Classical Planning Algorithms on the Atari Video Games

Nir Lipovetzky
The University of Melbourne
Melbourne, Australia
nirlipo@gmail.com

Miquel Ramirez
Australian National University
Canberra, Australia
miquel.ramirez@gmail.com

Hector Geffner
ICREA & Universitat Pompeu Fabra
Barcelona, SPAIN
hector.geffner@upf.edu

Abstract

The Atari 2600 games supported in the Arcade Learning Environment (Bellemare et al. 2013) all feature a known initial (RAM) state and actions that have deterministic effects. Classical planners, however, cannot be used for selecting actions for two reasons: first, no compact PDDL-model of the games is given, and more importantly, the action effects and goals are not known *a priori*. Moreover, in these games there is usually no set of goals to be achieved but rewards to be collected. These features do not preclude the use of classical algorithms like breadth-first search or Dijkstra's algorithm, but these methods are not effective over large state spaces. We thus turn to a different class of classical planning algorithms introduced recently that perform a *structured exploration* of the state space; namely, like breadth-first search and Dijkstra's algorithm they are "blind" and hence do not require prior knowledge of state transitions, costs (rewards) or goals, and yet, like heuristic search algorithms, they have been shown to be effective for solving problems over huge state spaces. The simplest such algorithm, called Iterated Width or IW, consists of a sequence of calls IW(1), IW(2), $\ldots$, IW(k) where IW(i) is a breadth-first search in which a state is pruned when it is not the first state in the search to make true some subset of i atoms. The empirical results over 54 games suggest that the performance of IW with the k parameter fixed to 1, i.e., IW(1), is at the level of the state of the art represented by UCT. A simple best-first variation of IW that combines exploration and exploitation proves to be very competitive as well.

Introduction

The Arcade Learning Environment (ALE) provides a challenging platform for evaluating general, domain-independent AI planners and learners through a convenient interface to hundreds of Atari 2600 games (Bellemare et al. 2013). Results have been reported so far for basic planning algorithms like breadth-first search and UCT, reinforcement learning algorithms, and evolutionary methods (Bellemare et al. 2013; Mnih et al. 2013; Hausknecht et al. 2014). The empirical results are impressive in some cases, yet a lot remains to be done, as no method approaches the performance of human players across a broad range of games.

Copyright © 2015, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

While all these games feature a known initial (RAM) state and actions that have deterministic effects, the problem of selecting the next action to be done cannot be addressed with state-of-the-art classical planners (Geffner and Bonet 2013). This is because there is no compact PDDL-like encoding of the domain, and more importantly, the goal to be achieved in each game is not given. Indeed, there are often no goals but rewards $r(a, s)$ that depend on the action a and the state s in which the action is performed, and these rewards are not known.[1] Thus no model of the goals or the rewards can be used to bias the search.

The action selection problem in the Atari games can be seen as a *reinforcement learning* problem (Sutton and Barto 1998) over a deterministic MDP where the state transitions and rewards are not known, or alternatively, as a *net-benefit planning problem* (Coles et al. 2012; Keyder and Geffner 2009) with unknown state transitions and rewards. Namely, we seek an action sequence $a_0, a_1, \ldots, a_m$ from the current state s_0 that generates a state sequence $s_0, s_1, \ldots, s_{m+1}$ with maximum total reward $\sum_{i=0}^{m} r(a_i, s_i)$, where m is a sufficiently large planning horizon, and the rewards $r(a_i, s_i)$ and the state-transitions $s_{i+1} = f(a_i, s_i)$ are known only after action a_i is applied in the state s_i.

The presence of unknown transition and rewards in the Atari games does not preclude the use of blind-search methods like breadth-first search, Dijkstra's algorithm (Dijkstra 1959), or learning methods such as LRTA* (Korf 1990), UCT (Kocsis and Szepesvári 2006), and Q-learning (Sutton and Barto 1998; Bertsekas and Tsitsiklis 1996). Indeed, the net-benefit planning problem with unknown state transitions and rewards over a given planning horizon, can be mapped into a standard *shortest-path problem* which can be solved optimally by Dijkstra's algorithm. For this, we just need to map the unknown rewards $r(a, s)$ into positive (unknown) action costs $c(a, s) = C - r(a, s)$ where C is a large constant that exceeds the maximum possible reward. The fact that the state transition and cost functions $f(a, s)$ and $c(a, s)$ are not known a priori doesn't affect the applicability of Dijkstra's algorithm, which requires the value of these functions precisely when the action a is applied in the state s.

[1]Actually, in the Atari games, the rewards $r(a, s)$ depend only on the state $f(a, s)$ that results from doing action a in the state s.

The limitation of the basic blind search methods is that they are not effective over large spaces, neither for solving problems off-line, nor for guiding a lookahead search for solving problems on-line. In this work, we thus turn to a recent class of planning algorithms that combine the scope of blind search methods with the performance of state-of-the-art classical planners: namely, like "blind" search algorithms they do not require prior knowledge of state transitions, costs, or goals, and yet like heuristic algorithms they manage to search large state spaces effectively. The basic algorithm in this class is called IW for Iterated Width search (Lipovetzky and Geffner 2012). IW consists of a sequence of calls $\text{IW}(1)$, $\text{IW}(2)$, .., $\text{IW}(k)$, where $\text{IW}(i)$ is a standard breadth-first search where states are pruned right away when they fail to make true some new tuple (set) of at most i atoms. Namely, $\text{IW}(1)$ is a breadth-first search that keeps a state only if the state is the first one in the search to make some atom true; $\text{IW}(2)$ keeps a state only if the state is the first one to make a pair of atoms true, and so on. Like plain breadth-first and iterative deepening searches, IW is complete, while searching the state space in a way that makes use of the *structure of states* given by the values of a finite set of state variables. In the Atari games, the (RAM) state is given by a vector of 128 bytes, which we associate with 128 variables X_i, $i = 1, \ldots, 128$, each of which may take up to 256 values x_j. A state s makes an atom $X_i = x_j$ true when the value of the i-th byte in the state vector s is x_j.

While a normal, complete breadth-first search runs in time that is exponential in the total number of variables X_i, the procedure $\text{IW}(k)$ runs in time that is exponential in the k parameter, $1 \leq k \leq n$. Elsewhere, it has been shown that IW exhibits state-of-the-art performance on most existing planning benchmark domains when goals are restricted to single atoms where IW provably runs in low polynomial time in the number of variables (Lipovetzky and Geffner 2012). For example, in a blocks world instance with any number of blocks and any initial configuration, the goal of placing a given block on top of another will be achievable in quadratic time in the number of blocks as any such instance can be shown to have *width* 2. This means that $\text{IW}(2)$ will not only be complete then, but also optimal. This is also true for most other benchmark domains in classical planning that turn out to have small, bounded widths for *any* instance as long as the goal is atomic. For dealing with the benchmarks where goals are not single atoms, e.g., where a given *tower* of blocks needs to be built, simple extensions of the basic IW procedure have been developed such as Serialized IW, where IW is used to achieve the goals one at at time following a simple goal ordering (Lipovetzky and Geffner 2012; 2014). The extension of IW for problems where there is no goal but an additive reward measure to be optimized, like in the Atari games, is direct.

The paper is organized as follows. We review the iterated width algorithm and where it comes from, look at simple variations of the algorithm that appear to be convenient for the Atari games, present and analyze the experimental results, and discuss related and future work.

Iterated Width

The Iterated Width (IW) algorithm has been introduced as a classical planning algorithm that takes a planning problem as an input, and computes an action sequence that solves the problem as the output (Lipovetzky and Geffner 2012). The algorithm however applies to a broader range of problems. We will characterize such problems by means of a finite and discrete set of states (the state space) that correspond to vectors of size n. Namely, the states are *structured* or *factored* , and we take each of the locations in the vector to represent a variable X_i, and the value at that vector location to represent the value x_j of variable X_i in the state. In addition to the state space, a problem is defined by an initial state s_0, a set of actions applicable in each state, a transition function f such that $s' = f(a, s)$ is the state that results from applying action a to the state s, and rewards $r(a, s)$ represented by real numbers that result from applying action a in state s. The transition and reward functions do not need to be known *a priori*, yet in that case, the state and reward that results from the application of an action in a state need to be *observable*. The task is to compute an action sequence $a_0, \ldots, a_m$ for a large horizon m that generates a state sequence $s_0, \ldots, s_{m+1}$ that maximizes the accumulated reward $\sum_{i=0}^{m} r(a_i, s_i)$, or that provides a good approximation.

The Algorithm

IW consists of a sequence of calls $\text{IW}(i)$ for $i = 0, 1, 2, \ldots$ over a problem P until a termination condition is reached. The procedure $\text{IW}(i)$ is a plain forward-state *breadth-first search* with just one change: right after a state s is generated, the state is pruned if it doesn't pass a simple *novelty test*. More precisely,

- The *novelty of a newly generate state s in a search algorithm* is 1 if s is the first state generated in the search that makes true some atom $X = x$, else it is 2 if s is the first state that makes a *pair* of atoms $X = x$ and $Y = y$ true, and so on.
- $\text{IW}(i)$ is a breadth-first search that prunes newly generated states when their novelty measure is greater than i.
- IW calls $\text{IW}(i)$ sequentially for $i = 1, 2, \ldots$ until a termination condition is reached, returning then the best path found.

For classical planning, the termination condition is the achievement of the goal. In the on-line setting, as in the Atari games, the termination condition is given by a time window or a maximum number of generated nodes. The best path found by IW is the path that has a maximum accumulated reward. The accumulated reward $R(s)$ of a state s reached in an iteration of IW is determined by the unique parent state s' and action a leading to s from s' as $R(s) = R(s') + r(a, s')$. The best state is the state s with maximum reward $R(s)$ generated but not pruned by IW, and the best path is the one that leads to the state s from the current state. The action selected to be done next is the first action along such a path.

Performance and Width

IW is a systematic and complete blind-search algorithm like breadth-first search (BRFS) and iterative deepening (ID),

#	Domain	I	IW(1)	IW(2)	Neither
1.	8puzzle	400	55%	45%	0%
2.	Barman	232	9%	0%	91%
3.	Blocks World	598	26%	74%	0%
4.	Cybersecure	86	65%	0%	35%
...	...	...	...	...	
20.	ParcPrinter	975	85%	15%	0%
21.	Parking	540	77%	23%	0%
22.	Pegsol	964	92%	8%	0%
23.	Pipes-NonTan	259	44%	56%	0%
24.	Pipes-Tan	369	59%	37%	3%
25.	PSRsmall	316	92%	0%	8%
26.	Rovers	488	47%	53%	0%
27.	Satellite	308	11%	89%	0%
28.	Scanalyzer	624	100%	0%	0%
29.	Sokoban	153	37%	36%	27%
30.	Storage	240	100%	0%	0%
31.	Tidybot	84	12%	39%	49%
32.	Tpp	315	0%	92%	8%
33.	Transport	330	0%	100%	0%
34.	Trucks	345	0%	100%	0%
35.	Visitall	21859	100%	0%	0%
36.	Woodworking	1659	100%	0%	0%
37.	Zeno	219	21%	79%	0%
	Total/Avgs	37921	37.0%	51.3%	11.7%

# Instances	IW	ID	BRFS	*GBFS* + h_{add}
37921	34627	9010	8762	34849

Table 1: *Top:* Number of classical planning instances per domain and percentages solved by IW(1), IW(2), or neither. Problems obtained by splitting benchmarks with N atomic goals into N problems with atomic goals. *Bottom:* Number of instances solved by IW in comparison with Iterative Deepening (ID), Breadth-First Search (BRFS), and Greedy Best First Search (*GBFS*) guided by additive heuristic. Time and memory outs after 30 minutes or 2 GB. Table from (Lipovetzky and Geffner 2012).

but unlike these algorithms, it uses the factored representation of the states in terms of variables to structure the search in a different way. This structured exploration has proved to be very effective over classical planning benchmark domains when goals are single atoms.[2] Table 1 from (Lipovetzky and Geffner 2012) shows the percentage of instances that are solved by the first iteration of IW, i.e. IW(1), by the second iteration IW(2), and by neither one. These are instances that have been obtained from the existing benchmarks by splitting problems with N atomic goals, into N problems with one atomic goal each. As the table shows, 37% of the 37921 instances are solved by IW(1) while 51.3% are solved by IW(2). Since IW(k) runs in time that is exponential in k, this mean that almost 90% of the 37,921 instances are solved in time that is either linear or quadratic in the number of problem variables, which in these encoding are all *boolean*.

[2]Any conjunctive goal can be mapped into a single dummy atomic goal by adding an action that achieves the dummy goal and that has the original conjunctive goal as a precondition. Yet, this changes the definition of the domain.

Furthermore, when the performance of IW is compared with breadth-first search and iterative deepening, on the one hand, and with a Greedy Best First Search guided by the additive heuristic h_{add} (Bonet and Geffner 2001) on the other (this algorithm is similar to the one used by the FF planner when hill climbing search fails (Hoffmann and Nebel 2001)), it turns out that "blind" IW solves as many problems as the informed search, 34,627 vs. 34,849, far ahead of the other blind algorithms BRFS and ID that solve 9,010 and 8,762 problems each. This is shown in the bottom part of Table 1. Moreover, IW is faster and results in shorter plans than a heuristic search algorithm (Lipovetzky and Geffner 2012).

The min k value for which IW(k) solves a problem is bounded and small in most of these instances. This is actually no accident and has a *theoretical explanation.* Lipovetzky and Geffner define a structural parameter called the problem *width* and show that for many of these domains, *any* solvable instance with atomic goals will have a bounded and small width that is independent of the number of variables and states in the problem. The min value k for which the iteration IW(k) solves the problem cannot exceed the problem width, so the algorithm IW *runs in time and space that are exponential in the problem width.*

Formally, the *width* $w(P)$ of a problem P is i iff i is the minimum positive integer for which there is a sequence $t_0, t_1, \ldots, t_n$ of atom sets t_k with at most i atoms each, such that 1) t_0 is true in the initial state of P, 2) any shortest plan π that achieves t_k in P can be extended into a shortest plan that achieves t_{k+1} by extending π with one action, and 3) any shortest plan that achieves t_n is a shortest plan for achieving the goal of P.

One way to understand this definition is that the problem width $w(P)$ is at most i if there is a "trail of stepping stones" $t_0, t_1, \ldots, t_n$ to reach the problem goal such that A) these "stepping stones" contain at most i atoms each, B) each stepping stone t_{k+1} is at distance 1 from the previous one t_k when t_k has been reached following the "trail", C) the "trail" preserves "optimality"; i.e., no tuple t_k can be reached in less than k steps.

While this notion of width and the iterated width algorithms that are based on it have been designed for problems where a goal state needs to be reached, the notions remain relevant in optimisation problems as well. Indeed, if a good path is made of states s_i each of which has a low width, IW can be made to find such path in low polynomial time for a small value of the k parameter. Later on we will discuss a slight change required in IW to enforce this property.

The Algorithms for the Atari Games

The number of nodes *generated* by IW(1) is $n \times D \times b$ in the worst case, where n is the number of problem variables, D is the size of their domains, and b is the number of actions per state. This same number in a breadth-first search is not linear in n but exponential. For the Atari games, $n = 128$, $D = 256$, and $b = 18$, so that the product is equal to $589,824$, which is large but feasible. On the other hand, the number of nodes generated by IW(2) in the worst case is $(n \times D)^2 \times b$, which is equal to $19,327,352,832$ which is too large, forcing us to consider only a tiny fraction of such states. For classical planning problems, the growth in the number of nodes from $\mathrm{IW}(1)$ to $\mathrm{IW}(2)$ is not that large, as the variables are boolean. Indeed, we could have taken the state vector for the Atari games as a vector of 1024 boolean variables, and apply these algorithms to that representation. The results would be different. In such a case, $\mathrm{IW}(1)$ and $\mathrm{IW}(2)$ would generate $n' \times D' \times b$ and $(n' \times D')^2 \times b$ states, which for $n' = 1024$ and $D = 2$ represent $36,864$ and $75,497,472$ states respectively. These are feasible numbers, but we haven't used this representation under the assumption that the correlations among the bits in each one of the 128 words of the state vectors are meaningful. In summary, from the basic IW algorithm we are testing only the first call $\mathrm{IW}(1)$.

IW is a purely exploration algorithm that does not take into account the accumulated reward for selecting the states to consider. As a simple variant that combines exploration and exploitation, we evaluated a *best-first search* algorithm with two queues: one queue ordered first by novelty measure (recall that novelty one means that the state is the first one to make some atom true), and a second queue ordered by accumulated reward. In one iteration, the best first search picks up the best node from one queue, and in the second iteration it picks up the best node from the other queue. This way for combining multiple heuristics is used in the LAMA planner (Richter and Westphal 2010), and was introduced in the planner Fast Downward (Helmert 2006). In addition, we break ties in the first queue favoring states with largest accumulated reward, and in the second queue, favoring states with smallest novelty measure. Last, when a node is expanded, it is removed from the queue, and its children are placed on both queues. The exception are the nodes with no accumulated reward that are placed in the first queue only. We refer to this best-first algorithm as 2BFS.

For the experiments below, we added two simple variations to $\mathrm{IW}(1)$ and 2BFS. First, in the breadth-first search underlying $\mathrm{IW}(1)$, we generate the children in random order. This makes the executions that result from the $\mathrm{IW}(1)$ lookahead less susceptible to be trapped into loops; a potential problem in local search algorithms with no memory or learning. Second, a discount factor $\gamma = 0.995$ is used in both algorithms for discounting future rewards like in UCT. For this, each state s keeps its depth $d(s)$ in the search tree, and if state s' is the child of state s and action a, $R(s')$ is set to $R(s) + \gamma^{d(s)+1} r(a, s)$. The discount factor results in a slight preference for rewards that can be reached earlier, which is a reasonable heuristic in on-line settings based on lookahead searches.

Experimental Results

We tested $\mathrm{IW}(1)$ and 2BFS over 54 of the 55 games considered in (Bellemare et al. 2013), from now on abbreviated as BNVB.[3] Table 2 shows the performance of $\mathrm{IW}(1)$ and 2BFS in comparison with breadth-first search (BrFS) and UCT. Videos of selected games played by $\mathrm{IW}(1)$, 2BFS, and UCT can be seen in Youtube.[4] The discount factor used by all the algorithms is $\gamma = 0.995$. The scores reported for BrFS and UCT are taken from BNVB. Our experimental setup follows theirs except that a maximum budget of $150,000$ simulated frames is applied to $\mathrm{IW}(1)$, 2BFS, and UCT. UCT uses this budget by running 500 rollouts of depth 300. The bound on the number of simulated frames is like a bound on lookahead time, as most of the time in the lookahead is spent in calls to the emulator for computing the next RAM state. This is why the average time per action is similar to all the algorithms except $\mathrm{IW}(1)$, that due to its pruning does not always use the full budget and takes less time per action on average.

Also, as reported by BNVB, all of the algorithms reuse the frames in the sub-tree of the previous lookahead that is rooted in the selected child, deleting its siblings and their descendants. More precisely, no calls to the emulator are done for transitions that are cached in that sub-tree, and such reused frames are not discounted from the budget that is thus a bound on the number of *new* frames per lookahead. In addition, in $\mathrm{IW}(1)$, the states that are reused from the previous searches are ignored in the computation of the novelty of new states so that more states can escape pruning. Otherwise, $\mathrm{IW}(1)$ often uses a fraction of the budget. This is not needed in 2BFS which does no pruning. $\mathrm{IW}(1)$ and 2BFS are limited to search up to a depth of $m = 1,500$ frames and up to $m = 150,000$ frames per root branch. This is to avoid the search from going too deep or being too committed to a single root action.

Last, in the lookahead, $\mathrm{IW}(1)$ and 2BFS select an action every 5 frames, while UCT selects an action in every frame. This means that in order to explore a branch 300 frames deep, UCT gets to choose 300 actions, while $\mathrm{IW}(1)$ and 2BFS get to choose 60 actions, both however using the same 300 frames from the budget. For this, we followed the setup of BrFS in BNVB that also selects actions every 5 frames, matching the behavior of the emulator that requests an action also every 5 frames. Since the lookahead budget is given by a maximum number of (new) frames, and the time is mostly taken by calls to the emulator, this may not be the best choice for $\mathrm{IW}(1)$ and 2BFS that may therefore not be exploiting all the options afforded by the budget. We'll look at this further in the future.

Table 2 shows that both $\mathrm{IW}(1)$ and 2BFS outperform BrFS, which rarely collects reward in many domains as the depth of the BrFS search tree results in a lookahead of 0.3 seconds (20 frames or 4 nodes deep). The notable exception to this is CENTIPEDE where abundant reward can be collected with a shallow lookahead. On the other hand, both $\mathrm{IW}(1)$ and 2BFS normally reach states that are up to 350–1500 frames deep (70–260 nodes or 6–22 seconds), even if $\mathrm{IW}(1)$ does not always use all the simulation frames allocated due to its agressive pruning. This can be observed in games such as BREAKOUT, CRAZY CLIMBER, KANGAROO, and POOYAN, where the average CPU time for each lookahead is up to 10 times faster than 2BFS. Computation

[3]We left out SKIING as the reported figures apparently use a different reward structure.

[4]http://www.youtube.com/playlist?list=PLXpQcXUQ_CwenUazUivhXyYvjuS6KQOI0.

	IW(1)		2BFS		BrFS	UCT
Game	Score	Time	Score	Time	Score	Score
Alien	**25634**	81	12252	81	784	7785
Amidar	**1377**	28	1090	37	5	180
Assault	953	18	827	25	414	**1512**
Asterix	153400	24	77200	27	2136	**290700**
Asteroids	**51338**	66	22168	65	3127	4661
Atlantis	159420	13	154180	71	30460	**193858**
Bank Heist	**717**	39	362	64	22	498
Battle Zone	11600	86	**330800**	87	6313	70333
Beam Rider	9108	23	**9298**	29	694	6625
Berzerk	**2096**	58	802	73	195	554
Bowling	**69**	10	50	60	26	25
Boxing	**100**	15	**100**	22	**100**	**100**
Breakout	384	4	**772**	39	1	364
Carnival	**6372**	16	5516	53	950	5132
Centipede	99207	39	94236	67	**125123**	110422
Chopper Command	10980	76	27220	73	1827	**34019**
Crazy Climber	36160	4	36940	58	37110	**98172**
Demon Attack	20116	33	16025	41	443	**28159**
Double Dunk	-14	41	21	41	-19	**24**
Elevator Action	13480	26	10820	27	730	**18100**
Enduro	**500**	66	359	38	1	286
Fishing Derby	30	39	6	62	-92	**38**
Freeway	**31**	32	23	61	0	0
Frostbite	902	12	**2672**	38	137	271
Gopher	18256	19	15808	53	1019	**20560**
Gravitar	3920	62	**5980**	62	395	2850
Hero	**12985**	37	11524	69	1324	12860
Ice Hockey	**55**	89	49	89	-9	39
James Bond	**23070**	0	10080	30	25	330
Journey Escape	40080	38	**40600**	67	1327	7683
Kangaroo	**8760**	8	5320	31	90	1990
Krull	**6030**	28	4884	42	3089	5037
Kung Fu Master	**63780**	21	42180	43	12127	48855
Montezuma Revenge	0	14	**540**	39	0	0
Ms Pacman	21695	21	18927	23	1709	**22336**
Name This Game	9354	14	8304	25	5699	**15410**
Pong	**21**	17	**21**	35	-21	**21**
Pooyan	11225	8	10760	16	910	**17763**
Private Eye	-99	18	**2544**	44	58	100
Q*Bert	3705	11	11680	35	133	**17343**
Riverraid	**5694**	18	5062	37	2179	4449
Road Runner	**94940**	25	68500	41	245	38725
Robot Tank	**68**	34	52	34	2	50
Seaquest	**14272**	25	6138	33	288	5132
Space Invaders	2877	21	**3974**	34	112	2718
Star Gunner	1540	19	**4660**	18	1345	1207
Tennis	**24**	21	**24**	36	-24	3
Time Pilot	35000	9	36180	29	4064	**63855**
Tutankham	172	15	204	34	64	**226**
Up And Down	**110036**	12	54820	14	746	74474
Venture	**1200**	22	980	35	0	0
Video Pinball	**388712**	43	62075	43	55567	254748
Wizard Of Wor	**121060**	25	81500	27	3309	105500
Zaxxon	**29240**	34	15680	31	0	22610

# Times Best (54 games)	**26**	13	1	19
# Times Better than IW (54 games)	–	16	1	**19**
# Times Better than 2BFS (54 games)	**34**	–	1	25
# Times Better than UCT (54 games)	**31**	26	1	–

Table 2: Performance that results form various lookahead algorithms in 54 Atari games. The algorithms, BrFS, IW(1), 2BFS, and UCT, are evaluated over 10 runs (episodes) for each game. The maximum episode duration is 18,000 frames and every algorithm is limited to a lookahead budget of 150,000 simulated frames. Figures for BrFS and UCT taken from (Bellemare et al. 2013). Average CPU times per action in seconds, rounded to nearest integer, shown for IW(1) and 2BFS. Numbers in bold show best performer in terms of average score, while numbers shaded in light grey show scores that are better than UCT's. Bottom part of the table shows pairwise comparisons among the algorithms.

time for UCT and BrFS are similar to 2BFS, as the most expensive part of the computation is the generation of frames through the simulator, and these three algorithms always use the full budget.

More interestingly, IW(1) outscores UCT in 31 of the 54 games, while 2BFS outscores UCT in 26. On the other hand, UCT does better than IW(1) and 2BFS in 19 and 25 games respectively. The relative performance between IW(1) and 2BFS makes IW(1) the best of the two in 34 games, and 2BFS in 16. In terms of the number of games where an algorithm is the best, IW(1) is the best in 26 games, 2BFS in 13 games, and UCT in 19 games. Also, BrFS is best in 2 games (CENTIPEDE, tied up in BOXING), while the other three algorithms are tied in another 2 games (PONG, BOXING).

Likewise, in FREEWAY and BERZERK both IW(1) and 2BFS attain a better score than the baseline semi-random algorithm *Perturb* in (Bellemare et al. 2013), that beats UCT on those games. *Perturb* is a simple algorithm that selects a fixed action with probability 0.95, and a random action with probability 0.05. For *Perturb*, BNVB do not report the average score but the best score. *Perturb* manages to do well in domains where rewards are deep but can be reached by repeating the same action. This is the case of FREEWAY, where a chicken has to run to the top of the screen across a ten lane highway filled with traffic. Every time the chicken gets across (starting at the bottom), there is one unit of reward. If the chicken is hit by a car, it goes back some lanes. In FREEWAY, only 12 out of the 18 possible actions have an effect: 6 actions move the chicken up (up-right, up-left, up-fire, up-right-fire, up-left-fire), 6 actions move the chicken down (down-right, down-left, down-fire, down-right-fire, down-left-fire), and 6 actions do nothing. *Perturb* does well in this domain when the selected fixed action moves the chicken up. As noted in Table 2 and seen in the provided video, UCT does not manage to take the chicken across the highway at all. The reason that UCT does not collect any reward is that it needs to move the chicken up at least 240 times[5] something that is very unlikely in a random exploration. IW(1) does not have this limitation and is best in FREEWAY.

IW(1) also outperforms the best learning algorithm in (Mnih et al. 2013) in the 7 games considered there, and 2BFS does so in 6 of the 7 games. Comparing with the scores reported for the reinforcement learning algorithms by BNVB, we note that both IW(1) and 2BFS do much better than the best learning algorithm in those games where the learning algorithms outperform UCT like MONTEZUMA REVENGE, VENTURE and BOWLING. In MONTEZUMA REVENGE rewards are very sparse, deep, and most of the actions lead to losing a life with no immediate penalty or consequence. All algorithms achieve 0 score, except for 2BFS that achieves an average score of 540, and a score of 2500 in one of the runs. This means however that even 2BFS is not able to consistently find rewards in this game. This game and several others like BREAKOUT and SPACE INVADERS could be much simpler by adding negative rewards for losing a life. We have indeed observed that our planning algorithms do not care much about losing lives until there is just one life left, when their play noticeably improves. This can be seen in the videos mentioned above, and suggest a simple form of learning that would be useful to both planners and reinforcement learning algorithms.

We are not reporting the performance of IW(k) with parameter $k = 2$ because in our preliminary tests and according to the discussion in the previous section, it doesn't appear to improve much on BrFS, even if it results in a lookahead that is 5 times deeper, but still too shallow to compete with the other planning algorithms.

Exploration and Exploitation

The notion of width underlying the iterated width algorithm was developed in the context of classical planning in order to understand why most of the hundreds of existing benchmarks appear to be relatively simple for current planners, even though classical planning is PSPACE-complete (Bylander 1994). A partial answer is that most of these domains have a low width, and hence, can be solved in low polynomial time (by IW) when goals contain a single atom. Benchmark problems with multiple atomic goals tend to be easy too, as the goals can often be achieved one at a time after a simple goal ordering (Lipovetzky and Geffner 2012).

In the iterated width algorithm, the key notion is the *novelty measure* of a state in the underlying breadth-first search. These novelty measures make use of the factored representation of the states for providing a structure to the search: states that have width 1 are explored first in linear time, then states that have width 2 are explored in quadratic time, and so on. In classical planning problems with atomic goals, this way of organizing the search pays off both theoretically and practically.

The use of "novelty measures" for guiding an optimization search while ignoring the function that is being optimized is common to the novelty-based search methods developed independently in the context of genetic algorithms (Lehman and Stanley 2011). In these methods individuals in the population are not ranked according to the optimization function but in terms of how "novel" they are in relation to the rest of the population, thus encouraging diversity and exploration rather than (greedy) exploitation. The actual definition of novelty in such a case is domain-dependent; for example, in the evolution of a controller for guiding a robot in a maze, an individual controller will not be ranked by how close it takes the robot to the goal (the greedy measure), but by the distance between the locations that are reachable with it, and the locations reachable with the other controllers in the population (a diversity measure). The novelty measure used by IW, on the other hand, is domain-independent and it is determined by the structure of the states as captured by the problem variables.

The balance between exploration and exploitation has received considerable attention in reinforcement learning (Sutton and Barto 1998), where both are required for converging to an optimal behavior. In the Atari games, as in other deterministic problems, however, "exploration" is not needed

[5]One needs to move the chicken up for at least 4 seconds (240 frames) in order to get it across the highway.

for optimality purposes, but just for improving the effectiveness of the lookahead search. Indeed, a best-first search algorithm guided only by (discounted) accumulated reward will deliver eventually best moves, but it will not be as effective over small time windows, where like breadth-first search it's likely not to find any rewards at all. The UCT algorithm provides a method for balancing exploration and exploitation, which is effective over small time windows. The 2BFS algorithm above with two queues that alternate, one guided by the novelty measures and the other by the accumulated reward, provides a different scheme. The first converges to the optimal behavior asymptotically; the second in a bounded number of steps, with the caveat below.

Duplicates and Optimality

The notions of width and the IW algorithm guarantee that states with low width will be generated in low polynomial time through *shortest* paths. In the presence of rewards like the Atari games, however, the interest is not in the shortest paths but in the *best* paths; i.e, the paths with maximum reward. IW may actually fail to find such paths even when calling IW(k) with a high k parameter. Optimality could be achieved by replacing the breadth-first search underlying IW(k) by Dijkstra's algorithm yet such a move would make the relation between IW and the notion of width less clear. A better option is to change IW to comply with a different property; namely, that if there is a "rewarding" path made up of states of low width, then IW will find such paths or better ones in time that is exponential in their width. For this, a simple change in IW suffices: when generating a state s that is a *duplicate* of a state s' that has been previously generated and not pruned, s' is replaced by s if $R(s) > R(s')$, with the change of reward propagated to the descendants of s' that are in memory. This is similar to the change required in the A* search algorithm for preserving optimality when moving from consistent to inconsistent heuristics (Pearl 1983). The alternative is to "reopen" such nodes. The same change is actually needed in 2BFS to ensure that, if given enough time, 2BFS will actually find optimal paths. The code used for IW and 2BFS in the experiments above does not implement this change as the overhead involved in checking for duplicates in some test cases did not appear to pay off. More experiments however are needed to find out if this is actually the most effective option.

Summary and Future Work

We have shown experimentally that width-based algorithms like IW(1) and 2BFS that originate in work in classical planning, can be used to play the Atari video games where they achieve state-of-the-art performance. This level of play is the result of a structured exploration of the state space that manages to combine the scope of blind search algorithms with the efficiency of heuristic search methods. There are several issues worth studying, we mention two of them. Regarding planning, the results show the potential of width-based algorithms for a broader class of planning problems. As shown, indeed, these algorithms do not require PDDL encodings and can deal with state successor functions encoded *procedurally* (as in simulators) as long as the structure of the states is known.

Concerning the Atari games, there are two problems in all the planning methods discussed. First, by using the RAM state of the Atari console one can consider that they are "cheating", as they use information that is not available to the human player. And second, they do not get better with experience. In the future, we would like to explore whether variations of these algorithms can be used to play from the information conveyed by the state of the screen pixels as opposed to the RAM state, and the potential uses of the new ideas for learning, such the use of the novelty measures for shaping rewards.

Acknowledgments. We thank the people involved in the creation and maintenance of the Arcade Learning Environment that have made this work possible. The research by Nir Lipovetzky is partially funded by the Australian Research Council Linkage grant LP11010015.

References

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(47):253–279.

Bertsekas, D., and Tsitsiklis, J. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.

Bylander, T. 1994. The computational complexity of STRIPS planning. *Artificial Intelligence* 69:165–204.

Coles, A.; Coles, A.; Olaya, A. G.; Jiménez, S.; López, C. L.; Sanner, S.; and Yoon, S. 2012. A survey of the seventh international planning competition. *AI Magazine* 33(1):83–88.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

Geffner, H., and Bonet, B. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool Publishers.

Hausknecht, M.; Lehman, J.; Miikkulainen, R.; and Stone, P. 2014. A neuroevolution approach to general atari game playing. *IEEE Transaction on Computational Intelligence and AI in Games* (99).

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

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Keyder, E., and Geffner, H. 2009. Soft goals can be compiled away. *Journal of Artificial Intelligence Research* 36:547–556.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *Proc. ECML-2006*, 282–293. Springer.

Korf, R. 1990. Real-time heuristic search. *Artificial Intelligence* 42:189–211.

Lehman, J., and Stanley, K. O. 2011. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation* 19(2):189–223.

Lipovetzky, N., and Geffner, H. 2012. Width and serialization of classical planning problems. In *Proc. ECAI*, 540–545.

Lipovetzky, N., and Geffner, H. 2014. Width-based algorithms for classical planning: New results. In *Proc. ECAI*, 1059–1060.

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

Pearl, J. 1983. *Heuristics*. Addison Wesley.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39(1):127–177.

Sutton, R., and Barto, A. 1998. *Introduction to Reinforcement Learning*. MIT Press.