Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)

Hierarchical Width-Based Planning and Learning

Miquel Junyent, Vicenç Gómez, Anders Jonsson
Universitat Pompeu Fabra
Barcelona, Spain
{miquel.junyent, vicen.gomez, anders.jonsson}@upf.edu

Abstract

Width-based search methods have demonstrated state-of-the-art performance in a wide range of testbeds, from classical planning problems to image-based simulators such as Atari games. These methods scale independently of the size of the state-space, but exponentially in the problem width. In practice, running the algorithm with a width larger than 1 is computationally intractable, prohibiting IW from solving higher width problems. In this paper, we present a hierarchical algorithm that plans at two levels of abstraction. A high-level planner uses abstract features that are incrementally discovered from low-level pruning decisions. We illustrate this algorithm in classical planning PDDL domains as well as in pixel-based simulator domains. In classical planning, we show how IW(1) at two levels of abstraction can solve problems of width 2. For pixel-based domains, we show how in combination with a learned policy and a learned value function, the proposed hierarchical IW can outperform current flat IW-based planners in Atari games with sparse rewards.

Introduction

The use of hierarchies in planning has proven to be a very successful way for significantly reducing the computational cost of finding good plans. Traditional methods include Hierarchical Task Networks (Currie and Tate 1991; Erol, Hendler, and Nau 1996), macro-actions (Fikes, Hart, and Nilsson 1972; Korf 1985), and state abstraction methods (Sacerdoti 1974; Knoblock 1990). Hierarchical planning can lead to exponential gains in complexity by exploiting the structure of a problem involving a reduced subset of the state components.

Iterated Width (IW) (Lipovetzky and Geffner 2012) is a search algorithm that makes use of the feature representation of the states to perform structured exploration. The original IW algorithm consists of successive breadth-first searches in which states are pruned if they fail to meet a novelty criterion. In particular, IW(w) only considers w features at a time, and prunes those states for which all combinations of w features are made true in previously generated states. IW(w) runs in time and space that are exponential in w, but independent of the size of the state space.

Initially proposed as a blind search method for classical planning, IW search has been extended in many different ways, resulting in several competitive width-based planners, including LW1 for partially observable domains (Bonet and Geffner 2014), or BFWS as an informed (best-first) width search planner (Lipovetzky and Geffner 2017).

One particular advantage of width-based planners is that, unlike other classical planners, they do not need a declarative representation of actions, costs or goals (Francès et al. 2017). Width-based planners are thus directly applicable in simulator environments, achieving state-of-the-art performance in the General Video Game competition (Geffner and Geffner 2015) and the Atari suite (Lipovetzky, Ramirez, and Geffner 2015; Shleyfman, Tuisov, and Domshlak 2016; Bandres, Bonet, and Geffner 2018).

The performance of IW strongly depends on how informative the state features are. Using poorly informed features requires a large value of w to reach a goal state, whereas using highly informative features reduces the problem width and, hence, makes it solvable using a lower value of w. This effect is known, e.g., in Atari, where using informative RAM states leads to better results than planning directly with pixels (Bandres, Bonet, and Geffner 2018). How to discover or learn such features to reduce the problem width is an open problem, and several ideas have been proposed, including the use of conjunctive features (Francès et al. 2017) or deep learning methods (Junyent, Jonsson, and Gómez 2019; Dittadi, Drachmann, and Bolander 2020).

In practice, IW is mostly used with $w = 1$ with complexity linear in the number of features (Geffner and Geffner 2015; Bandres, Bonet, and Geffner 2018; Ramirez et al. 2018; Dittadi, Drachmann, and Bolander 2020). In many challenging problems, even $w = 2$ with quadratic complexity is unfeasible (Geffner and Geffner 2015). Finding ways to run IW with a larger value of w can further extend the applicability of this class of planners.

In this work, we propose a hierarchical formulation of width-based planning that takes advantage of both the structured search performed by width-based algorithms as well as the concept of hierarchy, which captures explicitly the idea of using state abstraction to reduce effectively the width of a problem. The framework can be combined with other forms of learning to further extend the applicability of width-based planners.

Background

In this section we define Markov decision processes and the Iterated Width (IW) algorithm, and introduce notation that will be used throughout the paper.

Markov Decision Processes

A Markov decision process (MDP) is modeled as a tuple $M = \langle S, A, P, r \rangle$, where S is a finite set of states, A is a finite set of actions, P is a transition function and r is a reward function. We assume that the transition function P is *deterministic*, i.e., $P : S \times A \to S$ maps state-action pairs to next states, while the reward function $r : S \times A \to \mathbb{R}$ maps state-action pairs to real-valued rewards.

At each time step t, a learning agent observes state $s_t \in S$, selects an action $a_t \in A$, transitions to a new state $s_{t+1} = P(s_t, a_t)$ and receives reward $r_t = r(s_t, a_t)$. The aim of the learner is to compute a policy $\pi : S \to \Delta(A)$, i.e., a mapping from states to probability distributions over actions, that maximizes some measure of expected future reward. Here, $\Delta(A) = \{\mu \in \mathbb{R}^{|A|} : \sum_a \mu(a) = 1, \mu(a) \geq 0 \ (\forall a)\}$ is the probability simplex over A.

The expected future reward associated with policy π is governed by a value function V^π, defined in each state s as

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(S_t, A_t) \middle| S_0 = s \right].$$

Here, S_t and A_t are random variables representing the state and action at time t, respectively, satisfying $A_t \sim \pi(S_t)$ and $S_{t+1} = P(S_t, A_t)$ for each $t \geq 0$, and $\gamma \in (0, 1]$ is a discount factor. The *optimal value function* V^* is given by $V^* = \max_\pi V^\pi$, and the optimal policy π^* is the argument achieving this maximum, i.e., $\pi^* = \arg\max_\pi V^\pi$.

We assume that there exists a set of features F, each with finite domain D, and a mapping $\phi : S \to D^{|F|}$ from states to feature vectors. For each feature $f \in F$ and state $s \in S$, let $\phi(s)[f] \in D$ be the value that s assigns to f. It is common to approximate the value function in state s using the feature vector $\phi(s)$ and a parameter vector θ, i.e., the estimation of the value in state s is given by $\hat{V}_\theta(s) = g(\phi(s), \theta)$ for some function g, e.g. a neural network.

We can use deterministic MDPs to model goal-directed planning tasks. Such a planning task is also defined by a set of states S, a set of actions A and a deterministic transition function P. In addition, there is a set of designated goal states $S_G \subset S$. To model the task as an MDP, we make each goal state $s_G \in S_G$ absorbing by defining the transition function as $P(s_G, a) = s_G$ for each action $a \in A$. The reward function is defined as $r(s, a) = 1$ if $P(s, a) \in S_G$ and $r(s, a) = 0$ otherwise. Hence an optimal policy attempts to reach a goal state as quickly as possible and then stay there.

Iterated Width

Iterated Width (IW) (Lipovetzky and Geffner 2012) is a forward search algorithm that explores the state space of a deterministic MDP starting from a given initial state s_0. IW was initially developed for goal-directed planning tasks, attempting to find a goal state among the set of explored states. However, the algorithm has later been adapted to deterministic MDPs by instead attempting to maximize expected future reward (Lipovetzky, Ramirez, and Geffner 2015).

In its basic form, IW is a blind search algorithm that performs breadth-first search in the space of states, starting from s_0. However, unlike standard breadth-first search, IW uses a novelty measure to prune states. The novelty measure critically relies on the feature vector $\phi(s)$ associated with each state s. Concretely, IW defines a width parameter w, and remembers all visited tuples of feature values of size w in a so-called *novelty table*. During search, a state s is considered novel if its associated feature vector $\phi(s)$ contains at least one tuple of feature values of size w that has not been visited before. IW then prunes all states that are not novel.

For a given width w, because of pruning, the number of states visited by IW(w) is exponential in w. Since the state space is usually large, IW(w) is typically provided with a search budget, and terminates when the number of visited states exceeds the budget. Without a search budget, in most domains it is computationally infeasible to execute IW(w) for $w > 2$. However, many planning benchmarks turn out to have small width, at least when considering atomic goals, and in practice they can be solved by IW(1) or IW(2).

Several researchers have proposed extensions to IW. Rollout IW (Bandres, Bonet, and Geffner 2018) simulates a breadth-first search by repeatedly generating trajectories, or rollouts, from the initial state s_0. This is useful in domains for which it is expensive to store states in memory, making it impractical to perform an actual breadth-first search. The π-IW algorithm (Junyent, Jonsson, and Gómez 2019) maintains and updates a policy π, and uses the policy to decide in which order to expand states, rather than exploring blindly.

Complexity of IW(w)

In this section we provide a tighter upper bound on the number of states visited by IW(w). We use $n = |F|$ to denote the number of features, and $d = |D|$ to denote the domain size. We also assume that at most b actions are applicable in each state s. In Lipovetzky, Ramirez, and Geffner (2015) it was shown that IW(w) generates at most $b(nd)^w$ nodes.

Proposition 1. *Let $N(n, d, w)$ denote the maximum number of* novel *states visited by IW(w) for a given pair (n, d). Then, $N(n, d, w)$ is given by the recursive formula*

$$N(n, d, 0) = 1,$$
$$N(n, d, n) = d^n,$$
$$N(n, d, w) = (d - 1)N(n - 1, d, w - 1) + N(n - 1, d, w).$$

Given $N(n, d, w)$, the number of visited states (including those pruned) is bounded by $N(n, d, w) \cdot b$. There are two base cases: $w = 0$, in which case no state is novel apart from s_0, i.e., $N(n, d, 0) = 1$, and $w = n$, in which case all states are novel, i.e., $N(n, d, n) = d^n$.

The intuition for the recursion is as follows. Consider the case where IW(w) visits the maximum number of states. Given a feature $f \in F$, we can partition the subset of novel states into two subsets: states that are novel solely due to tuples that include f, denoted by S_f, and states that are novel (in part) due to tuples that exclude f, denoted by $S_{\neg f}$.

Since f is irrelevant in $S_{\neg f}$, IW(w) would generate the same novel states even if we removed f. Thus, the maximum amount of novel states in $S_{\neg f}$ is bounded by $N(n-1,d,w)$. Regarding S_f, we can divide it into $d-1$ subsets, each corresponding to a value of f different from its initial value $v_0 = \phi(s_0)[f]$. In each subset, since the value of f is the same, the novelty test can be simplified to checking tuples of size $w-1$ of features different than f. Therefore, the maximum number of novel states in S_f is $(d-1)\cdot N(n-1,d,w-1)$.

Note that we are not decomposing the problem into multiple subproblems; rather, the recursion defines an upper bound on the number of novel states in each subset.

Theorem 1. *For n features of size d, the maximum number of novel states visited by IW(w), $0 \leq w < n$, is*

$$N(n,d,w) = \sum_{k=0}^{w} \left[\binom{n-1-k}{w-k} d^k (d-1)^{w-k} \right].$$

The proof of Theorem 1 also establishes that $N(n,d,w)$ is indeed upper bounded by $(nd)^w$, which is consistent with previous results, but we omit the proof here for lack of space.

Hierarchical IW

In this section, we present our hierarchical approach to width-based planning. We start by defining a simple algorithm for hierarchical blind search. Then, we consider using width-based planners at all levels of the hierarchy, and show its effect on the width compared to planning at a single level.

For simplicity, WLOG we assume a two-level hierarchy: a high level (h) and a low level (ℓ). Each level is defined by its own feature set (F_h and F_ℓ, with domains D_h and D_ℓ, respectively) and feature mapping ($\phi_h : S \to D_h^{|F_h|}$ and $\phi_\ell : S \to D_\ell^{|F_\ell|}$, respectively). Each state s maps to a high-level state $s_h = \phi_h(s)$ and a low-level state $s_\ell = \phi_\ell(s)$.

A Hierarchical Approach to Blind Search

Blind search methods require two components: a successor function, that given a state and an action returns a successor state (e.g. a simulator), and a stopping condition, that will stop the search, for instance, when the goal is reached or after a budget is exhausted. In order to have different search levels, we modify these two components as follows:

- **High-level successor function:** Each call to this function triggers a low level search, that runs until a new high-level state is found (i.e., a state s that maps to a different $\phi_h(s)$).
- **Low-level stopping condition:** When a different high-level state is encountered, the search is stopped, returning control to the high-level planner. This stopping condition is added to the existing stopping conditions.

The control goes back and forth between the high and low-level planners. Each time that the high-level successor function is called, the according low-level search is resumed, generating new states until a new high-level state is found. We achieve this by storing a low-level search tree for each high-level state. If the low-level search terminates without finding a new high-level state, the high-level successor function returns *null*, and the high-level state is marked as *expanded*. The high-level planner will only generate successors from non-expanded high-level states, and can resume search from any state by retrieving it from memory.

The proposed framework allows many levels of abstraction, as well as the possibility to have different search methods at each level. For instance, we could have a breadth-first search at the high level and depth-first search at the low level, or combine different width-based search methods.

Hierarchical Width

The framework in the previous section partitions the states into subsets based on high-level features. To plan over the subsets, we can use any width-based search method as a high-level planner. For instance, we can apply IW(2) at the high level and IW(1) at the low level. We denote this by HIW$(2,1)$. We next define a type of high-level feature that we call *splitting*, and compare HIW with flat IW, showing the effect of the hierarchy on the width of the problem.

Definition 1. *A high-level feature $f \in F_h$ is splitting if, for each value $v \in D_h$, the induced subset of states $\{s \in S : \phi_h(s)[f] = v\}$ is a connected graph.*

Example: consider a simple problem where an agent needs to move along a corridor of length L, pick up a key, and go back along the same path to open a door. We can describe this problem using two features: p (the position) and k (whether or not the key is held). Initially $p = 0$ and $k = 0$. The goal is $p = 0$ and $k = 1$. If $k \in F_h$, then k is splitting: when k is false, the agent can still visit all the positions of the corridor, and likewise when k is true.

Theorem 2. *If all features in F_h are splitting, HIW(w_h, w_ℓ) is equivalent to a restricted version of IW$(w_h + w_\ell)$ with tuples of w_h features from F_h and w_ℓ features from F_ℓ.*

Proof. Since each feature in F_h is splitting, when we apply IW(w_ℓ) in a high-level state s_h, the subset of states induced by s_h is connected. Since the restricted version of IW$(w_h + w_\ell)$ considers exactly w_ℓ features in F_ℓ, it will explore the same low-level states as IW(w_ℓ). At the high-level, the restricted version of IW$(w_h + w_\ell)$ considers exactly w_h features in F_h, so it will explore the same high-level states as IW(w_h). Since the tuples in IW$(w_h + w_\ell)$ involve features in both F_h and F_ℓ, each state in the low-level search of a new high-level state is novel. Hence HIW(w_h, w_ℓ) explores the same states as the restricted version of IW$(w_h + w_\ell)$. □

Example (cont.): The corridor example has width 2, since IW needs to keep track of the key and visited position jointly. This example can be solved by HIW$(1,1)$ using $F_h = \{k\}$ and $F_\ell = \{p\}$, after two low-level searches (one for $k = 0$ and one for $k = 1$), and visits the same states as IW(2).

Theorem 2 compares HIW(w_h, w_ℓ) to flat IW$(w_h + w_\ell)$ when all the features in F_h are splitting. However, this is not a necessary condition for HIW(w_h, w_ℓ) to solve problems of width $w_\ell + w_h$. Without splitting features, HIW will not generate the same nodes as the restricted version of IW, but may still find the goal. We empirically show this in the experiments section.

Theorem 3. *Let $n_h = |F_h|$ and $d_h = |D_h|$ be the number of high-level features and domain sizes, and define (n_ℓ, d_ℓ) analogously. The maximum number of novel states expanded by HIW(w_h, w_ℓ) is $N(n_h, d_h, w_h) \cdot N(n_\ell, d_\ell, w_\ell)$.*

Proof. At the high level, HIW(w_h, w_ℓ) applies IW(w_h), which expands a maximum of $N(n_h, d_h, w_h)$ novel high-level states due to Theorem 1. For each novel high-level state, HIW(w_h, w_ℓ) applies IW(w_ℓ), which expands a maximum of $N(n_\ell, d_\ell, w_\ell)$ novel low-level states. □

Note that the maximum number of novel states expanded by the unrestricted version of IW$(w_h + w_\ell)$ on the feature set $F = F_h \cup F_\ell$ is $N(n_h + n_\ell, \max(d_h, d_\ell), w_h + w_\ell)$, which is much larger than $N(n_h, d_h, w_h) \cdot N(n_\ell, d_\ell, w_\ell)$ in general.
Example: The RAM memory in Atari, used in Lipovetzky, Ramirez, and Geffner (2015), consists of $n = 128$ features with $d = 256$ values. For IW(2), an upper bound on the number of novel states is $N(n, d, w) \sim 5 \cdot 10^8$. If we identify a splitting feature and define $n_h = 1$, $n_\ell = 127$, and $w_h = w_\ell = 1$, the upper bound due to Theorems 2 and 3 is $N(n_h, d, w_h) \cdot N(n_\ell, d, w_\ell) \sim 8 \cdot 10^6$, an improvement of almost two orders of magnitude.

Incremental Hierarchical IW (IHIW)

In classical planning, the states are defined by a set of atoms, and, although one atom may be more informative than others, there is no hierarchical structure. In this section, we present a simple method for identifying relevant features that may split the state space. Then, we introduce an algorithm that performs a sequence of hierarchical searches, using the aforementioned method to discover new high-level feature candidates at each step. In the experiments section, we test the algorithm in a range of classical planning domains[1].

Discovering High-Level Features

Consider a search tree generated by IW(1) for a problem of width 2. Is it possible to identify features that split the state space, so that the problem can be solved by HIW$(1, 1)$? In this section, we present a simple method for detecting candidate abstract features from a set of features F.

We consider all trajectories in the tree and hypothesize that a feature that changes only once before a trajectory is pruned is a good candidate for a high-level feature. Consider again the corridor example in which an agent has to use a key to open a door. IW(1) prunes any trajectory that repeats a position p, and will not solve the problem. However, feature k splits the state space into two sub-problems: reaching the key (k=true), and going back to the door (k=false).

We can detect high-level features using the method detailed in Algorithm 1. For each pruned leaf node, we retrieve the features that are shared with its parent that have not appeared in that branch before. The intuition is that when a splitting feature f changes value, from v_0 to v_1, the *next* state is likely to be pruned by IW(1), since v_1 has just been observed for f, and all other features may have been visited when f took value v_0.

[1]The code for all algorithms and experiments described in this paper can be found in https://github.com/aig-upf/hierarchical-IW

Algorithm 1 Method for finding high-level features

Input: node n
$N = \emptyset$
if IsLeaf(n) & Depth(n) > 2 **then**
  P = Atoms(n) $\cap$ Atoms(Parent(n)) *// common atoms*
  if $|P| <$ |Atoms(n)| **then** *// ensure different state*
    b = Branch($tree, n$) *// get branch root→n*
    $B = \bigcup_{i=1}^{\text{Depth}(n)-2}$ Atoms($b[i]$) *// all branch atoms*
    $N = P - B$ *// keep (branch) novel atoms*
return N

Algorithm 2 Incremental Hierarchical IW Search

Initialize: $H = \emptyset$, P = List(), *solved* = **false**
while not *solved* **do**
  pruned, *solved* = HIW(w_h, w_l)
  if not *solved* **then**
    Append(P, *pruned*)
    while $H == \emptyset$ **do**
      if P is empty **then**
        return
      n = Pop(P) *// Sample pruned node*
      H = FindAbstractFeatures(n) *// Algorithm 1*
    h = Pop(H) *// Sample candidate atom*
    RestructureTree(h) *// Create high-level nodes*

An Incremental Approach

A simple algorithm that takes advantage of the previous method would be:

1. Perform an IW(1) search, if the goal is found, return.
2. Run Alg. 1 on the IW(1) tree to find high-level features.
3. Run HIW$(1, 1)$ with the discovered high-level features.

This algorithm actually finds promising candidate features for small problems. For instance, it can solve the simple corridor example. However, it fails on more complex problems, possibly because a single IW(1) search may not be sufficient to visit states that contain relevant features.

To address this, we propose a slightly more sophisticated approach, Incremental HIW (Algorithm 2), that runs a series of HIW searches. It maintains a set of high-level feature candidates H, exploits one feature candidate at a time, and discovers new relevant features when necessary. First, we run HIW$(1, 1)$, which is equivalent to IW(1) since we start with $H = \emptyset$. While the task is not solved, we randomly sample a pruned node and update H using Algorithm 1. We may repeat this operation until new feature candidates are found or there are no more pruned nodes to sample from, in which case we stop the search. Then, a feature candidate is sampled from H, and the current search tree is restructured accordingly, in order to reuse the tree in the subsequent search.

Restructuring the tree mainly involves two operations: detaching subtrees at the low level and inserting new nodes at the high level. Although this may seem costly, both operations consist of modifying the data structure, while leav-

ing the data untouched. Modifying a search tree, however, implies that the associated novelty table cannot be reused. Thus, we generate a new novelty table, if necessary, when the according tree search is resumed.

Learning with Hierarchy

In this section we show how to combine HIW with a learning-based approach that uses a policy to direct search.

Count-Based Rollout IW

Bandres, Bonet, and Geffner (2018) presented Rollout IW (RIW), a width-based algorithm that performs breadth-first search implicitly, from independent rollout trajectories. RIW(w) maintains the notion of width by modifying the definition of novelty: a state s is considered novel if any w-tuple of features of s has not appeared at a lower depth. With this, the authors achieve an algorithm that is equivalent to IW(w), but with better anytime behavior. This novelty measure actually allows for many width-based algorithms, since it unties the order of expanding nodes from the novelty test.

In our scenario, a subset of states is encapsulated under the same high-level state (i.e., a set of high-level features). Selecting one high-level state or another directly determines which low-level states are generated. In order to balance exploration within high-level states, we extend RIW with a selection method that depends on state visitation counts.

Our method, named Count-based Rollout IW, is detailed in Algorithm 3. Similar to RIW, it consists of two phases: node selection and rollout. A non-pruned node of the search tree is selected according to a softmax probability distribution inversely proportional to the visitation counts of its feature vector. Then, a rollout is performed, generating nodes until one that does not pass the novelty test is found.

When a node n with features f passes the novelty test, there may be another node deeper in the tree with the same set of features f that needs to be pruned. In the implementation, we identify such nodes by keeping a mapping N from features to unpruned nodes. When pruning a node, we leave the visitation count for features f, $C[f]$, untouched. Thus, the new node n will be selected according to the existing visitation count. This way, we ensure a balance between different high-level states. Importantly, all nodes below a pruned node are not considered anymore for selection. Therefore, pruning a node implies removing it from the mapping N together with its descendants (function *Prune*).

Algorithm 3 Count-Based Rollout IW

```
function LOOKAHEAD(N, C)
    while not StopCondition() and not Empty(N) do
        n = Select(N, C)
        if Novel(n) then
            Rollout(n, N, C)
        else
            Prune(N, n)

function SELECT(N, C)
    c = GetCounts(N, C) // Feature counts of nodes in N
    p ∝ exp(1/τ(c + 1))
    n = Sample(N, p)
    return n

function ROLLOUT(n, N, C)
    while not StopCondition() do
        C[n.features]++
        n = Successor(n)
        if n == null or not Novel(n) or Terminal(n) then
            return
        Prune(N, N[n.features])
        N[n.features] = n
```

Modifications to π-IW

Junyent, Jonsson, and Gómez (2019) introduced Policy-Guided IW (π-IW), an on-line replanning algorithm that alternates planning and learning. π-IW learns a policy π from the rewards observed in the IW tree, and uses π to guide future searches. However, in sparse-reward tasks, IW(1) may not reach any reward, especially when the planning horizon is too short. Here we extend the original π-IW in two ways: adding a better tie breaking mechanism, and a value function estimate. In experiments, we call this (flat) version π-IW+.

When no reward is found during planning, the target policy for the learning step becomes the uniform distribution, and π-IW behaves as Rollout IW. In this case, π-IW may take a step towards a region of the search tree with low node count, and presumably with less novel states, losing valuable structure information provided by the IW search. To avoid that, we modify the target policy of π-IW to use the node counts in the search tree for tie-breaking (i.e., the amount of descendants per action at the root node). The new target policy takes the form $\pi^{\text{target}} \propto \pi^{\text{rewards}} \cdot \pi^{\text{counts}}$, where the product is element-wise, and π^{counts} is a softmax distribution:

$$\pi^{\text{counts}}(a|s) = \frac{\exp\left(1/(\tau c(s,a) + 1)\right)}{\sum_{a' \in A} \exp\left(c(s,a')\right)}$$

where τ is a temperature parameter and $c(s,a)$ is the amount of nodes in the subtree of action a. The temperature parameter for π^{rewards}, which is also defined as a softmax distribution but proportional to the returns $R(s,a)$, is typically close to zero to ensure a greedy target policy (Junyent, Jonsson, and Gómez 2019). Therefore, by performing the product, we achieve the effect of tie-breaking, especially if the temperature parameter for the counts is some orders of magnitude higher than the one for the rewards.

This tie-breaking may help finding deeper rewards. However, π-IW will not exploit this information in subsequent episodes, since π^{target} is still based on the rewards of the *current* planning horizon. To amend this, we learn a value function, which we combine with the observed rewards to generate a better estimate of π^{rewards}. When backpropagating the rewards from the leaves to the root, we take the maximum between the observed rewards and our value estimate.

To learn a parameterized policy estimate $\widehat{\pi}_\theta$, we follow the same approach of Junyent, Jonsson, and Gómez (2019). Specifically, we represent $\widehat{\pi}_\theta$ using a neural network, and at each time step t, we use the cross-entropy loss to update θ:

$$\mathcal{L} = -\pi_t^{\text{target}}(\cdot|s_t)^\top \log \widehat{\pi}_\theta(\cdot|s_t).$$

The difference in our work is that the target policy now uses visitation counts for tie-breaking. We also add an ℓ-2 regularization term. To learn the value function, we take the same approach as in MuZero (Schrittwieser et al. 2020).

Policy-Guided Hierarchical IW (π-HIW)

Hierarchical IW can be straightforwardly used for online replanning. At each step, we sample an action $a \sim \pi^{\text{target}} \propto \pi^{\text{rewards}} \cdot \pi^{\text{counts}}$. To generate π^{rewards}, we need to backpropagate the rewards through the hierarchical tree. Starting from the high-level leaf nodes, we first backpropagate the rewards of the associated low-level trees. Then, to propagate this return between two high-level nodes, we feed it to the corresponding low-level leaf nodes of the high-level parent, and repeat until we reach the high-level root. To generate π^{counts}, we backpropagate the counts in a similar manner.

After executing an action a, we cache the resulting subtree for subsequent searches, similar to previous work. In this case, we need to take into account that some high-level states will not be reachable anymore, and we should thus remove them from the high-level tree before resuming the search.

Experiments in Classical Planning

In this section, we evaluate experimentally the proposed hierarchical approach. We address the following questions:

- In practice, can HIW$(1, 1)$ solve problems of width 2?
- Can Algorithm 1 find good high-level feature candidates?
- Is IHIW$(1, 1)$ a good alternative to IW(2)?

Lipovetzky and Geffner (2012) empirically showed that most classical planning problems with atomic goals present a low width. In Table 1, we reproduce such results, and compare them to our algorithm. The table consists of 36 domains from the International Planning Competitions, prior to 2012. For each domain, we show the amount of single goal instances (I), generated by splitting each instance with G goal atoms into G single goal instances. Columns 3-11 show the amount of instances solved, together with the average number of nodes and time per solved instance, for IW(1), IW(2) and IHIW$(1, 1)$. Here, IHIW$(1, 1)$ consists of two standard IW(1) searches, one at each level of abstraction.

In some domains, IW(1) has greater coverage than IW(2), e.g. in Woodworking. This is because we set a budget of $10K$ nodes, and IW(2) may exhaust the budget before finding the goal. We observe that IHIW$(1, 1)$ outperforms IW(1) in all but five domains: Barman, OpenStacks, Parking, Scanalyzer and Woodworking. Compared to IW(2), IHIW$(1, 1)$ covers more or the same number of instances in 24 out of 36 domains. In 12 cases the average number of nodes per solved instance is lower in IHIW(1,1) than in IW(2), and in 18 cases IHIW solved it faster. Note that Table 1 only reports the average time for solved instances. Thus, we may find that IHIW is quicker than IW(2) even when solving more instances.

With these results we can conclude that HIW$(1, 1)$ can solve problems of width 2 in practice, and that Algorithm 1 is a good approach to identify promising high-level features. Finally, we can state that IHIW$(1, 1)$ is an efficient alternative to IW(2).

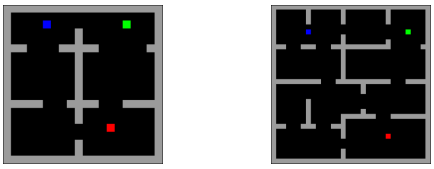

Figure 1: Snapshot of the two gridworld environments. Colors blue, red, green and gray represent the agent, key, door, and walls, respectively. The optimal policy takes 36 and 62 steps for the small (left) and large (right) tasks, respectively.

Pixel-Based Testbeds

In this section, we test our approach, π-HIW, in pixel-based gridworld environments and Atari games. We use two levels of abstraction: the high-level planner is Count-based Rollout IW (Algorithm 3) and the low-level planner is π-IW+ (i.e., Rollout IW guided by the current policy estimate). The set of abstract features $\phi_h(s)$ consists of a discretization of the image, similar to the one used in Go-Explore (Ecoffet et al. 2019, 2021), where the image is divided into tiles and the mean pixel value of each tile is taken as the feature value. Usually, this is further quantized into a smaller subset (e.g. 8 pixel values). For the low-level set of features, we follow the methodology of Junyent, Jonsson, and Gómez (2019) and define $\phi_\ell(s)$ as the boolean discretization of $z(s)$, where z is the last layer of the neural network representing $\widehat{\pi}_\theta$.

Gridworld Environments

We test our algorithm in two gridworld environments with sparse rewards (Figure 1). The agent (blue) has to pick up the key (red) and open the door (green), avoiding walls (gray). The agent is rewarded with $+1$ only when the door is reached while holding the key. Any other state has a reward of 0, except if the agent hits a wall, in which case the episode terminates with a reward of -1. We also end the episode after 200 and 500 steps for the small and large environment, respectively. The observation is a $84 \times 84 \times 3$ image and possible actions are {no-op, up, down, left, right}. The setting is similar to the one of Junyent, Jonsson, and Gómez (2019), but with larger environments and therefore sparser rewards.

We compare our hierarchical approach, π-HIW$(1, 1)$, to two baselines: π-IW, and our modified version π-IW+ that uses a value estimate and the subtree size for tie-breaking. For the latter, we use a temperature of 1 to generate π^{counts}. In order to bound the memory used by the planner, we set a maximum of 500 nodes that we keep in memory per step. The visitation count temperature used by the high-level planner (Algorithm 3) is set to 0.005. All other hyperparameters are the same as in Junyent, Jonsson, and Gómez (2019).

Figure 2 shows results for both environments. We observe that π-IW does not perform well, obtaining a reward close to zero in both environments. π-IW+ takes advantage of the value function and the tie-breaking counts and learns to solve the first task, while achieving a mean score of 0.5 for the second one in 10^6 interactions with the environment. For the hierarchical version, which also includes the aforementioned modifications, we report results of π-HIW$(1, 1)$ using

Domain	I	IW(1)			IW(2)			IHIW(1, 1)		
		Solved	Nodes	Time	Solved	Nodes	Time	Solved	Nodes	Time
8puzzle	32	40.6	34	0.00	**100**	475	0.04	**100**	**137**	**0.01**
Barman	232	**9.1**	215	0.02	**9.1**	215	0.13	**9.1**	215	**0.02**
Blocks World	302	37.4	91	0.01	79.5	1696	0.23	**96.4**	**869**	**0.06**
Cybersecurity	86	65.1	64	0.01	65.1	64	0.22	**67.4**	158	**0.02**
Depots	189	10.6	494	0.28	23.8	2393	1.58	**28.0**	**2268**	**0.97**
Driverlog	259	44.0	996	0.12	53.3	1249	0.18	**62.9**	**1085**	**0.11**
Elevator	510	0.0	-	-	11.4	5875	1.38	**16.9**	**4752**	1.79
Ferry	8	0.0	-	-	**100**	10	0.00	**100**	11	0.00
Floortile	538	96.3	515	0.04	93.5	1115	0.63	**99.3**	**567**	**0.04**
Freecell*	68	8.8	192	0.14	**22.1**	3558	4.00	19.1	504	0.48
Grid	19	5.3	2	0.00	**36.8**	2071	6.45	15.8	1244	2.51
Gripper	460	0.0	-	-	**100**	3355	1.70	**100**	**2140**	**0.36**
Logistics	249	18.1	2	0.00	**100**	763	0.16	28.5	87	0.01
Miconic	2325	0.0	-	-	0.0	-	-	**100**	2751	0.24
Mprime	50	8.0	2	0.01	18.0	3316	0.75	**20.0**	**2600**	**0.48**
Mystery	45	8.9	2	0.01	**37.8**	1200	0.57	31.1	1903	0.37
NoMystery	210	0.0	-	-	**80.0**	1917	1.61	24.8	1487	1.22
OpenStacks*	455	**0.0**	-	-	**0.0**	-	-	**0.0**	-	-
OpenStacksIPC6	1230	5.1	176	0.20	**14.2**	2637	11.46	13.8	2332	0.37
PSRsmall	316	89.9	2	0.00	92.1	2	0.00	**94.0**	3	**0.00**
ParcPrinter	990	85.6	195	0.01	84.6	695	0.63	**92.0**	**464**	**0.03**
Parking	540	**66.3**	2770	2.28	65.2	2963	5.79	**66.3**	**2770**	**2.27**
Pegsol	990	92.6	4	0.00	**100**	9	0.01	97.8	7	0.00
Pipes-NonTan	259	45.6	299	0.08	55.6	1937	0.85	**57.5**	**683**	**0.17**
Rovers*	488	31.6	2520	0.37	23.2	2504	1.59	**35.2**	2576	**0.37**
Satellite*	1324	5.7	367	0.19	7.2	675	0.23	**7.9**	1433	**0.22**
Scanalyzer	648	**99.1**	370	0.29	96.6	322	0.66	**99.1**	370	**0.28**
Sokoban	154	35.1	37	0.01	**74.0**	1049	5.36	40.3	84	0.01
Storage	240	**100**	327	1.87	**100**	1035	15.76	**100**	**327**	**1.88**
Tpp*	118	0.0	-	-	**44.9**	3313	26.01	35.6	1476	0.19
Transport	330	0.0	-	-	11.8	3765	1.20	**18.5**	4230	1.96
Trucks	345	0.0	-	-	**11.6**	5158	0.77	1.7	3342	0.47
Visitall	21880	**100**	2918	1.83	16.9	2912	1.34	**100**	2918	1.83
Woodworking	1801	**91.6**	1110	0.29	88.3	1063	3.43	**91.6**	1110	**0.29**
Zeno	219	21.0	10	0.00	**36.5**	1740	0.18	29.2	1035	0.10
		7			17			24	**12**	**18**

Table 1: Comparison between IW(1), IW(2) and IHIW(1, 1) in different classical planning domains. Column I shows the number of single goal instances. In domains with an asterisk not all available instances were evaluated due to time or memory constraints. In columns 3-11 we show, for each algorithm, the coverage in percentage, the average amount of expanded nodes, and the average time in seconds. Nodes and time values only take into account solved instances. All algorithms have a planning budget of 10,000 nodes. Best coverage and IHIW times or nodes that are lower than the ones of IW(2) are shown in bold.

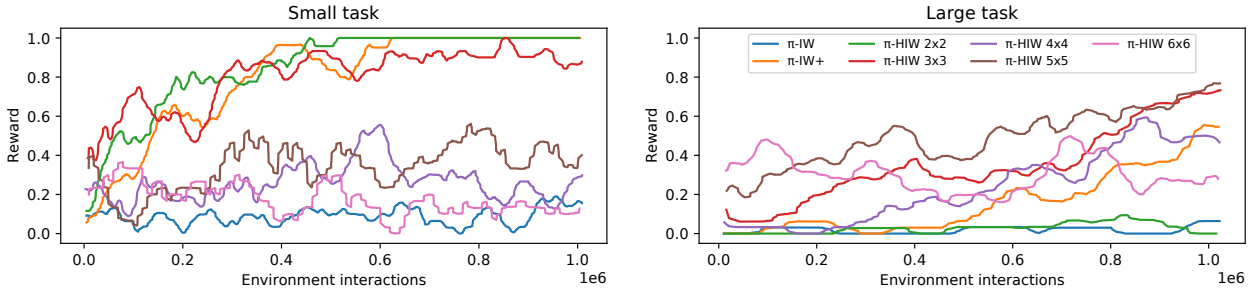

Figure 2: Comparison between π-IW, π-IW+, and π-HIW(1, 1) in the small and large gridworld environments.

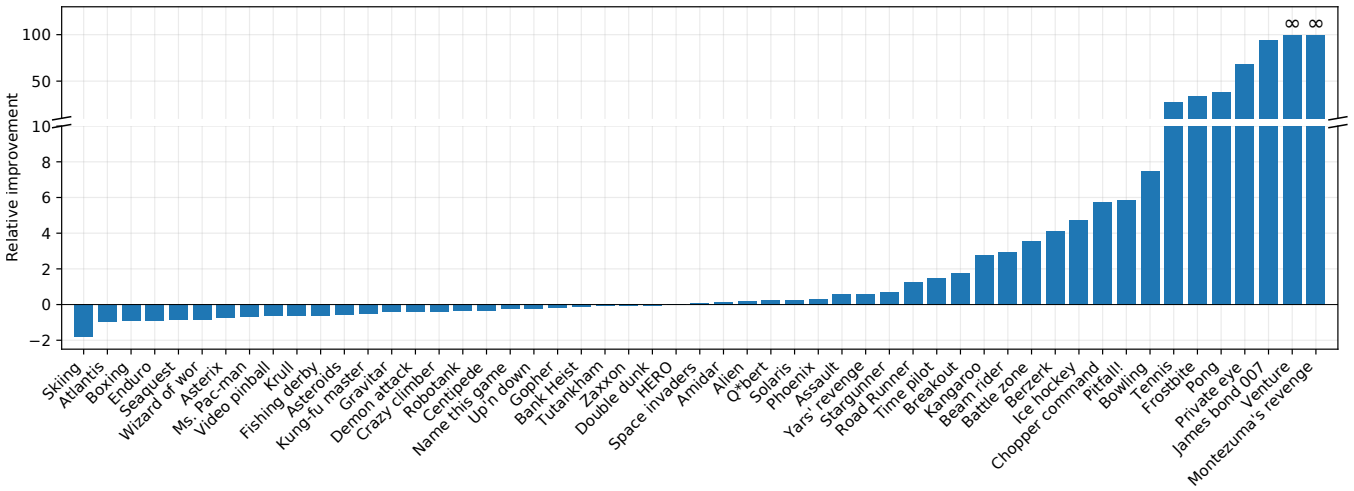

Figure 3: Comparison between π-IW and π-HIW in Atari in terms of relative improvement $(s_{\pi\text{-HIW}} - s_{\text{random}})/(s_{\pi\text{-IW}} - s_{\text{random}})$, where $s_{\pi\text{-IW}}$ and $s_{\pi\text{-HIW}}$ are the scores of the flat and hierarchical versions, respectively, and s_{random} is the score of a random agent taken from Wang et al. (2016). For Montezuma and Venture, the relative improvement is ∞, since π-IW has 0 score.

different number of tiles in $\phi_h(s)$, and 256 values per tile. We observe how, for the smaller task, 2x2 tiles is enough to get a good performance, similar to the baseline π-IW+, and the performance degrades when increasing the number of tiles. In the larger task, π-HIW$(1, 1)$ outperforms the baseline, but it needs at least 3x3 tiles to perform well.

Atari Games

We finish this section with a set of experiments using the Atari simulator. In this case, we do not optimize the hyper-parameters and define F_h using 32 pixels values and 8×11 tiles. Moreover, we use width $w_h = n = |F_h|$ at the high level, i.e., π-HIW$(n, 1)$. Even though IW(n) explores the entire high-level state space, there is a single combination of n features, which makes the novelty check efficient. In the original IW algorithm, IW(n) is equivalent to a breadth-first search without state duplicates. Nevertheless, we use Count-Based Rollout IW, described in Algorithm 3. With this, we aim to achieve effective widths larger than 2.

Figure 3 shows a comparison between π-HIW$(n, 1)$ and π-IW using the same setup as in Junyent, Jonsson, and Gómez (2019), but half the budget of simulator interactions. We observe that π-HIW improves over its predecessor π-IW in 28 games. Interestingly, games consisting of an agent moving in a fixed background present the best results e.g., James Bond, Private Eye, Pong, Frostbite, Chopper Command, etc. Within this type of games, π-HIW remarkably achieves a positive score in hard exploration games such as Montezuma's Revenge and Venture, a score not yet reported for any width-based planner. Figure 4 shows the learning curve in the game of Montezuma's Revenge. We also see an improvement in games with a moving background where the agent stays at a fixed position of the screen, for instance in Battle zone, Beam Rider, Road Runner, or Time Pilot. These results confirm that π-HIW benefits from the state abstractions provided by a simple down-sample of the image.

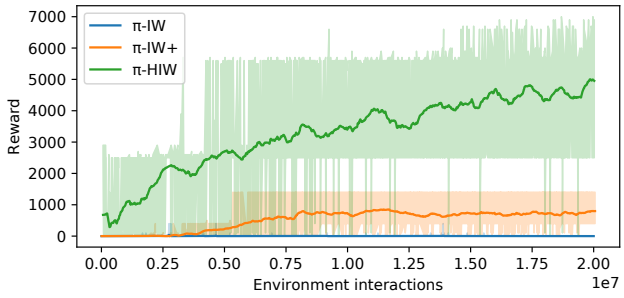

Figure 4: Performance of π-IW, π-IW+ and π-HIW in Montezuma's Revenge. Average over 5 runs with different random seeds. Shades show the maximum and minimum value.

Conclusions

We presented a novel hierarchical approach to width-based planning. Our method uses different feature mappings to create several levels of abstraction, allowing different search algorithms at different levels of the planning hierarchy. Specifically, we propose to use Iterated Width at two levels, resulting in the hierarchical search algorithm HIW(w_h, w_ℓ). We show that HIW(w_h, w_ℓ) can solve problems of width $w_h + w_\ell$ with the right choice of high-level features. Experiments in planning benchmarks show that an incremental version of HIW$(1, 1)$ is competitive with IW(2), solving single-goal instances using less time or nodes. When combined with a policy learning scheme, HIW achieves a positive score in hard exploration Atari games such as Montezuma's Revenge. For future work, a promising approach is to explore different combinations of search algorithms at different levels of the hierarchy.

Acknowledgements

V. Gómez has received funding from "La Caixa" Foundation (100010434), under the agreement LCF /PR/PR16/51110009 and is supported by the Ramon y Cajal program RYC-2015-18878 (AEI/MINEICO/FSE,UE). A. Jonsson is partially supported by Spanish grants PID2019-108141GB-I00 and PCIN-2017-082.

References

Bandres, W.; Bonet, B.; and Geffner, H. 2018. Planning With Pixels in (Almost) Real Time. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*.

Bonet, B.; and Geffner, H. 2014. Belief Tracking for Planning with Sensing: Width, Complexity and Approximations. *Journal of Artificial Intelligence Research* 50(1): 923970.

Currie, K.; and Tate, A. 1991. O-Plan: the open planning architecture. *Artificial intelligence* 52(1): 49–86.

Dittadi, A.; Drachmann, F. K.; and Bolander, T. 2020. Planning From Pixels in Atari With Learned Symbolic Representations. In *ICAPS 2020 Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning*.

Ecoffet, A.; Huizinga, J.; Lehman, J.; Stanley, K. O.; and Clune, J. 2019. Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995* .

Ecoffet, A.; Huizinga, J.; Lehman, J.; Stanley, K. O.; and Clune, J. 2021. First return, then explore. *Nature* 590(7847): 580586.

Erol, K.; Hendler, J.; and Nau, D. S. 1996. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence* 18(1): 69–93.

Fikes, R. E.; Hart, P. E.; and Nilsson, N. J. 1972. Learning and executing generalized robot plans. *Artificial intelligence* 3: 251–288.

Francès, G.; Ramrez, M.; Lipovetzky, N.; and Geffner, H. 2017. Purely Declarative Action Descriptions are Overrated: Classical Planning with Simulators. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, 4294–4301.

Geffner, T.; and Geffner, H. 2015. Width-based planning for general video-game playing. In *11th Artificial Intelligence and Interactive Digital Entertainment Conference*.

Junyent, M.; Jonsson, A.; and Gómez, V. 2019. Deep Policies for Width-Based Planning in Pixel Domains. In *29th International Conference on Automated Planning and Scheduling*, ICAPS, 646–654. AAAI Press.

Knoblock, C. A. 1990. Learning Abstraction Hierarchies for Problem Solving. In *Proceedings of the Eighth National Conference on Artificial Intelligence - Volume 2*, 923928. AAAI Press.

Korf, R. E. 1985. Macro-operators: A weak method for learning. *Artificial intelligence* 26(1): 35–77.

Lipovetzky, N.; and Geffner, H. 2012. Width and Serialization of Classical Planning Problems. In *Proceedings of the 20th European Conference on Artificial Intelligence*, 540545.

Lipovetzky, N.; and Geffner, H. 2017. Best-First Width Search : Exploration and Exploitation in Classical Planning. *Proceedings of the 31th Conference on Artificial Intelligence* 3590–3596.

Lipovetzky, N.; Ramirez, M.; and Geffner, H. 2015. Classical Planning with Simulators: Results on the Atari Video Games. In *Proceedings of the 24th International Conference on Artificial Intelligence*, 16101616.

Ramirez, M.; Papasimeon, M.; Lipovetzky, N.; Benke, L.; Miller, T.; Pearce, A. R.; Scala, E.; and Zamani, M. 2018. Integrated Hybrid Planning and Programmed Control for Real Time UAV Maneuvering. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, 13181326.

Sacerdoti, E. D. 1974. Planning in a hierarchy of abstraction spaces. *Artificial intelligence* 5(2): 115–135.

Schrittwieser, J.; Antonoglou, I.; Hubert, T.; Simonyan, K.; Sifre, L.; Schmitt, S.; Guez, A.; Lockhart, E.; Hassabis, D.; Graepel, T.; Lillicrap, T.; and Silver, D. 2020. Mastering atari, go, chess and shogi by planning with a learned model. *Nature* 588(7839): 604609.

Shleyfman, A.; Tuisov, A.; and Domshlak, C. 2016. Blind Search for Atari-Like Online Planning Revisited. In *International Joint Conference on Artificial Intelligence*, 3251–3257.

Wang, Z.; Schaul, T.; Hessel, M.; Hasselt, H.; Lanctot, M.; and Freitas, N. 2016. Dueling Network Architectures for Deep Reinforcement Learning. In *Proceedings of The 33rd International Conference on Machine Learning*, 1995–2003.