Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)

Deep Policies for Width-Based Planning in Pixel Domains

Miquel Junyent
Grupo Oesía
Barcelona, Spain
mjunyent@oesia.com

Anders Jonsson
Universitat Pompeu Fabra
Barcelona, Spain
anders.jonsson@upf.edu

Vicenç Gómez
Universitat Pompeu Fabra
Barcelona, Spain
vicen.gomez@upf.edu

Abstract

Width-based planning has demonstrated great success in recent years due to its ability to scale independently of the size of the state space. For example, Bandres et al. (2018) introduced a rollout version of the Iterated Width algorithm whose performance compares well with humans and learning methods in the pixel setting of the Atari games suite. In this setting, planning is done on-line using the "screen" states and selecting actions by looking ahead into the future. However, this algorithm is purely exploratory and does not leverage past reward information. Furthermore, it requires the state to be factored into features that need to be pre-defined for the particular task, e.g., the B-PROST pixel features. In this work, we extend width-based planning by incorporating an explicit policy in the action selection mechanism. Our method, called π-IW, interleaves width-based planning and policy learning using the state-actions visited by the planner. The policy estimate takes the form of a neural network and is in turn used to guide the planning step, thus reinforcing promising paths. Surprisingly, we observe that the representation learned by the neural network can be used as a feature space for the width-based planner without degrading its performance, thus removing the requirement of pre-defined features for the planner. We compare π-IW with previous width-based methods and with AlphaZero, a method that also interleaves planning and learning, in simple environments, and show that π-IW has superior performance. We also show that π-IW algorithm outperforms previous width-based methods in the pixel setting of Atari games suite.

Introduction

Width-based search algorithms have recently emerged as a state-of-the-art approach to automated planning (Lipovetzky and Geffner 2012). These algorithms assume that states are factored into *features* and rely on the concept of *state novelty*, which measures how novel a state is with respect to the states already visited during search. More precisely, a state is novel if at least one tuple of feature values appears for the first time during search, otherwise the state is pruned. The *width* of an algorithm bounds the size of the tuples used for novelty tests. Crucially, the complexity of width-based algorithms is exponential in the width, but independent of the size of the state space.

The defining trait of width-based algorithms is the ability to perform structured exploration of the state space, quickly reaching distant states that may be important for achieving planning goals. In classical planning, width-based search has been incorporated in a heuristic search framework, with algorithms such as Best-First Width Search (Lipovetzky and Geffner 2017) achieving remarkable results at the 2017 International Planning Competition.

Width-based search has also been adapted to reward-driven problems with unknown dynamics, relying on a simulator for interaction with the environment (Francès et al. 2017). This enables width-based search in domains intended for reinforcement learning, such as the Arcade Learning Environment (Bellemare et al. 2013). Several researchers have adapted the Iterated Width (IW) algorithm to this setting, using the bytes of the internal RAM state as features (Lipovetzky, Ramirez, and Geffner 2015; Shleyfman, Tuisov, and Domshlak 2016; Jinnai and Fukunaga 2017). Width-based search is used to select the action that maximizes long-term reward, and search is restarted after each action execution.

Since humans do not have access to the RAM state when playing a game, learning algorithms instead define the state in terms of the pixels of the game screen. Recently, Bandres, Bonet, and Geffner (2018) proposed a modified version of IW, Rollout IW, that uses pixel-based features and achieves comparable results to learning methods in almost real-time. The main bottleneck of IW is the cost of resetting the simulator to a previous state, and Rollout-IW gets around this by resetting the state only once before each rollout.

In spite of these achievements, IW remains a purely exploratory algorithm that is highly dependent on the quality of the given features and that expands actions at random. Since IW is sensitive to the order of action expansion, there is an opportunity to perform a more informed action selection.

In this work we leverage recent progress in deep reinforcement learning to train a *policy* π in the form of a neural network (NN) whose inputs are the pixels of an image, such as the Atari game screen. The policy is trained on the actions output by IW, reinforcing action trajectories that have proven promising in the past. The resulting algorithm, π-IW, uses the policy π to select actions during width-based search, resulting in more informed exploration of the state space.

We also show that the last intermediate layer of the

learned policy NN can be used as features for IW. Surprisingly, such dynamic features are often competitive with hand-crafted features, maintaining or sometimes increasing the performance of IW. In some domains, we even show that the dynamic features effectively reduce the width of a problem. Experimental results indicate that our approach to pixel-based planning is highly competitive in the Atari suite.

Background

In this section, we review the fundamental concepts of width-based planning and Markov decision processes.

Iterated Width

Iterated Width (IW) is a pure exploration algorithm originally developed for goal-directed planning problems with deterministic actions (Lipovetzky and Geffner 2012). It requires the state space to be factored into a set of *features* Φ. All features are usually assumed to have the same domain D, e.g. binary ($D = \{0, 1\}$) or integer ($D = \mathbb{Z}$). The original algorithm consists of a sequence of calls IW(i) for $i = 0, 1, 2, \ldots$ until a termination condition is reached. IW(i) performs a standard breadth-first search (BrFS) from a given initial state s_0, but prunes states that are not *novel*. When a new state s is generated, IW(i) contemplates all n-tuples of atoms of s with size $n \leq i$. The state is considered novel if at least one tuple has not appeared in the search before, otherwise it is pruned. This notion of novelty has been extended to produce several variants of the IW algorithm, e.g. those explained in the next sections.

IW(i) is thus a *blind* search algorithm that eventually traverses the entire state-space if we make i large enough. The traversal depends on how the states are structured, i.e., which features are used to represent the states, and on the order in which states are expanded. Each iteration IW(i) is an i-width BrFS that is complete for problems whose width is bounded by i and visits at most $\mathcal{O}((|\Phi| \cdot |D|)^i)$ states, where $|\Phi|$ is the number of features and $|D|$ is the size of their domains (Lipovetzky and Geffner 2012). Interestingly, most planning benchmarks turn out to have very small width and, in practice, they can be solved in linear or quadratic time.

Markov Decision Processes

A Markov decision process (MDP) is a tuple $M = \langle S, A, P, r \rangle$, where S is the finite state space, A is the finite action space, $P : S \times A \to \Delta(S)$ is the transition function, and $r : S \times A \to \mathbb{R}$ is the reward function. Here, $\Delta(S) = \{\mu \in \mathbb{R}^S : \sum_s \mu(s) = 1, \mu(s) \geq 0 \ (\forall s)\}$ is the probability simplex over S. Although MDPs are stochastic, planning algorithms typically assume that transitions are *deterministic*; this is the case for our algorithm as well.

At time t, the learner observes state $s_t \in S$, selects action $a_t \in A$, moves to the next state $s_{t+1} \sim P(\cdot|s_t, a_t)$, and obtains reward r_{t+1} such that $\mathbb{E}[r_{t+1}] = r(s_t, a_t)$. The aim of the learner is to select actions that maximize the expected cumulative discounted reward $\mathbb{E}\left[\sum_{k=t}^{\infty} \gamma^{k-t} r_{k+1}\right]$, where $\gamma \in (0, 1]$ is a discount factor. In the reinforcement learning setting, both the transition function P and the reward function r are unknown to the learner. Instead, the learner relies on a simulator in order to sample both functions.

The decision strategy is represented by a *policy* $\pi : S \to \Delta(A)$, i.e. a mapping from states to probability distributions over actions. In state s_t, action a_t is selected with probability $\pi(a_t|s_t)$. In a realistic setting, the state space is too large to explicitly represent the policy for each state. Instead, the policy estimate $\widehat{\pi}_\theta$ maintains a set of *parameters* θ that are combined with the state features to produce a distribution over actions. It is common to use an NN to represent the policy estimate, and define $\widehat{\pi}_\theta(a|s) = \frac{\exp(h_a(s,\theta)/\tau)}{\sum_{b \in A} \exp(h_b(s,\theta)/\tau)}$ as the softmax combination of the NN outputs $h_a(s, \theta)$, $a \in A$, where τ is a temperature that controls exploration.

Related Work

Our work lies at the intersection between width-based planning and policy learning in RL. We now review some relevant work of these two lines of research.

Width-Based Planning for MDPs

Several extensions of the original IW algorithm have been developed over the last years. One line of research has focused on the popular Atari 2600 benchmark (Bellemare et al. 2013), predominantly in the on-line planning setting, where actions are selected after a lookahead, and using the 128 bytes of the RAM memory to represent the state of the game.

First, Lipovetzky, Ramirez, and Geffner (2015) extended the original IW algorithm to MDPs by associating a reward $R(s)$ to each state s during search, equivalent to the reward $\sum_{t=0}^{d-1} \gamma^t r_{t+1}$ accumulated on the path from s_0 to $s_d = s$, where d is the depth of s in the search tree. Later, Shleyfman, Tuisov, and Domshlak (2016) introduced prioritized-IW (p-IW) that approximates breadth-first search with duplicate detection and state reopening. Jinnai and Fukunaga (2017) further extended p-IW by learning to avoid actions that lead to the same successor state. Both extensions considerably outperformed the original IW in the RAM setting of the Atari benchmark.

More recently, Bandres, Bonet, and Geffner (2018) adapted IW for the *pixel setting*. Specifically, their method uses the (binary) B-PROST features (Liang et al. 2016). The authors introduced a Rollout version of IW(1) which *emulates* the breadth-first traversal of IW, by keeping track of the minimum depth at which a feature is found for the first time and extending the notion of novelty accordingly. The pruned states are kept as leaves of the tree, and are considered as candidates for states with highest reward. This contribution has brought width-based planning closer to the RL setting. For example, Rollout-IW can deal with trajectories, as most RL methods do, with the restriction that a simulator needs to be reset to a previous state. Furthermore, the algorithm can work directly with pixels, instead of the RAM states, which are not always accessible. This Rollout version of IW is the basis of our work.

Besides the developments on the Atari benchmark, other works consider the discovery of features of IW and possible methods to inform the purely exploratory search of the original IW algorithm. For example, Francès et al. (2017)

discovered numeric features by means of a IW(1) search by keeping track of trajectories reaching a state where at least one of the goals in the task is true. Also, in Lipovetzky and Geffner (2017), an *informed* variant of IW was proposed where a standard goal-oriented search (exploitation) and width-based search (structural exploration) are combined to yield a search scheme, best-first width search, that results in classical planning algorithms that outperform many state-of-the-art planners.

These latter contributions have been developed in the classical planning setting. Our interest is to exploit the potential of deep learning methods for learning representations and policies directly from screen states by leveraging the structured exploration benefits of width-based planning.

Policy Iteration for Multi-Step Lookahead Policies

A natural way to combine planning and learning is to treat the planner as a "teacher" that provides *correct* transitions that are used to learn a policy, as in imitation learning (Ross, Gordon, and Bagnell 2011; Guo et al. 2014). A prominent example of this approach is AlphaGo, which achieved superhuman performance in the game of Go (Silver et al. 2016) by combining supervised learning from expert moves and self-play. AlphaZero (Silver et al. 2017b), a version of the same algorithm that learned solely from self-play, has outperformed previous variants, also showing stunning results in Chess and Shogui (Silver et al. 2017a). This combination of planning and learning can also be interpreted as a policy iteration algorithm with policy updates that consider multi-step (lookahead) greedy policies (Efroni et al. 2018), instead of the classical 1-step greedy policy improvement with respect to a value function estimate (Sutton and Barto 1998).

At every iteration t, AlphaZero generates a tree using Monte-Carlo tree search (MCTS) (Browne et al. 2012), guided by a policy and a value estimate. It keeps a visit count on the branches of the tree, and uses it to explore less frequent states (Kocsis and Szepesvári 2006) and to generate a target policy π_t^{target}. After tree expansion, an action is selected at the root following π_t^{target}, and the resulting subtree is kept for the next iteration. At the end of the episode, the win/loss result z_t is recorded and all transitions $(s_t, \pi_t^{target}, z_t)$ are added to a dataset. In parallel, the policy and value estimates are trained in a supervised manner with minibatches sampled from the dataset.

AlphaZero is known to perform well in two-player games. However, not much is known about its performance on general sequential decision problems. Here we show limitations of AlphaZero in simple environments. We postulate that its poor performance can be caused by the (unstructured) exploration of MCTS, in the sense that MCTS considers the state representation as a black-box.

Policy-Guided Iterated Width (π-IW)

In spite of its success, IW does not learn from experience, so its performance does not improve over time. When expanding a node, IW generates all possible states, one per action. The recently proposed Rollout IW algorithm (Bandres, Bonet, and Geffner 2018) generates whole branches by expanding one state per node, but selects actions randomly.

We now present our algorithm, Policy-Guided Iterated Width (π-IW), that enhances Rollout IW by incorporating an action selection policy, resulting in an *informed* IW search. More precisely, we leverage the exploration capacity of IW to train a policy estimate $\widehat{\pi}_\theta$, which is used in turn to guide subsequent search. We consider tuples of size 1, i.e., IW(1), which keeps planning tractable. Similar to Rollout IW, π-IW requires a simulator that provides the successor of a state s and a representation of s in terms of features Φ. Also, π-IW operates in an online setting, i.e., at each time-step, a planning step is followed by an action execution step. Importantly, we reset the novetly table after each action step, which enables us to solve problems of width higher than one.

Below we describe the two basic steps of the π-IW algorithm, and present a mechanism for extracting a feature space from the policy $\widehat{\pi}_\theta$. This second use of the policy is beneficial if no feature representation is initially available.

Planning Step

The planning step of π-IW is very similar to Rollout IW. For clarity, we describe four separate functions in Algorithm 1[1]. At every iteration of *Lookahead*, Rollout IW first selects a node n for expansion, and then performs a rollout from n. *Select* samples actions to traverse the tree until a state-action pair (n, a) is reached that has not yet been expanded. *Rollout* then samples actions starting from (n, a) until a state is reached that is either terminal or not novel. At that point, the final node is marked as *solved* and the process restarts until all nodes have been solved or a maximum budget of time or nodes is exhausted.

Following Bandres, Bonet, and Geffner (2018), a state is considered novel if one of its atoms is true at a smaller depth than the one registered so far in the novelty table. A node that was already in the tree will not be pruned if its depth is exactly equal to the one in the novelty table for one of its atoms (this condition appears on the third line of *Check_novelty*).

The only difference between Rollout IW and π-IW is how the function *Sample_action* is defined. In the original Rollout IW, *Sample_action* returns an action sampled with uniform probability, whereas π-IW uses a softmax policy $\widehat{\pi}_\theta(a|s_n) \propto \exp(h_a(s_n, \theta)/\tau)$, where h_a, $a \in A$, are the output logits of the NN and τ is a temperature parameter. Our planning step thus becomes the original Rollout IW in the limit $\tau \to \infty$. Just as in Rollout IW, actions that lead to nodes labelled as solved should not be considered. Thus, we set probability $\widehat{\pi}_\theta(a|s_n) = 0$ for each solved action a and normalize $\widehat{\pi}_\theta$ over the remaining actions before sampling.

Every time a node is labelled as solved, we try to propagate the label along the branch to the root (*Solve_and_propagate_label*). Each node of the branch will be marked as solved if all of its children appear as solved. Thus, the propagation of the label stops when at least one child has not yet been pruned. Initially, all nodes of the cached tree are marked as not solved, except for the ones that are terminal (*Initialize_labels*).

[1]The full code is available at https://github.com/aig-upf/pi-IW.

Algorithm 1 Planning step of π-IW(1)

```
function LOOKAHEAD(tree)
    Initialize_labels(tree)
    D := Make_empty_novelty_table()
    while within_budget and ¬tree.root.solved do
        n, a := Select(tree.root, D)
        if a ≠ ⊥ then
            Rollout(n, a, D)
function SELECT(n, D)
    loop
        novel := Check_novelty(D, n.atoms, n.depth, false)
        if is_terminal(n) or ¬novel then
            Solve_and_propagate_label(n)
            return n, ⊥
        a := Sample_action(n)
        if n[a] in tree then
            n := n[a]
        else
            return n, a
function ROLLOUT(n, a, D)
    while within_budget do
        n := expand_node(n, a)
        n.solved := false
        novel := Check_novelty(D, n.atoms, n.depth, true)
        if is_terminal(n) or ¬novel then
            Solve_and_propagate_label(n)
            return
        a := Sample_action(n)
function CHECK_NOVELTY(D, atoms, d, is_new)
    novel := false
    for f in atoms do
        novel := novel ∨ d < D[f] ∨ (¬is_new ∧ d = D[f])
        if d < D[f] ∧ is_new then
            D[f] := d
    return novel
```

Learning Step

Once the tree has been generated, the discounted rewards are backpropagated to the root: $R_i = r_i + \gamma \max_{j \in children(i)} R_j$. In general, a target policy $\pi_t^{target}(\cdot|s_t)$ can be induced from the returns at the root node by applying another softmax function, although in our experiments we applied the deterministic version (with $\tau \to 0$). The state s_t is stored together with the target policy in a dataset to train the model in a supervised manner. We use the cross-entropy error between the induced target policy $\pi_t^{target}(\cdot|s_t)$ and the current policy estimate $\widehat{\pi}_\theta(\cdot|s_t)$ to update the policy parameters θ, defining a loss function

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

In our experiments, we also add an ℓ-2 regularization term to avoid overfitting and help convergence. The model is trained by randomly sampling transitions from the dataset. The planning and learning steps can be executed in parallel, as in AlphaZero, or sequentially. In our experiments we choose the latter, sampling a batch of transitions at each iteration. We keep a maximum of T transitions, discarding outdated transitions in a FIFO manner.

Finally, a new root is selected from the nodes at depth 1 by selecting an action $a_t \sim \pi_t^{target}(\cdot|s_t)$, and the resulting subtree is kept for the next planning step. Cached nodes are not added to the novelty table, since it has been argued in previous work that this increases exploration and hence performance (Lipovetzky, Ramirez, and Geffner 2015). Note that cached nodes will contain outdated information. We did not find this to have a great impact on performance, and one possibility could be to rerun the model on all nodes of the tree at regular intervals (this is not done in our experiments).

Dynamic Features

The quality of the transitions recorded by IW greatly depends on the feature set Φ used to define the novelty of states. For example, even though IW has been applied directly to visual (pixel) features (Bandres, Bonet, and Geffner 2018), it tends to work best when the features are *symbolic*, e.g., when the RAM state is used as a feature vector (Lipovetzky, Ramirez, and Geffner 2015). Symbolic features make planning more effective, since the width of a problem is effectively reduced by the information encoded in the features. However, how to automatically learn powerful features for this type of structured exploration is an open challenge.

Unlike previous width-based methods, π-IW can use the representation learned by the policy NN to define a feature space, as in representation learning (Goodfellow, Bengio, and Courville 2016). With this dependence, the behavior of IW effectively changes when interleaving policy updates with runs of IW. If appropriately defined, these features should help to distinguish between important parts of the state space. In this work, we extract Φ from the last hidden layer of the NN. In particular, we use the output of the rectified linear units that we subsequently discretize in the simplest way, resulting in binary features (0 for zero outputs and 1 for positive outputs).

Experiments

In this section, we evaluate the performance of Policy-Guided Iterated-Width (π-IW) in different settings. First, we consider a simple problem where we compare our method against AlphaZero and current width-based methods. Second, we present results in the Atari 2600 testbed. The following questions are addressed:

- How do the different types of exploration (structured for π-IW and unstructured for MCTS) affect the performance of both algorithms?
- What is the benefit of learning a policy to guide the IW planner?
- Are the learned (dynamic) features effective? Is it possible to learn them without degrading the performance?

To answer the previous questions, we define a grid-like environment where an agent (blue pixel) has to navigate to first pick up a key (red pixel) and then go through a door (green pixel). An episode terminates with a reward of $+1$

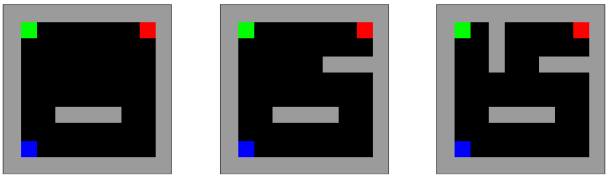

Figure 1: Snapshot of three versions of the maze. The blue, red and green squares represent the agent, the key and the door, respectively.

when the goal is accomplished, with a reward of -1 when a wall is hit, or with no reward after a maximum of 200 steps is reached. Intermediate states are *not* rewarded (including picking up the key), which makes the the problem more challenging. The observation is an 84×84 RGB image and possible actions are no-op, going up, down, left or right. See Figure 1 for an example.

We first analyze π-IW using static and dynamic features. For the first case, we take the set of BASIC features (Bellemare et al. 2013), where the input image is divided in tiles and an atom, represented by a tuple (i, j, k), is true if color k appears in the tile (i, j). In our simple environment, we make the tiles coincide with the grid, and we call this variant π-IW(1)-BASIC. For the second case, we take the (discretized) outputs of the last hidden layer of the policy network as binary feature vectors. We call this variant π-IW(1)-dynamic.

π-IW Can Reduce the Width of a Problem

Our first example is a simple corridor where the agent is located between a key and a door (see Figure 2). Using the BASIC features, this problem has width 2, since the agent needs to keep track of the paired features *having the key* (final pixel in blue or red) and *visited position* (color blue or other) jointly. Therefore, it is not solvable by IW(1) in the classical (off-line) setting. However, in the online planning setting (where novelty tables are reset after each action execution step), the task is solvable by IW(1). We are interested in analyzing the behavior of π-IW using dynamic features.

As expected, one planning step of π-IW using the set of BASIC features does not reach the goal, and results in a trajectory pruned at s_8, two steps after picking up the key. Similarly, π-IW(1)-dynamic is unable to generate the optimal plan in its first planning step, since the initial features are uninformed. However, using a sufficient number of features (13 boolean features in this example), after learning, π-IW(1)-dynamic *solves the task using a single planning (offline) step* in five out of five cases. This shows that the problem width is *reduced* from 2 to 1 in the learned representation of the policy. This is remarkable, since there is no explicit term in the loss function that encourages the policy to generate such a representation.

π-IW Improves MCTS Exploration

We now compare the two π-IW variants with current width-based methods and AlphaZero (Silver et al. 2017a) in a more complex task. We consider three variants of a maze, with increasing difficulty, shown in Figure 1. Although AlphaZero

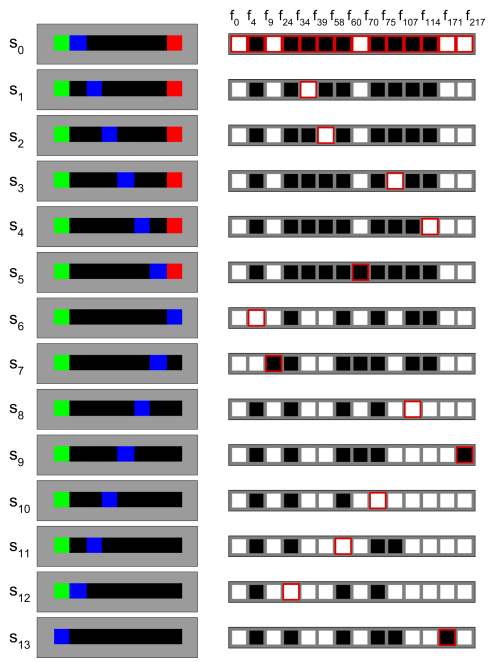

Figure 2: Feature learning in a corridor task. Left: from top to bottom, states expanded by π-IW(1) in one planning step once the policy has been trained. Right: subset of learned features for each state. Novel features at each step are marked in red.

was originally designed for two-player zero-sum games, it can be easily extended to the MDP setting. Each time a node s is generated, the statistics W_n of all nodes in the trajectory from the root to s are updated with the value of s. Since in the MDP setting there are rewards in the intermediate edges, we update W_n with the discounted sum of rewards of all edges between nodes n and s, including the value of s (e.g. $W_1 \leftarrow W_1 + r_1 + \gamma r_2 + \gamma^2 v_3$).

AlphaZero controls the balance between exploration and exploitation by a parameter p_{uct} together with a temperature parameter in the target policy τ, similar to ours. In the original paper, τ is set to 1 for a few steps at the beginning of every episode, and then it is changed to an infinitesimal temperature $\tau = \epsilon$ for the rest of the game (Silver et al. 2017b). Nevertheless, we achieved better results in our experiments with AlphaZero using $\tau = 1$ for the entire episode.

Both π-IW and AlphaZero algorithms share the same NN architecture and hyperparameters, specified in Table 1. We use two convolutional and two fully connected layers as in Mnih et al. (2013), which are trained using the non-centered version of the RMSProp algorithm. All hyperparameters of AlphaZero and π-IW have been optimized for the second version of the game, with two walls.

Figure 3 shows results comparing π-IW against existing width-based algorithms and AlphaZero for the three mazes (we also performed experiments using different random configurations of the walls and the results remained consistent). The top row shows the average reward as a function of the number of interactions with the environment. As expected,

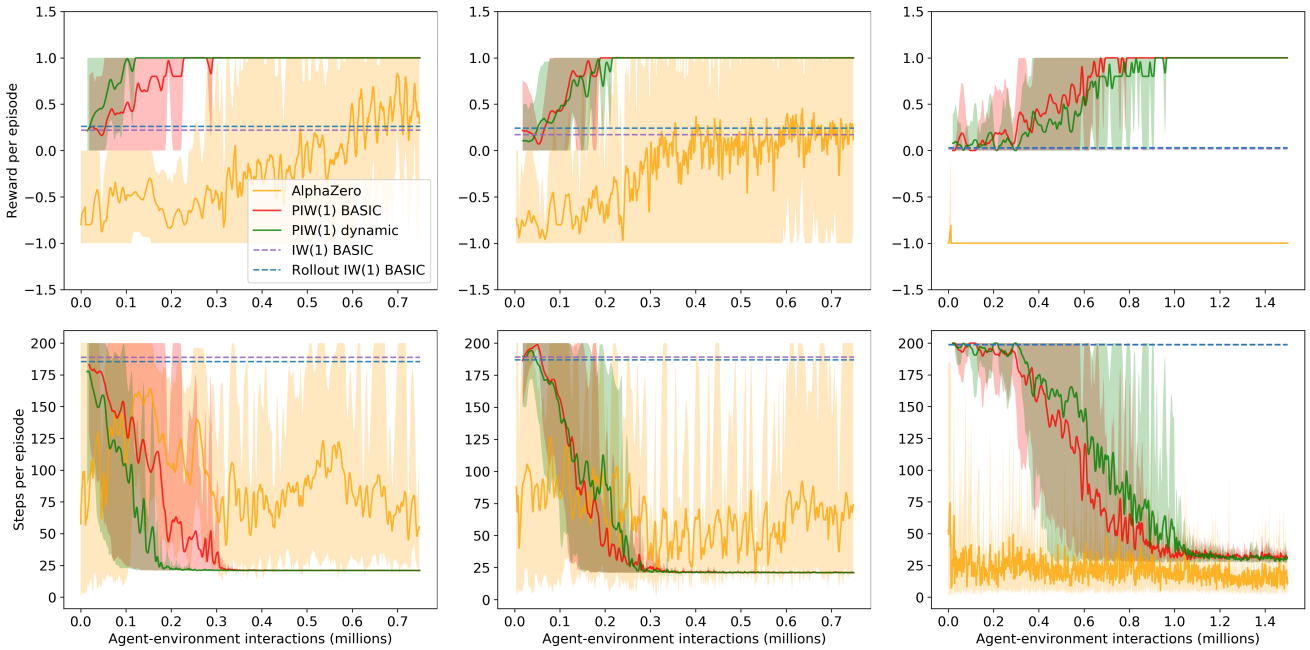

Figure 3: Top: Performance of width-based planning algorithms and AlphaZero in the three simple mazes is shown in the first row, increasing difficulty from left to right (1, 2 and 3 walls respectively). Bottom: The number of transitions per episode. We plot averages over five runs for learning algorithms, and shades show the minimum and maximum values. IW and RolloutIW baseline results are averages over 100 runs.

Hyperparameter	Value	Algorithm
Discount factor	0.99	Both
Batch size	32	Both
Learning rate	0.0005	Both
Clip gradient norm	40	Both
RMSProp decay	0.99	Both
RMSProp epsilon	0.1	Both
Tree budget nodes	50	Both
Dataset size T	10^3	Both
L2 reg. loss factor	10^{-3}	Both
Tree policy temp. τ	1	Both
p_{uct}	0.5	AlphaZero
Diritchlet noise α	0.03	AlphaZero
Noise factor	0.25	AlphaZero
Value loss factor	1	AlphaZero

Table 1: Hyperparameters used for π-IW and AlphaZero.

the number of interactions required to solve the problem increases with the level of difficulty. While π-IW variants reach top performance in less than $2 \cdot 10^5$ interactions in the first and the second versions of the game, they require nearly 10^6 to fully solve the third version.

The performance of the two other width-based algorithms (IW and Rollout IW) is independent of the number of interactions. These algorithms, despite using the structured exploration of IW, are limited to width 1 and do not learn from previous visited states. Consequently, only in a very few cases (depending on the tie breaking, basically), they find

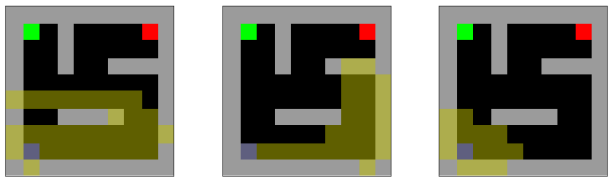

Figure 4: Three illustrative examples of visited states (in yellow) during one planning step before learning: **(left)** π-IW(1)-BASIC, **(middle)** π-IW(1)-dynamic, and **(right)** AlphaZero. The π-IW variants explore a larger region of the state space than AlphaZero.

the correct sequence of actions. Finally, AlphaZero shows less stable behavior and is unable to solve the most difficult scenario (rightmost plot). Comparing the two π-IW variants we observe few differences. This is remarkable, since it indicates that in this simple maze the features used by IW can be learned easily from scratch.

Figure 3 (bottom row) compares the number of steps per episode as a function of the number of interactions. For all versions of the game, we observe a decreasing trend in the π-IW variants until the optimal policy is learned. In this case, episodes require progressively fewer steps because the learned policy converges to the optimal one. The convergence occurs in alignment with the reward per episode (top row) and again, we observe no significant difference between π-IW(1)-BASIC and π-IW(1)-dynamic. In contrast, AlphaZero shows a more irregular behavior. In the hardest

instance, it ends up hitting a wall most of the time.

To illustrate the benefits of the structured exploration of π-IW compared to MCTS, we analyze the sample trajectories from the different algorithms after the first planning step, before any learning. We calculate the length of the trajectories (removing repeated states). On average, AlphaZero produces a tree whose longest trajectory is 3.83 (stdev $= 2.15$). On the other hand, the longest trajectories of π-IW are 7.3 (stdev $= 2.5$) and 7.02 (stdev $= 2.25$), respectively for BASIC and dynamic features (averages over 100 runs). Figure 4 illustrates this. Although there is no guidance towards rewarding states for any of the three algorithms (since the NNs have not been trained yet), π-IW reaches deeper parts of the state space than AlphaZero thanks to the structured exploration. As before, we do not observe significant differences between using handcrafted features or the ones extracted from the NN.

From these results we can draw the following conclusions. First, existing width-based algorithms can be significantly improved by incorporating a guiding policy, as in π-IW. Second, we have shown that for this simple problem, a small set of features can be effectively learned without degrading the performance of π-IW. Finally, π-IW outperforms AlphaZero because it uses the structure of the state to explore more systematically and reach deeper states. In contrast, the exploration of MCTS needs to go through an optimal branch several times to increase its probability for action selection, since the policy estimate is based on counts.

π-IW on Atari Games

We end this experimental section presenting results of π-IW(1)-dynamic on the pixel setting of the Atari suite. The aim of this section is to compare the performance of π-IW on a more challenging benchmark with existing width-based algorithms that use (predefined) pixel-based features. In particular, we consider IW and Rollout IW from Bandres, Bonet, and Geffner (2018). We do not provide results of AlphaZero in this benchmark, since our preliminary analysis showed poor performance, and the amount of hyperparameters to tune is considerably higher compared to π-IW.

We focus on a similar setting as in Bandres, Bonet, and Geffner (2018), where a short budget is given to the planner, although we do not aim to plan in real time. In our case, we set the budget of expanded nodes to 100, resulting in approximately one second per transition, considering *both planning and learning steps*. Note that this budget is very small compared to existing RAM-based methods, that allow 30,000 expanded nodes at each step (Lipovetzky, Ramirez, and Geffner 2015; Shleyfman, Tuisov, and Domshlak 2016).

Table 2 shows results comparing π-IW using dynamic features with IW and Rollout IW using the B-PROST feature set. All hyperparameters are kept the same as in Table 1 except for tree budget $= 100$, $T = 10^4$, $\tau = 0.5$, and frameskip $= 15$. The inputs of the NN are the last 4 grayscale frames stacked to form a 4-channel image. Results of π-IW are an average of the last 10 episodes for 5 runs with different random seeds. Performance is measured after 40M generated nodes, i.e., interactions with the simulator (excluding skipped frames).

First, if we compare π-IW against the methods that use a budget of 0.5 seconds (1st and 3rd columns vs 5th column), we observe that π-IW systematically outperforms both IW and rollout variants (see underlined values). Only in 11 games is either IW or Rollout IW better than our method, and only in two cases are both better. This shows that better performance can indeed be achieved at the cost of training the policy and learning the features.

Second, we observe (in bold) that π-IW outperforms all other methods in 22 games, and performance is comparable to the non-guided approaches in most other games (2nd and 4th columns vs 5th column). Remarkably, this is achieved with significantly less computational budget (approximately 30 times less) and without the need of predefined features.

These results suggest that a guiding policy can be beneficial, not only in terms of computational budget, but also in terms of the learned representation (in our case, the simple discretized features of the hidden layer) that can be directly exploited by the IW planner.

Conclusions and Future Work

We have presented π-IW, an algorithm that effectively combines planning and learning. Our approach learns a compact policy using the exploration power of IW(1), which helps reaching distant high-reward states. We use the transitions recorded by IW(1) to train a policy in the form of a neural network. Simultaneously, the search is informed by the current policy estimate, reinforcing promising paths.

We have shown that the learned representation by the policy network can be used as a feature space for IW, removing the need of pre-defined features. Interestingly, π-IW can even learn representations that reduce the width of a task. In our simple example, the minimum number of binary features required to decrease the width was determined by the distance to a rewarded state. For complex problems with a larger horizon, one could use more features, or even consider different feature discretizations other than binary ones. We believe this is a promising line of future research.

Our algorithm operates in a similar manner to AlphaZero, except that the exploration relies on the pruning mechanism of IW, it does not keep a value estimate, and the target policy is based on observed rewards rather than visitation counts. Compared to AlphaZero and previous width-based methods, π-IW has superior performance in simple environments. In the Atari 2600 games, π-IW achieves a similar performance to Rollout IW with a much smaller planning budget and without the need to provide pre-defined features.

In the future, we would like to investigate the use of a value estimate in our algorithm, and the possibility to decouple learning features for IW from the policy estimate. We also plan to bring π-IW closer to the RL setting by allowing stochastic state transitions. This requires analyzing how the final policy is determined by the interplay between the planning and the learning steps. Recent RL methods have proposed to exploit other notions of novelty for systematic exploration (Machado et al. 2018; Ostrovski et al. 2017; Martin et al. 2017; Bellemare et al. 2016). We believe that the combination of structured exploration and representation

Algorithm	IW	IW	Rollout IW	Rollout IW	π-IW	
Features	BPROST	BPROST	BPROST	BPROST	Dynamic	
Planning horizon	0.5s	32s	0.5s	32s	#100	time (s)
Alien	1,316.0	**14,010.0**	4,238.0	6,896.0	5,081.4	1.15
Amidar	48.0	1,043.2	659.8	**1,698.6**	1,163.6	1.07
Assault	268.8	336.0	285.6	319.2	**3,879.3**	1.14
Asterix	1,350.0	**262,500.0**	45,780.0	66,100.0	6,852.0	1.05
Asteroids	840.0	**7,630.0**	4,344.0	7,258.0	2,708.7	1.12
Atlantis	33,160.0	82,060.0	64,200.0	**151,120.0**	140,336.0	2.33
Bank Heist	24.0	739.0	272.0	**865.0**	324.2	1.02
Battle zone	6,800.0	14,800.0	39,600.0	**414,000.0**	137,500.0	1.22
Beam rider	715.2	1,530.4	2,188.0	2,464.8	**3,025.6**	1.08
Berzerk	280.0	**1,318.0**	644.0	862.0	757.2	0.81
Bowling	30.6	**49.2**	47.6	45.8	32.3	0.53
Boxing	**99.4**	79.0	75.4	79.4	89.5	1.33
Breakout	1.6	56.0	82.4	36.0	**175.1**	1.30
Centipede	88,890.0	**143,275.4**	36,980.2	65,162.6	32,531.7	1.70
Chopper command	1,760.0	1,800.0	2,920.0	5,800.0	**10,538.0**	1.13
Crazy climber	16,780.0	44,340.0	39,220.0	43,960.0	**101,246.0**	1.24
Demon attack	106.0	**23,619.0**	2,780.0	9,996.0	8,690.1	1.06
Double dunk	-22.0	-22.4	3.6	20.0	**20.1**	0.83
Enduro	2.6	229.2	169.4	**359.4**	225.5	1.27
Fishing derby	-83.8	-39.0	-68.0	-16.2	**18.7**	1.03
Freeway	0.6	25.0	2.8	12.6	**29.7**	2.57
Frostbite	106.0	182.0	220.0	**5,484.0**	3,995.6	1.24
Gopher	1,036.0	18,472.0	7,216.0	13,176.0	**197,496.8**	1.75
Gravitar	380.0	1,630.0	1,630.0	**3,700.0**	2,276.0	0.68
HERO	2,034.0	7,432.0	13,709.0	28,260.0	**33,109.8**	1.29
Ice hockey	-13.6	-7.0	-6.0	6.6	**47.0**	0.62
James bond 007	40.0	180.0	450.0	**22,250.0**	551.0	0.90
Kangaroo	160.0	3,820.0	1,080.0	**5,780.0**	2,216.0	0.91
Krull	3,206.8	**5,611.8**	1,892.8	1,151.2	4,022.5	1.30
Kung-fu master	440.0	8,980.0	2,080.0	14,920.0	**17,406.0**	1.26
Montezuma's revenge	0.0	0.0	0.0	0.0	0.0	0.53
Ms. Pac-man	2,578.0	**20,622.8**	9,178.4	19,667.0	9,006.5	1.26
Name this game	7,070.0	**13,478.0**	6,226.0	5,980.0	8,738.6	1.55
Phoenix	1,266.0	5,550.0	5,750.0	**7,636.0**	5,769.0	0.97
Pitfall!	**-8.6**	-92.2	-81.4	-130.8	-85.1	0.70
Pong	-20.8	0.8	-7.4	17.6	**20.9**	0.89
Private eye	2,690.8	-526.4	-322.0	3,157.2	**4,335.7**	0.66
Q*bert	515.0	16,505.0	3,375.0	8,390.0	**248,572.5**	1.21
Road Runner	200.0	0.0	2,360.0	37,080.0	**100,882.0**	1.59
Robotank	3.2	32.8	31.0	52.6	**60.3**	1.11
Seaquest	168.0	356.0	980.0	**10,932.0**	1,350.4	1.32
Skiing	-16,511.0	-15,962.0	**-15,738.8**	-16,477.0	-26,081.1	0.86
Solaris	1,356.0	2,300.0	700.0	1,040.0	**4,442.4**	1.18
Space invaders	280.0	1,963.0	**2,628.0**	1,980.0	2,385.9	1.07
Stargunner	840.0	1,340.0	13,360.0	**15,640.0**	7,408.0	1.38
Tennis	-23.4	-22.2	-18.6	**-2.2**	-12.2	0.65
Time pilot	2,360.0	5,740.0	7,640.0	8,140.0	**10,770.0**	1.20
Tutankham	71.2	172.4	128.4	184.0	**197.7**	1.22
Up'n down	928.0	62,378.0	36,236.0	44,306.0	**714,801.4**	1.39
Venture	0.0	**240.0**	0.0	80.0	0.0	0.74
Video pinball	28,706.4	**441,094.2**	203,765.4	382,294.8	133,521.9	2.40
Wizard of wor	5,660.0	**115,980.0**	37,220.0	73,820.0	44,508.0	0.88
Yars' revenge	6,352.6	10,808.2	5,225.4	9,866.4	**44,864.6**	1.00
Zaxxon	0.0	15,080.0	9,280.0	**22,880.0**	12,828.0	1.06
# best	2	13	2	14	22	

Table 2: Score comparison of width-based methods in 54 Atari games (pixel setting). Scores in bold are best overall and underlined values are π-IW scores higher than methods with 0.5s of budget. Performance of π-IW is an average of the last 10 episodes for 5 runs after 40M interactions (including all generated trees). Results taken from Bandres, Bonet, and Geffner (2018).

learning of π-IW is a promising direction for efficient exploration in RL.

Acknowledgements

Miquel Junyent's research is partially funded by project 2016DI004 of the Catalan Industrial Doctorates Plan. Anders Jonsson is partially supported by the grants TIN2015-67959 and PCIN-2017-082 of the Spanish Ministry of Science. Vicenç Gómez is supported by the Ramon y Cajal program RYC-2015-18878 (AEI/MINEICO/FSE,UE).

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, New Orleans, Louisiana, USA, February 2-7, 2018*.

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2013. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research* 47:253–279.

Bellemare, M.; Srinivasan, S.; Ostrovski, G.; Schaul, T.; Saxton, D.; and Munos, R. 2016. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems*, 1471–1479.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4(1):1–43.

Efroni, Y.; Dalal, G.; Scherrer, B.; and Mannor, S. 2018. Beyond the one-step greedy approach in reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning*, 1386–1395.

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

Goodfellow, I.; Bengio, Y.; and Courville, A. 2016. *Deep Learning*. MIT Press.

Guo, X.; Singh, S.; Lee, H.; Lewis, R. L.; and Wang, X. 2014. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in neural information processing systems*, 3338–3346.

Jinnai, Y., and Fukunaga, A. S. 2017. Learning to prune dominated action sequences in online black-box planning. In *AAAI Conference on Artificial Intelligence*.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*, 282–293. Springer.

Liang, Y.; Machado, M. C.; Talvitie, E.; and Bowling, M. 2016. State of the art control of atari games using shallow reinforcement learning. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, 485–493. International Foundation for Autonomous Agents and Multiagent Systems.

Lipovetzky, N., and Geffner, H. 2012. Width and serialization of classical planning problems. *Frontiers in Artificial Intelligence and Applications* 242:540–545.

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

Lipovetzky, N.; Ramirez, M.; and Geffner, H. 2015. Classical planning with simulators: Results on the atari video games. In *International Joint Conference on Artificial Intelligence*, volume 15, 1610–1616.

Machado, M. C.; Bellemare, M. G.; Talvitie, E.; Veness, J.; Hausknecht, M. J.; and Bowling, M. 2018. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research* 61:523–562.

Martin, J.; Sasikumar, S. N.; Everitt, T.; and Hutter, M. 2017. Count-Based Exploration in Feature Space for Reinforcement Learning. In *International Joint Conference on Artificial Intelligence*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*.

Ostrovski, G.; Bellemare, M. G.; van den Oord, A.; and Munos, R. 2017. Count-based exploration with neural density models. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70, 2721–2730. PMLR.

Ross, S.; Gordon, G.; and Bagnell, D. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 627–635.

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. AAAI Press.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529(7587):484–489.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2017a. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017b. Mastering the game of go without human knowledge. *Nature* 550(7676):354.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement learning: An introduction*. MIT press.