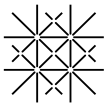

University of Basel

Hamming Width vs Novelty Width and Combinations

Bachelor's Thesis

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

Examiner: Prof. Dr. Malte Helmert
Supervisor: Simon Dold

Valdrin Sheremetaj
valdrin.sheremetaj@stud.unibas.ch
2021-059-779

03.07.2025

Acknowledgments

First and foremost, I want to thank my supervisor Simon Dold, who always helped me when I got stuck somewhere or did not understand an important (and sometimes easy) concept. Without him this thesis would not have turned out the same, I will always be thankful for that. I would also like to sincerely thank Prof. Malte Helmert and the Artificial Intelligence Research Group for giving me the opportunity to write this thesis and dive into this interesting topic. A special thanks also goes out to my friends and family, who never doubted me for a second.

Abstract

In classical planning, some tasks are easy to solve while others are significantly more difficult. This variance in problem difficulty can be explained and measured using width-based complexity measures, such as Novelty Width and Hamming Width. In this work, we analyze both Novelty Width and Hamming Width, comparing their empirical behavior and relative strengths. We will integrate both measures into a unified code framework, to directly compare them. Furthermore, we introduce combinations of these width notions and investigate whether such combinations are empirically effective in real-world planning applications.

Table of Contents

1 Introduction

Planning is an important area in artificial intelligence, which allows a system to decide which sequence of actions it needs to take, to achieve a certain goal. In the context of classical planning, we aim to find such a plan under the assumptions of a static, fully observable environment where all actions are deterministic. Since many general AI planning problems are PSPACE-complete or NP-hard (Bylander, 1994) [1], it makes them computationally infeasible in worst-case scenarios, but still many real-world instances are solved remarkably efficiently by modern planners. Which makes us question: how do we create these effective planners that work well in practice? One promising approach focuses on the width based complexity measures. These measures show how much of the problem's state space needs to be considered to make progress towards the goal. We will center on two such measures: Hamming Width by Chen and Giménez(2007)[3] and Novelty width by Lipovetzky and Geffner(2012)[9]. They both have different approaches on how to navigate the state space. Hamming Width by Chen and Giménez(2007)[3] focuses on the minimum number of state variables that must be changed in order to reach a new state where more goal conditions are satisfied, without undoing any previously satisfied goals. Novelty width by Lipovetzky and Geffner(2012)[9] on the other hand refers to the minimum number of facts that need to be examined simultaneously, in order to determine whether a state is novel or not. A state is called novel, if it is not a duplicate of any previously visited state during breadth-first search. Hence, Novelty width emphasizes the value of new states, while Hamming width emphasizes structural locality and incremental improvement (in contrast to the goal state). Understanding how these two approaches relate to each other is not only theoretically interesting but also practically useful for helping with the design of planning algorithms that are both efficient and robust across domains. That's why the goal of this thesis is to create a comprehensive analysis of Hamming Width and Novelty Width, both theoretically and empirically, and to investigate their relationships. We will also combine those two approaches and evaluate their practical value for AI planning. We do that by implementing them in the planning system Fast Downward (Helmert, 2006) [7], based on the Search Framework provided by Dold(2021)[4].

2 Background

In this chapter we will introduce different important definitions, that we will use throughout this thesis.

2.1 SAS^+ Planning Formalism

To argue about classical planning tasks in a formal way, we need to find a uniform representation for them, which allows us to express both Novelty Width and Hamming Width measures. While Lipovetzky and Geffner (2012) [9] used a classical $STRIPS$ planning with boolean variables (Fikes, Richard E. and Nilsson, Nils J. , 1971) [5], we will adopt the SAS^+ formalism (Bäckstrom and Nebel, 1995) [2] for this thesis. That is because it generalizes $STRIPS$ to support multi-valued variables and therefore is compatible with the definitions of Hamming Width as well.
A SAS^+ planning task Π is defined as a tuple $\Pi = \langle V, I, O, \gamma \rangle$, where V $= \{v_1, \ldots, v_m\}$ is a finite set of **state variables**. Each variable $v \in V$ has an associated finite domain $dom(v)$. A **partial state** p is a mapping:

$$p : \text{vars}(p) \rightarrow \bigcup_{v \in V} \text{dom}(v)$$

where $\text{vars}(p) \subseteq V$ is the set of variables for which p defines values. For each $v \in \text{vars}(p)$, we require that $p(v) \in \text{dom}(v)$. A tuple $\langle v, d \rangle$ with $v \in V$ and $d \in \text{dom}(v)$ is called a **fact**, and we write it as $v \mapsto d$. A set of such facts is called a **partial assignment**. A **state** s is a complete assignment of values to all variables, meaning $s(v) \in dom(v)$ for every $v \in V$. This means that each variable has exactly one value in each state, we write $s(v)$ to denote the value of the variable v in the state s. The **initial state** I is such an assignment. We say that a state s **agrees** with a partial state p if $s(v) = p(v)$ for every $v \in \text{vars}(p)$. We will use the notation $s \models p$ to denote that the complete state s agrees with the partial state p. The set of **operators** (also called actions) is denoted by O. Each operator $o \in O$ is defined by a pair $\langle pre(o), post(o) \rangle$, where $pre(o)$ is a partial assignment specifying the **precondition** of o, and $post(o)$ is a partial assignment specifying the **postcondition**. Informally, the precondition tells us in which states the operator is allowed to be applied and the postcondition tells us how the state changes when the operator is applied. An operator o is **applicable** in state s if $s \models pre(o)$. Applying operator o to state s results in a **successor state** s', where all variables in $post(o)$ are updated accordingly, while the remaining variables retain their values from s. Formally, $s'(v) = post(o)(v)$ if $v \in vars(post(o))$, and $s'(v) = s(v)$ otherwise. The **goal condition** γ is a partial state. A state s is a **goal state** if $s \models \gamma$, which means that it agrees with γ on all variables in $vars(\gamma)$. Contrary we denote **wrong variables**

in a state as $wrong(s) = \{v \in vars(\gamma) \mid s(v) \neq \gamma(v)\}$, this function shows us which goal variables are not yet satisfied in a state s. If $wrong(s) = \emptyset$, then $s \models \gamma$, which means s is a goal state. We define a **plan** P (for an instance Π) as a sequence of operators $o_0, \ldots, o_m$ that generates a state sequence $s_0, s_1, \ldots, s_{m+1}$, where $s_0 = I$, each o_i is applicable in s_i, and s_{i+1} is the successor state resulting from applying o_i to s_i. The final state s_{m+1} must satisfy the goal condition γ. Intuitively, this means that each operator o_i can be applied in the current state s_i, the application correctly moves us to the next state s_{i+1}, and the final state satisfies the goal. If such a plan exists starting from I, we call the problem instance Π **solvable**. Note that even solvable tasks may contain **dead ends**, which are states from which no solution can be reached. We define a notion for **plan execution**, to correctly talk about the outcomes of applying a plan P to a state s. Let $P = \langle o_1, o_2, \ldots, o_m \rangle$ be a plan, and let s be a state. We define the result of applying P to s, written $s[P]$, as the state obtained by applying all operators in sequence. A state s is called **reachable** if there exists a plan P such that $s = I[P]$ and a plan P is called a **solution** if $s[P]$ satisfies $s = I[P]$. We use the notion $s \xrightarrow{P} s'$ to denote that a state s' is reachable from a state s. The planning task Π is **solvable** if a plan exists from I.

2.2 Hamming Distance and Novelty

To define and analyze the different notions of width in planning problems, as introduced later in this thesis, we require to talk about the methodology with which we compute our width's. While the actual definitions of width measures will follow chapter in 3, this section introduces the formal elements they depend on.

Hamming distance. Originally introduced by Richard W. Hamming in the context of error-correcting codes (Hamming, 1950) [6], the *Hamming distance* between two states s_1 and s_2 over the same variable set V is defined as:

$$d_H(s_1, s_2) := |\{v \in V \mid s_1(v) \neq s_2(v)\}| .$$

It measures the number of variables whose assigned values differ between the two states and will be needed in our Hamming-based width measures. Informally, it will show us later, how far apart a state is from our wanted goal state.

Novelty. The notion of novelty was originally introduced by Lipovetzky and Geffner (2012) [9] as part of the Iterative Width (IW) search algorithm. We will now define the notion of novelty.
Unless stated otherwise, we assume each assignment in this work to be valid, meaning that each variable $v \in V$ appears in at most one fact of the partial assignment.
For a given integer $i \in \mathbb{N}$, let T_i denote the set of all i-sized tuples of facts. A tuple $t \in T_i$ is said to be **true** in a state s if $s(v) = d$ for every $v \to d \in t$.
Let $H \subseteq \bigcup_{i=1}^{k} T_i$ be a reference set of fact tuples observed in previously visited states. The *novelty* of a state s is the smallest integer $i \leq k$ such that there exists a tuple $t \in T_i$ which is true in s and not contained in H. If no such t exists, the novelty is said to be greater

than k. In other words, the novelty of a state measures how many facts we need to group together to find something "new" in s that hasn't been observed in earlier states and will be needed for our novelty-based width measures.

2.3 State Space and Transition System

To talk about the structure of the search process, we introduce a graph-based formalization of the state space induced by a planning task. We call this graph the **state space** of the planning task. It represents all possible ways in which states can evolve by applying actions. Formally, we define the state space induced by a planning task $\Pi = \langle V, I, O, \gamma \rangle$ as a directed graph:

$$\mathcal{S}_\Pi = \langle S, E, G, s_0 \rangle,$$

where:

- S is the set of states, i.e., all complete assignments over the variable set V. Each state corresponds to a node in the graph.
- $E = \{\langle s, s' \rangle \in S \times S \mid \exists o \in O$ such that o is applicable in s and leads to $s'\}$ is the set of directed edges (also called transitions) between states.
- $G \subseteq S$ is the set of goal states.
- $s_0 = I$ is the initial state of the task.

This allows us to represent how the different width-notions "operate" on a state space and how they traverse through it. the objective is to reach a goal state $s \in G$ starting from the initial state s_0.

2.4 Search Framework

We will introduce the **Search Framework** by Dold(2021) [4], which we will use for our implementation of the Hamming Width Search and Novelty Width Search algorithms. It allows us to specify search algorithms, by only 3 subroutines named **progressCheck**, **expandCheck** and **updateClosed**. We will implement these subroutines ourselves according to the properties of our Hamming- and Novelty Width Search and try to find combinatorial definitions for combinations between these two algorithms. The pseudocode for the algorithm provided already by Dold(2021) is shown here.
It defines a general skeleton for our search algorithms in planning tasks, by searching through states, while trying to reach a goal state γ and managing both open (states to explore) and closed states (states already explored). It does that by initializing first, while checking if the goal γ is already reached. If it does agree the search is already over. If it is not reached already, we initialize the open list containing only the initial state I. The closed set is initialized with the empty set, symbolizing that no state is reached yet and immediately updated by the subroutine **updateClose** with the initial state I. Dold(2021)[4] also added a reference state *reference*, which first remembers the starting state and later helps to see

Algorithm 1 Search Framework

```
1: Data: planning task Π = ⟨V, I, O, γ⟩, width k ∈ ℕ, heuristic h,
2: subroutines updateClosed, progressCheck, expandCheck
3: Result: plan π
4: if γ ⊆ I then
5:     return empty plan
6: end if
7: open := [I]
8: closed := ∅
9: updateClosed(I, closed, k)
10: reference := I
11: while open is not empty do
12:     current := pop first element of open
13:     for all candidate ∈ succ(current) do
14:         if γ ⊆ candidate then
15:             return extracted path to candidate
16:         else if progressCheck(candidate, reference, h) then
17:             open := [candidate]
18:             closed := ∅
19:             updateClosed(candidate, closed, k)
20:             reference := candidate
21:             break
22:         else if expandCheck(candidate, closed, k) then
23:             updateClosed(candidate, closed, k)
24:             append candidate to open
25:         end if
26:     end for
27: end while
28: return fail
```

if progress was made. After the initialization, we enter the main while loop, which runs as long as the open list contains at least one state or terminates if a goal was found. Inside the loop we first assign the first element of the open list to the current state *current*, then we iterate (in an arbitrary order) over each *candidate* that is a successor of current with a for-loop. For each of those *candidates*, we will first check if we already found a goal, if yes we return the path and the search is over. If this is not the case, we check if progress was made with the **progressCheck** subroutine. If that's true, we reinitialize the open list, the closed set and the *reference* state. The closed set becomes the empty set again, and will be updated again by **updateClosed** while using the *candidate* state. The open list then contains only the *candidate* state and *reference* gets updated to *candidate* and the for loop breaks. If candidate both provides no progress and is not agreeing with the goal, then we test *candidate* if it should be expanded further. We accomplish this, by checking it with the subroutine **expandCheck**. If *candidate* should be expanded further, we call the subroutine **updateClosed** on *candidate* to add elements into the closed set and append *candidate* to the open list. If the while loop ends, no solution was found and it terminates with a fail return. This Framework will not only serve for a more structured implementation of the different width measures, but also provides a unified basis for comparing Hamming Width and Novelty Width in a consistent and fair manner.

3 Width-Based Complexity Measures

3.1 Running Example

We will illustrate the following Width-Based Complexity Measures on a fairly easy domain, to just show how these Width Measures work and differentiate more clearly. We consider four different light switches, which can either be on(1) or off(0). The agent can exactly flip one switch at a time. The goal is to reach a desired configuration (for us it will be for all switches to be on) from an initial configuration (for us all switches are off at first). Expressed as a planning task:

$$\Pi = \langle V, O, I, \gamma \rangle$$

$$V = \{v_1, v_2, v_3, v_4\}, \qquad \text{dom}(v_i) = \{0, 1\} \text{ for } i \in \{1, 2, 3, 4\}$$

$$I = \{v_1 = 0, v_2 = 0, v_3 = 0, v_4 = 0\}, \qquad \gamma = \{v_1 = 1, v_2 = 1, v_3 = 1, v_4 = 1\}$$

The complete state space consists of all $2^4 = 16$ binary configurations from 0000 to 1111. The operator set O includes every pair of states that differ by exactly one bit. For example, $\langle 0000, 0001 \rangle$ flips the last switch, and $\langle 1010, 1110 \rangle$ flips the second switch.
We visualize this state space for this *light-switch domain* below, The node labeled 0000 is the initial state (indicated by an incoming arrow), and the goal state 1111 is marked with a double circle.

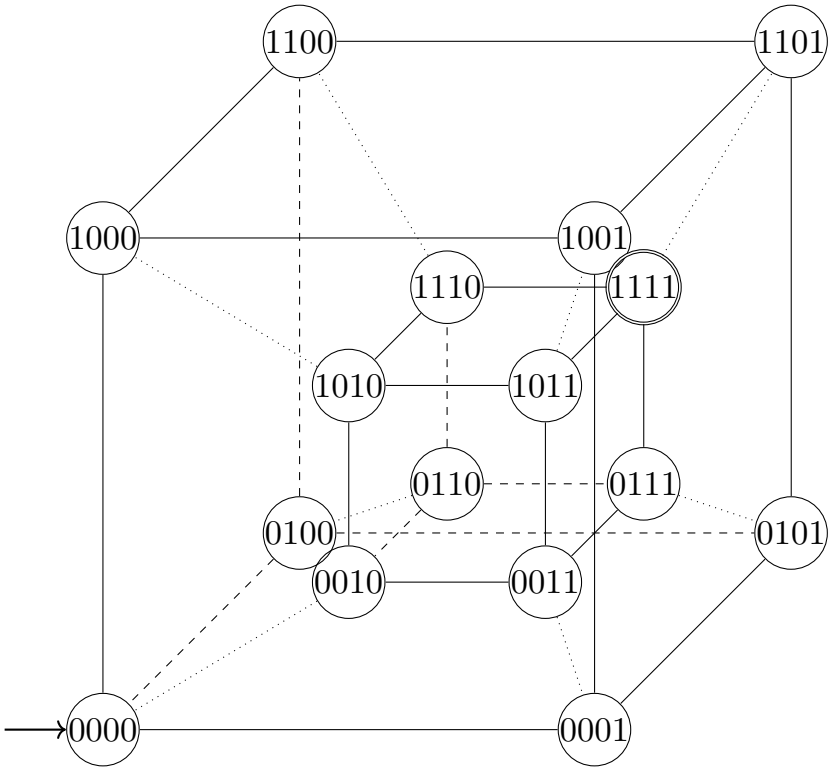

Figure 3.1: State space of the 4-switch light domain.

3.2 General Idea of Width in Planning

Heuristic planners like FF (Hoffman and Nebel, 2001) [8] often succeed in practice. However, this success lacks an explanation to why solving planning problems seems "easy". *Width* fills this gap of explaining the ability of current planners to solve planning problems in a few seconds, even though they are PSPACE-complete in general (Bäckström and Nebel, 1995) [2]. The concept of *width* in planning is a structural notion which captures the inherent difficulty of a planning problem. Intuitively, width shows us how many subgoals or facts have to be handled together at any time to find a solution. Hence, it measures the minimal number of "relevant facts" that need to be considered simultaneously to reach a goal from an initial state. Many planning problems, can be solved by focusing on a small number of those "relevant facts", which shows us that the planning problem is often much easier to solve. This means that there is a number (the width) which is bounded by a small constant. If the width of a planning instance is bounded by a constant, the search for a plan needs time exponential only in that constant (and polynomial in the size of the instance), making such problems tractable in practice. As mentioned before, we will focus on two different formal notions of width. Below we will explain each notion in turn, introduce combinations of those two width-notions and argue about how they could complement each other.

3.3 Hamming Width

Chen and Giménez(2007) introduced a set of width notions, trying to explain in a unified, domain-independent theory, why some AI planning benchmarks seem "easy" for planners. These width notions measure how hard it is to achieve the goal locally (via a small number of variable changes), even though the plan might be long. The intuition is, a problem is "easy" if, from any state, you can bring each variable to its goal by only changing a small number of variables. This approach explains tractability independently of domain-specific tricks like operator restrictions and therefore captures the benchmark domain's structure very well. We will define the introduced width notions now, starting with the *improvability*, because all of the other notions are based on it.

Definition 3.1 (k-improvable). Let $\Pi = \langle V, O, I, \gamma \rangle$ be a planning task. We say that plan P improves variable $u \in V$ in state s if:

1. for all $v \in V$, if $v \in vars(goal)$ and $s(v) = goal(v)$, then $(s[P])(v) = goal(v)$ and
2. if $u \in vars(goal)$, then $(s[P])(u) = goal(u)$.

This intuitively means that, if some goal variables are already satisfied in state s, then executing plan P must not undo them and the plan must bring a specific goal variable u to its goal value. So the plan P advances the solution, by making at least one variable (here u) correct (if u was not satisfied in the first place).

We call a state s **k-improvable** if there exists a plan P such that:

1. P improves some variable $u \in V$ and
2. all actions in P only reference (have preconditions or effects on) variables in a subset $U \subseteq V$ where $\mid U \mid \leq k$.

This expands our *improvability* notion, by adding an integer k, generally greater than or equal to 1, which bounds the number of variables that can be involved when making progress towards the goal. The idea is to, instead of solving the entire goal at once, provide only a "small window" of variables to "look at" and (if possible) incrementally improve the current state. For example, a variable is 3-improvable in a state s if there exists an improving plan P from s that only acts within a *3-dimensional subspace* of the full state space hypercube. In our *light-switch domain* example the state $s = 0001$ is 3-improvable because it is possible to reach the goal 1111 by flipping the remaining three switches one at a time. This plan only "touches" v_1, v_2, v_3 and therefore acts only within a 3-dimensional subspace. It is important to note, that it is also 1-and-2-improvable, since improving just one goal variable at a time (for example: flipping v_1 to reach 1001) is already sufficient to make progress without undoing the achieved part ($v_4 = 1$). Therefore, partial improvements can be made in smaller dimensions as well. We will visualize this, by showing a possible path P in our ongoing example below:

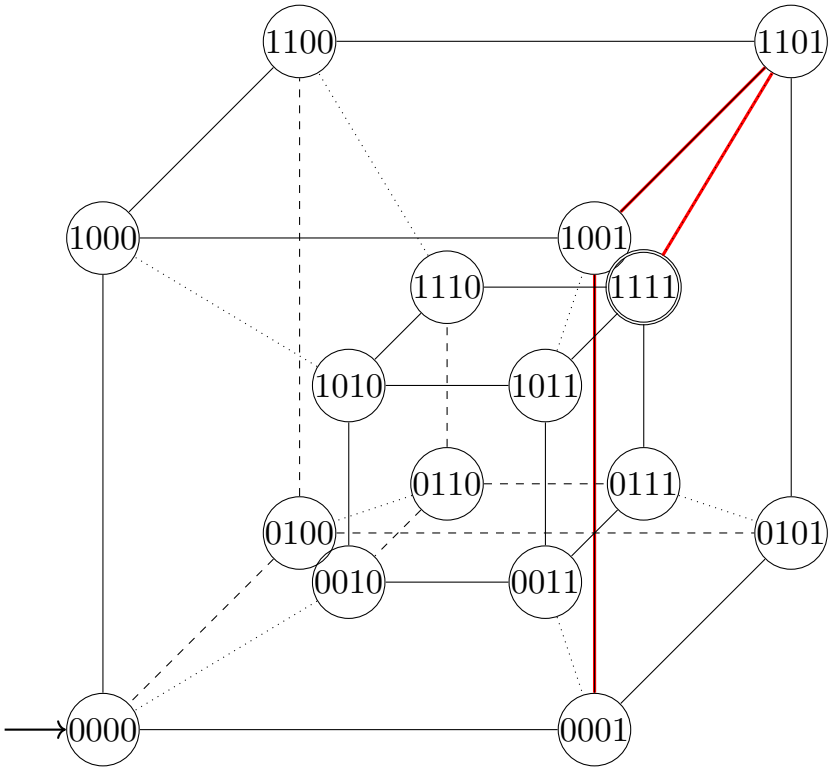

Figure 3.2: State space of the 4-switch light domain. The red path shows a 3-improvement sequence from state `0001` to the goal `1111`.

Definition 3.2 (Width k / Improvability Width) A planning task Π has *width k* if:

1. no plan solving it exists or
2. for every reachable state s that is not a goal state, every variable $u \in wrong(s)$ is k-improvable in s.

Chen and Giménez(2007) also introduced a width k algorithm, we will introduce it here: The algorithm is given a planning task Π:

1. Pick an arbitrary ordering $v_1 \dots v_n$ of the variables, set s to the initial state *init*, set Q to be the empty plan $\emptyset$
2. Loop from $1 \dots n$:
   If $v_i \in$ wrong(s), try to find a plan P using at most k variables that improves v_i in s. If such a plan P is found, replace s with $s[P]$, and append P to Q. If no plan is found, output "?" and halt.

This *width k algorithm* could be implemented into the Framework provided by Simon Dold (2021) [4], which we have seen in Section 2.4, by using the following subroutines shown in the table below:

Name	**Width-k Algorithm**
progressCheck	$wrong(candidate) \subset wrong(reference)$
expandCheck	$\exists W \subseteq V$ with $\lvert W \rvert \leq k$ such that $reference \xrightarrow{W} candidate$
updateClosed	insert *candidate* into *closed*

Table 3.1: Width-k algorithm subroutines in Dold's search framework

Definition 3.3 (Persistent Width) A planning task Π has **persistent width** k if:

1. either Π has no solution or
2. in every reachable state s that is not a goal state, there exists a variable $u \in wrong(s)$, such that u is *k-improvable* in s.

This is a "stronger" version of the improvability width, because regular improvability width only requires the existence of one successful sequence of k-improvements leading to a goal and persistent width requires that every reachable non-goal state non-goal state contains at least one variable that can still be safely improved. It therefore ensures, that you can always make progress, no matter where in the state space you are. It is important to note, that if a task Π has width k, it also has persistent width k. Chen and Giménez (2007) also introduced a *persistent width k algorithm*. It resembles the *width k algorithm* discussed before, but has some subtle changes. The ordering of the variables $v_1, \ldots, v_n$ does not rely on a fixed ordering anymore and rather builds the ordering incrementally during execution. In each iteration the algorithm tries to find a variable v that is *k-improvable* in the current state using only variables not used yet$(v_1 \ldots v_{n-1})$. If such a variable exists, it becomes v_n otherwise it halts. It is important to note, that only the subroutine *expandCheck* in our *width k algorithm* would be changed (see Table 3.1), because instead of only checking the reachability of candidate, it would also check if the candidate state improves some wrong goal variable.
Now we get to the Hamming-based width variants, which build up on the improvability notions talked before, but have their own properties.

Definition 3.4 (k-Hamming Improvable) We say that a variable u in state s is *k-Hamming Improvable* if there exists a plan $P = o_0, \ldots, o_m$ improving u in s such that for all $i = 1 \ldots n$: $d_H(s, s[o_1, \ldots, o_i]) \leq k$. Intuitively, this means that we can improve a variable u in state s if there exists a plan that improves u, without moving to far away from the initial state s. Here the key difference to normal *k-improvability* is that the plan must stay within a Hamming-distance-bound $\leq k$, which restricts how far the execution is allowed to drift away from the initial state, instead of restricting the number of variables an action in a plan can reference. Returning to our example for *k-improvability*, we can see that the state $s = 0001$ is 3-Hamming-improvable as well, because it's Hamming distance to the goal 1111 is 3 (since v_1, v_2, v_3 differ). But here, we are not interested in which variables we change, but only how far the state actually is from the goal state.

Definition 3.5 (Hamming Width k) We say that a planning task Π has *Hamming Width k* if:

1. no plan exists or
2. for every reachable state s that is not a goal state, every variable $u \in wrong(s)$ is k-Hamming improvable.

We again expand the *k-Hamming Improvable* notion with an integer k. This time this integer is bounded to the Hamming distance between an initial state and all intermediate states during plan execution. This forms a sort of "radius" or neighborhood around the current state: any improving plan must remain within this radius at every step along the way. Below you can see the Hamming distances on our running light-switch domain with a Hamming Width of 2. This will illustrate which states can be considered for improvement and which one's are not in the Hamming Width "radius":

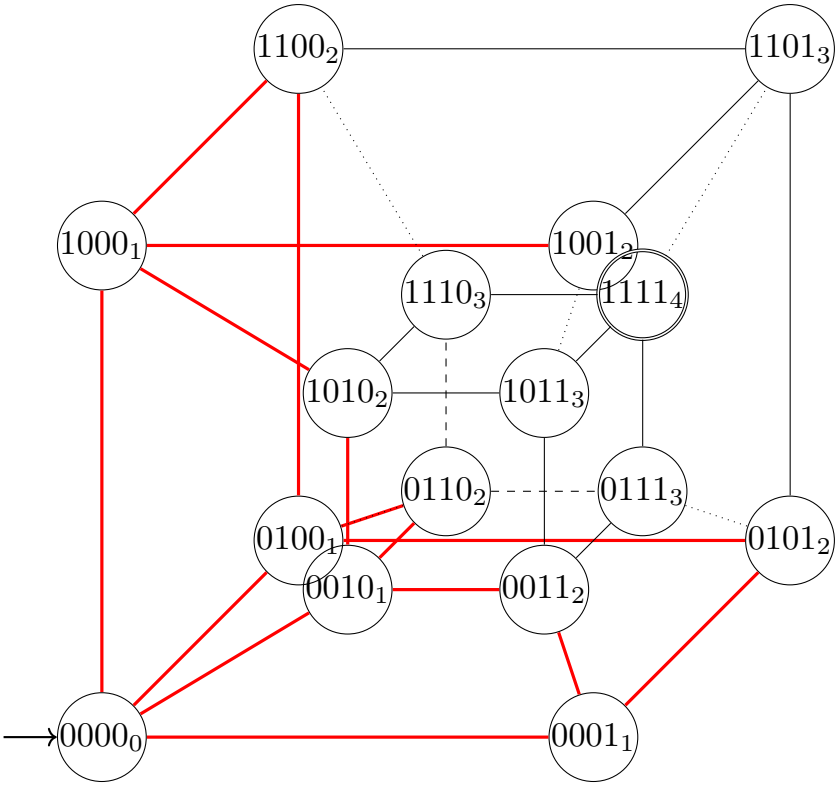

Figure 3.3: State space of the 4-switch light domain. Each node is annotated with its Hamming distance from the initial state 0000. Red edges indicate all transitions between states within Hamming distance ≤ 2 from the initial state 0000.

Definition 3.6 (Persistent Hamming Width k) A planning task Π has *persistent Hamming Width k* if:

1. no plan exists or
2. in every reachable state s that is not a goal state, there exists a variable $u \in wrong(s)$, such that u is *k-Hamming-improvable* in s.

We focused on implementing the subroutines for the framework provided by Dold(2021) [4] which use the properties of persistent Hamming Width, because it gives a uniform explanation of the tractability of many domains. We can see the subroutines needed below:

Name	Persistent Hamming Width Algorithm
progressCheck	$wrong(candidate) \subset wrong(reference)$
expandCheck	$candidate \notin closed$ and $d_H(candidate, reference) \leq k$
updateClosed	insert *candidate* into *closed*

Table 3.2: Persistent Hamming Width subroutines in Dold's search framework

3.4 Novelty Width

Lipovetzky and Geffner (2012) provided a different width notion for planning problems and a blind-search planning algorithm which is called the *Novelty Width Algorithm (NWA)*. With that, they offered a similar way of explaining the structural complexity of a planning problem, by focusing not on Hamming distance-based width measures but rather on a novelty-based approach. We will talk about the different ideas they introduced below. We will start with explaining the introduced *tuple graph*, which gives a formal structure to discuss planning complexity.

Definition 3.7 (Tuple Graph) For $\Pi = \langle V, I, O, \gamma \rangle$ the Graph $\mathcal{G}_i$ is inductively defines as follows:

- T_i is the set of all tuples t of at most i facts
- A tuple $t \in T_i$ is a **root node** in $\mathcal{G}_i$ iff t is true in the initial state I.
- There is a directed edge $t \rightarrow t'$ in $\mathcal{G}_i$ iff for every optimal plan π for $\Pi(t)$, there exists an action $a \in O$ such that π followed by a is an optimal plan $\Pi(t')$.

This intuitively means that root nodes are true in the initial state and that we can "draw an edge" to t', if for every optimal plan to t there exists an action a which by adding it to the end of the plan results in an optimal plan for t'. This is theoretically important because the *Novelty Width Algorithm (NWA)* tries to traverse $\mathcal{G}_i$ by generating new states. The *Novelty Width* is also defined with the Graph $\mathcal{G}_i$.

Definition 3.8 (Novelty Width) For a formula ϕ over the state variables in V that is not true in the initial state I, the *novelty width* of ϕ is the smallest integer k such that the tuple graph $\mathcal{G}_k$ contains a tuple t that *implies* ϕ. If ϕ is already true in the initial state, its novelty width is defined as 0. The *Novelty Width* of a problem gives us a bound on the complexity of solving that problem. If the width of a planning problem $w(P)$ is i then P can be solved in time that is exponential in i. Now we can focus on the *Novelty Width Algorithm (NWA)* derived from these notions.

Definition 3.9 (Novelty Width Algorithm) The Novelty Width Algorithm consists of a sequence of calls $NWA(i)$ for $i = 1, 2 \ldots$ over a planning task Π until it is solved. This sequence of calls is bounded to a number k (the Novelty Width). The *Novelty Width Algorithm* is a breadth-first search with extra pruning: Right after a state s is generated, the state is pruned if it does not pass a *novelty test* which depends on k. This *novelty test* intuitively means, that every state with a *novelty* larger than k must be pruned. For example, if the first state makes a fact p true, its novelty is 1, because the single tuple $\langle p \rangle$ was not previously seen. If s does not generate a single *new* tuple but a pair $\langle p, q \rangle$ then its novelty is 2, and so on. Therefore, the algorithm treats newly generated states with a novelty greater than k as if they were duplicate states. We implemented the *Novelty Width Algorithm* into Dold's Framework (2021) [4], by using the following subroutines shown in the table below:

Name	**Novelty Width Algorithm**
progressCheck	`false`
expandCheck	$\exists p \subseteq$ *candidate* with $\lvert p \rvert \leq k$, $p \notin$ *closed*
updateClosed	insert each $p \subseteq$ *candidate* with $\lvert p \rvert = k$ into *closed*

Table 3.3: Novelty-Width algorithm subroutines in Dold's search framework

3.5 Hamming Width vs Novelty Width on our Running Example

We will later run experiments on the *Persistent Hamming Width Algorithm* (see Table 3.2) and on the *Novelty Width Algorithm* (see Table 3.3) as well. But first we will further explain how these two algorithms work on our running example. Our *light-switch domain* is fairly simple, since the agent can only flip one switch at a time and we want all switches to be on, we just need to flip each flip without "unflipping" one. This has many solutions (24 to exact), but we will focus on the plan : $flip(v_1), flip(v_2), flip(v_3), flip(v_4)$. This example still differentiates in the width needed for the two algorithms. While the *Persistent Hamming Width Algorithm* only needs Hamming Width = 1, because you can reach the goal by flipping one switch at a time, while always making progress. The *Novelty Width Algorithm* shows a more interesting behavior. It will halt at Novelty Width = 1 and Novelty Width = 2 and find a solution at Novelty Width = 3. For Novelty Width = 1 it would halt at depth 1. Because we can reach the states $1000, 0100, 0010, 0001$ but after reaching these states we have already seen $v_1 = 1, v_2 = 1, v_3 = 1, v_4 = 1$. Therefore when flipping another switch (which one does not matter here), we get a state which has already been seen and is pruned. For Novelty Width = 2 we would halt at depth 2, because after expanding every state with novel *pairs*, we can't expand to the states with three switches on, because all combinations of two variables being on have now been seen, so no novel pairs remain to justify further expansion. With novelty width = 3 we can expand to the states $1110, 1101, 1011, 0111$ and get to our goal state, since we get new novel *triples*. It is important to note that the goal state 1111 is not novel with Novelty Width = 3, because it does not consist of any new triple. But because of early-goal checking in our algorithms it still expands the goal state in the open list and returns the solution. Figure 3.4 shows the search tree of a *Novelty Width*

algorithm with Novelty Width = 3 on our running example.

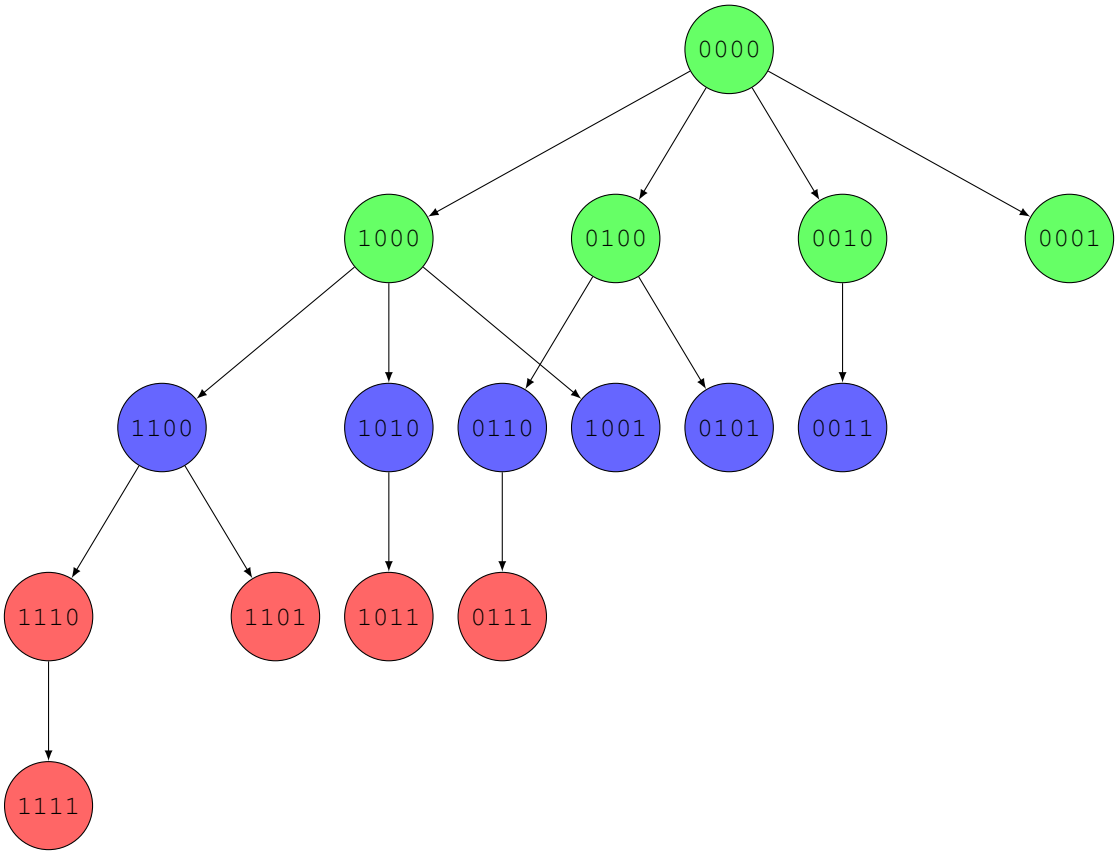

Figure 3.4: Novelty Width Search Tree (k = 3) on the Light-Switch Domain. Green nodes represent the root and all states found with novelty 1, blue nodes are states with novelty 2, and red nodes are states discovered at novelty 3. The algorithm is able to reach the goal state 1111 only at $k = 3$, as lower novelty bounds prune the search too early. The tie-breaking is indicated by nodes of the same depth being more to the left if they are generated earlier.

This is of course not a general case, we will see in Chapter 4 that the *Novelty Width Algorithm* has planning problems with a smaller width then the *Hamming Width Algorithm* as well.

3.6 Combinations of Hamming Width and Novelty Width

We will combine the Novelty Width Algorithm and Persistent Hamming Width Algorithm and try to explore how both concepts can complement each other. Each of these approaches have well-known strengths and limitations. The Novelty Width Algorithm is great at exploring the search space early on. It looks for states that introduce something "new" , which finds a combination of features that hasn't been seen before and therefore explores the search space widely. This kind of search often finds solutions quickly in simple problems. But it has a downside, it is not very goal-directed. The search might get close to a solution but then get stuck, simply because there are no new features left to discover. In those cases, even though the goal is nearby, the algorithm does not know which direction to take. That's where the Persistent Hamming Width Algorithm can help. It measures how close a state is to the goal by counting how many variables are different. This gives the search a sense of direction, especially when novelty alone isn't enough. If the novelty search hits a dead end near the goal, Hamming width can push it forward. On the other hand, Hamming width also has issues. It can be misled by irrelevant variables. Just because a state is only a few variable changes away from the goal doesn't mean it's actually useful, following them could lead the search in the wrong direction. Novelty can help here by adding a filter, among the many "close" states, it highlights the ones that actually contribute new or meaningful features. A combination would therefore introduce a tie-breaking system. Sometimes the search will encounter multiple states with the same novelty or the same Hamming distance. When that happens, we break the tie by applying a secondary rule. For example, if there are equally novel states, we pick the one closer to the goal in terms of Hamming distance. Or the other way around. This helps avoid getting stuck and makes the algorithm more robust overall. Overall combining both of these ideas, might give us a more balanced planning strategy. We will talk now about the several approaches and how we might implement it in Dold's framework (2021) [4].

We implemented two different combinations of the two algorithms. First we will talk about the new combinatorial algorithm *Hamming-AND-Novelty*. It is a logical combination of the two pruning strategies: a state is only considered for expansion if it satisfies both conditions. Hence, it must have a novelty of at most k and a Hamming distance of at most k to the goal. In practice, this means that the state must be both "new enough" and "close enough" to the goal to be promising. This strict filtering rule can lead to more focused exploration, but also runs the risk of pruning away useful states too early.

In Dold's framework (2021) [4], this combination is implemented by modifying the **expandCheck** and **updateClosed** subroutine (*progressCheck* stays the same as in the Novelty Width Algorithm). It is important to note that in our implementation, we maintained two different closed lists. One filled with partial states used by the Novelty Width Algorithm and one filled with the states used by the Persistent Hamming Width Algorithm. This is of course not the only way to do this, but helped us with keeping the closed lists separate. After a call of **updateClosed**, we therefore have to update both closed lists as well. Here they are called $closed_{ham}$ for the Persistent Hamming Width Algorithm closed list and $closed_{nov}$ for the Novelty Width Algorithm closed list. Instead of checking just one pruning condition,

we check both. We therefore get:

Name	Hamming-AND-Novelty Algorithm
progressCheck	`false`
expandCheck	$\exists p \subseteq \mathit{candidate},\ \lvert p\rvert \leq k,\ p \notin \mathit{closed}_{\text{nov}} \wedge d_H(\mathit{candidate}, \mathit{reference}) \leq k$
updateClosed	insert $p \subseteq \mathit{candidate}$ with $\lvert p\rvert = k$ into $\mathit{closed}_{\text{nov}}$, and insert *candidate* into $\mathit{closed}_{\text{ham}}$

Table 3.4: Hamming-AND-Novelty algorithm subroutines in Dold's Framework

The second combination, which we call *Hamming-OR-Novelty*, takes a more relaxed approach. Instead of requiring both conditions, it allows a state to be expanded if it satisfies at least one of them. That is, a state is accepted if it either, introduces a novel fact tuple (novelty $\leq k$), or Its Hamming distance to the reference state is $\leq k$. This variant prioritizes broader coverage. It allows the algorithm to explore more states, including some that might not look useful under one measure but are promising under the other.
In Dold's framework (2021) [4], the only change from the AND-version is that we use a logical OR in `expandCheck`, rather than AND.

Name	Hamming-OR-Novelty Algorithm
progressCheck	always `false`
expandCheck	$\exists p \subseteq \mathit{candidate},\ \lvert p\rvert \leq k,\ p \notin \mathit{closed}_{\text{nov}} \vee d_H(\mathit{candidate}, \mathit{reference}) \leq k$
updateClosed	insert $p \subseteq \mathit{candidate}$ with $\lvert p\rvert = k$ into $\mathit{closed}_{\text{nov}}$, and insert *candidate* into $\mathit{closed}_{\text{ham}}$

Table 3.5: Hamming-OR-Novelty algorithm subroutines in Dold's Framework

In summary, these two combinations represent different trade-offs: *Hamming-AND-Novelty* is more selective and goal-focused, while *Hamming-OR-Novelty* is more exploratory and less strict in pruning.

4 Experiments

In this chapter, we look at how the different width-based algorithms introduced in Chapter 3 perform in practice. We want to see if the ideas of Hamming Width, Novelty Width, and their combinations actually help in solving planning tasks efficiently. Instead of just sticking to the theoretical definitions, we now run them on standard benchmark domains to compare how well they work. This includes measuring how fast they are, how many nodes they expand, how many states get generated and how often they can actually solve a task. Additionally every measurement will also give us the width needed to solve the task, which will provide insights for the complexity of each domain, explaining most of the metrics results. Before talking about the results, we will first explain how the experiments are set up.

4.1 Experimental Setup

Calculations were performed at sciCORE [1] scientific computing center at the University of Basel. We used the *infai_2* cluster for all experiments. Jobs are submitted through Slurm using the *autonice* wrapper. Each job runs with 12 CPU cores and gets 3900MB of memory per core. The time limit per run is set to 30 minutes. We used the Fast Downward planning system[2] (Helmert, 2006) [7] and extended it with our implementation of the algorithms *Persistent Hamming Width Algorithm, Novelty Width Algorithm, Hamming-OR-Novelty Algorithm and Hamming-OR-Novelty Algorithm* based on Dold's (2021) [4] Search Framework. We used the optimal-strips suite of the downward-benchmark repository [3] [10], which provided us with the needed *PDDL benchmarks*. The domains we investigated are Gripper (IPC 2008), VisitAll (IPC 2011), Blocks (IPC 2000), Movie (IPC 1998), Pegsol (IPC 2008), and Openstacks (IPC 2006). We selected these because they vary in structure and difficulty, and they allow us to compare how each width-based algorithm performs across different types of planning problems. Some of them have known low width bounds, while others are more "challenging" and require a higher value of width.
We ran each algorithm on all tasks in this suite and measured the following metrics:

- **Coverage:** The number of tasks successfully solved by the algorithm within the given time and memory limit. This metric provides a basic measure of algorithm robustness and practical applicability across different domains.

[1] http://scicore.unibas.ch/
[2] http://www.fast-downward.org/
[3] https://github.com/aibasel/downward-benchmarks

- **Runtime:** The total time taken to find a solution, measured from the start of the planner until a plan is found. This indicates how fast the algorithm traverses the search space and reaches a goal state. It reflects both the efficiency of the search strategy and the overhead introduced by pruning.
- **States Generated:** The total number of unique states generated during the search. This metric reflects the size of the explored search space and can provide insight into the overhead introduced by the algorithm's branching behavior. Especially for the Novelty Width Algorithm and the two combinations, which need to generate Partial States to perform the *novelty test*.
- **Nodes Expanded:** The number of search nodes expanded by the planner. This metric is independent of hardware and datastructure details and reflects how effectively the search algorithm focuses on relevant parts of the search space. Fewer expanded nodes indicate more effective pruning.
- **Width Required:** The minimal width value k that allowed the algorithm to find a solution within the given limits. This value is used to estimate the structural complexity of each task with respect to the algorithm.

4.2 Results and Analysis

We will now discuss the results of our experiments, focusing on how each width-based algorithm performed across the selected benchmark domains. Each subsection presents one of the evaluation metrics introduced earlier and compares the results across all domains. We will also abbreviate the algorithms in this section: *HW* for Persistent Hamming Width Algorithm, *NW* for Novelty WIdth Algorithm, *AND* for Hamming-AND-Novelty Algorithm and *OR* for Hamming-OR-Novelty Algorithm.

4.2.1 Coverage

Coverage	**HW**	**AND**	**NW**	**OR**
blocks (11)	11	6	7	7
gripper (6)	6	2	2	5
movie (30)	30	30	30	30
openstacks (5)	**5**	0	0	5
pegsol-08-strips (2)	2	1	1	2
visitall-opt11-strips (4)	4	4	4	4
Sum (58)	58	43	44	53

Table 4.1: Coverage (number of tasks solved per domain) for each width-based algorithm. The number in parentheses next to each domain indicates how many different problem instances were run for that domain.

Analysis. The HW algorithm achieves full coverage, solving all 58 tasks across all domains. This indicates that, HW is highly effective in our implementation. In contrast, the AND and NW variants show significantly lower coverage, solving only 43 and 44 tasks. This drop in performance can be seen particularly in domains like *gripper* and *openstacks*, where both algorithms solve fewer than half of the gripper tasks and none of the openstacks tasks. This happens due to the overhead introduced by generating partial states, which we will examine more closely in the *States Generated* subsection.
Interestingly, the OR combination recovers a bit of this lost performance, solving 53 out of 58 tasks. This suggests that the strategy (using either Hamming or Novelty) is more efficient in practice than NW and AND, but is still affected from the overhead introduced by partial states. It even solves all *openstacks* tasks, which seemed too "hard" for NW and AND. Overall, HW and OR appear to be more robust across domains, while AND and NW are more sensitive to time or memory restrictions, because of the generation of partial states. From now on, every openstacks run will return None to each metric, because our experiment only considers runs where all algorithms finished for a specific problem (NW and AND did not).

4.2.2 Runtime

Total Time (s)	**HW**	**AND**	**NW**	**OR**
blocks (6)	0.01	4.11	0.66	0.17
gripper (2)	0.01	1.35	4.58	0.36
movie (30)	0.01	0.04	0.61	0.01
openstacks (0)	None	None	None	None
pegsol-08-strips (1)	0.00	836.43	0.01	0.00
visitall-opt11-strips (4)	0.00	0.03	0.01	0.00

Table 4.2: Total runtime (in seconds) per domain for each width-based algorithm.

Analysis. The number in parentheses next to each domain indicates how many different problem instances have returned a value for the "Total runtime". As we can see, HW and OR are consistently fast across all domains, always requiring less than a second in total. In contrast, AND and NW show a noticeable increase in total runtime, especially in domains like *gripper* (for NW) and *pegsol* (for AND), where generating and checking partial states leads to significant overhead. Especially, AND takes over 800 seconds to solve just one instance in *pegsol*, compared to practically zero time for HW and OR. This shows that this strict pruning with Novelty Width and Hamming Width introduces a lot of computational cost, when generating the partial states, but also that a lot of "helpful" states then get pruned, because they can not fulfill both width tests, even though Novelty and Hamming Width alone work quite good. One reason for HW's strong performance (compared to NW and AND) is therefore that it operates directly on full states without the need to generate or evaluate subsets of states, which gives it a clear advantage in runtime over all algorithms.

4.2.3 States Generated

Generated States	**HW**	**AND**	**NW**	**OR**
blocks (6)	343.06	1161.68	1059.17	766.64
gripper (2)	818.37	7375.55	8238.79	4181.88
movie (30)	8050.97	39109.18	39109.18	4157.63
openstacks (0)	None	None	None	None
pegsol-08-strips (1)	21.00	37.00	25.00	14.00
visitall-opt11-strips (4)	10.99	105.18	36.28	23.51
Geometric Mean (43)	**220.54**	**1054.52**	**790.93**	**337.64**

Table 4.3: Number of states generated per domain for each width-based algorithm. Values are summed up.

Analysis. We will now talk about the values of *Generated States* between these algorithms, which will show us why the results we have seen so far make sense. The number in parentheses next to each domain indicates how many different problem instances have returned a value for the "Generated States". As expected, HW generates by far the fewest states in every domain, besides in *pegsol* compared to OR. This is because HW works directly on full states, which avoids the need to construct and test multiple subsets per step. On the other hand, AND and NW generate significantly more states, especially in domains like *movie*, where both reach over 39,000 generated states. This overhead comes from the need to build partial states to perform the *novelty test* during search, which combined with a high width value (comparably complex problem for NW), leads to a high state generation. OR generates fewer states than both AND and NW by avoiding unnecessary subset generation when a Hamming-based decision is already sufficient. This again explains the better runtime results of HW and OR in the previous section, and confirms that partial-state-based pruning can be expensive in terms of state space growth.

4.2.4 Nodes Expanded

Nodes Expanded	**HW**	**AND**	**NW**	**OR**
blocks (6)	125.18	446.62	402.78	282.37
gripper (2)	182.32	1919.72	2203.73	1076.85
movie (30)	91.00	442.00	442.00	47.00
openstacks (0)	None	None	None	None
pegsol-08-strips (1)	16.00	25.00	16.00	9.00
visitall-opt11-strips (4)	4.64	43.88	15.55	10.02
Geometric Mean (43)	**43.41**	**210.74**	**157.72**	**66.37**

Table 4.4: Number of search nodes expanded per domain for each width-based algorithm.

Analysis. The number in parentheses next to each domain indicates how many different problem instances have returned a value for "Nodes Expanded". HW and OR consistently expand fewer nodes than NW and AND. We have to keep in mind that this is also due to the higher value of generated states. In domains like *gripper*, AND and NW expand more than 2000 nodes, while HW stays below 200 and OR cuts this number in half. We can also observe that OR expands even fewer nodes than HW in some domains like *movie*, which shows how combining novelty and Hamming in a more flexible way allows the planner to skip large unpromising areas of the search space, even though we generate some partial states for it. This supports our hypothesis that the OR strategy benefits from both exploration and goal direction. On the other hand, AND suffers from being too strict, it only expands fewer nodes than NW in gripper, but otherwise its pruning leads also to often missing useful paths and leading to inefficient search, as we have seen in runtime and coverage. These results show the idea that a well-designed combination of both width notions can make the search both efficient and effective, but also that a bad-designed one might make the two algorithms even worse.

4.2.5 Width Required

Width Required (k)	**HW**	**AND**	**NW**	**OR**
blocks (6)	33	46	20	15
gripper (2)	6	14	12	6
movie (30)	60	180	180	30
openstacks (0)	None	None	None	None
pegsol-08-strips (1)	3	6	2	1
visitall-opt11-strips (4)	4	16	7	4
Sum (43)	**106**	**262**	**221**	**56**

Table 4.5: Minimal width values k required to solve each task for the different algorithms.

Analysis. The number in parentheses next to each domain indicates how many different problem instances have returned a value for "width k". We observe that HW and OR consistently solve tasks with much smaller k-values compared to NW and especially AND. This is particularly visible in the *movie* domain: both AND and NW require $k = 180$ to solve the tasks, while OR only needs $k = 30$, and HW solves them with $k = 60$.
This result makes sense when we consider how novelty-based algorithms behave. NW relies on finding novel partial states, but once no new ones are found, the only way to make progress is by increasing k. This explains why it needs large values of generated states to complete tasks. AND suffers even more because it combines both novelty and Hamming checks: a state has to be both "new" and "close" to the goal. This strictness prunes many potentially useful states too early, which forces the algorithm to raise k just to keep moving. HW, on the other hand, ignores novelty completely and only uses Hamming distance to guide search. This avoids the novelty bottleneck but still needs a relatively high k in some domains (for example: $k = 60$ in *movie*) because it sometimes overestimates the usefulness

of states that are close in terms of variable changes but not helpful in reaching the goal. It follows promising-looking but misleading paths, especially when the problem has many irrelevant facts.

OR clearly shows the strength of combining both concepts. It can rely on novelty early to explore the space broadly, and when that fails, it falls back on Hamming to guide the search towards the goal. This flexibility allows it to keep k much lower than both NW and AND, and even HW in all domains. With just $k = 30$ in *movie*, and a total width sum of only 56 (compared to 106 for HW, 221 for NW, and 262 for AND), it is by far the most width-efficient approach. This confirms that a more relaxed combination like OR benefits from the strengths of both width-measures.

Overall, this metric strongly supports the idea that combining Hamming Width and Novelty Width, provides a more adaptive, efficient pruning signal and leads to more tractable planning even in complex domains.

5 Conclusion

In this thesis, we explored different notions of structural width in classical planning and investigated how they can be used. Our focus was on the *Hamming Width* by Chen and Giménez (2007) [3], *Novelty Width* by Lipovetzky and Geffner (2012) [9], and two hybrid algorithms that combine both ideas using logical conjunction (AND) and disjunction (OR). The theoretical motivation for each width measure was discussed, along with an implementation template of all subroutines within Dold's width-based search Framework [4] for Fast Downward [7].
We then had an empirical evaluation across a set of benchmark domains, analyzing runtime, coverage, node expansions, number of generated states, and the minimum width value k required to solve tasks, for our different width-based algorithms. This allowed us to gain both qualitative and quantitative insights into the trade-offs of each approach.
The results show that:

- The **Persistent Hamming Width Algorithm** is really effective. Despite its simplicity, it solves all tasks in our test suite and is very efficient in runtime. Its main strength lies in working directly with full states, avoiding the costly generation of partial states required by novelty-based methods. But it is not as width-efficient as Hamming-OR-Novelty.

- The **Novelty Width Algorithm**, while powerful in theory, struggles for us in practice when novelty is quickly exhausted. Its performance drops in harder domains, where the required k-values grow fast, leading to high runtime and large search spaces, due to partial states generation. It is important to note that the generation of partial states, still can be optimized by other programming techniques.

- The **Hamming-AND-Novelty Algorithm**, though conceptually appealing as a strict filter, suffers from over-pruning. It is often too selective and fails to find solutions even when both individual strategies would succeed alone.

- The **Hamming-OR-Novelty**, on the other hand, strikes a good balance. It leverages the exploratory power of novelty early on and switches to Hamming distance when novelty fails (or the other way around). This flexibility translates into strong empirical performance, with good coverage, low runtime, and minimal required width across domains.

Taken together, these findings support the idea that combining multiple width measures can lead to more adaptive and efficient planning. The OR-based combination especially proves

that structural features of the search space can be exploited more effectively when planners are allowed to reason via Novelty and via Hamming distance.

Bibliography

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

[2] Christer Bäckström and Bernhard Nebel. Complexity results for sas+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

[3] Hubie Chen and Omer Giménez. Act local, think global: Width notions for tractable planning. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS)*, pages 73–80. AAAI Press, 2007.

[4] Simon Dold. Correlation complexity and different notions of width. Master's thesis, University of Basel, 2021.

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

[6] Richard W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950.

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

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

[9] Nir Lipovetzky and Hector Geffner. Width and serialization of classical planning problems. In *Proceedings of the Twentieth European Conference on Artificial Intelligence (ECAI)*, pages 540–545. IOS Press, 2012.

[10] AI Group University of Basel. Downward benchmarks. https://github.com/aibasel/downward-benchmarks, 2022. Accessed: 2025-07-01.