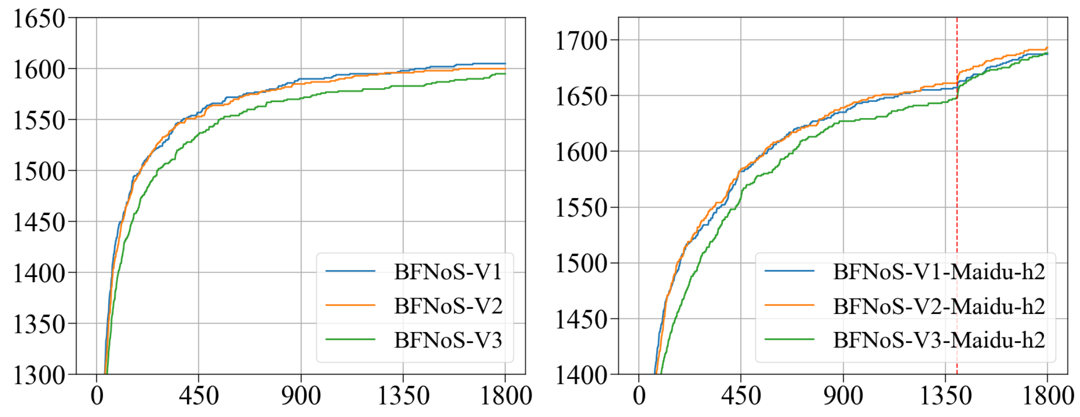
The first variant is $\text{BFNoS}_t(f_5(C_1), f_5(LFC))_{8:1}$. This solver differs from the $\text{BFNoS}_t(f_5(C_1), f_5(W_2))$ solver presented in previous chapters in two main characteristics. Firstly, it substitutes the width-based W_2 heuristic with the newly introduced LFC heuristic. The solver thus foregoes any means of directly measuring the presence of novel size-2 tuples. The second crucial difference is found in the '$8 : 1$' subscript, which refers to the solution of expanding 8 nodes from the first open list, with evaluation function $f_5(C_1)$, for every 1 node expanded by the second open list with evaluation function $f_5(LFC)$. The two open lists are therefore expanded asymmetrically, with more weight placed on the first one, and thus on the count-based novelty heuristic C_1.

The second variant, described BFNoS$_t(f_5(AC_{1,2}), f_5(LFC))_{8:1}$, further modifies the former through the use of the other heuristic proposed in this chapter, $AC_{1,2}$, and maintaining the LFC secondary open list heuristic and the 8-to-1 primary-secondary open list expansion ratio.

8.2.1.4 Analysis of BFNoS Variants

Solver	**BFNoS-only**	**BFNoS-Maidu-h^2; memory threshold only (6GB)**	**BFNoS-Maidu-h^2; memory + time threshold (6GB, 1400 *sec*)**
	All IPC Benchmark Domains		
BFNoS$_t(f_5(C_1), f_5(W_2))$	**1600±3.9 (83.32%±0.18)**	1662±4.72 (88.02%±0.21)	1688±3.29 (89.79%±0.22)
BFNoS$_t(f_5(C_1), f_5(LFC))_{8:1}$	1595±6.11 (82.92%±0.39)	**1669±3.61 (88.46%±0.21)**	**1692±1.15 (90.12%±0.10)**
BFNoS$_t(f_5(AC_{1,2}), f_5(LFC))_{8:1}$	1588±6.24 (82.49%±0.41)	1658±4.51 (87.76%±0.32)	1688±1.00 (89.84%±0.09)
	IPC 2023		
BFNoS$_t(f_5(C_1), f_5(W_2))$	87±1.10	87±1.30	98±0.45
BFNoS$_t(f_5(C_1), f_5(LFC))_{8:1}$	84±0.58	85±1.15	97±0.00
BFNoS$_t(f_5(AC_{1,2}), f_5(LFC))_{8:1}$	**91±0.58**	**91±1.53**	**103±1.15**
	IPC 2018		
BFNoS$_t(f_5(C_1), f_5(W_2))$	**149±1.34**	159±1.52	166±2.17
BFNoS$_t(f_5(C_1), f_5(LFC))_{8:1}$	148±3.21	**170±0.58**	**171±1.15**
BFNoS$_t(f_5(AC_{1,2}), f_5(LFC))_{8:1}$	141±3.06	159±1.00	165±0.58

TABLE 8.1: Coverage comparison of BFNoS and BFNoS-Maidu-h^2 variants. % score is the average of the % of instances solved in each individual domain, and is provided for results calculated over the full set of benchmark domains. The best results for each solver class are highlighted in bold.



(A) BFNoS-only variants comparison (B) Hybrid variants comparison

FIGURE 8.1: Instance coverage vs. time (*sec*). Comparison of BFNoS variants discussed in this section, and the respective BFNoS-Maidu-h^2 configurations. BFNoS-V1 is BFNoS$_t(f_5(C_1), f_5(W_2))$, BFNoS-V2 is BFNoS$_t(f_5(C_1), f_5\ (LFC))_{8:1}$, and BFNoS-V3 is BFNoS$_t(f_5(AC_{1,2}), f_5(LFC))_{8:1}$. All BFNoS variants use the full 8 *GB* memory allowance when ran on their own in Figure (A), and 6 *GB* memory threshold and 1400 *sec* time thresholds when ran as the frontend solver in the in BFNoS-Maidu-h^2 configurations in Figure (B). The red vertical line in Figure (B) signals the time threshold.

Both BFNoS$_t(f_5(C_1), f_5(LFC))_{8:1}$ and BFNoS$_t(f_5(AC_{1,2}), f_5(LFC))_{8:1}$ demonstrate similar, but marginally lower coverage in Table 8.1 when ran on their own, compared to the original $\text{BFNoS}_t(f_5(C_1), f_5(W_2))$ variant. On the other hand, the dual-configuration $\text{BFNoS}_t(f_5(C_1), f_5(LFC))_{8:1}$-Maidu-$h^2$ outperforms all other solvers, achieving the best overall coverage and %-score of all solvers evaluated in this thesis, albeit by a very small margin.

These results provide interesting insights. Firstly, we show that a "better" frontend solver does not guarantee improved results in the dual configuration. Other factors, including the orthogonality between frontend and backend coverage, and the frontend solver's characteristics with respect to memory failures (as discussed in Section 7.3.2), overpower $\text{BFNoS}_t(f_5(C_1), f_5(W_2))$'s advantage both in the memory-only and time+memory dual configurations. $\text{BFNoS}_t(f_5(AC_{1,2}), f_5(LFC))_{8:1}$-Maidu-$h^2$ also manages to catch up to $\text{BFNoS}_t(f_5(C_1), f_5(W_2))$-Maidu-$h^2$ thanks to the time threshold, however as mentioned in the previous chapter, this is a weaker result than matching the memory-only performance, as a larger number of challenging problems are solved much later in time, as shown in Figure 8.1b.

The second key insight is that both BFNoS variants adopting the LFC heuristic in a secondary open list with asymmetric expansion, can almost match the performance of $\text{BFNoS}_t(f_5(C_1), f_5(W_2))$ without the need for a width-novelty heuristic, complementing the primary $f_5(C_1)$ open list heuristic with similar effectiveness. This is surprising, as the W_2 heuristic is known to be powerful on its own, and has extensive theoretical analysis behind its performance [6]. The fact that a count-based approach proves to be an effective means of implementing an exploratory heuristic that does not rely on the use of tuples – employed in classical count-based novelty, and all previous literature on width-novelty – is an important result in its own right. Such results provide a positive outlook on the potential of studying alternative exploratory heuristics which may be extracted from the state representation, and the adoption of count-based novelty as a more general means of promoting the exploration of the state space in specific dimensions.

Addressing the lower performance of $\text{BFNoS}_t(f_5(AC_{1,2}), f_5(LFC))_{8:1}$, a main identified limitation of the $AC_{1,2}$ heuristic is its slower evaluation time and the greater memory usage compared to C_1, which cause $\text{BFNoS}_t(f_5(AC_{1,2}), f_5(LFC))_{8:1}$ to lag behind the other two variants in Figure 8.1a. This is caused by the constant factor increase in tuples whose count must be checked and incremented, as well as the need to evaluate the set of added atoms in a state. $\text{BFNoS}_t(f_5(C_1), f_5(LFC))_{8:1}$, on the other hand, denotes a trend which is on par with that of $\text{BFNoS}_t(f_5(C_1), f_5(W_2))$.

Still, while $AC_{1,2}$'s negatives outweigh its overall benefits, it does demonstrate strong performance in specific subsets of domains. $\text{BFNoS}_t(f_5(AC_{1,2}), f_5(LFC))_{8:1}$ is the best solver tested so far on the IPC 2023 dataset, solving 91 instances on average on its own, and outperforms every other solver, including benchmark solvers and dual-configuration BFNoS variants in Table 7.3, in 'folding', 'agricola' and 'flashfill' domains, where on average it solves 13, 19 and 20 instances. Interestingly, there seem to be some trade-offs between the performances of $\text{BFNoS}_t(f_5(C_1), f_5(LFC))_{8:1}$ and $\text{BFNoS}_t(f_5(AC_{1,2}), f_5(LFC))_{8:1}$ in IPC 2023 and IPC 2018 datasets, causing the two variants to achieve complementary results across the two datasets.

The BFNoS variants adopting alternative count-based heuristics described in this section thus offer an interesting set of strengths and weaknesses compared to the base $\text{BFNoS}_t(f_5(C_1), f_5(W_2))$ solver, and solutions that diverge more markedly from previous work on width and novelty in Classical Planning. Further study of these differences may provide valuable insights into count-based novelty and the structure of specific domains. These results also suggest promising future directions for expanding the set of exploratory heuristics in Classical Planning. This could lead to the development of powerful new novelty heuristics, and search algorithms that leverage the complementary strengths of these heuristics to improve performance across a range of challenging problems.

8.2.2 Other Future Directions

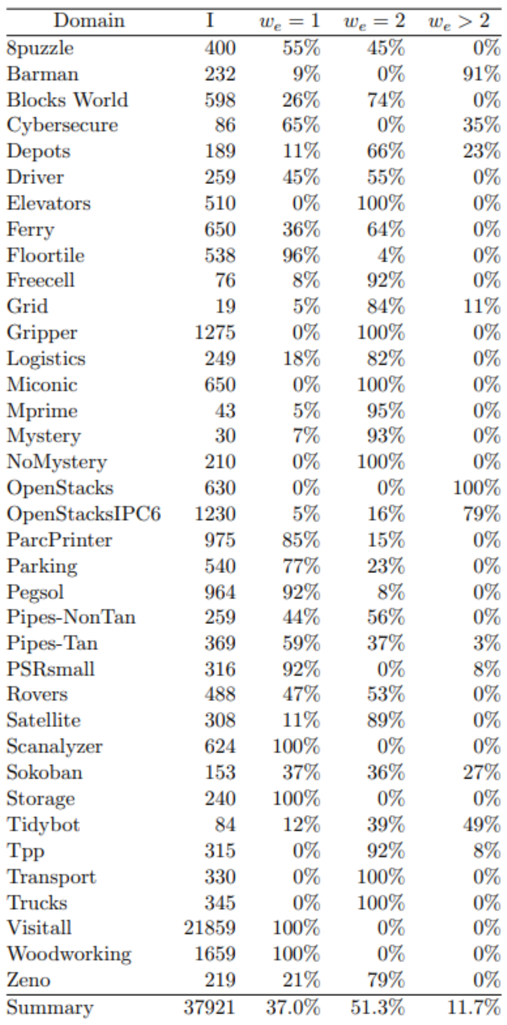
In Section 3.6 we provide an initial link between classical count-based novelty and pseudocounts as defined by Bellemare et al. [53]. This presents a basis for bridging count-based exploration techniques across classical planning and RL. The current count-based novelty framework is only defined over discrete and binary state representations. Expanding on those results, we believe that leveraging knowledge transfer from RL count-based exploration methods, which offer multiple techniques that tackle numerical and continuous state-space representations, is a promising avenue to generalize our own count-based novelty framework to numerical and continuous state features, potentially making it adoptable across planning sub-fields that extend beyond classical planning.

We believe that research gaps still exist also in reference to our results on memory thresholds. Time and memory thresholds have been implemented as separate limits in our hybrid solvers. Further study and development of the technique can merge the time and memory limits, and potentially additional factors, into a single threshold that relies on a probability-of-failure measure of a planner. This may improve the performance of dual-configuration planners, but also crucially provide developments to the large body of work that exists on *portfolio planners*[1], which currently rely heavily on time thresholds, enabling individual solver run-times that vary more dynamically based on the underlying solver characteristics and planning problem.

[1]In classical planning, a portfolio planner is a system that selects and combines multiple planning algorithms to solve a given problem by leveraging their complementary strengths.

Appendix A

Atomic Width of Domains



Domain	I	$w_e = 1$	$w_e = 2$	$w_e > 2$
8puzzle	400	55%	45%	0%
Barman	232	9%	0%	91%
Blocks World	598	26%	74%	0%
Cybersecure	86	65%	0%	35%
Depots	189	11%	66%	23%
Driver	259	45%	55%	0%
Elevators	510	0%	100%	0%
Ferry	650	36%	64%	0%
Floortile	538	96%	4%	0%
Freecell	76	8%	92%	0%
Grid	19	5%	84%	11%
Gripper	1275	0%	100%	0%
Logistics	249	18%	82%	0%
Miconic	650	0%	100%	0%
Mprime	43	5%	95%	0%
Mystery	30	7%	93%	0%
NoMystery	210	0%	100%	0%
OpenStacks	630	0%	0%	100%
OpenStacksIPC6	1230	5%	16%	79%
ParcPrinter	975	85%	15%	0%
Parking	540	77%	23%	0%
Pegsol	964	92%	8%	0%
Pipes-NonTan	259	44%	56%	0%
Pipes-Tan	369	59%	37%	3%
PSRsmall	316	92%	0%	8%
Rovers	488	47%	53%	0%
Satellite	308	11%	89%	0%
Scanalyzer	624	100%	0%	0%
Sokoban	153	37%	36%	27%
Storage	240	100%	0%	0%
Tidybot	84	12%	39%	49%
Tpp	315	0%	92%	8%
Transport	330	0%	100%	0%
Trucks	345	0%	100%	0%
Visitall	21859	100%	0%	0%
Woodworking	1659	100%	0%	0%
Zeno	219	21%	79%	0%
Summary	37921	37.0%	51.3%	11.7%

FIGURE A.1: Atomic width of solving serialized single atom subgoals over a set of benchmark planning domains. I is the number of resulting instances. Other columns denote the number of solved atomic instances using iterative width with width w_e. From Lipovetzky [2].

Appendix B

Extended Proofs

Theorem 3.7. *Lower and upper bounds for $\alpha_{0:t}(n^c)$ are given by:*

$$\alpha_{0:t}(n^p) - \frac{t-1}{t}\frac{e}{L} \leq \alpha_{0:t}(n^c) \leq \alpha_{0:t}(n^p) + \frac{t-1}{t}\frac{e}{L}$$

Proof. When comparing the Hamming distances of n^p and n^c with respect to a third node n', the greatest decrease in Hamming distances e, corresponding to all e effects changing variables v_i where $n^p(v_i) \neq n'(v_i)$ and $n^c(v_i) = n'(v_i)$. Thus in the lower bound we get that all e effect variables change their corresponding valuation to match with all states in history except for the parent node, reducing Hamming distance to each state by 1 for each effect e. The upper bound is symmetric, and we take into account the fact that the distance of n^c to n^p is already accounted as the distance of n^p to n^c in $\alpha_{0:t}(n^p)$, since $n^c = n_t$, and it thus does not change. Since parent and child states share all variable valuations except for e effects, which change valuation from parent to child node. This yields, for all cases where $n' \in n_{0:t}$, $n' \neq n^c$ and $n' \neq n^p$:

$$\delta(n^c, n') \geq \frac{1}{L}(H(n^p, n') - e) = \delta(n^p, n') - \frac{e}{L} \tag{B.1}$$

$$\delta(n^c, n^p) = \frac{1}{L}(H(n^p, n^p) + e) = \frac{e}{L} \tag{B.2}$$

We can redefine the average $\alpha_{0:t}(n^c)$ as

$$\alpha_{0:t}(n^c) = \frac{1}{t}\left[\sum_{i=0;n_i\notin\{n^p,n^c\}}^{t}\Big(\delta(n^c,n_i)\Big) + \delta(n^c,n^p)\right] \tag{B.3}$$

We have that, for $t-1$ comparisons, $\sum_{j=1}^{t-1}\alpha_{0:t-1}(n^p) = \sum_{j=1}^{t-1}\frac{1}{t-1}\sum_{i=0;n_i\neq n^p}^{t-1}\delta(n^p,n_i) = \sum_{i=0;n_i\neq n^p}^{t-1}\delta(n^p,n_i)$, therefore we can update the average for n^p which includes node $n^c = n_t$:

$$\begin{aligned}\alpha_{0:t}(n^p) &= \frac{1}{t}\left[\sum_{j=1}^{t-1}\alpha_{0:t-1}(n^p) + \delta(n^c,n^p)\right]\\ &= \frac{1}{t}\left[\sum_{i=0;n_i\neq n^p}^{t-1}\Big(\delta(n^p,n_i)\Big) + \delta(n^c,n^p)\right]\end{aligned} \tag{B.4}$$

Substituting (B.1) into (B.3), noting that $\sum_{i=0;n_i\notin\{n^p,n^c\}}^{t}(\frac{e}{L}) = (t-1)\frac{e}{L}$, and that $\sum_{i=0;n_i\notin\{n^p,n^c\}}^{t}\big(\delta(n^p,n_i)\big) = \sum_{i=0;n_i\notin\{n^p\}}^{t-1}\big(\delta(n^p,n_i)\big)$ since $n^c = n_t$, we obtain:

$$\begin{aligned}\alpha_{0:t}(n^c) &\geq \frac{1}{t}\left[\sum_{i=0;n_i\notin\{n^p,n^c\}}^{t}\Big(\delta(n^p,n_i) - \frac{e}{L}\Big) + \delta(n^c,n^p)\right]\\ &= \frac{1}{t}\left[\sum_{i=0;n_i\notin\{n^p\}}^{t-1}\Big(\delta(n^p,n_i)\Big) + \delta(n^c,n^p)\right] - \frac{t-1}{t}\frac{e}{L}\end{aligned} \tag{B.5}$$

Substituting (B.4) into (B.5) we obtain

$$\alpha_{0:t}(n^c) \geq \alpha_{0:t}(n^p) - \frac{t-1}{t}\frac{e}{L} \tag{B.6}$$

For the upper bound, we note that it is symmetrical in that in the upper bound all effects e are novel, that is for some effect variable v_i we have that $n^c(v_i) \neq n'(v_i)$ for all $n' \in n_{0:t-1}$, thus we get $\delta(n^c,n') \leq \delta(n^p,n') + \frac{e}{L}$. Following the same procedure yields the upper bound. □

Example B.1. *For states of size* $L = 3$*, we give a history of* t *nodes, one of which the parent. All* $t-1$ *nodes that are not the parent node* n^p *are represented by state vector [1,0,1]. Parent node* n^p *is represented by [1,1,0]. For an action with 2 effects,* $e = 2$*, knowing that the Hamming distance between parent and child node must be* $e = 2$ *by assumption, then the greatest decrease occurs when child node* n^c *also has value*

[1,0,1]. Parent node n^p has average normalized Hamming distance of $\frac{2}{3}$ to all other nodes, where 3 is given by $L = 3$. Child node n^c has average normalized Hamming distance of $\frac{1}{t}\frac{2}{3} = \frac{t}{t}\frac{2}{3} - \frac{t-1}{t}\frac{2}{3} = \alpha_{0:t}(n^p) - \frac{t-1}{t}\frac{2}{L}$.

Theorem 3.8. *Upper bound $\alpha_{0:t}(n^c)$ with respect to $\mu = \mu_{t-1}^{min}(n^c)$ is given by:*

$$\alpha_{0:t}(n^c) \leq \alpha_{0:t}(n^p) + \frac{t-1}{t}\frac{e(1-2\mu)}{L}$$

Proof. We seek to maximize the Hamming distance of the child node with respect to its parent by minimizing the number of nodes in history $n' \in n_{0:t-1}$ where value $n^c(v_i) = n'(v_i)$ for effect variables v_i. Since μ is the minimum feature occurrence, this acts as a constraint, and the upper bound occurs when all effects e have occurrence equal to the minimum occurrence μ. Thus there are $(1-\mu)\cdot(t-1)$ nodes in which, for effect variables v_i, $n'(v_i) \neq n^c(v_i)$ and $n'(v_i) = n^p(v_i)$, and $\mu \cdot (t-1)$ nodes in which $n'(v_i) = n^c(v_i)$ and $n'(v_i) \neq n^p(v_i)$. Thus, for $t-1$ nodes $n' \in n_{0:t-1}$, $n' \neq n^p$, we have that $\delta(n^c, n') = \frac{1}{L}(H(n^p, n') + e)$ a total of $(1-\mu)\cdot(t-1)$ times, and $\delta(n^c, n') = \frac{1}{L}(H(n^p, n') - e)$ a total of $\mu \cdot (t-1)$ times.

The summation over all comparisons becomes:

$$\begin{aligned} &\sum_{i=0; n_i \notin \{n^p, n^c\}}^{t} \Big(\delta(n^c, n_i)\Big) \\ &= \sum_{i=0; n_i \notin \{n^p, n^c\}}^{t} \Big(\delta(n^p, n_i)\Big) + (1-\mu)(t-1)\frac{e}{L} - \mu(t-1)\frac{e}{L} \\ &= \sum_{i=0; n_i \notin \{n^p\}}^{t-1} \Big(\delta(n^p, n_i)\Big) + (t-1)\cdot\frac{e(1-2\mu)}{L} \end{aligned} \tag{B.7}$$

Substituting (B.7) into (B.3) as in Theorem 3.7 we obtain:

$$\begin{aligned} \alpha_{0:t}(n^c) &\leq \frac{1}{t}\Bigg[\sum_{i=0; n_i \notin \{n^p\}}^{t-1} \Big(\delta(n^p, n_i)\Big) + (t-1)\cdot\frac{e(1-2\mu)}{L} + \delta(n^c, n^p)\Bigg] \\ &= \frac{1}{t}\Bigg[\sum_{i=0; n_i \notin \{n^p\}}^{t-1} \Big(\delta(n^p, n_i)\Big) + \delta(n^c, n^p)\Bigg] + \frac{t-1}{t}\frac{e(1-2\mu)}{L} \end{aligned} \tag{B.8}$$

Substituting (B.4) into (B.8) we obtain:

$$\alpha_{0:t}(n^c) \leq \alpha_{0:t}(n^p) + \frac{t-1}{t}\frac{e(1-2\mu)}{L}$$

□

Theorem 3.9. *Lower bound for $\alpha_{0:t}(n^c)$ when $\mu_{t-1}^{min}(n^c) = 0$ is given by:*

$$\alpha_{0:t}(n^c) \geq \alpha_{0:t}(n^p) - \frac{t-1}{t}\frac{e-2}{L}$$

Proof. In the lower bound, one effect of the action from parent to child node is constrained to be novel, resulting in a Hamming distance of $+1$ compared to the parent node, and $e-1$ effects match all previous history except n^p, resulting in a hamming distance of -1 compared to the parent. Thus we have that:

$$\begin{aligned}\delta(n^c, n') &= \frac{1}{L}(H(n^p, n') - (e-1) + 1) \\ &= \delta(n^p, n') - \frac{e-2}{L}\end{aligned} \tag{B.9}$$

Inserting (B.9) into (B.3) and following the derivation from Theorem 3.7 yields Theorem 3.9. □

Proposition 3.13. *Given a tuple size k and an underlying set of tuples $U = U^{(k)}$ over features V of states in S, if $\rho_t = \mu_t^{min}(s)$, then the pseudocount $\hat{N}_t = \min_{u\in U}(N_t^u(s_{t+1}))$.*

Proof. This proof adopts the notation described in Section 3.6. The pseudocount function (as defined in [53], introduced in Section 2.6.4.3) is defined

$$\hat{N}_t(s) = \frac{\rho_t(s)(1-\rho'_t(s))}{\rho'_t(s) - \rho_t(s)} \tag{B.10}$$

and the minimum empirical count distribution is $\mu_t^{min}(s) = \min_{u\in U}(\mu_t^u(s))$, where $\mu_t^u(s) = \frac{N_t^u(s)}{t}$ is the tuple empirical count distribution for u. We define the minimum count as

$$N_t^{min}(s) := \min_{u\in U}(N_t^u(s))$$

thus we have that

$$\mu_t^{min}(s) = \min_{u \in U}(\mu_t^u(s)) = \frac{\min_{u \in U}(N_t^u(s))}{t} = \frac{N_t^{min}(s)}{t}$$

The prediction gain of the minimum empirical count distribution is given by

$$(\mu_t^{min})'(s) = \mu_t^{min}(s; s_{1:t} \cdot s) = \frac{N_t^{min}(s) + 1}{t + 1}$$

as it simply increments the minimum count and the total count by 1. Substituting $\rho_t(s) = \mu_t^{min}(s)$ and $\rho_t'(s) = (\mu_t^{min})'(s)$ into Equation B.10 we then solve the pseudocount function, proving the proposition:

$$\begin{aligned}
\hat{N}_t(s) &= \frac{\frac{N_t^{min}(s)}{t}\left(1 - \frac{N_t^{min}(s)+1}{t+1}\right)}{\frac{N_t^{min}(s)+1}{t+1} - \frac{N_t^{min}(s)}{t}} \\
&= \frac{N_t^{min}(s)(t - N_t^{min}(s))}{\left(\frac{N_t^{min}(s)+1}{t+1} - \frac{N_t^{min}(s)}{t}\right)(t)(t+1)} \\
&= \frac{N_t^{min}(s)(t - N_t^{min}(s))}{\left(\frac{t(N_t^{min}(s)+1) - N_t^{min}(s)(t+1)}{t(t+1)}\right)t(t+1)} \\
&= \frac{N_t^{min}(s)(t - N_t^{min}(s))}{t - N_t^{min}(s)} \\
&= N_t^{min}(s)
\end{aligned} \tag{B.11}$$

$\square$

Appendix C

Extended Tables of Results

Domain	BFNoS	Dual-BFWS	Apx-BFWS (Tarski)	LAMA-First	Maidu	Maidu with h^2	BFNoS-Dual-back	BFNoS-LAMA	BFNoS-Maidu-h^2
agricola-sat18-strips	15±0.0	12	18±0.9	12	12	13	15±0.5	15±0.5	15±0.5
airport	47±0.6	46	47±0.6	34	38	45	46±0.6	46±0.5	46±0.6
assembly	30±0.0	30	30±0.0	30	30	30	30±0.0	30±0.0	30±0.0
barman-sat14-strips	20±0.0	20	20±0.0	20	20	20	20±0.0	20±0.0	20±0.0
blocks	35±0.0	35	35±0.0	35	35	35	35±0.0	35±0.0	35±0.0
caldera-sat18-adl	18±0.0	19	19±0.6	16	16	16	16±0.0	17±0.5	18±0.0
cavediving-14-adl	8±0.5	8	8±0.5	7	7	7	8±0.0	8±0.0	8±0.5
childsnack-sat14-strips	1±1.1	9	5±0.6	6	6	6	8±0.0	6±0.0	6±0.5
citycar-sat14-adl	20±0.0	20	20±0.0	5	6	6	20±0.0	20±0.0	20±0.0
data-network-sat18-strips	17±0.6	13	19±0.5	13	16	16	16±0.8	15±1.1	16±0.8
depot	22±0.0	22	22±0.0	20	22	22	22±0.0	22±0.0	22±0.0
driverlog	20±0.0	20	20±0.0	20	20	20	20±0.0	20±0.0	20±0.0
elevators-sat11-strips	20±0.0	20	20±0.0	20	20	20	20±0.0	20±0.0	20±0.0
flashfill-sat18-adl	14±1.3	17	15±1.6	14	15	14	17±0.5	16±0.6	16±0.9
floortile-sat14-strips	2±0.5	2	2±0.0	2	2	20	2±0.0	2±0.0	20±0.0
folding	9±0.0	5	5±0.5	11	11	11	9±0.0	9±0.0	9±0.0
freecell	80±0.0	80	80±0.0	79	80	80	80±0.0	80±0.0	80±0.0
ged-sat14-strips	20±0.0	20	20±0.0	20	20	20	20±0.0	20±0.0	20±0.0
grid	5±0.0	5	5±0.0	5	5	5	5±0.0	5±0.0	5±0.0
gripper	20±0.0	20	20±0.0	20	20	20	20±0.0	20±0.0	20±0.0
hiking-sat14-strips	20±0.0	18	20±0.0	20	20	20	20±0.0	20±0.0	20±0.0
labyrinth	15±0.5	5	18±0.5	1	0	2	15±0.5	15±0.5	15±0.5
logistics00	28±0.0	28	28±0.0	28	28	28	28±0.0	28±0.0	28±0.0
maintenance-sat14-adl	17±0.0	17	17±0.0	11	13	13	17±0.0	17±0.0	17±0.0
miconic	150±0.0	150	150±0.0	150	150	150	150±0.0	150±0.0	150±0.0
movie	30±0.0	30	30±0.0	30	30	30	30±0.0	30±0.0	30±0.0
mprime	35±0.0	35	35±0.0	35	35	35	35±0.0	35±0.0	35±0.0
mystery	18±0.6	19	19±0.0	19	19	19	19±0.0	19±0.0	19±0.0
nomystery-sat11-strips	14±0.8	19	14±0.5	11	19	18	19±0.0	15±0.6	17±0.0
nurikabe-sat18-adl	16±0.6	14	17±0.5	9	11	16	17±0.6	17±0.0	18±0.0
openstacks-sat14-strips	20±0.0	20	20±0.6	20	20	20	20±0.0	20±0.0	20±0.0
organic-synthesis-split-sat18-strips	8±0.5	12	8±0.0	14	14	14	11±0.5	14±0.0	14±0.9
parcprinter-sat11-strips	9±0.6	16	11±1.3	20	20	20	20±0.0	20±0.0	20±0.0
parking-sat14-strips	20±0.0	20	20±0.0	20	20	20	20±0.0	20±0.0	20±0.0
pathways	26±0.9	30	28±1.1	23	25	25	30±0.0	27±0.8	27±0.7
pegsol-sat11-strips	20±0.0	20	20±0.0	20	20	20	20±0.0	20±0.0	20±0.0
pipesworld-notankage	50±0.0	50	50±0.0	43	45	45	50±0.0	50±0.0	50±0.0
pipesworld-tankage	43±1.6	42	44±0.6	43	43	43	43±0.8	43±0.5	43±0.6
psr-small	50±0.0	50	50±0.0	50	50	50	50±0.0	50±0.0	50±0.0
quantum-layout	20±0.0	20	20±0.0	20	20	20	20±0.0	20±0.0	20±0.0
recharging-robots	14±0.6	12	14±0.8	13	13	13	14±0.5	14±0.0	14±0.5
ricochet-robots	20±0.5	20	18±0.6	14	18	18	20±0.0	20±0.0	20±0.0
rovers	40±0.4	40	40±0.4	40	40	40	40±0.4	40±0.0	40±0.0
rubiks-cube	5±0.0	6	5±0.6	20	20	20	5±0.0	20±0.0	16±0.6
satellite	34±0.8	33	34±0.5	36	36	36	34±0.6	35±0.0	35±0.0
scanalyzer-sat11-strips	20±0.0	20	20±0.5	20	20	20	20±0.0	20±0.0	20±0.0
schedule	149±1.3	150	149±1.3	150	150	150	149±0.7	150±0.0	150±0.0
settlers-sat18-adl	13±1.5	7	12±0.7	17	18	18	12±0.5	17±0.0	17±0.5
slitherlink	5±0.6	5	5±0.7	0	0	0	5±0.5	3±0.6	4±0.7
snake-sat18-strips	20±0.0	17	20±0.0	5	14	14	20±0.0	20±0.0	20±0.0
sokoban-sat11-strips	15±1.1	17	14±0.9	19	19	20	15±0.5	19±0.0	20±0.0
spider-sat18-strips	17±1.3	16	16±1.1	16	16	17	18±0.0	18±0.0	18±0.9
storage	30±0.5	29	30±0.0	20	25	25	29±0.5	29±0.0	29±0.6
termes-sat18-strips	10±0.8	10	5±1.5	16	14	14	10±0.5	14±0.0	14±0.0
tetris-sat14-strips	20±0.0	17	20±0.0	16	17	20	20±0.0	20±0.0	20±0.0
thoughtful-sat14-strips	20±0.0	20	20±0.2	15	19	19	20±0.0	20±0.0	20±0.0
tidybot-sat11-strips	20±0.0	18	20±0.2	17	20	20	20±0.0	20±0.0	20±0.0
tpp	30±0.5	30	30±0.3	30	30	30	30±0.0	30±0.0	30±0.0
transport-sat14-strips	20±0.0	20	20±0.2	17	18	16	20±0.5	20±0.0	20±0.0
trucks-strips	8±0.8	19	13±1.5	18	20	22	17±0.5	16±0.0	20±0.0
visitall-sat14-strips	20±0.0	20	20±0.0	20	20	20	20±0.0	20±0.0	20±0.0
woodworking-sat11-strips	20±0.0	20	12±1.1	20	20	20	20±0.0	20±0.0	20±0.0
zenotravel	20±0.0	20	20±0.0	20	20	20	20±0.0	20±0.0	20±0.0
Coverage (1831)	1600±3.9	1603	1606±3.9	1535	1590	1626	1641±1.9	1662±2.3	1688±3.3
% Score (100%)	83.32% ±0.18	83.23%	83.51% ±0.27	79.06%	82.84%	85.31%	86.23% ±0.09	87.87% ±0.17	89.79% ±0.22
Front-end % coverage share	-	-	-	-	-	-	97%	96%	94%

TABLE C.1: Comparative performance analysis across the full set of benchmark domains. *% score* is the average of the % of instances solved in each domain. *Front-end % coverage share* refers to the % of covered instances solved by the BFNoS front-end. Values for BFNoS variants and Approximate-BFWS represent the mean and include the standard deviation across 5 measurements.

Domain	BFNoS-Dual-back	BFNoS-LAMA	BFNoS-Maidu	BFNoS-Maidu-h^2
agricola-sat18-strips	15±0.0	15±0.0	15±0.0	15±0.5
airport	47±0.6	47±0.6	47±0.6	47±0.6
assembly	30±0.0	30±0.0	30±0.0	30±0.0
barman-sat14-strips	20±0.0	20±0.0	20±0.0	20±0.0
blocks	35±0.0	35±0.0	35±0.0	35±0.0
caldera-sat18-adl	16±0.0	17±0.0	17±0.0	18±0.0
cavediving-14-adl	8±0.0	8±0.6	8±0.6	8±0.6
childsnack-sat14-strips	4±0.5	4±0.5	4±0.5	4±0.6
citycar-sat14-adl	20±0.0	20±0.0	20±0.0	20±0.0
data-network-sat18-strips	16±0.7	16±0.6	17±0.6	16±0.6
depot	22±0.0	22±0.0	22±0.0	22±0.0
driverlog	20±0.0	20±0.0	20±0.0	20±0.0
elevators-sat11-strips	20±0.0	20±0.0	20±0.0	20±0.0
flashfill-sat18-adl	17±0.5	16±0.0	15±0.5	15±0.5
floortile-sat14-strips	2±0.0	2±0.0	2±0.0	20±0.0
folding	9±0.0	10±1.0	10±0.6	9±0.6
freecell	80±0.0	80±0.0	80±0.0	80±0.0
ged-sat14-strips	20±0.0	20±0.0	20±0.0	20±0.0
grid	5±0.0	5±0.0	5±0.0	5±0.0
gripper	20±0.0	20±0.0	20±0.0	20±0.0
hiking-sat14-strips	20±0.0	20±0.0	20±0.0	20±0.0
labyrinth	15±0.5	15±0.5	15±0.5	15±0.5
logistics00	28±0.0	28±0.0	28±0.0	28±0.0
maintenance-sat14-adl	17±0.0	17±0.0	17±0.0	17±0.0
miconic	150±0.0	150±0.0	150±0.0	150±0.0
movie	30±0.0	30±0.0	30±0.0	30±0.0
mprime	35±0.0	35±0.0	35±0.0	35±0.0
mystery	19±0.0	19±0.0	19±0.0	19±0.0
nomystery-sat11-strips	19±0.0	14±0.8	17±0.0	17±0.0
nurikabe-sat18-adl	16±0.5	16±0.6	16±0.6	16±0.6
openstacks-sat14-strips	20±0.0	20±0.0	20±0.0	20±0.0
organic-synthesis-split-sat18-strips	11±0.7	14±0.6	13±0.5	13±1.3
parcprinter-sat11-strips	20±0.0	20±0.0	20±0.0	20±0.0
parking-sat14-strips	20±0.0	20±0.0	20±0.0	20±0.0
pathways	30±0.0	26±0.9	27±0.8	26±0.9
pegsol-sat11-strips	20±0.0	20±0.0	20±0.0	20±0.0
pipesworld-notankage	50±0.0	50±0.0	50±0.0	50±0.0
pipesworld-tankage	43±1.1	42±1.5	42±1.5	42±1.5
psr-small	50±0.0	50±0.0	50±0.0	50±0.0
quantum-layout	20±0.0	20±0.0	20±0.0	20±0.0
recharging-robots	14±0.6	14±0.6	14±0.6	14±0.6
ricochet-robots	20±0.0	20±0.6	20±0.0	20±0.0
rovers	40±0.5	40±0.5	40±0.5	40±0.5
rubiks-cube	5±0.0	5±0.0	5±0.0	5±0.0
satellite	33±0.6	34±1.1	35±0.0	34±1.0
scanalyzer-sat11-strips	20±0.0	20±0.0	20±0.0	20±0.0
schedule	148±1.3	148±1.3	148±1.3	148±1.3
settlers-sat18-adl	12±0.8	16±0.9	16±0.9	16±0.6
slitherlink	5±0.0	4±0.7	4±0.7	4±0.7
snake-sat18-strips	20±0.0	20±0.0	20±0.0	20±0.0
sokoban-sat11-strips	15±0.6	19±0.5	19±0.5	20±0.0
spider-sat18-strips	18±0.0	18±0.0	18±0.0	18±0.0
storage	29±0.6	29±0.6	29±0.5	29±0.6
termes-sat18-strips	10±0.5	11±0.8	11±0.5	11±0.5
tetris-sat14-strips	20±0.0	20±0.0	20±0.0	20±0.0
thoughtful-sat14-strips	20±0.0	20±0.0	20±0.0	20±0.0
tidybot-sat11-strips	20±0.0	20±0.0	20±0.0	20±0.0
tpp	30±0.0	30±0.0	30±0.0	30±0.0
transport-sat14-strips	20±0.0	20±0.0	20±0.0	20±0.0
trucks-strips	18±0.5	16±0.6	19±0.5	20±0.9
visitall-sat14-strips	20±0.0	20±0.0	20±0.0	20±0.0
woodworking-sat11-strips	20±0.0	20±0.0	20±0.0	20±0.0
zenotravel	20±0.0	20±0.0	20±0.0	20±0.0
Coverage (1831)	1636±3.3	1638±4.9	1643±3.7	1662±4.7
% Score (100%)	85.90%±0.14	86.13%±0.26	86.45%±0.17	88.02%±0.21
Front-end % coverage share	97%	97%	97%	96%

Table C.2: Comparative performance analysis across the full set of benchmark domains of 'memory-only' threshold dual-configuration BFNoS variants. *% score* is the average of the % of instances solved in each domain. *Front-end % coverage share* refers to the % of covered instances solved by the BFNoS front-end. Coverage values represent the mean and include the standard deviation across 5 measurements.

Domain	**BFNoS-Only**		**Memory-Only TH**		**Memory+Time TH**	
	BFNoS-V2	BFNoS-V3	BFNoS-V2-Maidu-h^2	BFNoS-V3-Maidu-h^2	BFNoS-V2-Maidu-h^2	BFNoS-V3-Maidu-h^2
agricola-sat18-strips	17±1.7	19±1.5	16±0.6	18±0.6	16±0.6	16±0.6
airport	46±0.6	47±0.0	46±0.0	47±0.0	46±0.0	47±0.0
assembly	23±1.0	27±0.6	30±0.0	30±0.0	30±0.0	30±0.0
barman-sat14-strips	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0
blocks	35±0.0	35±0.0	35±0.0	35±0.0	35±0.0	35±0.0
caldera-sat18-adl	18±0.0	18±0.6	18±0.0	17±1.0	18±0.0	17±0.6
cavediving-14-adl	8±0.0	7±0.0	8±0.0	7±0.0	8±0.0	7±0.0
childsnack-sat14-strips	1±0.6	2±1.0	2±0.0	2±1.0	6±0.0	6±0.0
citycar-sat14-adl	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0
data-network-sat18-strips	18±0.6	17±0.6	18±1.0	16±0.0	18±1.0	16±0.6
depot	22±0.0	22±0.0	22±0.0	22±0.0	22±0.0	22±0.0
driverlog	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0
elevators-sat11-strips	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0
flashfill-sat18-adl	18±0.0	20±0.6	18±0.0	19±0.6	18±0.0	19±0.6
floortile-sat14-strips	1±0.0	1±0.6	20±0.0	20±0.0	20±0.0	20±0.0
folding	9±0.0	13±0.0	9±0.0	13±0.6	9±0.0	14±0.0
freecell	80±0.0	80±0.0	80±0.0	80±0.0	80±0.0	80±0.0
ged-sat14-strips	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0
grid	5±0.0	5±0.0	5±0.0	5±0.0	5±0.0	5±0.0
gripper	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0
hiking-sat14-strips	20±0.6	20±0.0	20±0.6	20±0.0	20±0.0	20±0.0
labyrinth	15±0.0	15±0.0	15±0.0	15±0.0	15±0.0	15±0.0
logistics00	28±0.0	28±0.0	28±0.0	28±0.0	28±0.0	28±0.0
maintenance-sat14-adl	17±0.6	15±0.0	17±0.6	15±0.6	17±0.0	16±0.0
miconic	150±0.0	150±0.0	150±0.0	150±0.0	150±0.0	150±0.0
movie	30±0.0	30±0.0	30±0.0	30±0.0	30±0.0	30±0.0
mprime	34±0.0	34±0.0	34±0.0	34±0.0	35±0.0	35±0.0
mystery	19±0.6	19±0.0	19±0.0	19±0.0	19±0.0	19±0.0
nomystery-sat11-strips	13±0.6	13±0.0	17±0.0	17±0.0	17±0.0	17±0.0
nurikabe-sat18-adl	17±0.0	13±0.0	17±0.0	13±0.0	18±0.0	15±0.6
openstacks-sat14-strips	20±0.0	19±0.6	20±0.0	19±0.6	20±0.0	20±0.0
organic-synthesis-split-sat18-strips	8±0.0	8±0.0	15±0.0	14±0.6	15±0.6	14±0.6
parcprinter-sat11-strips	12±0.0	12±0.6	20±0.0	20±0.0	20±0.0	20±0.0
parking-sat14-strips	20±0.0	18±0.6	20±0.0	18±0.6	20±0.0	20±0.0
pathways	29±0.6	27±0.6	29±0.0	27±0.0	29±0.0	27±0.6
pegsol-sat11-strips	20±0.0	19±0.0	20±0.0	20±0.0	20±0.0	20±0.0
pipesworld-notankage	50±0.0	50±0.0	50±0.0	50±0.0	50±0.0	50±0.0
pipesworld-tankage	42±1.5	42±1.0	42±1.5	42±1.0	44±0.6	43±0.0
psr-small	50±0.0	50±0.0	50±0.0	50±0.0	50±0.0	50±0.0
quantum-layout	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0
recharging-robots	13±0.6	14±0.0	14±0.6	14±0.0	14±0.0	14±0.0
ricochet-robots	19±0.0	19±0.0	20±0.0	20±0.0	20±0.0	20±0.0
rovers	40±0.0	40±0.0	40±0.0	40±0.0	40±0.0	40±0.0
rubiks-cube	4±0.6	5±0.0	5±0.6	5±0.0	16±0.0	16±0.6
satellite	35±0.0	35±0.0	35±0.0	35±0.0	35±0.0	35±0.0
scanalyzer-sat11-strips	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0
schedule	150±0.0	147±1.2	150±0.0	147±1.2	150±0.0	150±0.0
settlers-sat18-adl	8±1.0	4±1.0	17±0.0	17±0.0	17±0.0	17±0.0
slitherlink	3±0.0	5±0.6	3±0.0	4±1.0	3±0.0	4±1.0
snake-sat18-strips	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0
sokoban-sat11-strips	14±1.1	13±0.6	20±0.0	20±0.0	20±0.0	20±0.0
spider-sat18-strips	18±0.0	16±0.0	18±0.0	16±0.0	18±0.0	17±0.0
storage	29±0.6	29±0.6	29±0.6	29±0.6	29±0.6	29±0.6
termes-sat18-strips	6±1.0	7±1.0	13±0.0	9±1.2	14±0.0	14±0.6
tetris-sat14-strips	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0	20±0.6
thoughtful-sat14-strips	19±0.0	20±0.0	19±0.0	20±0.0	19±0.0	20±0.0
tidybot-sat11-strips	20±0.0	20±0.6	20±0.0	20±0.6	20±0.0	20±0.0
tpp	30±0.0	29±0.0	30±0.0	30±0.6	30±0.0	30±0.0
transport-sat14-strips	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0
trucks-strips	10±1.5	12±0.6	18±0.6	20±0.6	20±0.0	20±0.0
visitall-sat14-strips	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0
woodworking-sat11-strips	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0
zenotravel	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0	20±0.0
Coverage (1831)	1595±6.1	1588±6.2	1669±3.6	1658±4.5	1692±1.2	1688±1.0
% Score (100%)	82.92%±0.39	82.49%±0.41	88.46%±0.21	87.76%±0.32	90.12%±0.10	89.84%±0.09
Front-end % coverage share	-	-	95%	95%	94%	93%

TABLE C.3: Comparative performance analysis across the full set of benchmark domains of BFNoS variants. BFNoS-V2 is $\text{BFNoS}_t(f_5(C_1), f_5\ (LFC))_{8:1}$, and BFNoS-V3 is $\text{BFNoS}_t(f_5(AC_{1,2}), f_5(LFC))_{8:1}$. *% score* is the average of the % of instances solved in each domain. *Front-end % coverage share* refers to the % of covered instances solved by the BFNoS front-end. Coverage values represent the mean and include the standard deviation across 3 measurements.

Bibliography

[1] Nir Lipovetzky. Ai planning for autonomy: Search algorithms. University Lecture Slides, 2022.

[2] Nir Lipovetzky. Structure and inference in classical planning. 2013.

[3] Tom Bylander. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[4] Daniel Fišer and Florian Pommerening. International planning competition 2023 classical tracks. In *International Planning Competition 2023*, 2023. [Online; accessed 02-September-2023].

[5] Nir Lipovetzky and Hector Geffner. Width and serialization of classical planning problems. In *ECAI 2012*, pages 540–545. IOS Press, 2012.

[6] Nir Lipovetzky. Planning for novelty: Width-based algorithms for common problems in control, planning and reinforcement learning. *arXiv preprint arXiv:2106.04866*, 2021.

[7] Nir Lipovetzky and Hector Geffner. Width-based algorithms for classical planning: New results. In *ECAI 2014*, pages 1059–1060. IOS Press, 2014.

[8] Nir Lipovetzky and Hector Geffner. Best-first width search: Exploration and exploitation in classical planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[9] Michael Katz, Nir Lipovetzky, Dany Moshkovich, and Alexander Tuisov. Adapting novelty to classical planning as heuristic search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 27, pages 172–180, 2017.

[10] Anubhav Singh, Nir Lipovetzky, Miquel Ramirez, and Javier Segovia-Aguas. Approximate novelty search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 349–357, 2021.

[11] Joschka Groß, Alvaro Torralba, and Maximilian Fickert. Novel is not always better: On the relation between novelty and dominance pruning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9875–9882, 2020.

[12] Simon Dold and Malte Helmert. Novelty vs. potential heuristics: A comparison of hardness measures for satisficing planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 20692–20699, 2024.

[13] Alexander L Strehl and Michael L Littman. An analysis of model-based interval estimation for markov decision processes. *Journal of Computer and System Sciences*, 74(8):1309–1331, 2008.

[14] Nir Lipovetzky and Hector Geffner. A polynomial planning algorithm that beats lama and ff. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 27, pages 195–199, 2017.

[15] Jörg Hoffmann and Bernhard Nebel. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[16] Nir Lipovetzky and Hector Geffner. Searching for plans with carefully designed probes. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 21, pages 154–161, 2011.

[17] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, Christian Muise, Ronald Brachman, Francesca Rossi, and Peter Stone. *An introduction to the planning domain definition language*, volume 13. Springer, 2019.

[18] Edsger Wybe Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[19] Silvia Richter and Matthias Westphal. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39: 127–177, 2010.

[20] Augusto B. Corrêa, Guillem Francès, Markus Hecher, Davide Mario Longo, and Jendrik Seipp. Scorpion Maidu: Width search in the Scorpion planning system. In *Tenth International Planning Competition (IPC-10): Planner Abstracts*, 2023.

[21] Michael Katz. Cerberus: Red-black heuristic for planning tasks with conditional effects meets novelty heuristic and enchanced mutex detection. *Ninth International Planning Competition (IPC-9): planner abstracts*, pages 47–51, 2018.

[22] Michael Katz, Nir Lipovetzky, Dany Moshkovich, and Alexander Tuisov. Merwin planner: Mercury enchanced with novelty heuristic. *IPC 2018 planner abstracts*, pages 53–56, 2018.

[23] Christer Bäckström. Five years of tractable planning. In *New directions in AI planning*, pages 19–33. 1996.

[24] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[25] David Chapman. Planning for conjunctive goals. *Artificial intelligence*, 32(3):333–377, 1987.

[26] Kutluhan Erol, Dana S Nau, and VS Subrahmanian. When is planning decidable? In *Artificial Intelligence Planning Systems*, pages 222–227. Elsevier, 1992.

[27] Stephen V Chenoweth. On the np-hardness of blocks world. In *AAAI*, pages 623–628, 1991.

[28] Jörg Hoffmann. Where ‘ignoring delete lists’ works: Local search topology in planning benchmarks. *Journal of Artificial Intelligence Research*, 24:685–758, 2005.

[29] Malte Helmert and Robert Mattmüller. Accuracy of admissible heuristic functions in selected planning domains. In *Proceedings of the 23rd national conference on Artificial intelligence-Volume 2*, pages 938–943, 2008.

[30] Jendrik Seipp, Florian Pommerening, Gabriele Röger, and Malte Helmert. Correlation complexity of classical planning domains. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 3242–3250. IJCAI/AAAI Press, 2016.

[31] Stefan Edelkamp. Planning with pattern databases. In *Proceedings of the Sixth European Conference on Planning (ECP 2001)*, 2001.

[32] Patrik Haslum and Héctor Geffner. Admissible heuristics for optimal planning. In *AIPS*, pages 140–149. Citeseer, 2000.

[33] Florian Pommerening, Malte Helmert, Gabriele Röger, and Jendrik Seipp. From non-negative to general operator cost partitioning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 29, 2015.

[34] Hubie Chen and Omer Giménez. Act local, think global: Width notions for tractable planning. In *ICAPS*, pages 73–80. Citeseer, 2007.

[35] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[36] David Hall, Alon Cohen, David Burkett, and Dan Klein. Faster optimal planning with partial-order pruning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 23, pages 100–108, 2013.

[37] Alvaro Torralba and Jörg Hoffmann. Simulation-based admissible dominance pruning. In *IJCAI*, pages 1689–1695, 2015.

[38] Dominik Drexler, Jendrik Seipp, and Hector Geffner. Learning sketches for decomposing planning problems into subproblems of bounded width. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, pages 62–70, 2022.

[39] Alexander Tuisov and Michael Katz. The fewer the merrier: Pruning preferred operators with novelty. In *ICAPS 2021 Workshop on Heuristics and Search for Domain-independent Planning*, 2021.

[40] Michael Katz and Alexander Tuisov. Tftm-co1 planner: Pruning preferred operators with novelty. 2023.

[41] Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks revisited. In *AAAI*, volume 8, pages 975–982, 2008.

[42] Dimitri P Bertsekas. Dynamic programming and optimal control. *Journal of the Operational Research Society*, 47(6):833–833, 1996.

[43] Gabriele Röger and Malte Helmert. The more, the merrier: Combining heuristic estimators for satisficing planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 20, pages 246–249, 2010.

[44] Vidal Alcázar and Alvaro Torralba. A reminder about the importance of computing and exploiting invariants in planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 25, pages 2–6, 2015.

[45] Ayal Taitler, Ron Alford, Joan Espasa, Gregor Behnke, Daniel Fišer, Michael Gimelfarb, Florian Pommerening, Scott Sanner, Enrico Scala, Dominik Schreiber, et al. The 2023 international planning competition, 2024.

[46] Blai Bonet and Hector Geffner. General policies, representations, and planning width. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11764–11773, 2021.

[47] Dominik Drexler, Jendrik Seipp, and Hector Geffner. Expressing and exploiting the common subgoal structure of classical planning domains using sketches: Extended version. *arXiv preprint arXiv:2105.04250*, 2021.

[48] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47:235–256, 2002.

[49] Marco Wiering and Jürgen Schmidhuber. Efficient model-based exploration. pages 223–228, 1998.

[50] Sham Machandranath Kakade. *On the sample complexity of reinforcement learning.* University of London, University College London (United Kingdom), 2003.

[51] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[52] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[53] Marc G Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. *Advances in neural information processing systems*, 29, 2016.

[54] Georg Ostrovski, Marc G Bellemare, Aäron Oord, and Rémi Munos. Count-based exploration with neural density models. In *International conference on machine learning*, pages 2721–2730. PMLR, 2017.

[55] Jarryd Martin, S Suraj Narayanan, Tom Everitt, and Marcus Hutter. Count-based exploration in feature space for reinforcement learning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 2471–2478, 2017.

[56] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[57] Nir Lipovetzky, Miquel Ramirez, and Hector Geffner. Classical planning with simulators: Results on the atari video games. In *Proc. IJCAI*, 2015.

[58] Guillem Frances, Miquel Ramirez, Nir Lipovetzky, and Hector Geffner. Purely declarative action representations are overrated: Classical planning with simulators. In *International Joint Conferences on Artificial Intelligence Organization (IJCAI)*, 2017.

[59] Miquel Junyent, Anders Jonsson, and Vicenç Gómez. Deep policies for width-based planning in pixel domains. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 646–654, 2019.

[60] Thomas M Moerland, Joost Broekens, Aske Plaat, and Catholijn M Jonker. A unifying framework for reinforcement learning and planning. *Frontiers in Artificial Intelligence*, 5:908353, 2022.

[61] Jürgen Schmidhuber. A possibility for implementing curiosity and boredom in model-building neural controllers. In *'From animals to animats: proceedings of the first international conference on simulation of adaptive behavior'*, 1991.

[62] Florent Teichteil-Königsbuch, Miquel Ramirez, and Nir Lipovetzky. Boundary extension features for width-based planning with simulators on continuous-state domains. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 4183–4189, 2021.

[63] Augusto B. Corrêa, Guillem Francès, Markus Hecher, Davide Mario Longo, and Jendrik Seipp. Scorpion maidu satisficing ipc2023-classical. https://github.com/

ipc2023-classical/planner8/tree/ipc2023-classical, 2023. Accessed: 2024-04-20.

[64] Miquel Ramirez, Nir Lipovetzky, Christian Muise, Anubhav Singh, and Giacomo Rosa. Lightweight Automated Planning ToolKiT - Extended with BFNoS Solvers. https://github.com/grosa97/LAPKT-BFNoS, 2024. Accessed: 2024.

[65] Miquel Ramirez, Nir Lipovetzky, and Christian Muise. Lightweight Automated Planning ToolKiT. http://lapkt.org/, 2015. Accessed: 2024.

[66] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl-the planning domain definition language. 1998.

[67] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[68] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. https://doi.org/10.5281/zenodo.790461, 2017.

[69] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward lab, 2017.

[70] Jendrik Seipp, Thomas Keller, and Malte Helmert. Saturated cost partitioning for optimal classical planning. *Journal of Artificial Intelligence Research*, 67:129–167, 2020.

[71] Malte Helmert. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.

[72] Guillem Francés, Miquel Ramirez, and Collaborators. Tarski: An AI planning modeling framework. https://github.com/aig-upf/tarski, 2018.

[73] Anubhav Singh, Nir Lipovetzky, Miquel Ramirez, and Javier Segovia-Aguas. Approximate novelty search - technical appendix, 2021.

[74] Daniel Fiser and Rebecca Eifler. Ricochet robots pddl domain. https://github.com/ipc2023-classical/domain-ricochet-robots, 2023. Accessed: 2024-04-20.

[75] Anubhav Singh, Nir Lipovetzky, Miquel Ramirez, and Javier Segovia-Aguas. Approximate novelty search ipc2023-classical. *Proceedings International Planning Competition (IPC-10)*, 2023.

[76] Florian Pommerening and Alvaro Torralba. International planning competition 2018 - classical tracks. https://ipc2018-classical.bitbucket.io/, 2018. Accessed: 2024-04-20.

[77] Mojtaba Elahi and Jussi Rintanen. Optimizing the optimization of planning domains by automatic action schema splitting. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence*, 2024.