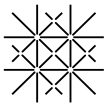

University of Basel

Computing Planning Task Width via SAT Compilation

Bachelor's thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
AI Research Group
ai.dmi.unibas.ch

Examiner: Prof. Dr. Malte Helmert
Supervisor: Remo Christen

Khadzhimurad Gadzhiev
khad.gadzhiev@stud.unibas.ch
2023-060-247

11/06/2026

Abstract

In classical planning, an area of research in the field of artificial intelligence, the "width" of a task was developed as an objective metric to gauge its computational difficulty. Search algorithms can use the width to find optimal plans for tasks efficiently, computing the width itself however proved difficult, recently a non deterministic, theoretical algorithm was developed. This thesis to bridge this gap by providing a concrete software implementation of this algorithm to allow for the computation of width for a greater number of planning tasks. The implementation relies on verifying conditions and properties by encoding the planning task into a Boolean Satisfiability (SAT) problem and was developed in Python. Additionally several optimizations were developed to speed up computation and the implementation was tested on a great number of tasks.

Table of Contents

1 Introduction

Classical Planning is a fundamental area of research in the field of artificial intelligence. You start with a set of states that represent the universe of the problem and a set of actions, an action can be applied to a state if specific preconditions are met, in that case applying the action will modify the state and lead to a new one. Classical assumptions mean that the environment is **single-agent**, meaning that only one instance of our program is acting on the state space at a time. It is **fully observable** because all of the information about the current universe is conveyed through the current state. It is **static** because only actions taken by our agent can change the current state. Finally it is **discrete** and **deterministic** because there is a finite number of states/actions and actions have deterministic effects on the states.

1.1 Solving a planning task

A state space contains an initial state and a set of goal states. The goal of a planning system is to find a sequence of actions that can be used to get to the goal state from the initial state, this sequence is called a plan. This can be done via search algorithms that explore the state space by applying actions, or via other methods like SAT solving (more on that later). But solving is not the only thing we can do with state spaces, we can also try and learn more about their properties, or about the properties of their domains (families of tasks) that could give us a deeper understanding of the field in general, one such property is the width.

1.2 Width

In [13] Nir Lipovetzky and Héctor Geffner prove that certain algorithms (like **Iterated width**) can solve a planning task optimally in time exponential in w where w is the width of the task (if w is known). This is interesting since automated planning in general is **PSPACE-Complete** [2] thus width is an accurate measure as to the difficulty of a task or even domain, as they also manage to define upper width bounds on certain domains. To actually compute the width one approach would be to just run the **Iterated width** algorithm with increasing width (startin from 1) until it finds a plan, unfortunately the

value this would produce, called the effective width, will not always match the *true* width that we will be computing in this thesis, which is why a more complicated computation is required to find it.
For some tasks we can confidently deduce the width without computing it specifically, thanks to proofs in [13] we now that tasks with single atom goals (more on those later) from the Blocks World, Logistics and n-puzzle domains have a width of at most 2. Practically this means that these can be solved very efficiently without computing the width but just assuming it is 2.
As no software for the computation of width exists, studying width and finding other uses or properties of it was time consuming, thus, a theorethical algorithm for computing the width was developed in [15]. This algorithm however was non deterministic and was not meant to be actually implementable as it was used simply as a stepping stone in their proofs, the aim of this thesis is to implement a concrete version of this algorithm to allow for an automated way of computing width for a greater number of tasks.

1.3 SAT Solving

SAT solving refers to the process of solving the **Boolean satisfiability problem**. Given a formula over boolean variables, a so called SAT Solver will determine if there exists an assignment of those boolean variables that makes the whole formula evaluate to true, most actual solvers also output a possible assigment if the formula is satisfiable, this problem is NP-Complete in general.
In practice SAT Solvers usually only operate on formulas in **C**onjunctive **N**ormal **F**orm (CNF), i.e. a conjunction of clauses, but any general formula is guaranteed to have a CNF representation, even if the conversion can cause the formula to become exponentially larger in the worst case. Classical planning can benefit from SAT solving since it is possible to encode a planning task as a formula over boolean variables as proposed in [11], if an assignment is found then a plan can be extracted from it. On top of this we will be able to use propositional logic to encode statements about the task and then prove or disprove them via SAT solving to help with the computation of width.

2
Background

In the following section we are going to formally define the concepts and objects discussed above, starting with the state space:

Definition 1. *A* state space *is a 6-tuple* $\mathcal{S} = \langle S, A, cost, T, s_I, S_G \rangle$ *with:*

- S *: a finite set of states*
- A *: a finite set of actions*
- $cost : A \to \mathbb{R}_0^+$ *is a function representing action costs*
- $T \subseteq S \times A \times S$ *: a transition relation, deterministic in* $\langle s, a \rangle$
- $s_I \in S$ *: the initial state*
- $S_G \subseteq S$ *: the set of goal states*

However, since we have to explicitly define every possible state, and all the possible transitions this state space definition may be hard to work with, as such many formalisms were developed that represent a state space with structures that outline rules about how to modify states when actions are applied instead of listing all of them. One such formalism uses propositional logic to define a task while being exponentially smaller than the above definition:

Definition 2. *A* propositional planning task *is a 4-tuple* $\Pi = \langle V, I, O, \gamma \rangle$ *with:*

- V *: A finite set of state variables*
- *Initial state* I *: an interpretation of* V
- O *: A finite set of operators over* V
- γ *: Goal formula over* V

An operator in this case represents an action, it has a precondition, to restrict the states it can apply to, and has an effect that describe how it affects states it's applied to:

Definition 3. *An* operator *o over variables V is an object with these properties:*

- *the precondition $pre(o)$: a formula over V*
- *the effect $eff(o)$ over V*

Where an effect is defined inductively as follows :

- *$\top$ is the empty effect*
- *v and $\neg v$ are atomic effects for $v \in V$*
- *$e_1 \wedge e_2$ is an effect if e_1 and e_2 are both effects*

This definition of a task is said to induce a state space (i.e. one can be constructed from this definition), so formally we are still working in one.

Note. *This definition also allows operators to have costs but we will be ignoring this as the concepts and definitions discussed in this thesis only apply to unit-cost tasks, i.e. tasks where all actions have the same cost.*

For a state s_1, s_2 is its successor for some applicable operator o if s_2 is s_1 but with the effect of o applied to it.
A plan $p = \langle o_0, o_1, ..., o_n \rangle$ with $o_i \in O$ for a planning task Π is a tuple of operators such that applying the operators in p to the initial state yields a state s such that $s \models \gamma$. Such a plan is said to be optimal if there exists no other plan for Π with a lower length than that of p.
We also define $\Pi(\varphi)$, with φ a formula over V, to be the planning task Π but with goal formula $\gamma = \varphi$.

Example
Let $\Pi_E = \langle V, I, O, \gamma \rangle$ be a planning task in the visit-all domain with 3 $cells = \{A, B, C\}$, then:

- V contains:
  - (robot-on-c) $\forall c \in cells$
  - (visited-c) $\forall c \in cells$
  - (connected-c_1-c_2) $\forall c_1, c_2 \in cells, c_1 \neq c_2$
- I = { (robot-on-A) $\mapsto$ **T**, (visited-A) $\mapsto$ **T**, (connected-A-B) $\mapsto$ **T**, (connected-B-A) $\mapsto$ **T**, (connected-B-C) $\mapsto$ **T**, (connected-C-B) $\mapsto$ **T** }
  with the rest of the variables $v \in V$ mapped to **F**.
- O contains:
  - (move-from-c_1-to-c_2) $\forall c_1, c_2 \in cells, c_1 \neq c_2$
    with $eff(o) = \neg$(robot-on-c_1) $\wedge$ (robot-on-c_2) $\wedge$ (visited-c_2)
    and $pre(o)$ = (robot-on-c_1) $\wedge$ (connected-c_1-c_2)
- γ = (visited-A) $\wedge$ (visited-B) $\wedge$ (visited-C)

We will use Π_E to illustrate the concepts discussed in this thesis, intuitively this task is about visiting all of the available cells by moving the robot between connected cells.

2.1 Regression

An important concept for propositionnal planning tasks is regression. When we talk about regressing a formula φ through an operator o the intuitive question that the regression answers is "What formula must hold in the state s for φ to hold after o is applied to s ?". But first we must define a couple of helper notions:

Definition 4. *The* effect condition *effcond(l, e) under which the atomic effect l triggers given the effect e is a propositional formula defined as follows:*

- *effcond$(l, \top) = \bot$*
- *effcond$(l, e) = \top$ for atomic effect $e = l$*
- *effcond$(l, e) = \bot$ for atomic effect $e \neq l$*
- *effcond$(l, e_1 \wedge e_2) =$ effcond$(l, e_1) \vee$ effcond(l, e_2)*

From there we can easily define the regression of a variable through an effect, i.e. "What formula should hold for v to be *true* after the effect e is applied ?"

Definition 5. *The* regression *of variable v through effect e:*

$$regr(v, e) = \textit{effcond}(v, e) \vee (v \wedge \neg \textit{effcond}(\neg v, e))$$

Intuitively this formula encodes the logic : v is *true* if e makes it *true* or if v was already *true* and e doesn't make it *false*. We can now generalize the definition to work with any formula φ over V:

Definition 6. *The* regression *of formula φ over V through effect e is defined inductively as follows:*

- *$regr(\top, e) = \top$ and $regr(\bot, e) = \bot$*
- *$regr(\neg\varphi, e) = \neg regr(\varphi, e)$*
- *$regr(\varphi \wedge \psi, e) = regr(\varphi, e) \wedge regr(\psi, e)$*
- *$regr(\varphi \vee \psi, e) = regr(\varphi, e) \vee regr(\psi, e)$*

Finally, we can define the regression through an operator as the regression through its effect and its precondition being satisfied:

Definition 7. *The* regression *of formula φ over V through operator o:*

$$regr(\varphi, o) = pre(o) \wedge regr(\varphi, \textit{eff}(o))$$

Example
For formula φ_E = (visited-A) $\wedge$ (robot-on-C) and operator o_E = (move-from-B-to-C) the regression would be:

$$\begin{aligned} regr(\varphi_E, o_E) &= pre(o_E) \wedge regr(\varphi_E, \mathit{eff}(o_E)) \\ &= \text{(robot-on-}B\text{)} \ \wedge \ \text{(connected-}B\text{-}C\text{)} \ \wedge \ \text{(visited-}A\text{)} \end{aligned}$$

This means that (robot-on-B) $\wedge$ (connected-B-C) $\wedge$ (visited-A) has to hold before applying o_E for φ_E to hold afterwards. Note how (robot-on-C) doesn't appear in the regression because o_E would always make it *true* anyways.

2.2 Width

The notion of width has several definitions in classical planning, one such definition exists for constraint satisfaction problems (a specialization of the state space search problem) where it measures the number of variables that have to be collapsed to ensure that the graph underlying the CSP becomes a tree [4, 6]. For a propositional planning task the term "width" was already used in [3] to mean a sort of Hamming Distance [8]. These definitions however do not comply with the intuitive understanding we have of the word "width" which is that a lower "width" of a task would mean a "simpler" task (i.e. easier to solve), the notion of width we will use, defined in [13] does indeed yield a lower number for tasks that are simpler to solve even if their definitions are not necessarily small, as such this width leads to an objective metric we could use to gauge the difficulty of a planning task.
Since our notion of width may be small even for large complex tasks, instead of working with full states that may potentially contain hundreds of variables (as they are interpretations and must assign a value to every state variable), we will restrict ourselves to using smaller conjunctions of state variables:

Definition 8. *T^i for some planning task Π is the set of all possible conjunctions of state variables from V where each variable can only appear once, no negations of state variables are allowed, and the number of literals can be no greater than a given positive integer i.*

With that [13] defines the notion of a reachability graph that formalizes the concept of reachability between these conjunctions, not to be confused with the classical definition of a state being reachable from the initial state, the below definition enforces stricter rules:

Definition 9. *The* reachability graph *G^i of a problem $\Pi = \langle V, I, O, \gamma \rangle$ is defined inductively as follows:*

1. *$t \in T^i$ is a root vertex in G^i iff $I \models t$*
2. *$t_1 \rightarrow t_2$, with $t_2 \in T^i$ is a directed edge in G^i iff $t_1 \in G^i$ and for every optimal plan p for $\Pi(t_1)$ there is an operator $o \in O$ such that p followed by o is an optimal plan for $\Pi(t_2)$*

Intuitively this graph represents all conjunctions that would be "easy" to find a plan for, the lower the i in G^i the easier.

Example
For task Π_E, the reachability graph G^1 would look like this:

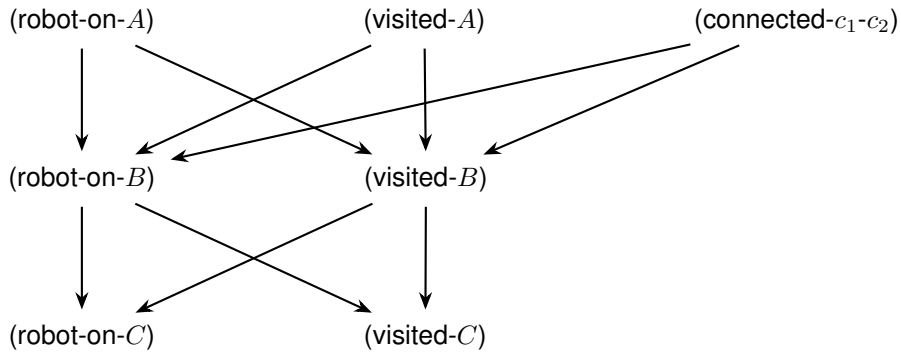

Figure 2.1: G^1 reachability graph for Π_E

Note. *The node (connected-c_1-c_2) represents all variables of this type that are true under I in Π_E because they all have the same transitions and would just clutter the figure.*

We can see how a structure emerges in the graph, we have to go through B to get to C and this is reflected in the edges in Figure 2.1 and since $i = 1$ we are limited to conjunction with just one literal.

2.2.1 Goal Implication

Now that we have the graph, representing the conjunctions that are easy to reach we need to determine whether finding a plan for the actual goal formula is as easy as the current i, for that we introduce a new notion of implication:

Definition 10. *A formula γ_1 implies formula γ_2 for a task Π if all optimal plans for $\Pi(\gamma_1)$ are also optimal plans for $\Pi(\gamma_2)$.*

This notion of implication differs from the classical logical implication definition that just states that the formula $\gamma_1 \implies \gamma_2$ must be true. The classical definition would be too restrictive, since we're working with planning tasks and not pure formulas, sometimes while reaching a conjunction, the "side-effects" (meaning the effects that don't affect variables that are part of the conjunction) of the operators we applied will lead to some other conjunction also being satisfied, this "implication" adresses this issue with a more verbose and complicated definition.

Example
In the context of task Π_E we can say that formula $\varphi_1 =$ (visited-B) $\wedge$ (visited-C) implies formula $\varphi_2 =$ (robot-on-C), because the optimal plan to visit C and B always ends on C (since you go through B to get to C) however $\varphi_1 \implies \varphi_2$ does not hold.

With this notion of implication we can define the width for a formula (i.e. how hard it would be to satisfy this formula by applying actions to the initial state):

Definition 11. *For a formula φ over the state variables V, its* width *relative to* Π *is defined as follows:*

- *0 if φ is true under I*
- *The minimum w such that G^w contains a vertex representing a conjunction t that implies φ otherwise.*

We can see that predictably, the width is the lowest if ϕ is already satisfied in the initial state, and only grows from there. Finally, the difficuly of a planning problem relies on the difficulty of finding a plan that satisfies its goal formula, so we can simply define the width of a planning task as:

Definition 12. *The* width *of a planning problem* Π, $w(\Pi)$ *is the width of its goal G relative to* Π .

This definition of width is completely equivalent (if slightly reformulated) to the one presented in [13].

Example
For task Π_E and graph G^1 shown in Figure 2.1, we can say that formula (robot-on-C) implies the goal formula $\gamma =$ (visited-A) $\wedge$ (visited-B) $\wedge$ (visited-C) because you always start on A and go through B to get to C. Since G^1 contains a vertex that implies the goal formula of Π_E we can say that $w(\Pi_E) = 1$.

2.3 SAT Encoding

During the width computation we will have to prove a lot of properties about tasks and their optimal plans, the approach we will make use of is encoding the planning task as a boolean satisfiability problem. This approach was first introduced in [11] and expanded upon in [12]. The central idea is to encode each step of the plan as a selection between the possible operators, this is possible thanks to the regression introduced in Definition 7. To know which step the boolean variable corresponds to, we superscript it with an index (more on that later). Finally, to force the task to respect the initial state and the goal formula, we make the goal formula part of the encoding at the last step and encode the initial state as a conjunction for the 0-th step. Of course this approach necessitates to know in advance the length of the plan we're expecting to extract, this is length is more commonly known as the "horizon", and when the goal is to solve a task with no prior knowledge we start with a horizon of 1 and increase it until we get a formula that is satisfiable as an unsatisfiable formula signals that the horizon is too low and that the task cannot be solved with a plan of this length.

2.3.1 Sequential Encoding

As previously discussed we will encode each step of the plan into the propositional formula, for that we define time-stamped formulas:

Definition 13. *Let φ be a propositional logic formula over the variables V. Let $0 \leq i \leq T$ where T is the horizon of the encoding :*
φ^i is the time-stamped formula *obtained from φ by replacing each $v \in V$ with v^i*

With this we can finally start encoding the task, since the ultimate goal is to feed this formula to a SAT solver we will aim to have a formula in CNF (i.e. a conjunction of clauses), we begin with the straight forward initial state clauses:

- v^0 for all $v \in V$ with $I(v) = \mathbf{T}$
- $\neg v^0$ for all $v \in V$ with $I(v) = \mathbf{F}$

The other easy one is the goal formula which just becomes γ^T to indicate that it must hold at the horizon (at the end of the plan).

2.3.2 Transitions

Next we will encode the transitions, i.e. the operator applications, this will be done in two parts : clauses that force the SAT Solver to pick exactly one operator on each step and clauses that will force the selected operator's effects to ultimately lead us to the goal.
Let $O = \{o_1, o_2, ..., o_n\}$, first we have the selection clauses:

- $o_1^i \vee o_2^i \vee ... \vee o_n^i$ for all $1 \leq i \leq T$ to force at least one operator to be selected on each step.
- $\neg o_j^i \vee o_k^i$ for all $1 \leq i \leq T$, $1 \leq j < k \leq n$ to force at most one operator to be selected on each step.

Now the tranition clauses, if an operator is applied in step i, then its precondition was satisfied at step $i-1$ and any variable is *true* in i if and only if its regression was *true* at step $i-1$:

- $o^i \rightarrow pre(o)^{i-1}$
- $o^i \rightarrow (v^i \leftrightarrow regr(v, \mathit{eff}(o))^{i-1})$

Note. *These formulas are not currently in CNF but can be converted in several steps that will not be discussed here.*

If we now put all these clauses together for every $0 \leq i \leq T$, we will get one CNF formula that we can feed to a SAT solver, if the formula is satisfiable we can then extract the plan from the returned assignment by just looking at which operator was selected on each step.

2.4 STRIPS

While a propositional planning task is great to work with, the propositional formulas it is based on are hard to represent on a computer, as such when working with concrete implementations, the formalisms are often more restricted, one such formalism is the **St**anford **R**esearch **I**nstitute **P**roblem **S**olver developed in [5]:

Definition 14. *A* STRIPS planning task *is a 4-tuple* $\Pi = \langle V, I, G, A \rangle$ *with:*

- V *: a finite set of state variables, also called atoms or facts*
- $I \subseteq V$ *: the inital state*
- $G \subseteq V$ *: the set of goals*
- A *: a finite set of actions, where for all actions* $a \in A$*, the following functions are defined :*
  - $pre(a) \subseteq V$ *: the preconditions of* a
  - $add(a) \subseteq V$ *: the add effects of* a
  - $del(a) \subseteq V$ *: the delete effects of* a

This definition is almost identical to Definition 2, with the following notable differences:

- States (including the inital one) are no longer interpretations but are just sets of variables, this representation is equivalent as we just assume that the variables present in the set are set to *true*, while the rest are set to *false*.
- The goal can no longer be any arbitrary formula over V, it is now a set of variables, we can turn this back into a formula by interepreting it as a conjunction of all the variables present in the set (negative literals can't be represented in STRIPS).
- Actions are almost equivalent to operators except they can't have any arbitrary formula as a precondition, only a set of variables that have to be *true* for the action to apply.

This formalism, is much easier to work with programmatically as sets are much simpler objects than whole formulas over variables, and whether a state satisfies a condition can be checked simply via subset operators rather than whole satisfiability analyses. Thus this representation is the one that will be handled in the actual implementation, but all concepts discussed above apply to it just the same.

Example

Converting task Π_E to a STRIPS formalism $\Pi_E = \langle V, I, G, A \rangle$ is pretty straight forward with:

- V contains:
  - (robot-on-c) $\forall c \in cells$
  - (visited-c) $\forall c \in cells$
  - (connected-c_1-c_2) $\forall c_1, c_2 \in cells, c_1 \neq c_2$
- I = { (robot-on-A), (visited-A), (connected-A-B), (connected-B-A), (connected-B-C), (connected-C-B) }
- G = { (visited-A), (visited-B), (visited-C) }
- A contains:
  - (move-from-c_1-to-c_2) $\forall c_1, c_2 \in cells, c_1 \neq c_2$
    with $add(a)$ = { (robot-on-c_2), (visited-c_2) }
    $del(a)$ = { (robot-on-c_1) }
    and $pre(o)$ = { (robot-on-c_1), (connected-c_1-c_2) }

3
Algorithms

The goal of the thesis is to develop a concrete implementation that would be capable of computing the width of a planning task, as such the first step is to define an algorithm that would compute the width while respecting the definition given above, we will start with Definition 11. Since the goal is to find a *minimum* width that satisfies a condition, the overarching algorithm will just be a loop over all possible values of w:

Algorithm 1 $w(\Pi)$

```
1: if I ⊨ γ then
2:     return 0
3: end if
4: for w ← (1 to |V|) do
5:     if Π has width w then
6:         return w
7:     end if
8: end for
9: return −1
```

Recall that in Definition 11, we get width 0 if the initial state already models the goal formula, which is what line 2 is doing. In most cases we have to resort to the reachability graph G^i, which itself only contains nodes from T^i, to check the width, as such Definition 8 gives us a way to set an upper bound on the width:
T^i only contains conjunctions of non repeating positive literals from V with size no greater than i. This means that increasing i past the number of state variables will not yield any new conjunctions in T^i and thus no new nodes in G^i, as such when i reaches $|V|$ and the width check on line 5 still fails, we can assume that the task is unsolvable and return -1 (width is technically only defined for solvable tasks).

The actual computation happens in the "Π has width w" step:

Algorithm 2 Π has width w

```
1: T^w ← conjunctions no greater than w from Π
2: G^w ← conjunctions from T^w that are satisfied in I
3: while G^w changes with each iteration do
4:     for ⟨t_1, t_2⟩ with t_1, t_2, ∈ T^w, t_1 ≠ t_2 do
5:         if t_1 ∈ G^w and transition t_1 → t_2 exists then
6:             add t_1 → t_2 to G^w
7:         end if
8:     end for
9: end while
10: for conjunction t in G^w do
11:     if t implies γ then
12:         return True
13:     end if
14: end for
15: return False
```

This algorithm does two things, the first part (lines 1 to 9) constructs the reachability graph G^w by checking all possible pairs of conjunctions from T^w and deciding whether they should have a transition between them. The second part (lines 10 to 15) checks all conjunctions in the constructed G^w and returns *True* only if at least one of them implies the goal formula, otherwise it returns *False*. Some important details like "t implies G" and "transition $t_1 \rightarrow t_2$ exists" are abstracted away into different algorithms, they will be explored later. First there is an easy optimization we can implement that would save us from checking most of the transitions several times. For it we will use the Iterated Width $IW(i)$ algorithm, and the notion of novelty introduced in [13] :

Definition 15. *The* novelty *of a state is a value calculated by during the search through a state space. When a new state s is explored (or generated), it produces a new conjunction of atoms t iff s is the first explored state that such that $s \models t$. The novelty of s is then defined as the size of the smallest conjunction t it produced. If s produces no such conjunction, its novelty is $\geq n + 1$ (practically the exact value doesn't matter) where n is the number of atoms in the planning task.*

From there $IW(i)$ is defined simply as a breadth-first search that prunes newly generated states when their novelty measure is greater than i. There are several properties of this algorithm we can exploit, starting with the optimality of the plan it finds. As a quick reminder, a breadth-first search builds the search tree layer by layer, which for unit cost tasks means that the first time a state is encountered, the path taken to get to that state will necessarily be an optimal one and the depth at which the state was encountered at represents the length of an optimal plan for that state.
The first optimization we can use, relates to a special property of the $IW(i)$ search, when it was introduced in [13], it was proven that if i is the true width of a planning task, $IW(i)$ is guaranteed to find an optimal plan, thus if it does not encounter a goal state, we know that the width we're checking must be too low and we can exit early. We do that by running

$IW(i)$ on the task with the current width we're checking before even starting the main computation.

Note. *This property does not work in reverse, i.e. the width i is not guaranteed to be correct even if $IW(i)$ encounters a goal state because it might not always encounter it thanks to tie-breaking between states and which ones get pruned first, this is the reason that effective width sometimes differs from actual width.*

Since we are already running $IW(i)$ on the task, we can exploit a different property of it. For ease of use we define $IW_G(i)$ as the $IW(i)$ search but that keeps going even after finding the goal (until all new states get pruned). We can use the search tree of $IW_G(i)$ as a sort of scaffolding to base our graph G^i on, but first we are going to have to prove that this approach is equivalent to the naive one, for that we prove the following:

Lemma 1. *The graph G^i for a task Π is a directed acyclic graph where the depth of a node t corresponds to the length of the optimal plan for $\Pi(t)$*

Proof. Let t_n be a conjunction of positive variables from V at depth n in G^i.
Base Case
The initial nodes contained in G^i are all conjunctions t_0 produced by the initial state I, and thus the length of their optimal plan is 0.
Induction Step
Let t_k be some conjunction at depth k of G^i with the length of an optimal plan for $\Pi(t_k)$ being k, if $t_k \to t_{k+1}$ is a directed edge in G^i then according to Definition 9:

"For every optimal plan p_k for $\Pi(t_k)$ there is an operator $o \in O$ such that p_k followed by o is an optimal plan for $\Pi(t_{k+1})$"

Thus an optimal plan p_{k+1} for $\Pi(t_{k+1})$ is always one longer than an optimal plan for $\Pi(t_k)$, i.e. it has length $t + 1$.
Conclusion
Since both the base case and the induction step hold, then Lemma 1 holds □

Theorem 1. *It is enough to only check transitions between $t_1 \to t_2$ in the reachability graph G^i for a task Π where n is the shallowest depth that t_1 is produced at, and $n+1$ is the shallowest depth that t_2 is produced at in the search tree of $IW_G(i)$ for task Π, to compute the correct width.*

Proof. From Lemma 1 we know that directed edges $t_1 \to t_2$ only exist between conjunctions whose optimal plans differ by 1, thus we know that if they were produced by $IW_G(i)$ during its search then they would be produced at depths n and $n+1$ for some integer n, as $IW_G(i)$ is a breadth first search and thus the first time it encounters a state (and thus the conjunctions it produces) it is guaranteed to be at a depth equivalent to the length of an optimal plan for this state.
One issue that might arise is pruning, how do we know that using the search tree of $IW_G(i)$ doesn't prune too many states ? Ultimately, we're just looking for *one node* in G^i that would imply the goal, since if we find one we know that the width is correct, recall that the

first optimization makes us immediately reject if $IW_G(i)$ does not encounter a goal state, thus it is guaranteed that at least one goal implying conjunction is present in the search tree of $IW_G(i)$ (the one produced by the goal state it encountered) and we can be sure that we won't "miss" the correct width. □

With this we can rewrite Algorithm 2 to only check transitions between conjunctions in successive depth in the search tree of $IW_G(i)$:

Algorithm 3 Π has width w

```
1:  if IW_G(w) does not encounter goal state then
2:      return False
3:  end if
4:  S^w ← search tree of IW_G(w)
5:  G^w ← conjunctions no greater than w from Π that are satisfied in I
6:  for pair ⟨t_1, t_2⟩ from S^w at depths n and n + 1 do
7:      if t_1 ∈ G^w and transition t_1 → t_2 exists then
8:          add t_1 → t_2 to G^w
9:      end if
10: end for
11: for conjunction t in G^w do
12:     if t implies G then
13:         return True
14:     end if
15: end for
16: return False
```

Line 2 now exits early if $IW_G(w)$ does not encounter a goal state, then lines 4 to 10 extract the search tree S^w of $IW_G(w)$ before looping over successive depths pairs from it instead of over all possible pairs from T^w.

Note. *It is important that the depths are looped over in the order of shallowest to deepest, as new transitions rely on the source node already being in the graph.*

3.1 Transition checking

The next part of the algorithm that we will analyze in depth is the "transition $t_1 \rightarrow t_2$ exists" check:

Algorithm 4 transition $t_1 \rightarrow t_2$ exists

1: $p_1 \leftarrow$ optimal plan for $\Pi(t_1)$
2: $p_2 \leftarrow$ optimal plan for $\Pi(t_2)$
3: **if** length of $p_2 \neq$ length of $p_1 + 1$ **then**
4: **return** False
5: **end if**
6: $T \leftarrow$ length of p_1
7: $\varphi \leftarrow \bigwedge_{o \in O} \neg regr(t_2, o)$
8: $\psi \leftarrow$ CNF formula for $\Pi(t_1)$ with horizon T
9: $\psi \leftarrow \psi \wedge \varphi^T$
10: **if** ψ is satisfiable **then**
11: **return** False
12: **end if**
13: **return** True

Reminder that this sections verifies the following condition from Definition 9:

"for every optimal plan p for $\Pi(t_1)$ there is an operator $o \in O$ such that p followed by o is an optimal plan for $\Pi(t_2)$"

The first section (lines 1 to 5) computes optimal plans for $\Pi(t_1)$ and $\Pi(t_2)$ (either using the SAT encoding or any other search algorithm) and comparing their lengths, because if p_2 is not one longer than p_1 we can immediately reject as the difference between the plans is not exactly one operator.
The next part (lines 6 to 9) is encoding the statement:

"there exists an optimal plan for $\Pi(t_1)$ that can't be extended into a plan for $\Pi(t_2)$ with just one operator $o \in O$"

as ψ (note how this statement makes no claims as to the optimality of the plan for $\Pi(t_2)$ but this condition is already enforced by the length check not rejecting at the beginning). If we can prove that ψ is unsatisfiable, this would mean that the opposite always holds, i.e. :

"all optimal plans for $\Pi(t_1)$ can be extended into plans for $\Pi(t_2)$ with just one operator $o \in O$"

which is equivalent to the statement we're trying to prove. To encode φ^T we use the negated regression on t_2, which gives a formula that if satisfied would mean that t_2 could not hold at step $T+1$, since in φ^T we regress through all operators $o \in O$ it means that if it is satisfied, no operator can cause t_2 to hold at $T+1$. Finally we can combine φ^T with an encoding of $\Pi(t_1)$ (at horizon T since T is the length of its optimal plan p_1) through a conjunction to ensure that both have to be satisfied, if that proves to be impossible we know that the transition verifies the condition and we can return *True*.

3.2 Goal implication

The final part that is still in need of an explanation is the "t implies G" check:

Algorithm 5 t implies G

```
1: p ← optimal plan for Π
2: p_t ← optimal plan for Π(t)
3: if length of p ≠ length of p_t then
4:     return False
5: end if
6: T ← length of p_t
7: φ ← CNF formula for Π(t) with horizon T
8: φ ← ψ ∧ ¬γ^T
9: if φ is satisfiable then
10:    return False
11: end if
12: return True
```

We can see that the structure is very similar to that of Algorithm 4 because it is also checking a statement about all optimal plans for a task, specifically we're veryfing the following condition from Definition 10:

"all optimal plans for $\Pi(t)$ are also optimal plans for Π"

We start (lines 1 to 5) by computing optimal plans for $\Pi(t)$ and Π and comparing their lengths, because if they are not equal we can immediately reject as the condition clearly doesn't hold.
The next part (lines 6 to 8) is encoding the statement:

"there exists an optimal plan for $\Pi(t)$ that isn't also a plan for Π"

as ϕ (note how this statement again makes no claims as to the optimality of the plan for Π but this condition is already enforced by the length check not rejecting at the beginning). If we can prove that ϕ is unsatisfiable, this would mean that the opposite always holds, i.e. :

"all optimal plans for $\Pi(t)$ are also a plan for Π"

which is equivalent to the statement we're trying to prove. This time we can simply encode $\Pi(t)$ and combine it via conjunction with $\neg\gamma^T$ which would ensure the goal formula is not satisfied at the end of the plan, if ψ is unsatisfiable we know that t implies G (or γ).

4

Implementation

The concrete implementation of the width computation was developed in Python using the Pyperplan [1] and PySAT [9, 10] packages. Pyperplan was used for the parsing of the problem files and in general for its data structures representing the planning task. While Pyperplan does offer SAT encoding capabilities, they were too limited (more on that further down) and as such PySAT was used for its more advanced SAT API.

4.1 Instances

The program starts by parsing the task files, specifically the PDDL problem and domain files. The PDDL [7] format is the standard encoding used to represent a classical planning task and is supported by most planners. It encodes objects as entities of different types and predicates that are a sort of "blueprint" on how to construct the possible facts of the task. Similarly actions are defined through these blueprints, referencing object types rather than concret objects. The object types, predicates and action are defined in a domain PDDL file, while the actual objects, initial state and goals are defined is specific problem files, as such many problems may share the same domain but operate in different state spaces.
Pyperplan was used to parse the PDDL files and turn them into useful data structures, with the main one being its `Task` class. In Pyperplan facts were represented as simple strings, with a state being a `frozenset` of those strings (a built-in python class that represents an immutable set). Thus the Task class contained the initial state, the set of goals, the set of all possible facts and the set of `Operator` objects. However Pyperplan, as the name suggest is focused solely on finding plans for STRIPS tasks, not any other of its properties like the width, this lead a severe problem. The issue was related to an optimization that Pyperplan applied during its parsing : when creating possible facts from the PDDL predicates, Pyperplan would purposefuly omit facts that could not be influenced by any operators (they would not appear in the facts list or in any states), this would work perfectly fine for a simple search algorithm as only facts that can be changed matter, but the width search relies on the number of facts as an upper bound, thus this part of the Pyperplan source code had to be modified.

4.2 Computation

After parsing, the tasks get feeded to an implementation of Algorithm 1 that loops over w and calls to `has_width()` which as the name suggests is an implementation of Algorithm 3. This one starts off by calling to our custom implementation of the $IW(i)$ algorithm. To compute novelty the algorithm stores all encountered encountered conjunctions as a set of sets, additionaly functions to extract all possible conjunctions from a set were developed, from there a simple subset check is enough to know whether a new conjunction was produced. The rest of the algorithm loops over pairs and unlike the definition of Algorithm 3 immediately checks goal implications for nodes that get added to the graph, this saves computation time if we find one that implies the goal early but was ommitted from the definition for clarity.

4.3 SAT

The pairs get passed to `does_transition_exist()`, an implementation of Algorithm 4, where the first instance of SAT encoding occurs, specifically we're aiming to encode the task into DIMACS. The DIMACS format is the most widely used text encoding of a CNF formula, since a formula in conjunctive normal form is just a collection of clauses, each clause in a DIMACS file is a separate line. One each line DIMACS uses positive integers to represent variables and their negation to represent the corresponding negated variable (0 is used at the end of each line to signify the end of a clause). The format also specifies mandatory boiler plate lines that we won't be discussing.

While Pyperplan does offer encoding the task into DIMACS, it was found that it's immediate representation (before getting into the DIMACS format) was not user friendly as it was not in CNF, additionally the task was not solved completely via the SAT solver, instead offloading some additional computation to the part that extracts the plan from the boolean assignment. The combination of these two factors meant that it was easier to encode the task ourselves, with the help of a more general library like PySAT, which did not provide a complete planning task pipeline but instead a set of robust tools to manipulate any propositional logic formula.

However even PySAT was not perfect, for example it did offer classes such as `AND()`, `NOT()`, `OR()` that would then be convertible directly to CNF using the provided `CNF` class, but after implementing the encoding using those, it was found that the hundreds of objects (each boolean operation was a separate object) it instantiates quickly fill up the memory, which was unacceptable for an operation that is supposed to run that frequently. We instead opted to use the more memory efficient but less user friendly "clauses" representation that the `CNF` class could also parse.

In that representation a CNF formula was just a list of clauses, where each clause was also a list that contain integer IDs that represented boolean variables.

Example
The following list:

$$[[1], [2,3], [4,5,6]]$$

Represents this CNF formula:

$$v_1 \wedge (v_2 \vee v_3) \wedge (v_4 \vee v_5 \vee v_6)$$

The variable have to be integers to adhere to the DIMACS standard, to map variables to and from these IDs, PySAT provided the `IDPool` class.
Since facts are just strings in the Pyperplan representation, when creating a time-stamped formula according to Definition 13, the step is just suffixed at the end of the variable string (with a separating '-') and the new, suffixed string is given an ID via `IDPool`.
With that, we encode the tasks and the additional formulas required like φ and ψ and then pass it on to one of the built-in SAT solvers that PySAT provides to checks its satisfiability.

4.4 Caching

While checking transitions, every encoding would technically be different as the goal formula constantly changes, however the entire rest of the formula (initial state, operators, transitions) would stay the same. Thus to allow for intelligent caching, all of the functions are actually methods in a `WidthComputation` class, when a task is encoded for a specific horizon, the program first checks if the cache contains an encoding for this horizon, if it does then it simply appends the goal clauses to the returned formula instead of re-encoding the whole task, if it doesn't it encodes it and stores it in the cache *before* appending the goal formula. This optimization allows for a much shorter runtime overall and thus allows us to check higher values of the width.
Finally if the transition does indeed exist, the destination conjunction immediately gets checked for goal implication.
The source conjunction does not need to be checked as it was either checked when it was added previously or it was part of the initial state in which case if it could imply the goal, then the program would have already returned a width of 0 thanks to the check on line 2 of Algorithm 1.
The goal implication check itself happens inside `does_imply_goal()`, an implementation of Algorithm 5 that works exactly the same as `does_transition_exist()` and since it's part of the same class, it benefits from the same cache and thus usually does little actual encoding. Finally, if a goal implying conjunction is found, then we can immediately return knowing that the current width is the correct one and can print it to the user, otherwise we need to keep checking until we go through all of the conjunctions in G^i at which point we can confidently say that the width is too low.

5 Results

To test the efficacy and results of our implementation, test instances were run on the sciCore computing grid. The different task instances were from the followng domains:

`barman-opt14-strips, blocks, childsnack-opt14-strips, depot, driverlog, grid, gripper, logistics00, miconic, movie, mystery, pipesworld-notankage, rovers, satellite, storage, tpp, visitall-opt11-strips, zenotravel`

5.1 Instances

The instance files were taken from the downward-benchmarks repository,[1] all of these were in the PDDL format and were STRIPS instances.

Note. *The repository did also contain non STRIPS instances which were ignored.*

In [13] the authors looked closely at several domains but with the restriction that the instances had to have one atom goals, as for a STRIPS task removing an atom from the goal can only ever make the plan shorter, thus by fabricating these simpler instances the authors were able to analyze them more easily. Specifically, they arrived at an analytical upper bound on the width for one atom goal instances for tasks in the `Blocks World`, `Logistics` and `n-puzzle` domains (this will be explored more in-depth further down). Additionally *effective width* calculations were run on a lot more domains, but all still restricted to only one atom goal instances.
As we did not manage to get access to the actual instances used in [13] we resorted to modifying the instances we already had. All availables instances were programmatically split into separate one atom goal instances (by just taking one atom from the goal at a time), this lead to a total of 7710 PDDL files (not counting the domain files).

[1] https://github.com/aibasel/downward-benchmarks

5.2 Specifications

Calculations were performed at sciCORE[2] scientific computing center at University of Basel. With the use of lab [14] for the orchestration and compilation of results for the several thousand instances. For each task a seperate instances of the width computation was run on an `Intel Xeon Silver 4114` with a limit of 30 minutes of CPU time and a memory limit of 3.5 GiB.

5.3 Test Results

Domain	**OOM Error**	**Timeout**	**Success**
barman-opt14-strips (43)	36	7	0
blocks (302)	18	219	65
childsnack-opt14-strips (15)	15	0	0
depot (188)	176	2	10
driverlog (250)	173	44	33
grid (19)	18	0	1
gripper (460)	35	425	0
logistics00 (249)	151	53	45
miconic (2325)	2143	167	15
movie (210)	0	0	210
mystery (45)	37	4	4
pipesworld-notankage (259)	183	65	11
rovers (558)	455	95	8
satellite (1742)	1735	4	3
storage (240)	205	23	12
tpp (85)	85	0	0
visitall-opt11-strips (505)	284	185	36
zenotravel (215)	163	21	31
Sum (7710)	**5912**	**1314**	**484**
Percentage (100%)	**76.68%**	**17.04%**	**6.28%**

Table 5.1: The outcome of running the width computation on the 18 domains

We can see that despite the instances only containing one atom in the goal conjunction, most of them run out of memory, primarly this happens when the program tries to encode the task as a SAT formula. A good portion of the instances also end up running out of time, with only around 6.3% of the instances finishing the computation and returning a value for the width. In fact there are 4 domains where no instances managed to finish, specifically `barman-opt14-strips`, `childsnack-opt14-strips`, `gripper` and `tpp`.

[2] http://scicore.unibas.ch/

Domain (Successes)	Minimum width	Average width	Maximum width
barman-opt14-strips (0)	-	-	-
blocks (65)	0	0,69	1
childsnack-opt14-strips (0)	-	-	-
driverlog (33)	0	0,06	1
gripper (0)	-	-	-
miconic (15)	2	2	2
movie (210)	1	1	1
pipesworld-notankage (11)	1	1	1
rovers (8)	1	1	1
storage (12)	1	1	1
tpp (0)	-	-	-
visitall-opt11-strips (36)	0	0,72	1
zenotravel (31)	0	0,06	1
Average (421)	**0,67**	**0,84**	**1,11**

Table 5.2: Width computation results across 13/18 domains

Note. *The average was taken as an average accross the domains, not across all of the instances separately.*

The 5 domains that are missing, were ommitted from this table because all of their width were 0, they will be discussed later. We can see that across the 421 instances present in this table, the average width is only 0,84. The values being so low is to be expected for two reasons :

- Firstly, we are limiting ourselves to single atom goal instances only, leading to "easier" to solve tasks.
- Secondly, we saw earlier than only around 6.3% of the instances finished their computation and returned a width value, which means that most of them got cut short, thus the likelihood that a smaller width gets returned is much higher, because instances with higher widths simply did not have enough time or memory to finish the computation.

Domain (Successes)	Width
depot (10)	0
grid (1)	0
logistics00 (45)	0
mystery (4)	0
satellite (3)	0

Table 5.3: Width computation results for domains with zero widths.

These are the domains missing from Table 5.2, they were ommitted as they all have a width of 0 accross all instances that suceeded in the computation, specifically they weren't used in the averages because they are most likely an artifact of the way we created the instances. Since we blindly split existing tasks into several with each having a one atom goal from the original's goal condition, some happened to already be satisfied in the initial state (which is the only way an instance can have a width of 0).

5.4 Analyzing Results

Predictably we can see that the single atom instances from the `blocks` and `logistics00` domains do not exceed the width upper bound of 2 set in [13], which at least gives us a sense that the implementation is doing what it's supposed to. In the same paper, an *effective* width calculation was run on a similar set of instances, in there they also prove that the *effective* width is always lower or equal to the *true* width we're computing. Since the test sets are not exactly the same we can't compare them entirely but we can compare specific domains :

- **Miconic** : Out of 650 instances, they found all of them had an effective width of 2. In our case only 15 instances finished computation but all of them reported a width of 2 as well.
- **Visitall** : Out of 21859 instances, they found all of them had an effective width of 1. In our case only 36 instances finished computation but the reported maximum width was indeed 1.
- **Storage** : Out of 240 instances, they found all of them had an effective width of 1. In our case only 12 instances finished computation but all of them reported a width of 1 as well.

Interestingly, they found that the instaces from the **Barman**, **Gripper** and **Tpp** domains all had mostly high effective widths (2 or bigger) which might explain why these domains did not succeed at computing the width even for a single instance in our testing (the higher the width the harder the computation).
Overall the results do not contradict their theoretical or practical findings and thus lead us to believe that the implementation truly does compute the correct width.

Surprisingly, all `movie` domain instances finished their computation and returned a value of 1, from this we could conjecture that all one atom goal instances in the domain have a width of at most 1, but this would need to be proven rigourously as was done in [13] and this is not the aim of this thesis.
Due to the really low success rate for the rest of the domains it is not really possible to extract any more valuable statistics from this data.

5.5 Future Work

Since the results seem correct, the next logical step would be to improve the efficiency of the program to allow it to run successfully on a greater number of instances. This can be done in two main ways, either by finding optimization in the algorithms themselves, for example by restricting the number of transitions that need to be checked even further just like we did for Algorithm 3 or by finding different but equivalent conditions on the graph in Definition 9 or on the goal implication in Definition 10 that would be easier to validate programmatically.

The second way is to optimize the implementation itself, one area that could benefit greatly from that is the part that encodes the task as a SAT formula, as this was just a quick implementation and does not build upon the decades of optimizations in that area, additionally it could be specialized even further as usually encoding engines are not specialized to encode the same task over and over with a different goal. This was adressed briefly with the caching but more could be done like for example reusing the work done for a lower horizon when encoding with a new one. In a similar vein it would also theoretically be possible to store the encodings incrementally (meaning each one uses parts of the previous one) to reduce the memory footprint.
An optimization that was considered but that we simply did not have the time to implement was multi-threading, specifically when checking the transitions and the goal implications, one check does not influence the other and they could be run in parallel, for the goal implication they could simply all stop if one of them finds a goal implying conjunction. Finally the implementation could be rewritten in a more performance oriented programming language, as Python was used simply for its ease of use and quick prototyping, something like Rust or Go could greatly speed up the computation and probably lower the memory footprint with no additional optimizations (and could also simplify the implementation of other optimizations like multi-threading). Finally the SAT solver itself was mostly ignored in this thesis and used as a black box, but since we are running similar formulas through it one after the other, maybe a specialized implementation that takes advantage of that could lead to better performance.

Bibliography

[1] Yusra Alkhazraji, Matthias Frorath, Markus Grützner, Malte Helmert, Thomas Liebetraut, Robert Mattmüller, Manuela Ortlieb, Jendrik Seipp, Tobias Springenberg, Philip Stahl, and Jan Wülfing. Pyperplan. Zenodo, 2020. https://github.com/aibasel/pyperplan.

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

[3] Hubie Chen and Omer Giménez. Act local, think global: Width notions for tractable planning. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS 2007)*, pages 73–80. AAAI Press, 2007.

[4] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

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

[6] Eugene C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[7] Malik Ghallab, Craig Knoblock, David Wilkins, Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, David ... Smith, and Daniel Weld. PDDL – the planning domain definition language. 1998.

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

[9] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT 2018)*, pages 428–437, 2018.

[10] Alexey Ignatiev, Zi Li Tan, and Christos Karamanos. Towards universally accessible SAT technology. In *Proceedings of the 27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, pages 4:1–4:11, 2024.

[11] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, pages 359–365. AAAI Press, 1992.

[12] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 1194–1201. AAAI Press, 1996.

[13] Nir Lipovetzky and Héctor Geffner. Width and serialization of classical planning problems. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI)*, pages 540–545. IOS Press, 2012.

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

[15] Jiajia Song, Seeun William Umboh, Malte Helmert, Nir Lipovetzky, and Sebastian Sardina. Computing planning width: How hard is it, really? In *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS 2026)*, 2026 (To Appear).