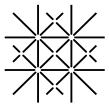


University of Basel

Preferred Operators for Abstraction Heuristics

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence Group
https://ai.dmi.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Florian Pommerening

Carina Fehr
carina.fehr@unibas.ch
2023-054-505

01.06.2026

Acknowledgments

Most of all, I want to thank Dr. Florian Pommerening for his enormous patience and for inspiring me with his seemingly endless knowledge. I deeply appreciate all his detailed explanations, valuable input and interesting ideas. I would also like to thank Prof. Dr. Malte Helmert for giving me the opportunity to work on this thesis in his research group. Thanks also to my family and friends for their support and for always giving me hope.

Abstract

Preferred operators have been shown to be successful in greedy best-first search, but so far they have not been used with abstraction heuristics. In this thesis, we introduce preferred operators for abstraction heuristics. Abstraction heuristics estimate goal distances by solving simplified versions of the original planning task. Preferred operators are identified as operators that lie on an optimal abstract plan. They are implemented for several abstraction heuristics in Fast Downward: single pattern databases, canonical pattern databases and Cartesian abstractions generated by CEGAR. Furthermore, we implement a live version of preferred operators, where they are determined during search, and a precomputed version, where they are computed during abstraction creation.

We evaluate the impact of preferred operators on different performance metrics in greedy best-first search. The experiments show that the usefulness of preferred operators depends on the quality of the heuristic and the employed search strategy. In eager greedy best-first search, preferred operators only introduce additional overhead, providing nearly no benefit. In contrast, lazy greedy best-first search benefits from preferred operators, especially when used with weaker abstraction heuristics. Precomputed preferred operators require more preprocessing time and memory, making them less practical compared to the live approach. Overall, the results indicate that preferred operators are most useful in settings where the search is weakly guided and produces high numbers of state expansions.

Table of Contents

1 Introduction

Classical planning is the task of finding a sequence of operators that transforms an initial state into a goal state. In classical planning, the reachable state space often grows exponentially with the size of the problem, which makes it infeasible to search the entire space. Consequently, uninformed search algorithms such as Dijkstra's algorithm are impractical for most complex planning tasks.

To navigate these large search spaces efficiently, heuristic search is employed. Heuristics estimate the remaining cost to reach a goal state, influencing the order in which states are expanded. Modern planning systems such as Fast Downward [6] implement different heuristics and heuristic search algorithms.

One important class of heuristics in classical planning is the class of abstraction heuristics. These heuristics are based on simplifying the original planning task by constructing an abstract version of the state space in which certain distinctions between states are ignored. The solution cost in the abstract space is then used as a heuristic estimate for the original problem.

Beyond heuristic values, preferred operators can provide additional guidance during search. Preferred operators are operators that are considered particularly promising from a given state. In many approaches, they are landmarks or part of a relaxed plan. In satisficing planners, preferred operators are often used to prioritize certain successor states during search [9]. By giving priority to these successors, the search can be directed more strongly toward promising regions of the state space, which has been shown to significantly improve performance.

In this thesis, we discuss preferred operators that are derived from abstraction heuristics. When computing an abstraction heuristic, we obtain not only a heuristic value for each abstract state, but also the optimal distances between abstract states. By analysing the abstract state space, we can identify operators that lie on an optimal path towards the abstract goal state. We define the corresponding concrete operators as preferred in the original task. We analyse whether this type of guidance improves the performance of greedy best-first search (GBFS).

In Chapter 2, we will introduce the necessary formal definitions. In Chapter 3, we extend

preferred operators to abstraction heuristics. We describe two different strategies to identify and implement them. Then, in Chapter 4, we present our experimental evaluation of preferred operators in three different abstraction heuristics and analyse the results. Finally, Chapter 5 concludes the thesis.

2
Background

This chapter introduces the fundamental concepts on which this thesis builds. We formally define planning tasks and state spaces as the basis of classical planning and then present heuristics, preferred operators and abstractions.

2.1 Classical Planning

The objective of classical planning is to find a sequence of operators that will lead from an initial state to a goal state. A planning task is modelled as a search problem in a state space. Each state corresponds to a configuration of the environment, while operators represent how one state can be transformed into another.

Definition 1 *(SAS^+ planning task).* A SAS^+ planning task [1] is defined as a 4-tuple $\Pi = \langle V, O, I, g \rangle$ where:

- V is a finite set of state variables. Each state variable $v \in V$ is associated with its domain dom(v). The domain is a finite non-empty set of values. Let $D = \bigcup_{v \in V} dom(v)$. A partial state s is a partial assignment of the state variables: $V \nrightarrow D$ such that for all $v \in vars(s) : s(v) \in dom(v)$ where $vars(s)$ denotes the variables on which s is defined. A state is a partial state with $vars(s) = V$. A state s is consistent with a partial variable assignment f if $f(v) = s(v)$ for all $v \in vars(f)$.
- O is a finite set of operators. Each $o \in O$ is defined as a triple $\langle pre(o), \mathit{eff}(o), cost(o) \rangle$, where *pre(o)*, the preconditions, and *eff(o)*, the effects, are partial variable assignments, and $cost(o) \in \mathbb{R}_0^+$.
- I is the initial state.
- g is a goal. It is a partial assignment of the state variables. A state is a goal state if it is consistent with g.

An operator $o \in O$ is considered applicable in a state s if its precondition holds in that state, meaning that s is consistent with $pre(o)$. The state resulting in applying the applicable

operator o in s, written $s' = s[\![o]\!]$, is defined as:

$$s[\![o]\!](v) = \begin{cases} \mathit{eff}(o)(v) & \text{if } v \in \mathit{vars}(\mathit{eff}(o)), \\ s(v) & \text{otherwise} \end{cases}$$

A sequence of operators $\pi = \langle o_1, o_2, ..., o_n \rangle$ is applicable in a state s if there exists a sequence of states $\langle s_0, ..., s_n \rangle$ such that $s_0 = s$, o_i is applicable in s_{i-1} for all i, and $s_i = s_{i-1}[\![o_i]\!]$. A plan is a path that starts in the initial state and ends in a goal state. The cost of a plan is the sum of the costs of its operators. A plan with minimal cost is called an optimal plan.

Definition 2 *(State space).* A state space is a 6-tuple $\mathcal{S} = \langle S, O, \mathit{cost}, T, s_I, S_G \rangle$ where:

- S is a finite set of states,
- O is a finite set of operators,
- $\mathit{cost} : A \to \mathbb{R}_0^+$ are the operator costs,
- $T \subseteq S \times O \times S$ is the transition relation,
- $s_I \in S$ is the initial state,
- $S_G \subseteq S$ is the set of goal states.

An operator $o \in O$ is applicable in a state $s \in S$ if there exists a state s' such that $\langle s, o, s' \rangle \in T$. In this case, s' is the successor state of s under operator o. A solution is a sequence of operators $\pi = \langle o_1, ..., o_n \rangle$ if there exists a state sequence $\langle s_0, ..., s_n \rangle$ such that $s_0 = s_I$, $\langle s_i, o_{i+1}, s_{i+1} \rangle \in T$ for all i, and $s_n \in S_G$. The cumulative cost of a sequence of transitions is obtained by summing the costs of the corresponding operators: $C(\pi) = \sum_{i=1}^{n} \mathit{cost}(o_i)$. An optimal solution is one with minimal cost.

To solve a planning task, we need to define how such a task induces a state space.

Definition 3 *(induced state space).* Given a planning task $\Pi = \langle V, O, I, g \rangle$, the induced state space $\mathcal{S}(\Pi) = \langle S, O, \mathit{cost}, T, s_I, S_G \rangle$ is defined as a tuple where:

- S is the set of all states over V,
- the operators are the same,
- the cost function is as defined in O,
- T contains all tuples $\langle s, o, s' \rangle$ where o is applicable in s and $s' = s[\![o]\!]$,
- $s_I = I$,
- S_G is the set of all states consistent with g.

An optimal solution in the induced state space corresponds to an optimal plan in the planning task.

2.2 Heuristics

A heuristic for a state space $\mathcal{S}$ is a function $h : S \to \mathbb{R}_0^+ \cup \{\infty\}$, assigning a non-negative number or infinity to each state. It estimates the cost of the optimal plan from a given state s to the nearest goal state. For each state $s \in S$, the perfect heuristic function h^* maps

to the optimal plan cost if one exists. Otherwise, it maps to ∞. A heuristic h is called admissible if $h(s) \leq h^*(s)$ for all states $s \in S$.
Heuristic search algorithms use heuristic functions to guide the exploration of the state space towards goal states. A prominent example in classical planning is greedy best-first search (GBFS). In each step, GBFS selects for expansion a state s with minimal heuristic value $h(s)$, thereby expanding the currently most promising node. GBFS ignores the path cost $g(s)$ from the initial state to s and relies only on the heuristic estimate.
The standard version of GBFS is called *eager*. In eager GBFS, the heuristic value is computed for every successor when it is generated. If heuristic evaluation is deferred until a node is selected for expansion, GBFS is called *lazy* [9]. In lazy GBFS, newly generated successors are inserted into the open list with the heuristic value of their parent state. The true heuristic value is only computed when the state is selected for expansion [10]. While GBFS is not optimal, it is commonly used in satisficing search due to its computational efficiency in quickly finding a solution.

2.3 Abstraction Heuristics

Abstractions reduce the size of the state space induced by a planning task by ignoring certain distinctions between states.
Definition 4 *(Abstractions).* Let $\Pi = \langle V, O, I, g \rangle$ be a planning task with induced state space $\mathcal{S} = \langle S, A, \mathit{cost}, T, s_I, S_G \rangle$. An abstraction function $\alpha : S \to S'$ maps states of $\mathcal{S}$ to abstract states. The abstraction of $\mathcal{S}$ induced by α is the state space $\mathcal{S}^\alpha = \langle S', A, \mathit{cost}, T', s'_I, S'_G \rangle$:

- The state space $\mathcal{S}^\alpha$ preserves all operators and operator costs of $\mathcal{S}$,
- $T' = \{\langle \alpha(s), o, \alpha(s') \rangle \mid \langle s, o, s' \rangle \in T\}$: for every transition $t = \langle s, o, s' \rangle \in T$ there is a corresponding transition $t' = \langle \alpha(s), o, \alpha(s') \rangle \in T'$,
- $s'_I = \alpha(s_I)$,
- $S'_G = \{\alpha(s) \mid s \in S_G\}$

The abstraction heuristic induced by α, denoted by h^α, is defined as follows: for every state s, the value $h^\alpha(s)$ equals the minimal cost $h^*_{\mathcal{S}^\alpha}(\alpha(s))$ of any path in the abstract state space $\mathcal{S}^\alpha$ that leads from the abstract state $\alpha(s)$ to an abstract goal state. Every abstraction heuristic h^α is admissible, meaning it can underestimate but never overestimate the true cost to a goal state [7]. This follows because every transition in the concrete state space $\mathcal{S}$ corresponds to a transition or a self-loop with the same cost in the abstract space $\mathcal{S}^\alpha$. Hence, any concrete path π with cost $h^*(s)$ from s to a goal state induces an abstract path from $\alpha(s)$ to an abstract goal state with cost of $h^*(s)$. Since $h^\alpha(s)$ is the minimal cost of such abstract paths, we obtain $h^\alpha(s) \leq h^*(s)$.

2.3.1 Pattern Databases

The most common abstraction heuristics are pattern database heuristics (PDBs) [3]. A PDB is constructed by selecting a subset of state variables, which are then preserved with perfect precision, while the distinctions involving all other variables are ignored. PDB heuristics are formalized as projections to a pattern $P \subseteq V : \pi_P(s) = \{v \mapsto s(v) \mid v \in P\}$. Given a planning task and subset of its variables P, consider the abstraction defined by the projection

π_P onto P. The corresponding abstraction heuristic is called pattern database heuristic with pattern P.

A pattern database is a look-up table that stores $h^*_{\mathcal{S}^\alpha}(s')$ for every state s' of the abstract task. It is typically implemented as a one-dimensional, zero-indexed vector of size $N = \prod_{i=1}^{k} |dom(v_i)|$, where $|dom(v_i)|$ denotes the domain size of variable v_i in the pattern. Each entry in this vector corresponds to one abstract state. To allow efficient heuristic look-ups, abstract states are mapped to indices using perfect hash functions [4]. The indices are referred to as ranks, computing them is known as ranking, and the inverse process is called unranking. Assuming that variable domains are encoded as consecutive integers starting from zero, the rank of a state s can be computed as $\text{rank}(s) = \sum_{i=1}^{k} N_i s[v_i]$ where $N_i = \prod_{j=1}^{i-1} |dom(v_j)|$ can be precomputed. The corresponding unranking function retrieves the value of variable v_i from a rank r as $\text{unrank}(r)[v_i] = \lfloor \frac{r}{N_i} \rfloor \bmod |dom(v_i)|$ [14]. Both ranking and unranking can be performed in $O(k)$ time, while accessing the heuristic value for a given state after ranking is constant time. This allows fast retrieval of stored heuristic values during search.

2.3.2 Canonical Pattern Databases

A canonical pattern database heuristic extends the idea of individual pattern databases to a collection of multiple patterns. The usefulness of a single PDB heuristic is limited by its space requirements, as the size of a pattern database grows exponentially with the number of state variables included in the pattern. As a result, using a single pattern database heuristic quickly becomes infeasible for large planning tasks. Instead, several patterns are evaluated. This allows multiple smaller PDBs to be combined into a pattern collection.

A simple admissible and consistent heuristic estimate for two patterns P_1 and P_2 is the maximum of their heuristic values h^{P_1} and h^{P_2} [5]. However, when possible, it is preferable to use the sum of both heuristic values as a heuristic estimate, since $h^{P_1} + h^{P_2} \geq \max\{h^{P_1}, h^{P_2}\}$. This idea can also be extended to more than two patterns. The sum of multiple patterns is an admissible and consistent heuristic if there exists no operator that has an effect on two different variables from the patterns.

Consider a pattern collection $\mathcal{C} \subseteq 2^V$ for a planning task. We construct the compatibility graph for $\mathcal{C}$, where vertices correspond to patterns $P \in \mathcal{C}$ and there is an edge between two vertices if and only if no operator affects both incident patterns. Next, we compute the set of all maximal cliques $cliques(\mathcal{C})$ of the graph. These correspond to maximal additive subsets of $\mathcal{C}$.

The canonical heuristic [5] for $\mathcal{C}$ is defined as

$$h^{\mathcal{C}}(s) = \max_{\mathcal{D} \in cliques(\mathcal{C})} \sum_{P \in \mathcal{D}} h^P(s).$$

The canonical heuristic function is the best possible admissible heuristic that can be derived from $\mathcal{C}$ using the additivity criterion. It is also admissible and consistent for all choices of $\mathcal{C}$.

2.3.3 Counterexample Guided Abstraction Refinement for Cartesian Abstractions

Counterexample Guided Abstraction Refinement (CEGAR) is a technique that was originally introduced for model checking [2]. CEGAR incrementally constructs abstractions by refining

initially very coarse abstractions.
Seipp and Helmert proposed to use it in combination with Cartesian abstractions to construct abstraction heuristics that can be applied in classical planning [12]. Each abstract state is represented as a Cartesian set of concrete states. We call a set of states[1] for a planning task with variables $\langle v_1, ..., v_n \rangle$ Cartesian if it can be written as a product $A_1 \times A_2 \times ... \times A_n$, where $A_i \subseteq dom(v_i)$ for all $1 \leq i \leq n$. An abstraction is called Cartesian if all its abstract states are Cartesian sets. The abstraction function α maps each state $s \in S$ to the Cartesian set that contains s.
The procedure starts with a single abstract state that represents the entire concrete state space. CEGAR then identifies flaws in the current abstraction and refines only those parts that are responsible for them. Flaws can arise in different situations. First, it could be that an abstract transition cannot be realized by any concrete path. Second, even if the operator is applicable, the resulting concrete state may not be contained in the successor abstract state. Last, an abstract state may be classified as a goal state even though the corresponding concrete state is not a goal. To eliminate these flaws, the abstraction is refined by splitting abstract states along selected state variables. This is done by selecting a variable v_i and partitioning the corresponding set A_i into two subsets, one containing the values that are relevant for the flaw and a second subset containing all remaining values. The original abstract state is then replaced by two new Cartesian sets, where A_i is replaced once by each of the two subsets, while all other components remain unchanged. This increases the precision of the abstraction because each refinement step reduces the set of concrete states represented by an abstract state.
Due to time and memory limits, the refinement process is typically terminated before the problem is solved completely. Nevertheless, the result is an abstraction where the resulting heuristic $h^\alpha(s)$ is given by the optimal goal distance in the abstract state space $h^*_{\mathcal{S}^\alpha}(\alpha(s))$. This early termination saves resources because it takes more and more iterations to obtain further improvements as the number of CEGAR iterations grows.
Heuristics based on Cartesian abstractions allow more fine-grained abstractions than pattern databases. Pattern databases are a special kind of Cartesian abstractions where for all i, $A_i = \{s(v_i)\}$ if v_i is in the pattern P, and $A_i = dom(v_i)$ otherwise. Increasing the precision of the abstraction requires adding another variable to the pattern, which at least doubles the number of abstract states.

2.4 Preferred Operators

Preferred operators (POs) are operators identified by the heuristic as promising in a given state. As they are likely to be beneficial in the original problem, they can be preferred during search to guide exploration towards relevant parts of the state space.
Algorithm 1 shows how Fast Downward uses preferred operators in GBFS. It employs a *dual-queue* approach in which states generated by preferred operators (preferred successors) are inserted into a separate open list (line 19) in addition to the regular open list (line 17) [9]. The search alternates between these lists when selecting states for expansion (line 6). As a result, preferred successors are expanded earlier on average, while all generated states

[1] Technically, we defined a set of states as a set of total functions, while the Cartesian product is a set of tuples. We implicitly map tuples to the corresponding states in the obvious way.

remain eligible for expansion. A similar approach is the *boosted dual-queue* method: if the search algorithm identifies that preferred operators lead to improved heuristic values, the preferred queue is temporarily prioritized. During this phase, states for expansion are taken exclusively from the preferred queue for a fixed number of steps, typically around 1000, giving preferred successors stronger influence on the search direction. If another operator is found during this phase that leads to an improvement, the boosts accumulate and operators continue to be taken from this queue for even longer [10]. Boosting can be applied in both eager and lazy search. The default settings of Fast Downward enable boosting for lazy search, whereas eager search uses the standard dual-queue method without boosting [11]. Richter and Helmert showed that this configuration works best for four different admissible heuristics [9]. In an experiment, we tested eager search with boosting in abstraction heuristics. This did not result in any difference compared to eager search without boosting, so we will keep the default settings in all our further experiments.

Algorithm 1 Pseudocode for GBFS with preferred operators, dual-queue approach

```
1: BEGIN
2: INIT OpenList ← {initial_state}
3: INITPreferredOpenList ← {}
4: turn ← 0
5: while OpenList ≠ ∅ do
6:     if turn mod 2 = 0 and PreferredOpenList ≠ ∅ then
7:         s ← remove_min(PreferredOpenList)
8:     else
9:         s ← remove_min(OpenList)
10:    end if
11:    turn ← turn +1
12:    if is_goal(s) then
13:        return plan
14:    end if
15:    for each operator op in applicable_operators(s) do
16:        s' ← apply(op, s)
17:        insert s' into OpenList with priority h(s') (eager) or h(s) (lazy)
18:        if (op is preferred operator) then
19:            insert s' into PreferredOpenList
20:        end if
21:    end for
22: end while
23: return unsolvable
24: END
```

Preferred operators therefore do not change the heuristic value itself but influence the order in which states that have the same heuristic value (in eager search) or the same parent heuristic value (in lazy search) are explored. They provide additional information derived from the heuristic, which can improve practical search performance.

3

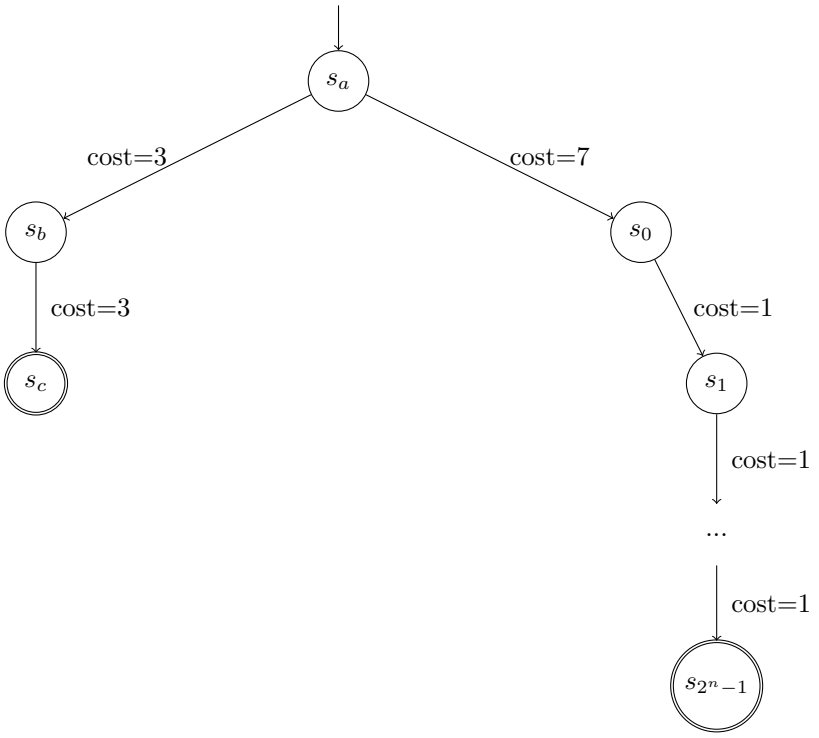
Preferred Operators for Abstraction Heuristics

In the Fast Downward planner, preferred operators are currently implemented only for inadmissible heuristics such as the additive heuristic. In this chapter, we extend this idea to abstraction heuristics. We first explain how preferred operators can be identified in abstraction heuristics and then how we implement and store them.

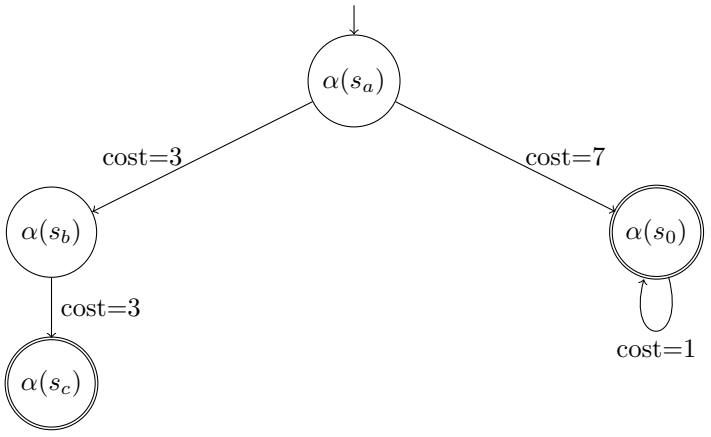
3.1 Identification of Preferred Operators

In abstraction heuristics, operators that lie on an optimal plan in the abstract state space are marked as preferred operators. While the abstraction heuristic primarily provides a numerical estimate of the remaining distance to a goal state, the structure of the abstract state space also contains information about which transitions are part of an optimal plan.
The following example illustrates the use of preferred operators. We consider a concrete state space $\mathcal{S} = \langle S, A, cost, T, s_I, S_G \rangle$. The state variables consist of a variable $x \in \{0, 1, 2, 3\}$ and n additional variables $y_1, ..., y_n \in \{0, 1\}$. The variables y_i can be interpreted as the bits of a binary counter: we write y for the integer encoded by the vector $(y_1, ..., y_n)$.
The operator costs, the transitions, the initial state and the goal states are illustrated in the graph below.

$$s_a = \{x \mapsto 0, y \mapsto 0\} \qquad s_b = \{x \mapsto 2, y \mapsto 0\}$$
$$s_c = \{x \mapsto 3, y \mapsto 0\} \qquad s_i = \{x \mapsto 1, y \mapsto i\}$$
$$\text{for } i \in \{0, ..., 2^n - 1\}$$



The left branch leads quickly to a goal, while the right branch enters a large subgraph in which the value of y is increasing step by step. With n boolean variables y_i, the number of concrete states with $x \mapsto 1$ is 2^n. We now consider a projection onto the single variable x. In this abstraction, all states with a different y value but the same x value are merged into a single abstract state. The resulting abstract state space is shown in the graph below.



$\alpha(s_a) = \{x \mapsto 0\}$ $\quad h(\alpha(s_a)) = 6$

$\alpha(s_b) = \{x \mapsto 2\}$ $\quad h(\alpha(s_b)) = 3$

$\alpha(s_0) = \alpha(s_1) = ... = \alpha(s_{2^n-1}) = \{x \mapsto 1\}$ $\quad h(\alpha(s_0)) = 0$

$\alpha(s_c) = \{x \mapsto 3\}$ $\quad h(\alpha(s_c)) = 0$

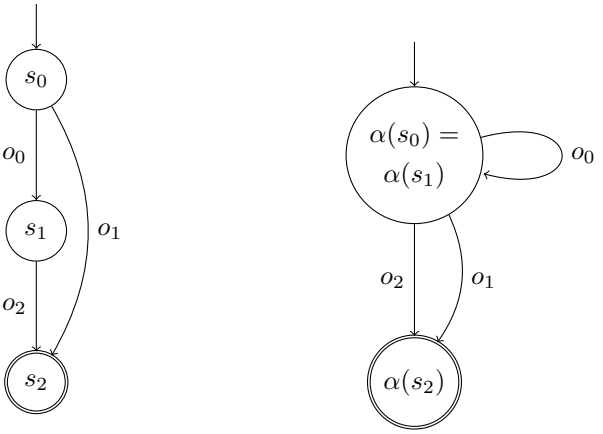
Consider GBFS using this abstraction as a heuristic. Starting from the initial state $\alpha(s_a)$, the heuristic value reflects the abstract distance to a goal. From the initial state, GBFS would choose the right successor with heuristic value 0, a goal state in the abstraction. GBFS bases its decision solely on the heuristic value and therefore prefers the state with the smallest heuristic estimate. However, in the concrete space, this successor is not a goal state.

The search therefore continues within the large subgraph where $x \mapsto 1$, repeatedly selecting successors with heuristic value 0. As a result, the search will expand an exponential number of states before finding a solution.

The left successor of the initial state, despite being reached by a cheaper operator and lying on the optimal path, is not selected because its heuristic value is higher. Only the operator with cost 3 leading to s_b lies on an optimal abstract path and is therefore identified as preferred. A search algorithm that prioritizes preferred operators will follow the left branch directly to a goal, expanding only two states (s_b and s_c) and finding a solution faster than if it relied purely on the heuristic value.

A similar situation occurs when a state has multiple successors with identical heuristic value. This commonly happens in lazy search, where all successors initially inherit the heuristic value of their parent. In such cases, GBFS provides no tie-breaking between successors and may select any of them. Preferred operators provide additional guidance by identifying which operators lie on an abstract plan, thereby reducing the risk of exploring large irrelevant parts of the search space.

Due to the loss of information introduced by the abstraction, preferred operators are not guaranteed to correspond to applicable transitions in the concrete state space. Since multiple concrete states are merged into a single abstract state, an operator may lie on the optimal path in the abstract space without being applicable in the concrete state space. The effect can be illustrated by the following planning task. The left graph shows the concrete state space, while the right graph shows the abstraction obtained by projecting onto the variable x. In this abstraction, both states with $x \mapsto 0$ are merged into a single abstract state. In the abstract state space, operator o_2 with cost 1 leads directly from the initial state to the goal state. Since this transition lies on the optimal abstract plan, it is identified as a preferred operator. However, in the concrete state space, o_2 is not applicable from the initial state, as it has $y \mapsto 1$ as a precondition. Due to the abstraction, this distinction is lost and the abstraction suggests the operator o_2 that appears optimal but whose preconditions are not satisfied in the current concrete state s_0.



$$s_0 = \{x \mapsto 0, y \mapsto 0\},\ s_1 = \{x \mapsto 0, y \mapsto 1\},\ s_2 = \{x \mapsto 1, y \mapsto 1\}$$
$$\text{cost}(o_0) = 1,\ \text{cost}(o_1) = 3,\ \text{cost}(o_2) = 1$$
$$\alpha(s_0) = \alpha(s_1) = \{x \mapsto 0\},\ h(\alpha(s_0)) = 1\ ,\ \alpha(s_2) = \{x \mapsto 1\},\ h(\alpha(s_2)) = 0$$

At any given state, only operators whose preconditions are satisfied in this state can be applied. Therefore, when identifying preferred operators, it is sufficient to consider only those operators that are applicable in this state. We will show that an operator which is not

applicable in the abstract state also cannot be applicable in the concrete state by a proof by contrapositive.

Proof. Consider a state space $\mathcal{S}$ and its abstraction $\mathcal{S}^\alpha$ induced by the abstraction function α. Suppose the operator o is applicable in the concrete state s. By Definition 2, this means there exists a concrete state transition $\langle s, o, s' \rangle \in T$. By Definition 4, for every transition $\langle s, o, s' \rangle \in T$ there exists a corresponding transition $\langle \alpha(s), o, \alpha(s') \rangle \in T'$. By Definition 2, the existence of $\langle \alpha(s), o, \alpha(s') \rangle$ means o is applicable in $\alpha(s)$. Therefore, by contrapositive, whenever an operator is not applicable in the abstract state $\alpha(s)$, it also cannot be applicable in the concrete state s.

We show that it is sufficient to consider only the first operator of each optimal abstract plan when identifying preferred operators. Let $\pi = \langle o_1, ..., o_n \rangle$ be an optimal abstract plan from $\alpha(s)$ to a goal state, and let o be some operator in π. We consider two cases:

Case 1: The operator o is not applicable in the current abstract state $\alpha(s)$ and therefore also not in the concrete state s. We can ignore this operator and not mark it as preferred. The only interesting situation is when such an operator is already applicable, as seen in the second case.

Case 2: o is applicable in $\alpha(s)$ and s. Let $\alpha(s')$ be the abstract successor state reached by applying o in $\alpha(s)$. If $\alpha(s') \neq \alpha(s)$, the suffix of π starting at o is a plan from $\alpha(s)$ to the goal. It costs at most as much as π and since π is optimal, this suffix must also be optimal. Therefore, o is the first operator of an optimal abstract plan starting from $\alpha(s)$. If $\alpha(s') = \alpha(s)$, then o has to be a zero-cost loop. Otherwise, removing it would yield a cheaper path, contradicting optimality. Removing o yields a plan of equal cost starting from the same state $\alpha(s)$, so o can again be seen as the first operator of an optimal plan and may be marked as a PO.

In both cases, every operator on an optimal abstract plan is either not applicable or can be considered the first operator of some optimal abstract plan. It is therefore sufficient to consider only the first operator of optimal abstract plans when evaluating preferred operators. Subsequent operators of an abstract plan will become relevant only in successor states, where they will be evaluated as first operators of the remaining plan.

The main advantage of preferred operators in abstraction heuristics is that they expose additional structural information about the abstract plan that the heuristic value alone does not show. While the heuristic value summarizes the estimated remaining cost to the goal, it does not indicate which transitions contribute to achieving it. By identifying operators that start an optimal path and marking them as preferred, the search algorithm can prioritize these operators rather than treating all successors with the same heuristic value as equal.

3.2 Precomputed Preferred Operators

To integrate preferred operators into abstraction heuristics, we extend the pattern database implementation in Fast Downward. In the *precomputed preferred operators* approach, POs are determined during pattern database generation.

The heuristic values stored in the pattern database are generated once prior to the search by running a backward Dijkstra search operating directly on ranks rather than explicit abstract states. Since abstract states are represented as indices and the ranking is a weighted sum of variable values, applying an operator always shifts the index by a fixed, precomputed hash

effect. This hash effect is constant regardless of which state the operator is applied to. Since the search proceeds backwards, operators are applied in reverse direction.
Algorithm 2 illustrates the backward Dijkstra search and how preferred operators are recorded during it. Starting from all abstract goal states, which are initialized with distance zero (lines 6-9), the algorithm repeatedly extracts the state with minimal distance from a priority queue (line 13). For each processed state, the algorithm retrieves all abstract operators whose application results in the current state, i.e., operators for which the current state is a successor (line 17). The algorithm determines the predecessor state corresponding to each of those operators by adding the hash effects to the current rank (line 19) and its distance value by adding the operator cost to the distance of the current state (line 20). When a shorter path to a predecessor state is discovered, the stored distance value of that state is updated (lines 21-22). The process continues until the priority queue is empty, at which point the resulting distance vector corresponds directly to the pattern database.
Our implementation of precomputed preferred operators integrates the identification of preferred operators into this backward search. Whenever a shorter path to a predecessor state is found, we update the set of preferred operators associated with that state. Since this new operator is part of the currently best path to the goal, the list of preferred operators for that state is cleared (line 24) and the corresponding operator is inserted as the new preferred operator (line 25).
If an alternative operator with the same optimal cost is discovered, the operator is added to the list of preferred operators of that state (lines 26-28). This ensures that all operators that contribute to shortest paths and therefore lie on optimal paths in the abstract state space are stored.
To validate the implementation, we create a distance test that can be applied in the precomputed version. It determines whether the heuristic value of the start state exactly matches the sum of the path when a random preferred operator is chosen in each step. The distance test was successful in all 1986 tested instances.

3.3 Live Preferred Operators

Another possible approach for computing POs in abstraction heuristics is *live preferred operators*. In this approach, preferred operators are not stored during abstraction construction. Instead, they are determined dynamically when the heuristic value of a state is requested during search.
Algorithm 3 illustrates how preferred operators are evaluated for a state s in abstraction heuristics such as single pattern databases, canonical pattern databases or Cartesian abstractions.
When the heuristic value is evaluated for a state, we generate all operators that are applicable in the current state (line 6). For each applicable operator, the corresponding successor state is computed (lines 7-8). The heuristic value of this successor state is then obtained by querying the abstraction heuristic (line 9). An operator lies on the optimal abstract path if its cost exactly matches the reduction in the heuristic value. Therefore, we consider this operator preferred if and only if it satisfies the equation $h(s) = cost(o) + h(s')$. Operators satisfying this condition contribute to an optimal path from the current abstract state to the goal and are therefore marked as preferred (lines 10-11). We prove both directions of this equivalence.

Algorithm 2 Pseudocode for precomputed preferred operators in backward Dijkstra search

```
1: BEGIN
2: INIT pq ← emtpy priority queue
3: INIT distances[s] ← ∞ for all states s
4:
5: for each state s do
6:     if s is goal state then
7:         distances[s] ← 0
8:         insert ⟨0, s⟩ into pq
9:     end if
10: end for
11:
12: while pq ≠ ∅ do
13:     ⟨distance, state⟩ ← remove_min(pq)
14:     if distance > distances[s] then
15:         continue
16:     end if
17:     applicable_ops ← operators whose successor is s
18:     for operator in applicable_ops do
19:         predecessor ← s + hash_effect(operator)
20:         alternative_cost ← distances[s] + cost(operator)
21:         if alternative_cost < distances[predecessor] then
22:             distances[predecessor] ← alternative_cost
23:             insert ⟨alternative_cost, predecessor⟩ into pq
24:             clear preferred_operators[predecessor]
25:             add operator to preferred_operators[predecessor]
26:         else if alternative_cost = distances[predecessor] then
27:             add operator to preferred_operators[predecessor]
28:         end if
29:     end for
30: end while
31: END
```

Proof. ($\Rightarrow$) Suppose o satisfies $h(s) = cost(o) + h(s')$. Let π' be the optimal abstract plan from $\alpha(s')$ to a goal state, with cost $h^*_{\mathcal{S}^\alpha}(\alpha(s')) = h(s')$. Since o is applicable in s, the transition $\langle \alpha(s), o, \alpha(s') \rangle \in T'$ exists. By prepending the operator o to this optimal plan π', we get π, which is a plan starting from $\alpha(s)$. Its total cost is $cost(o) + h(s') = h(s)$. Since $h(s)$ is the cost of the optimal abstract plan starting from $\alpha(s)$, no plan starting from this abstract state can have cost less than $h(s)$. Therefore π is an optimal abstract plan and since o is the first operator in this plan, o lies on an optimal abstract plan from $\alpha(s)$.
($\Leftarrow$) Let $h^*_{\mathcal{S}^\alpha}(\alpha(s))$ be the optimal cost from an abstract state $\alpha(s)$ to a goal state in $\mathcal{S}^\alpha$, and therefore $h(s) = h^*_{\mathcal{S}^\alpha}(\alpha(s))$. Suppose the operator o lies on an optimal abstract path from $\alpha(s)$ towards a goal state. Then there exists a transition $\langle \alpha(s), o, \alpha(s') \rangle \in T'$. Since o is part of the optimal plan, the remaining optimal cost from $\alpha(s')$ satisfies $h^*_{\mathcal{S}^\alpha}(\alpha(s')) = h^*_{\mathcal{S}^\alpha}(\alpha(s)) - cost(o)$. Therefore, we can conclude $h(s) = cost(o) + h(s')$.
For canonical PDBs and Cartesian abstractions, the heuristic is defined as the sum over multiple functions. Therefore, the heuristic value of each successor state has to be computed for all functions and aggregated. An operator is considered preferred if the total heuristic value decreases exactly by the operator's cost.
This live version avoids additional memory usage for storing preferred operators for all abstract states. However, it introduces additional computational overhead during search, as the optimality condition must be checked for every applicable operator whenever a state is expanded.

Algorithm 3 Pseudocode for live preferred operators for state s

```
1: BEGIN
2: INIT h ← get_value(s)
3: if h = ∞ then
4:     return DEAD_END
5: end if
6: applicable_operators ← generate_applicable_ops(s)
7: for each op in applicable_operators do
8:     successor ← get_successor(s, op)
9:     h_succ ← get_value(successor)
10:    if h = cost(op) + h_succ then
11:        set_preferred(op)
12:    end if
13: end for
14: END
```

4

Experiments

The goal of the experiments is to evaluate the impact of preferred operators on the performance of GBFS when used in combination with abstraction heuristics. We investigate whether preferred operators improve search efficiency and how their effect depends on the search strategy and the method used to compute them.

4.1 Experimental Setup

We run all tests using the Downward Lab Package [13]. The implementation was tested on the benchmarks from the satisficing tracks of all International Planning Competitions. We excluded all domains that use axioms and conditional effects, as these features are not supported by pattern databases.
Calculations were performed at sciCORE (http://scicore.unibas.ch/) scientific computing core facility at University of Basel. The experiments are tested with a time limit of 5 minutes and a memory limit of 3584 MB for each task. Our implementation of preferred operators is done on top of Fast Downward 24.06.1 [6]. Each run uses a single CPU core of an Intel Xeon Silver 4114 2.2 GHz Processor.
We compare different configurations of GBFS using abstraction heuristics. In particular, we consider three heuristics. For pattern database heuristics with a single pattern, we test all configurations with the greedy pattern generator. In this approach, a single pattern is constructed incrementally by adding variables until the size of the resulting abstract state space reaches a predefined size limit of one million states. The selection of variables follows an ordering based on goal variables and causal dependencies. Variables that are part of the goal are selected first, followed by variables that are causally connected to already selected ones.
The iPDB heuristic relies on a hill-climbing pattern collection generator using multiple patterns [5]. This method starts with a pattern for each goal variable and then iteratively improves the collection. In each step, candidate patterns are generated by extending existing patterns along causal graph connections. The pattern that yields the best improvement is included in the pattern collection. This process continues until no further improvement is made or the improvement is smaller than a predefined minimal improvement. The resulting

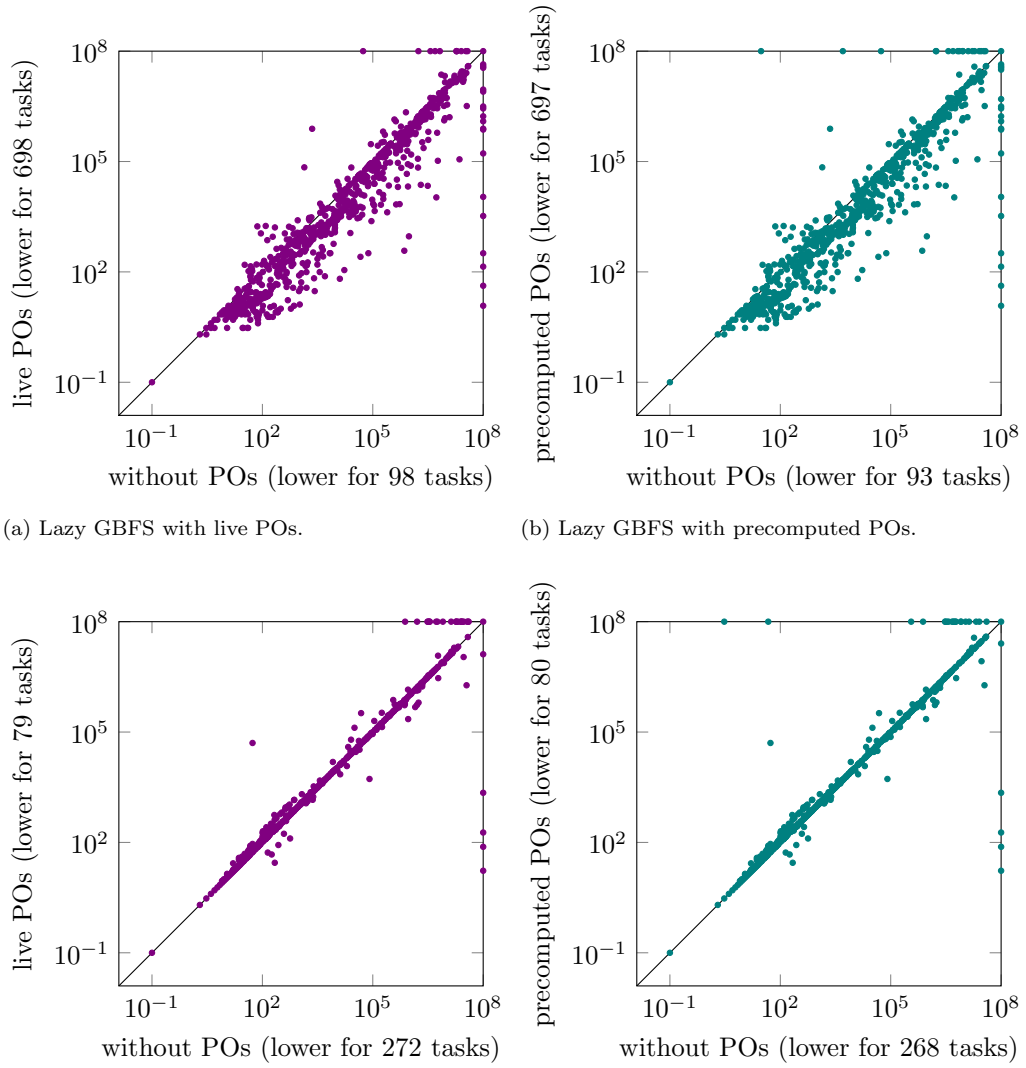
pattern collection is then used with the canonical PDB heuristic.
Cartesian abstraction heuristics use CEGAR to generate abstractions. Rather than building one abstraction, CEGAR decomposes the problem into several subtasks that focus on one subgoal each. For each subtask, a separate Cartesian abstraction is computed. The resulting heuristic values are made additive with cost partitioning.
First, we evaluate the performance of GBFS using a single PDB without preferred operators, with live preferred operators and with precomputed preferred operators, considering both lazy and eager GBFS. Additionally, we evaluate lazy GBFS using iPDB with live POs. Finally, we evaluate lazy GBFS using Cartesian abstractions with live POs.

4.2 Effect of Preferred Operators on Expansions

We first evaluate the impact of preferred operators on the number of expanded nodes. The number of expansions is a measure of how much of the state space the algorithm had to explore before finding a solution. If POs can help to guide the search better, the search algorithm should expand fewer states. We use expansion score to compare across different configurations. A score of 1 corresponds to the best possible performance of 100 expansions, while a score of 0 corresponds to either failure or worst-case performance (10^6 expansions) [13]. For solved tasks, the score is computed using a logarithmic scale between the lower bound and the upper bound. Higher score values indicate better performance.

4.2.1 Evaluation with Pattern Databases

Figure 4.1a compares the absolute number of expansions in lazy greedy best-first search using a pattern database heuristic with and without live POs. Each point represents a single planning problem from a specific domain. In 698 out of 818 commonly solved tasks, the configuration with POs results in fewer expansions, while in 98 tasks the search without POs performs better. A similar pattern can be observed when using precomputed POs, where 697 tasks show fewer expansions with POs and 93 tasks show fewer expansions without them (Figure 4.1b). This indicates that preferred operators reduce the number of expansions required in the majority of tasks for both PO variants. As expected, both variants guide the search comparably well in terms of expansions, but due to differences in runtime and memory overhead, they do not solve exactly the same set of tasks. Additionally, differences in tie-breaking can occur due to the order in which preferred operators are computed and stored, which may lead to small differences in the number of expansions.
In contrast, the effect of preferred operators in eager GBFS differs significantly. With live POs, expansions are lower in only 79 tasks, while they are higher in 272 out of 824 commonly solved tasks (Figure 4.1c). A similar pattern is observed in Figure 4.1d for precomputed POs. This indicates that, in about one third of all tasks, POs lead to higher expansion counts in eager GBFS and in nearly 60% of all tasks they have no influence on the number of expansions.
Table 4.1 shows the expansion scores for all four configurations. The difference between live and precomputed POs is negligible with respect to expansions. In eager GBFS, the scores change only marginally. This is consistent with the observation from the plots that both PO variants have only a very small impact on the number of expansions in eager search. When the guidance provided by the POs is accurate, the search may follow a better transition as



(a) Lazy GBFS with live POs.

(b) Lazy GBFS with precomputed POs.

(c) Eager GBFS with live POs.

(d) Eager GBFS with precomputed POs.

Figure 4.1: Expansions across all four configurations: eager and lazy GBFS with live and precomputed POs using a single PDB heuristic.

Configuration	w/o POs	with POs	Change
Lazy GBFS; Precomputed	459.06	507.44	+48.38
Lazy GBFS; Live	459.06	509.36	+50.30
Eager GBFS; Precomputed	519.80	519.36	-0.44
Eager GBFS; Live	519.80	521.23	+1.43

Table 4.1: Expansion scores across all four search configurations using a single PDB heuristic.

without POs. If POs are misleading, the search may first expand an additional state from the preferred queue before returning to the regular open list.
Overall, these results show that preferred operators are effective in reducing expansions and avoiding unnecessary exploration in lazy GBFS, but they do not provide consistent benefits in eager GBFS. In lazy GBFS, the heuristic alone cannot distinguish between successors, because all successors of a state are inserted into the open list with the parent's heuristic value. As a result, additional information, such as preferred operators, becomes more valuable. POs therefore provide meaningful guidance by highlighting promising successors that would otherwise appear equally good. In eager GBFS, each successor is assigned its own heuristic value, already providing a strong distinction between successors and reducing the benefit of additional guidance. This explains why our results show that preferred operators are most effective in lazy GBFS, while their impact is limited in eager GBFS. Based on this result, we will no longer consider eager GBFS in future evaluations and instead focus on lazy GBFS.

4.2.2 Evaluation with iPDB

When using an iPDB heuristic in lazy GBFS, the expansion score without preferred operators is 594.51, while enabling live POs increases it to 640.48 (+45.97). The absolute change in score is similar to that in the single pattern database setting, suggesting that the effect of POs persists when multiple patterns are used. Despite the already higher performance of iPDB without POs compared to single PDBs, preferred operators still provide additional guidance and reduce the number of expansions.

4.2.3 Evaluation with Cartesian Abstractions

When integrating preferred operators into the Cartesian abstraction heuristic, we do not observe an improvement in expansions. This indicates that preferred operators provide no additional guidance in this setting.
A closer look at individual domains shows that, in *miconic*, the largest domain with 150 of the 1986 benchmark problems, POs significantly harm the expansion score, decreasing it from 118.23 to 79.79 (-38.44). This strongly biases the overall score, as differences in the other domains are small (at most 8, and often even less), and the version with POs is often better.
Another possible explanation for weak improvements could be that Cartesian abstraction heuristics already offer better guidance through the combination of multiple abstraction functions. Since the abstractions are refined along optimal paths, the heuristic may already capture the relevant search direction.

4.2.4 Comparing Preferred Operators Quality across Abstractions

We evaluate which abstraction heuristic leads to the most effective preferred operators. For this, we directly compare the impact of live POs on the expansion score in lazy GBFS for single PDBs, iPDB and Cartesian abstractions. Table 4.2 shows that the single PDB heuristic, which produces the weakest heuristic guidance, benefits most from POs. IPDB achieves a similar reduction. Cartesian abstractions generated by CEGAR do not profit from

Heuristic	without POs	with live POs	Change (in %)
Single PDB	459.06	509.36	+10.96%
iPDB	594.51	640.48	+7.73%
Cartesian Abstraction	572.31	572.78	+0.08%

Table 4.2: Comparing expansion scores with live and without POs in lazy GBFS using three different abstraction heuristics.

including POs, despite yielding the lowest amount of total expansions over all commonly solved tasks and therefore the highest-quality heuristic.
This suggests that the abstraction heuristic with the strongest guidance benefits the least from POs, while the heuristic with the weakest guidance can make use of them the most. One possible explanation for this inverse relation is that a strong heuristic already contains sufficient information about the state space, whereas a weaker heuristic benefits from additional search guidance.

4.3 Effects of Preferred Operators on Time

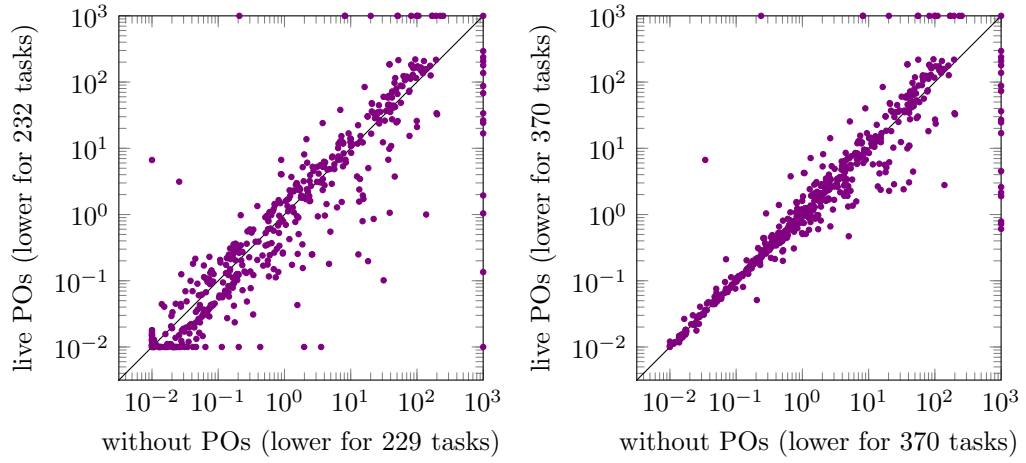
We now analyse the impact of preferred operators on time using abstraction heuristics for lazy GBFS. We compare preprocessing time, which is the time spent on setup and precomputation before search begins, search time, which covers the actual state space exploration, and total time, which consists of the sum of preprocessing and search time. We use the score that was originally used for measuring time in the agile tracks for the International Planning Competition in 2023 [8]. The score on a solved task is 1 if it was solved within 1 second and 0 if the task was not solved within the time limit of 5 minutes (300 s). If the task was solved in T seconds ($1 < T < 300$), then its score is $1 - \frac{log(T)}{log(300)}$.

4.3.1 Evaluation with Pattern Databases

Table 4.3 shows that all time scores increase when using live preferred operators in lazy GBFS. The largest increase occurs in search time score. However, only 232 tasks have a lower absolute search time with POs, while another 229 tasks show lower search time without POs (Figure 4.2a). The total time of GBFS with POs is lower than that of a search without POs for 370 tasks, as seen in Figure 4.2b. These results suggest that POs provide useful guidance that can reduce the overall effort during search, for example by avoiding unnecessary state expansions. At the same time, this positive effect on search efficiency is offset in roughly half of the tasks by the additional computation required to determine POs.
In addition to the scores evaluated over all tasks, we conduct a second experiment in which we limit the tasks taken into consideration for time scores to commonly solved tasks. In this setting, the total time scores are nearly identical: 711.33 with live POs and 711.22 without. Only a few problems benefit so much from POs that they achieve a high score with POs but remain unsolved without POs. We found seven tasks that achieve a score of 0.73 or higher with POs but are not solved without POs. Three of these belong to the *schedule* domain and another three to the two *pipesworld* domains. These tasks explain a large part of the higher total time score over all tasks compared to the score restricted to commonly solved tasks. There are fewer tasks where the opposite effect occurs. Therefore, the score over all tasks is higher for lazy GBFS with live POs. Preferred operators tend to be beneficial when they

Metric	Without POs	live POs	Change
Coverage (sum)	833	836	+3
score preprocessing time	809.54	813.01	+3.47
score search time	732.94	739.26	+6.32
score total time	715.63	720.18	+4.55

Table 4.3: Time measurements in lazy GBFS using a single PDB heuristic with live and without preferred operators.



(a) Search time with live POs.

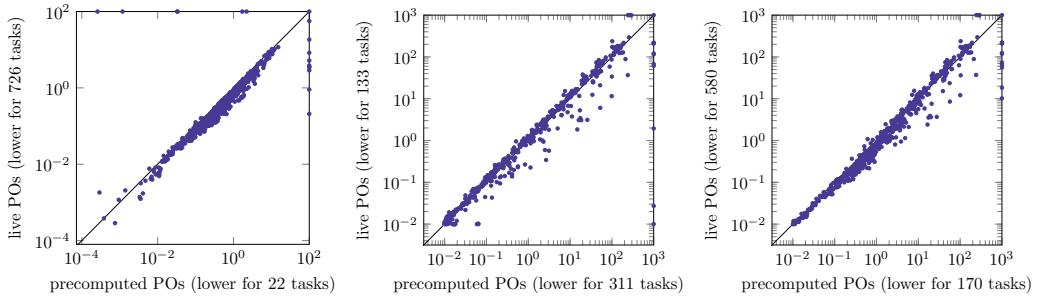
(b) Total time with live POs.

Figure 4.2: Time comparison in lazy GBFS using a single PDB heuristic with live and without preferred operators.

successfully identify operators that lead to promising regions of the search space. When this guidance is less effective than expected, their negative impact remains limited due to the dual-queue approach, which ensures that non-preferred operators are still considered. As a result, POs primarily improve coverage in a small subset of all instances rather than by consistent improvements across commonly solved tasks. The preprocessing score over the commonly solved tasks is 796.04 without and 795.99 with POs. This is expected, because live POs do not require additional preprocessing, and it confirms that the difference over all tasks is solely due to the higher coverage.

To measure these effects more directly, we compute the time per expansion over the commonly solved tasks. The results show that the additional computation required to determine live POs leads to a more than doubled time per expansion, from 99 to 210 microseconds on average. Combined with the expansion score in the commonly solved tasks, which, unlike the time scores, is still higher with POs (+45.41), we can conclude that most of the similar time scores result from fewer but more time-consuming expansions, which ultimately balances out.

When directly comparing the two variants of preferred operators in lazy GBFS, we also observe a better performance of live POs. As expected, precomputed POs require more time for preprocessing. This is reflected in the higher preprocessing time score of live POs (813.01 vs. 798.09) and in Figure 4.3a, where only 22 tasks have a lower preprocessing time when precomputed POs are used, and 726 tasks have a higher time. We expect that



(a) Preprocessing time (b) Search time (c) Total time

Figure 4.3: Lazy GBFS using a single PDB heuristic with precomputed POs compared to live POs.

precomputed POs would then perform better in search time, since the live version has to do the work that in the precomputed version has already been done and can now simply be read. However, the results do not show a clear benefit. Precomputed POs perform better in terms of lowering search time over many tasks. This is visible in Figure 4.3b, where 311 points lie slightly above the diagonal, indicating small improvements in search time for precomputed POs. However, there are also 133 points far below the diagonal, showing that in some tasks the live variant achieves substantially lower search times. This distribution explains why the advantage of precomputed POs is not reflected in the search time score, where the live variant achieves a higher search time score. When considering total runtime, the advantage of the live variant becomes more obvious. It achieves lower total time in 580 tasks, compared to only 170 tasks for the precomputed version (Figure 4.3c). It also attains a higher total time score (720.18 compared to 709.07). This suggests that the time savings from fewer expansions with precomputed POs are not sufficient to compensate for the increased initialization cost.

To get a better insight into the overhead introduced by precomputed preferred operators, we measure the time spent in the Dijkstra computation. The results show a substantial increase in PDB construction time on commonly solved tasks when preferred operators are computed. The geometric mean increases from 0.29 seconds to 0.43 seconds. This corresponds to an increase of nearly 50%.

Overall, the results show that POs can improve runtime, but their effectiveness depends on the methods used. Only in the lazy setting with live POs, they slightly reduce time. In lazy GBFS with precomputed POs, the positive effect is ruled out by the negative impact of the overhead, leading to no visible effect.

4.3.2 Evaluation with iPDB

Coverage increases from 1005 to 1020 when using live POs in lazy GBFS with iPDB. The better heuristic provided by iPDB compared to a single PDB leads to a high search time score, both with and without POs: it improves from 913.01 to 919.15 (+6.14). However, preprocessing is also significantly more expensive due to the computation of multiple patterns, which negatively affects total time. The total time score increases from 684.47 to 694.80 (+10.33). Overall, these improvements are slightly stronger than those observed with a single PDB heuristic. Using preferred operators seems to be worth it for the objective of

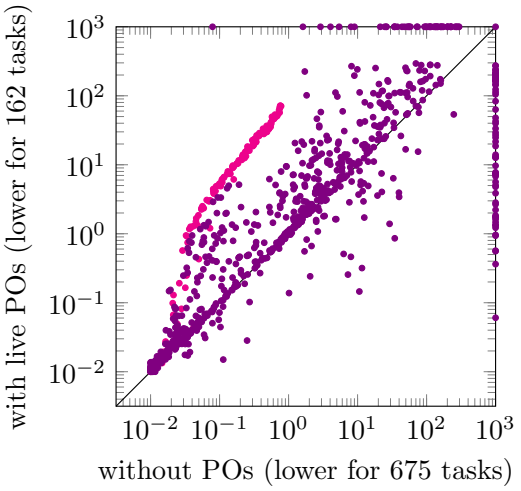


Figure 4.4: Total time in lazy GBFS using Cartesian abstractions without POs compared to live POs. The points in pink represent all 150 tasks from the *miconic* domain.

solving more problems in less time, as they are able to provide more informed guidance and therefore allow more tasks to be solved before the time limit.

4.3.3 Evaluation with Cartesian Abstractions

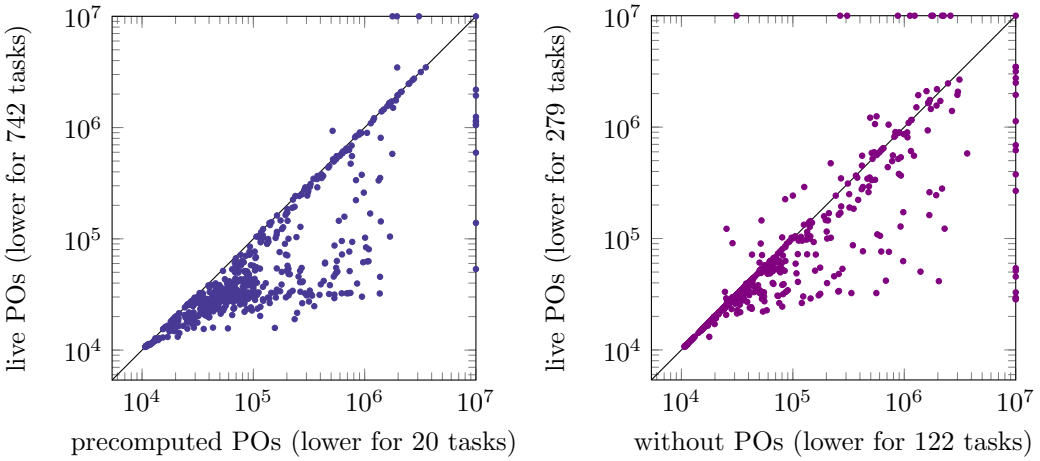
When using POs in lazy GBFS with Cartesian abstractions, coverage increases from 933 to 941, indicating that the planner is able to solve more tasks when preferred operators are used. At the same time, the total time score decreases from 787.85 to 724.55 (-63.30). We analyse if the *miconic* domain is again responsible for a large part of the overall decrease, and we observe that in this domain the total time score drops from 150.00 to 102.62 (-47.38). This is a significant part, but even without this domain, the search with POs would perform worse. Figure 4.4 also illustrates the negative impact of POs in the *miconic* domain: the 150 problems from this problem form a nearly parallel cluster above the diagonal.

4.4 Effects of Preferred Operators on Memory

We also evaluate the impact of preferred operators on memory consumption by comparing different configurations of GBFS. Memory describes the amount of storage required for search. This includes both dynamically maintained data structures such as generated and expanded states, as well as precomputed structures like pattern databases. The memory score for a task is 1 if the task was solved with 2000 KB of memory and 0 if it exceeds the memory limit of 3584 MB. If the task was solved within S KB of memory ($2000 < S < 3584 \cdot 1024$), then its score is $\frac{log(S)-log(3584*1024)}{log(2000)-log(3584*1024)}$ [13].

4.4.1 Evaluation with Pattern Databases

We first directly compare the two variants of preferred operators by running the same benchmarks with lazy GBFS, once using live POs and once using precomputed POs. As expected, precomputing preferred operators significantly increases memory usage, because the PO information for all reachable abstract states must be stored.



(a) Comparing memory usage between precomputed POs and live POs.

(b) Comparing memory usage between search without POs and with live POs.

Figure 4.5: Memory usage in lazy GBFS using a single PDB heuristic.

Figure 4.5a shows that many data points lie below the diagonal, indicating substantially higher memory consumption for the precomputed variant. Only a small fraction of instances (20 tasks) benefit from precomputing in terms of memory usage. The memory score drops from 499.00 with live POs to 432.26 with precomputed POs.
Additionally, we compare the memory usage without POs to lazy GBFS with live POs. The memory score increases by 17.69, and more problems are solved with less memory with POs (Figure 4.5b). The improvement in memory usage is also reflected in the amount of out-of-memory errors, decreasing from 995 without POs to 818 with live POs. This is likely related to the reduced number of expansions with POs, which leads to lower memory usage due to fewer states being stored.
Similar to the timer added to measure the time spent in the Dijkstra loop, we also record memory usage before and after this phase. The results show a substantial increase in memory consumption when precomputed preferred operators are stored, with the geometric mean of memory usage rising from approximately 9.96 KB to 39.44 KB. This increase of nearly 300% is due to the need to store preferred operator information for each abstract state. While the distance vector holding the heuristic values stores a single integer (4 bytes) per abstract state, the precomputed variant stores a vector per abstract state. Each inner vector occupies 24 bytes regardless of its contents ($3 \cdot 8$ byte pointers). Additionally, the preferred operators IDs contribute $4 \cdot \sum_s |pref(s)|$ bytes, where $|pref(s)|$ denotes the number of POs stored for abstract state s. The live computation avoids this overhead entirely.
Overall, these results show that while live preferred operators have a small positive impact on memory consumption in lazy search, precomputing preferred operators introduces a noticeable overhead, which generally outweighs any potential savings.

4.4.2 Evaluation with iPDB

For the iPDB heuristic, the memory score increases from 508.50 to 534.35 when using live POs (+25.85). At the same time, the number of out-of-memory errors is reduced from 421 to 213. This again indicates a more pronounced improvement compared to the single Pattern

Database setting. In contrast to the total time score, iPDB achieves a higher result than the single PDB in terms of memory score, with and without POs. Therefore, if the focus is on fast computation, a single PDB with POs should be used. If the goal is to maximize coverage or minimize memory usage, iPDBs with preferred operators are a better choice.

4.4.3 Evaluation with Cartesian Abstractions

When using Cartesian abstractions, the memory score improves from 422.74 to 444.24 with preferred operators (+21.50). Additionally, out-of-memory errors are drastically reduced, from 359 to 15. On the other hand, search time score decreases as seen in the section before. This indicates that the effort shifted towards less memory consumption but higher time efforts. We need more time but are also able to solve more problems using less memory with POs.
Surprisingly, in the *miconic* domain, there is no change in memory score. This indicates that although expansions and the required time increase in this domain, memory consumption remains stable. Therefore, the overall score is not negatively affected and shows an increase.

4.5 Comparison to LAMA

We also compare our implementation of live POs in iPDB and single PDBs with the LAMA planner using lazy GBFS. LAMA was the best-performing planner in the sequential satisficing track of the International Planning Competition 2008 [10]. In this setting, both approaches perform significantly worse than LAMA. When running iPDB with POs and LAMA, fewer tasks are solved with iPDB (1730 LAMA; 1020 PDB). Both search time score (1565.26 vs. 919.15) and total time score (1525.14 vs. 694.80) decrease considerably. This indicates that pattern database heuristics, even when extended with POs, are not efficient. This is consistent with previous observations that POs do not yield extensive improvements.
We additionally tested a time limit of 60 seconds for the hill-climbing computation to prevent preprocessing from consuming the entire available time. This leads to an increase in coverage to 1256. At the same time, all scores improve by roughly 100, but this configuration still remains far from competitive with LAMA, which consistently achieves higher scores, often doubling them.
We restrict one experiment to commonly solved tasks and examine the time per expansion. The results show that iPDB with preferred operators performs significantly worse than LAMA: the geometric mean of expansions increases by more than a factor of ten and the time per expansion roughly doubles (0.542 milliseconds vs. 1.001 milliseconds). This indicates that not only more nodes are expanded, but each expansion is also more expensive. Consequently, all scores decrease.
Finally, we run one experiment with a LAMA configuration and a configuration that includes all LAMA settings together with an additional single PDB. This also harms performance. The coverage decreases from 1730 to 1726, and all scores drop by about 20 to 150 score points.
Therefore, we conclude that our implementation of POs across different abstraction heuristics cannot compete with a state-of-the-art planner such as LAMA.

Metric	eager GBFS w/o POs	lazy GBFS with POs	Change
score expansions	519.80	509.36	-10.44
score memory	498.82	499.00	+0.18
score total time	736.91	720.18	-16.73
time per expansion (ms)	0.429	0.397	-0.032

Table 4.4: Comparison of eager GBFS without POs and lazy GBFS with live POs using the single PDB heuristic.

4.6 Comparison to Eager Search

We observe that live POs can have a strong positive impact on expansions in lazy GBFS, however this effect partially disappears when considering time and memory. From this, we conclude that POs improve guidance, but much of this advantage is offset by the additional computational effort required. We want to know whether the remaining advantages are sufficient to make lazy GBFS with POs a competitive alternative to eager GBFS without POs. Lazy GBFS without POs requires less time per expansion than eager GBFS, since it avoids evaluating all successors. We lose this advantage when we have to compute heuristic values for all successors in order to determine preferred operators. The behaviour of lazy GBFS with POs then becomes similar to eager GBFS, where all successor states are evaluated heuristically, resulting in more informed but also more expensive guidance compared to lazy search without POs.

In domains with uniform operator cost, lazy GBFS with POs and eager GBFS without POs rely on the same information. Differences arise mainly from the dual-queue in lazy search and from variations in tie-breaking caused by different orders in the queues. With operators of non-uniform cost, there is a bigger difference between those two methods. Lazy GBFS takes the cost of the operator leading to the successor also into consideration for the ordering for the next state, whereas eager GBFS ranks successors by heuristic value alone. Depending on the cost structure of the state space, one configuration or the other can perform better, though as illustrated in the example in Chapter 3, a quite specific cost structure is required for the configuration with POs to have a benefit.

To compare these two configurations, we run an experiment comparing eager GBFS without POs to lazy GBFS using a PDB heuristic with live POs to evaluate whether the additional cost of POs is justified. With a coverage of 852, eager GBFS solves more problems than lazy GBFS (836). The memory score is nearly identical, while in expansions and total time, eager GBFS performs slightly better (Table 4.4).

The higher expansion score indicates that standard eager GBFS can guide the exploration more effectively. The difference in the configurations is even larger for the total time score. This is partly due to differences in exploration effort and partly from the additional computation needed to determine POs. These results suggest that eager GBFS without POs can achieve better performance and using lazy GBFS with POs is not worth it.

To examine the differences between uniform and non-uniform cost domains, we extract the results from only the non-uniform-cost domains. There, eager GBFS achieves a coverage of 177, while the expansion score advantage shrinks to 5. This means that uniform-cost domains account for about 80% of coverage but only 50% of the difference in expansion score between the two configurations. These numbers indicate that in uniform cost, the performance of both configurations is more similar than in non-uniform cost domains.

5 Conclusion

In this thesis, we explored the effect of preferred operators on greedy best-first search when used with abstraction heuristics. The experiments show that the usefulness of preferred operators depends strongly on the quality of the existing search guidance.

In eager GBFS, POs have almost no effect on the number of expanded states. Since eager search already evaluates every successor individually, the heuristic itself provides sufficient discrimination between states. Adding POs introduces overhead without offering meaningful additional guidance, making them counterproductive.

In lazy GBFS, POs are much more effective. Because lazy search initially assigns the parent heuristic value to all successors, it lacks detailed guidance when deciding which state to explore next. In this setting, POs provide additional information by identifying promising successors, leading to fewer expansions in many tasks.

The experiments also reveal differences between live and precomputed POs. While both variants provide the same guidance in terms of expansions, up to tie-breaking, precomputed POs are not practical in combination with abstraction heuristics. Storing POs for all abstract states significantly increases preprocessing time and memory consumption, while the resulting improvements during search are too small to compensate. In contrast, live POs avoid this large memory increase and therefore are the more efficient and effective approach when combining preferred operators with abstraction heuristics. However, because of additional heuristic evaluations during search, live POs undo the advantage of lazy search over eager search.

The effectiveness of preferred operators additionally depends on the abstraction heuristic used. Single pattern databases, which provide the weakest heuristic guidance, benefit the most from POs regarding expansions. Canonical PDBs show consistent improvements across all metrics, with the highest increase in coverage, runtime and memory usage among the three heuristics considered. In Cartesian abstractions generated by CEGAR, however, POs provide very little benefit and in one domain even harm time and expansion performance substantially, leading to an overall neutral result for expansions.

Overall, the results indicate an inverse relationship between search and heuristic quality and the usefulness of POs: the weaker the heuristic guidance provided by the abstraction or search algorithm, the more expansions POs can reduce. POs thus can be seen as a mechanism

for compensating for weak guidance rather than as a universal beneficial addition. When the heuristic is already informative, preferred operators add overhead without adding value, when the heuristic is coarse, they help focus the search toward promising regions. However, the reduction in expansions does not automatically lead to proportional improvements in runtime and memory usage. Computing POs introduces additional cost per expansion and much of the benefit from fewer expansions is offset by this overhead. This leads to varying effects across metrics and abstractions, without revealing a clear overall correlation.
Finally, the comparison with the LAMA planner puts these results into context. It demonstrates that abstraction heuristics with POs are still not competitive with state-of-the-art satisficing planners.

Bibliography

[1] Christer Bäckström and Bernhard Nebel. Complexity Results for SAS+ Planning. *Computational Intelligence*, 11:625–656, 1995.

[2] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 154–169, 2000.

[3] Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[4] Stefan Edelkamp. Planning with pattern databases. In *Proceedings of the 6th European Conference on Planning*, pages 84–90, 2014.

[5] Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, pages 1007–1012, 2007.

[6] Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[7] Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling*, pages 176–183, 2007.

[8] IPC 2023. IPC 2023: The International Planning Competition (Classical Tracks). `https://ipc2023-classical.github.io`. Accessed: 2026-11-04.

[9] Silvia Richter and Malte Helmert. Preferred operators and deferred evaluation in satisficing planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling*, pages 273–280, 2009.

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

[11] Gabriele Röger, Florian Pommerening, and Jendrik Seipp. Fast Downward Stone Soup 2014. In *Proceedings of the 8th International Planning Competition*, pages 28–31, 2014.

[12] Jendrik Seipp and Malte Helmert. Counterexample-guided cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research*, 62:535–577, 2018.

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

[14] Silvan Sievers, Manuela Ortlieb, and Malte Helmert. Efficient implementation of pattern database heuristics for classical planning. In *Proceedings of the 5th International Symposium on Combinatorial Search*, pages 49–56, 2012.

Declaration of Authorship

I used ChatGPT to debug code and to fix grammatical errors. Additionally, I used DeepL to translate specific sentences. Everything else was done by myself.