Width-based Planning for General Video-Game Playing

Tomas Geffner
Universidad de Buenos Aires
Buenos Aires, ARGENTINA
tomas.geffner@gmail.com

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

Abstract

Iterated Width is a simple search algorithm that assumes that states can be characterized in terms of a set of boolean features or atoms. In particular, IW(1) consists of a standard breadth-first search with one variation: a newly generated state is pruned if it does not make a new atom true. Thus, while a breadth-first search runs in time exponential in the number of atoms, IW(1) runs in linear time. Variations of the algorithm have been shown to yield state-of-the-art results in classical planning and more recently in the Atari video games. In this paper, we use the algorithm for selecting actions in the games of the general video-game AI competition (GVG-AI) which, unlike classical planning problems and the Atari games, are stochastic. We evaluate a variation of the algorithm over 30 games under different time windows using the number of wins as the performance measure. We find that IW(1) does better than the sample MCTS and OLMCTS controllers for all time windows with the performance gap growing with the window size. The exception are the puzzle-like games where all the algorithms do poorly. For such problems, we show that much better results can be obtained with the IW(2) algorithm which is like IW(1) except that states are pruned in the breadth-first search when they fail to make true a new *pair* of atoms.

1 Introduction

In its early years, AI researchers used computers for exploring intuitions about intelligence and for writing programs displaying intelligent behavior. Since the 80s, however, there has been a shift from this early paradigm of writing programs for specific, sometimes ill-defined problems to developing solvers for well-defined mathematical models like constraint satisfaction problems, STRIPS planning, SAT, Bayesian networks, partially observable Markov decision processes, logic programs, and general game playing [Geffner, 2014]. Unlike the early AI programs, solvers are general as they must deal with any instance that fits the underlying model. The generality captures a key component of intelligence and raises a crisp computational challenge, as these models are all computationally intractable. Indeed, for these solvers to scale up, they must be able to exploit the structure of the problems automatically. The generality and scalability of solvers is tested experimentally through benchmarks and in most cases through competitions.

The general video-game AI playing competition (GVG-AI) is one of the latest developments in this trend toward empirically tested and scalable solvers [Perez *et al.*, 2015]. The competition provides a setting for evaluating action selection mechanisms on a class of Markov decision processes (MDPs) that represent video-games. As in other competitions, some of the games are known and some are new. The information available to the decision mechanisms are the states, the rewards, and the status of the game as resulting from a simulator. The games are defined in a compact manner through a convenient high-level language called VGDL [Schaul, 2013] but the resulting descriptions are not available to the solvers which have to select an action from a small pool of actions every 40 milliseconds. The competition is closely related to the existing MDP planning competition [Younes *et al.*, 2005; Coles *et al.*, 2012], except that the MDPs express video games, the MDPs are specified in a different language that is not available to the solvers, and the time windows for action selection are very short precluding much exploration before decisions.

Like in the MDP competition, the algorithms that do best in the GVG-AI setting tend to be based on variations of Monte-Carlo Tree Search [Chaslot *et al.*, 2008]. While Monte-Carlo planning methods evaluate each of the applicable actions by performing a number of stochastic simulations starting with each of the actions, in Monte-Carlo Tree Search, the average rewards obtained from such simulations are used to incrementally build a tree of the possible executions, which may deliver good decisions over short time windows, while ensuring optimal decisions asymptotically. Since its first success in the game of Go, where leaves of the tree are initialized with values obtained from a given informed base policy [Gelly and Silver, 2007], variations of the basic MCTS algorithm called UCT [Kocsis and Szepesvári, 2006] have been successfully used in a number of contexts, including the MDP and GVG-AI competitions.

MCTS methods, however, do not do well in problems with large spaces, sparse rewards, and non-informed base policies where they generate random actions and bootstrap very slowly. Indeed, plain MCTS methods make no attempt at exploiting the structure of such problems. This is in contrast with the methods developed for example in classical planning, SAT, and CSP solving, where good, general methods have been devised for exploiting structure successfully in the form of automatically derived heuristic functions or effective forms of inference [Geffner and Bonet, 2013; Biere *et al.*, 2009; Rossi *et al.*, 2006].

The aim of this paper is to gain further insights on the computational challenges raised by the GVG-AI framework by using a different type of algorithm whose origin is precisely in classical planning where it has been shown to be very effective over large spaces [Lipovetzky and Geffner, 2012]. The complete algorithm, called Iterated Width or IW, is a sequence of calls IW(1), IW(2), ..., where IW(k) is a breadth-first search in which a new state is pruned right away when it is not the first state in the search to make true some subset of k atoms or boolean features. In classical planning, the algorithm has been shown to be effective for decomposing a problem into subproblems and for solving the subproblems. More recently, the algorithm IW(k) with the parameter k fixed to 1 [Lipovetzky *et al.*, 2015] has been shown to outperform MCTS in the Atari games within the Arcade Learning Environment [Bellemare *et al.*, 2013]. In this work, we consider variations of IW(1) and IW(2) within the GVG-AI framework where the games can be stochastic. The results are no less interesting. A key difference between IW and most other classical planning algorithms is that the latter require a compact encoding of the problems in a PDDL language or similar for either inferring heuristic values or translating them into SAT. IW, on the other hand, works with simulators, very much like MCTS and brute-force search methods, but unlike them IW makes an attempt at exploiting the *factored structure of states* in a way that has been shown to pay off in many settings.

The paper is organized as follows. First, we review the IW algorithm, the GVG-AI framework, and the way IW(1) is used. We look then at the empirical results, consider a version of the IW(2) algorithm, and end with a summary and discussion.

2 Basic Algorithms: IW and IW(1)

The iterated width algorithm (IW) 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 which can be characterized in terms of a finite set of states where each state s has a *structure* given by a set of features or variables that can take a finite set of values. A state assigns values to all the variables. In classical planning, the state variables are boolean, while in ALE, for example, the state is a vector of 1024 bits organized into 128 words, which can be regarded as composed of 1024 boolean variables or as 128 variables that can take up to 2^8 values each. Width-based algorithms are sensitive to these choices, and in particular, to the resulting set of atoms. An *atom* $X = x$ represents the assignment of value x to variable X, while a *tuple of atoms* is a set of atoms involving different variables. The *size* of a tuple is the number of atoms that it contains. A state s makes an atom $X = x$ true if the value of X in s is x, and makes a tuple of atoms t true if it makes each atom in t true. Likewise, the state s makes the atom false if it doesn't make it true, and makes a tuple of atoms false if it makes false one of the atoms in the tuple.

IW is a sequence of calls IW(i) for $i = 1, 2, \ldots$ where IW(i) is a plain *breadth-first search* with one change: right after a new state s is generated in the search, the state is pruned if s does not make true a new tuple of at most i atoms. That is, the state s is *not* pruned in the breadth-first search only if there is a tuple t of size no greater than i such that t is true in s and false in all the states generated in the search before s.

As an illustration, IW(i) for $i = 1$, i.e., IW(1), is a breadth-first search where a newly generated state s is pruned when there is no *new atom* made true by s. Similarly, IW(2) is a breadth-first search where a newly generated state s is pruned when there is no *new atom pair* made true by s, and so on.

A key property of the algorithm is that while the number of states is *exponential* in the number of atoms, IW(i) runs in time that is exponential in i only. In particular, IW(1) is linear in the number of atoms, while IW(2) is quadratic. Furthermore, Lipovetzky and Geffner define a general *width* measure for problems P and prove that IW(i) solves P when the width of P is no greater than i. Moreover in such a case, IW(i) solves P optimally (i.e., it finds a shortest solution). For example, *any* blocks world problem where the goal is to have one block x on top of another block y can be shown to have width 2 no matter the number of blocks or configuration. As a result, all such problems can be solved in quadratic time by IW(2) although the number of states is exponential. On the other hand, the same problem with joint goals, as when a tower of blocks is to be built, does not have a bounded width. The serialized iterated algorithm (SIW) proposed by Lipovetzky and Geffner uses IW sequentially for achieving the joint goal, one subgoal at a time. The algorithm is effective on most classical planning benchmarks where there are general and effective ways for serializing goals into subgoals, and where the width of the resulting subproblems is bounded and low. At the same time, the algorithm IW(i) with i set to the number of problem variables is equivalent to breadth-first search and hence complete for all problems provided that different states disagree on the truth of some atom or feature.

The algorithm IW(1) has been used recently for playing Atari video-games *on-line* in the Arcade Learning Environment [Bellemare *et al.*, 2013] where it was shown to achieve state-of-the-art performance [Lipovetzky *et al.*, 2015]. For this, the atoms $X_i = x$ were defined so that there is one variable X_i for each of the 128 words in the state vector, and one atom $X_i = x$ for each of its 2^8 possible values. IW runs in time linear in the number of atoms, which in this representation is $128 \times 256 = 2^{15}$. The alternative representation where the variables X_i are associated with each of the 1024 bits in the state vector results into a smaller set of $1024 \times 2 = 2^{11}$ atoms, making IW(1) run faster but with poorer results.

In the Atari games, there is no crisp goal to achieve but rewards to be collected, and planning is done *on-line* by selecting actions from the *current state* using the simulator in a lookahead search. This search is performed with the IW(1) algorithm with little modification, keeping the (discounted) total reward accumulated in each non-pruned path in the breadth-first search, and selecting the first action of the path with most reward. A similar idea will be used in the GVG-AI setting.

Last, for using the IW(i) algorithms it is not necessary for the states s to explicitly encode the value x_i of a set of variables X_i. An alternative is to associate the variables X_i with a set of *features* ϕ_i so that the atoms made true by the state s are of the form $X_i = \phi_i(s)$. Indeed, the algorithm IW(i) can be understood as the algorithm IW(1) with a set of boolean features ϕ_k, each of which checks whether a tuple t_k of size no greater than i is true in s.

3 The GVG-AI Framework

The games supported in the GVG-AI framework are MDPs that represent video games [Perez *et al.*, 2015]. The video-games consist of objects of different categories that live in a rectangular grid and have dynamic properties. There are four actions for moving an avatar in the four directions and a fifth "use" action whose effects are game and state dependent but which basically makes use of the resources that the avatar may have. The categories of objects includes avatar, non-playing character (NPC), movable and static objects, resources, portal, and from–avatar (like bullets fired by avatar). Changes occur due to the actions performed, object collisions, and the internal dynamics of objects. The games are defined in a compact manner through a convenient high level language called VGDL [Schaul, 2013] that is mostly declarative except for the changes resulting from collisions that are expressed in Java code. The game descriptions are not available to the solvers which have to select an action every 40 milliseconds by interacting with the simulator. The simulator provides information about the state, the rewards, and the status of the game (won/lost/on-going) while applying the actions selected. The range of problems that can be expressed elegantly in VGDL is very broad and includes from simple shooter games to Sokoban puzzles

An important part of the game description is the object type hierarchy tree (sprite set). For example, the tree for the game Butterfly has nodes *cocoon* and *animal*, the latter with children *avatar* and *butterfly*. We call the nodes in these hierarchies, *stypes* (sprite types). The stype of an object is associated with its dynamic and behaviors and is readable from the state observation. Stypes are important in our use of the IW algorithms as the set of total stypes in a game is finite and usually small (smaller than 20 in general), while the set of objects and the stype of an object may change dynamically.

The GVG-AI setting is similar to the one used in the MDP planning competition [Younes *et al.*, 2005; Coles *et al.*, 2012], the main difference being that the MDPs are described in a different language and that the game descriptions are not available to the solvers.

4 IW Algorithms in GVG-AI

For using the IW algorithms in the GVG-AI setting, two issues need be addressed: first the definition of the boolean features or atoms; second, the stochasticity of the games. We consider these two issues next.

Since every object has a unique ID id and a number of dynamic features ϕ_i, we could associate problem variables $X[id, i]$ with values $\phi_i(id, s)$ representing the value of feature ϕ_i of object id in the state s. There are however two problems with this. First, the set of objects is dynamic. Second, the resulting number of atoms ends up being too large even for a linear time algorithm like IW(1) given that the time window for decisions is very small: 40 milliseconds. Indeed, running IW(1) to completion in such a representation may require up to two orders-of-magnitude more time in some games.

In order to avoid dealing with either a *dynamic* set of variables or a set of atoms that is *too large*, we have chosen to define the set of boolean features as the ones representing just whether an object of a certain *stype* is in a given grid cell. If there are N grid cells and T stypes in the object hierarchy, the number of atoms $at(cell, stype)$ would be $N \times T$, no matter the number of actual objects. In addition, since the actions control the avatar, we treat the avatar stypes (stypes beneath the avatar node in the hierarchy if any, else the stype avatar itself) in a different way. For avatar subtypes we consider the atoms $avatar(cell, angle, atype)$ where $angle$ is the possible avatar orientations (between 1 and 4 depending on the game), and $atype$ is the avatar stype (1 or 2; e.g. avatar-with-key and avatar-with-no-key). With this provision, the total number of atoms is in the order of $N \times T + N \times O \times A$ where O and A represent the avatar orientations and stypes respectively. For a grid with 100 cells and a hierarchy of 10 stypes, this means a number of atoms no greater than 1800. The IW(1) lookahead search will not expand then more than 1800 nodes.

The second issue to be addressed is the stochasticity of the games. In principle, IW algorithms can be used off the box oblivious to the fact that calling the simulator twice from the same state and with the same action may result in different successor states. Indeed, IW(i) would never call the simulator twice in that fashion. This is because IW(i) will not apply the same action twice in the same node, and the search tree generated by IW(i) can never have two nodes representing the same state (the tuples made true by the second node must have been made true by the first node). Yet ignoring stochasticity altogether and considering only the first outcome produced by the simulator can be too risky. For example, if the avatar has a killing monster to its right that moves randomly, it would be wise to move away from the monster. However, IW(i) may find the action of moving right safe, if the simulator returns a state where the monster also happened to move right. Thus to temper the risky optimism that follows from ignoring all but the state transitions returned first by the simulator when other transitions are possible, we add to IW(i) a simple safety check. Before running IW(i) for selecting an action in a state s, we take M samples of the successor state of each action a applicable in s, and count the number of times $D(a, s)$ where the avatar dies as a result of that one step (game lost). The actions a that are then regarded as *safe* in s are all the actions applicable in s that have a minimum count $D(a, s)$. This minimum count does not have to be zero, although hopefully it will be smaller than M. Equivalently, if $D(a, s) < D(a', s)$ for two applicable actions a and a' in s, action a' will be declared not safe in s.

The applicable actions that are labeled as *not safe* in the current state s are then treated as if they were not applicable in s. That is, when deciding which action to apply in s, actions that are not safe in s are deemed not applicable in s when performing the looking ahead from s using the $\text{IW}(i)$ algorithm. We call this pruning, *safety prepruning* or just *prepruning*. This pruning is shallow and fast, but helps to tame optimism when risk is immediate. The $\text{IW}(i)$ algorithms below use this simple form of prepruning. For comparison, a plain breadth-first search is also used in the experiments with and without prepruning. In all cases, the number M of samples is set to 10.

This way of dealing with stochastic state transitions is myopic but practical. Actually, it's common to design closed-loop controllers for stochastic systems using simplified deterministic models, letting the feedback loop take care of the modelling error. Similar "determinization" techniques have been used for controlling MDPs even without risk detection. Indeed, the on-line MDP planner called FF-replan [Yoon *et al.*, 2007] performed very well in the first two MDP competitions making the assumption that the uncertain state transitions are under the control of the planning agent. This converts the MDP into a classical planning problem that can be solved very efficiently. The resulting plans are executed and monitored, and when a state is observed that is not the one predicted by the simplified model, a new call to the classical planner is made from the observed state (replanning) and the process iterates until reaching the goal.

In our use of the IW algorithms, the determinization is not made by the planning agent but by the simulator, replanning is done at every step, and the prepruning trick is added to deal with immediate risk.

5 Experimental Results

For testing the IW algorithms in the GVG-setting, we used the software available in the competition site along with the 3 sets of 10 games each available.[1] For comparison, we consider the vanilla Monte Carlo Tree Search (MCTS) and the Open Loop MCTS (OLMCTS) controllers provided by the organizers [Perez *et al.*, 2015], a breadth-first search lookahead (BrFS), a simple 1-step lookahead (1-Look), and random action selection (RND). Moreover, for BrFs, we consider a variant with prepruning and one without. The 1-lookahead algorithm uses the $M = 10$ samples not only to prune unsafe actions but to choose the safe action with the most reward. Initially we evaluate the IW(1) algorithm but then consider IW(2) and an additional variant. In order to learn how time affects the performance of the various algorithms, we we consider three time windows for action selection: 40 milliseconds, 300 milliseconds, and 1 second. The experiments were run on an AMD Opteron 6300@2.4Ghz with 8GB of RAM.

[1]The competition site is `http://www.gvgai.net`.

Tables 1–3 show the performance of the algorithms for each of the game sets. The tables include BrFS (with prepruning), MCTS, OLMCTS, IW(1), 1-Look, and RND. We focus on a crisp performance measure: the number of games won by each algorithm. Since there are 5 levels in each game, and for each level we run 5 simulations, the maximum number of wins per game is 25. The total number of wins in each set of games is shown in the bottom row, which is bounded by the total number of games played in the set (250).

For the first set of games, shown in Table 1, we can see that IW(1) wins more games than MCTS and OLMCTS for each of the three time windows. For 40 msecs, the number of wins for MCTS, OLMCTS, and IW(1) are 89, 93, and 145 respectively, and the gap in performance grows with time: with 260 msecs more, MCTS and OLMCTS win 20 and 17 more games respectively, while IW(1) wins 36 more. Then, MCTS and OLMCTS do not win additional games when the window is extended to 1 second, while IW(1) wins 10 games more then. The total number of games won by each of these three algorithms when the time window is 1 second is thus 108, 108, and 191 respectively. Surprisingly, breadth-first search with prepruning does also very well in these games; indeed, better than MCTS and OLMCTS for all time windows, although not as well as IW(1), which also takes better advantage of longer times. BrFS with prepruning wins 125 games with 40 msecs, while normal BrFS without the pruning (not shown), wins 107. The two algorithms win 137 and 123 games respectively with 1 second. 1-step lookahead and random action selection end up winning 67 and 20 games respectively.

The pattern for the second set of games, shown in Table 2 is similar: IW(1) does best for all time windows and in this case by larger margins. For 40 msecs, MCTS, OLMCTS, and IW(1) win 68, 70, and 110 games respectively, while the numbers are 77, 90, and 183 for 1 second. Again, BrFS with prepruning does slightly better than MCTS for all time windows, although for this set it is tied with OLMCTS. BrFS without prepruning, on the other hand, does slightly worse. 1-step lookahead does almost as well as MCTS winning 62 of the games, while random selection wins 28.

Table 3 for the third set of games is however different. These games are more puzzle-like, like Sokoban, and as a result all the algorithms do poorly, including IW(1), that actually does worse than MCTS, OLMCTS, and even BrFS. The number of wins by MCTS and OLMCTS reach the 59 and 60 games, while IW(1) does not get beyond 45. Moreover, in these games, IW(1) does not get better with time. Of course, these games are more challenging, and indeed, none of the algorithms manages to solve 25% of the games for any time window. In the previous two sets of games, IW(1) managed to win more than 72% of the games with 1 second.

IW(2) and Variations

A key question is what makes the puzzle-like problems in the third set actual puzzles and thus more challenging. It's definitely not the size of the state space; indeed, the famous blocks world has a state space that is exponential in the number of blocks, and yet it's not a puzzle (unless one is looking for provably shortest solutions). A second question is why on

Time	40ms				300ms				1s					
Game	BrFS	MC	OLMC	IW(1)	BrFS	MC	OLMC	IW(1)	BrFS	MC	OLMC	IW(1)	1-Look	RND
Aliens	24	**25**	**25**	**25**	25	25	25	25	25	25	25	25	10	5
Boulderdash	0	0	**2**	1	5	1	0	**7**	5	2	0	**17**	1	0
Butterflies	24	24	**25**	**25**	25	25	25	25	25	25	25	25	22	9
Chase	3	0	0	**6**	5	1	1	**17**	8	0	0	**14**	1	0
Frogs	17	4	5	**20**	15	6	8	**25**	17	6	8	**25**	12	0
Missile Command	15	13	13	**17**	19	22	20	**25**	16	23	21	**25**	8	5
Portals	15	2	3	**17**	14	3	0	**20**	14	1	1	**22**	5	0
Sokoban	5	**7**	6	2	8	**10**	9	3	9	**11**	10	1	0	0
Survive Zombies	**13**	10	12	12	9	13	**14**	**14**	8	12	**14**	**14**	7	1
Zelda	9	4	2	**20**	9	3	8	**20**	10	3	4	**23**	1	0
Total	125	89	93	**145**	134	109	110	**181**	137	108	108	**191**	67	20

Table 1: Performance over first set of games for three time windows. Rows show wins over 5 simulations per level, 5 levels.

Time	40ms				300ms				1s					
Game	BrFS	MC	OLMC	IW(1)	BrFS	MC	OLMC	IW(1)	BrFS	MC	OLMC	IW(1)	1-Look	RND
Camel Race	**1**	0	**1**	0	1	3	0	**24**	1	0	3	**25**	0	1
Digdug	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0
Firestorms	**13**	2	2	11	14	7	6	**25**	12	3	4	**23**	10	0
Infection	22	21	19	**23**	21	19	**22**	21	20	22	20	**25**	19	22
Firecaster	0	0	0	0	0	0	**1**	0	0	0	0	**3**	0	0
Overload	10	4	7	**19**	17	3	5	**23**	17	5	5	**25**	0	0
Pacman	**1**	0	0	**1**	1	1	4	**14**	2	2	7	**22**	0	0
Seaquest	13	**16**	14	15	11	17	**22**	9	4	20	**24**	18	12	0
Whackamole	23	23	**24**	20	22	23	**25**	21	24	22	**25**	19	21	5
Eggomania	1	2	3	**15**	0	0	2	**22**	3	3	2	**23**	0	0
Total	84	68	70	**104**	87	73	87	**159**	84	77	90	**183**	62	28

Table 2: Performance over second set of games for three time windows. Rows show wins over 5 simulations per level, 5 levels.

such problems IW(1) does poorly, and moreover, doesn't get better with time. And finally, how can such problems be addressed computationally using generic methods.

Fortunately, for these questions, the theory behind the IW(i) algorithms helps [Lipovetzky and Geffner, 2012]. From this perspective, a problem is easy either if it has a low width, or if it can be easily serialized into problems with low width. This is certainly true for blocks world problems but not for problems like Sokoban. Indeed, while IW(1) runs in time that is linear in the number of features, it is *not* a complete algorithm in general; it is provably complete (and actually optimal) for width 1 problems only. Problems like Sokoban have a higher width even when a single stone needs to be placed in the goal, in the presence of other stones. One way to deal with such problems is by moving to a IW(i) algorithm with a higher width-parameter i.

Table 4 shows the results of the IW(2) algorithm for the third set of puzzle-like problems. The algorithm runs in time that is quadratic in the number of atoms and thus is less efficient than IW(1) but as it can be seen from the table, by 1 second, it has won 81 of the games, many more than those won in 1 second by MCTS and OLMCTS: 60 and 58. The three algorithms have a similar performance for 40 milliseconds but, while MCTS and OLMCTS do not improve much then, the opposite is true for IW(2). The atoms used by IW(2) are the same as those used by IW(1) but these atoms are used in *pairs* for pruning states in the breadth-first search. Given the special role of the avatar in the GVG-AI games, we also considered a variant of IW(2) that we call IW(3/2), which pays attention to *some* atom pairs only: those where one of the atoms is an avatar atom, i.e., an atom of the form $avatar(cell, angle, atype)$. As it can be seen in the table, IW(3/2) does better than IW(2), as while the resulting search considers a smaller set of nodes, it gets deeper in the search tree faster. The same is true for IW(1) in the first two sets of "easier" problems where it does better than both IW(2) and IW(3/2) for the same reasons (see Tables 5 and 6). The IW(i) algorithms are all breadth-first search algorithms that perform a more aggressive pruning the smaller the width-parameter i. More aggressive pruning means that the breadth-first search gets deeper sooner, while less aggressive pruning means that more paths are explored. There is thus a tradeoff: on easy games and small time windows, IW(1) is the best choice, but for harder problems and more time IW(3/2) and IW(2) will do better. As mentioned above, neither of these algorithms by themselves, nor MCTS or OLMCTS, will do well in simple serializable problems with joint goals, like when a tower of blocks needs to be built, even if the individual goals are rewarded. The reason is that just one serialization will do the job; namely, the one that achieves the goals bottom up. Methods for obtaining such serializations automatically using width-based algorithms are discussed in [Lipovetzky and Geffner, 2012], yet such methods, like other classical planning methods do not work with simulations and require a factored representation of the action effects and goals.

5.1 Profiles in Time

The performance of the algorithms over all the games is summarized in the curves shown in Figure 1. For each algorithm, the curve shows the total number of games won in the Y axis as a function of the three time windows shown in the X axis. The flat line shows the performance of 1-step lookahead. The curves for MCTS, OLMCTS, and BrFS show a higher number of wins for 40 milliseconds that however becomes prac-

Time	40ms				300ms				1s					
Game	BrFS	MC	OLMC	IW(1)	BrFS	MC	OLMC	IW(1)	BrFS	MC	OLMC	IW(1)	1-Look	RND
Bait	3	2	2	**7**	1	2	2	**5**	1	3	2	**5**	2	2
Bolo Adventures	0	0	0	0	0	0	**1**	0	0	**1**	0	0	0	0
Brain Man	**1**	**1**	**1**	0	**2**	**2**	**2**	0	1	0	**3**	0	0	0
Chips Challenge	4	**5**	4	0	**6**	3	4	0	**7**	5	5	2	4	0
Modality	**8**	6	5	6	**9**	6	8	5	5	**6**	**6**	5	5	2
Painters	**25**	21	24	**25**	**25**	**25**	24	**25**	25	25	25	25	6	8
Real Portals	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Real Sokoban	0	0	0	0	0	0	**9**	0	0	0	0	0	0	0
The Citadel	2	**5**	1	1	3	**4**	2	3	4	**7**	4	2	1	0
Zen Puzzle	5	**9**	5	6	6	15	**16**	6	6	**13**	**13**	5	2	2
Total	48	**49**	42	45	52	57	**59**	44	49	**60**	58	44	20	14

Table 3: Performance over third set of games for three time windows. Rows show wins over 5 simulations per level, 5 levels.

Time	40ms			300ms			1s		
Game	IW1	W2	IW3/2	IW1	IW2	IW3/2	IW1	IW2	IW3/2
Bait	**7**	5	5	5	5	**14**	5	10	**19**
Bolo Adv	0	0	0	0	0	0	0	0	0
Brain Man	0	1	**2**	0	0	**5**	0	3	**8**
Chips Ch	0	**2**	0	0	**2**	**2**	2	4	**5**
Modality	**6**	5	**6**	5	15	**20**	5	**20**	**20**
Painters	25	25	25	25	25	25	25	25	25
Real Port	0	0	0	0	0	0	0	0	0
R Sokoban	0	0	0	0	0	0	0	0	0
Citadel	1	**2**	0	3	3	**7**	2	12	**17**
Zen Puzz	**6**	**6**	5	6	6	**8**	5	7	**16**
Total	45	**46**	43	44	56	**81**	44	81	**100**

Table 4: IW(1), IW(2) and IW(3/2) in third set of games

Time	40ms			300ms			1s		
Game	IW1	IW2	IW3/2	IW1	IW2	IW3/2	IW1	IW2	IW3/2
Camel Race	0	**1**	0	**24**	0	5	**25**	0	5
Digdug	0	0	0	0	0	0	0	0	0
Firestorms	**11**	4	9	**25**	6	15	**23**	6	14
Infection	**23**	16	21	21	17	**24**	**25**	21	22
Firecaster	0	0	0	0	0	0	**3**	0	0
Overload	**19**	12	17	23	19	**24**	**25**	23	**25**
Pacman	**1**	0	**1**	**14**	0	3	**22**	0	6
Seaquest	**15**	11	9	9	9	**10**	**18**	9	5
Whackamole	20	20	**22**	**21**	**21**	11	19	24	**25**
Eggomania	**15**	0	0	**22**	0	14	**23**	1	20
Total	**104**	64	79	**159**	72	117	**183**	84	122

Table 6: IW(1), IW(2) and IW(3/2) in second set of games

Time	40ms			300ms			1s		
Game	IW1	IW2	IW3/2	IW1	IW2	IW3/2	IW1	IW2	IW3/2
Aliens	**25**	13	**25**	25	25	25	25	25	25
Boulder	1	0	**2**	**7**	0	6	**17**	0	6
Butterf	**25**	21	24	**25**	22	**25**	25	25	25
Chase	**6**	1	5	**17**	3	14	14	4	**16**
Frogs	**20**	0	17	**25**	0	21	**25**	0	**25**
Missile	**17**	10	13	**25**	18	20	**25**	20	24
Portals	**17**	10	11	**20**	12	16	**22**	13	16
Sokoban	2	2	**6**	3	12	**15**	1	15	**20**
Survive	**12**	8	9	**14**	6	9	**14**	10	11
Zelda	**20**	9	16	**20**	16	18	**23**	18	17
Total	**145**	74	128	**181**	114	169	**191**	130	185

Table 5: IW(1), IW(2) and IW(3/2) in first set of games

tically flat after 300 milliseconds. The curve for IW(1) starts higher than these curves and increases until reaching its peak at 1 second where the gap with the previous algorithms is very large. The curve for IW(2), on the other hand, starts lower but overtakes MCTS, OLMCTS, and BrFS after 300 milliseconds. Finally, the curve for IW(3/2) is similar to the curve for IW(1) although the two algorithms do best on different game sets: IW(1) on the easier games, and IW(3/2) on the last set.

6 Summary and Discussion

We have evaluated the IW(1) algorithm in the games of the GVG-AI framework and shown that it appears to outperform breadth-first search and plain Monte-Carlo methods like MCTS and OLMCTS. The exception are the puzzle-like games where all the algorithms do rather poorly. The theory behind the IW(i) algorithms provides useful hints for understanding why. The algorithm IW(2) does indeed much better over such problems but needs more time. The width parameter i in IW(i) captures a tradeoff: the higher the number, the higher the quality of the decision when given more time.

IW algorithms operate on state spaces where the relevant information about states can be encoded through a number of boolean features. For the games in the GVG-AI setting, these boolean features have been identified with atoms of the form $at(cell, stype)$ and $avatar(cell, angle, atype)$ where the stypes (atypes) correspond to the types of objects (avatars) in the sprite set. IW(1) preserves new states that bring new atoms and hence runs in time linear in this set of atoms, while IW(2) preserves new states that bring new pair of atoms, and runs in quadratic time. The algorithm IW(3/2) achieves a further tradeoff by considering only some pairs: those that include an avatar atom $avatar(cell, angle, atype)$.

IW algorithms are oblivious to the stochasticity of the domain as they only consider one successor state s' for every applicable action a in a state s: the one returned by the stochastic simulator the first time it's called with a and s. Since this may be too risky sometimes, before running IW(i) (and BrFS) from a state s in the GVG-AI games, actions that are deemed not safe in s after a 1-lookahead step that uses $M = 10$ samples, are deemed as not applicable in s. This way of dealing with uncertainty has similarity with forms of optimistic planning used for dealing with MDPs [Yoon *et al.*, 2007].

A key difference between IW algorithms and MCTS is that the former make an attempt at exploiting the structure of the problems and states. This is why IW algorithms can do so well in classical planning problems that often involve huge state spaces. On the other hand, in problems with large spaces and sparse rewards, uninformed MCTS methods act randomly and bootstrap slowly. Similarly, a key difference

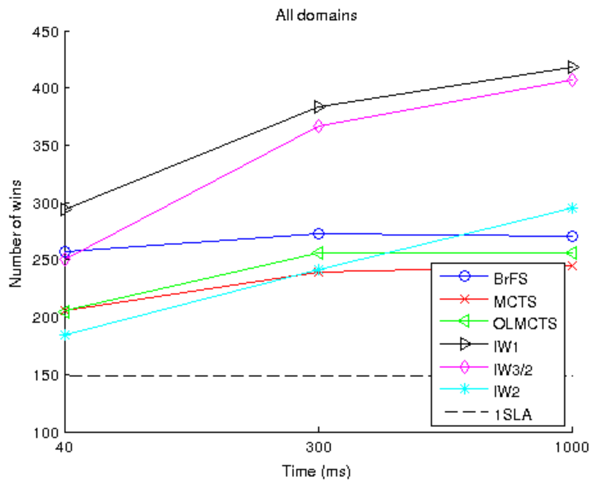

Figure 1: Total number of games won by each algorithm over the three sets of games as function of time window

between IW and standard classical planners is that the latter require an explicit, compact encoding of the action effects and goals.

These relations suggest a very concrete and crisp challenge; namely, how to plan effectively with a simulator when there is no explicit encoding of the action effects and goals in problems that are easy to serialize into simple problems but where not every serialization would work. The simplest example illustrating this challenge is no puzzle at all: it's the classical blocks world where the goal is to build a given tower of blocks. Getting rewards for achieving parts of the tower does not make the problem easier as the agent must realize that it has to work bottom up. Modern classical planners that have explicit descriptions of the action effects and goals do not run into this problem as they can easily infer such goal orderings. For simulation-based approaches, however, that is a challenge.

Acknowledgements: We thank the team behind the GVG-AI framework for a wonderful environment and set of problems, and Ivan Jimenez at UPF for help running the experiments. We also thank Nir Lipovetzky and Miquel Ramirez for the feedback.

References

[Bellemare *et al.*, 2013] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47(47):253–279, 2013.

[Biere *et al.*, 2009] A Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*. IOS Press, 2009.

[Chaslot *et al.*, 2008] GMJ Chaslot, M.H.M. Winands, H. Herik, J. Uiterwijk, and B. Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(3):343, 2008.

[Coles *et al.*, 2012] A. Coles, A.and Coles, A. Garcia Olaya, S. Jimenez, C. Linares, S. Sanner, and S. Yoon. A survey of the seventh international planning competition. *AI Magazine*, 33(1):83–88, 2012.

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

[Geffner, 2014] H. Geffner. Artificial Intelligence: From programs to solvers. *AI Communications*, 27(1):45–51, 2014.

[Gelly and Silver, 2007] S. Gelly and D. Silver. Combining online and offline knowledge in uct. In *Proc. ICML*, pages 273–280, 2007.

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

[Lipovetzky and Geffner, 2012] Nir Lipovetzky and Héctor Geffner. Width and serialization of classical planning problems. In *Proc. ECAI*, pages 540–545, 2012.

[Lipovetzky *et al.*, 2015] N. Lipovetzky, M. Ramirez, and H. Geffner. Classical planning algorithms on the atari video games. In *Proc. of 2015 AAAI Workshop on Learning for General Competency in Video Games*, 2015.

[Perez *et al.*, 2015] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, A. Couetoux, J. Lee, Chong-U Lim, and Tommy Thompson. The 2014 general video game playing competition. *IEEE Transactions on Computional Intelligence and AI in Games*, 2015. Forthcoming. At julian.togelius.com/Perez20152014.pdf.

[Rossi *et al.*, 2006] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[Schaul, 2013] T Schaul. A video game description language for model-based or interactive learning. In *Proc. IEEE Computational Intelligence in Games (CIG-2013)*, pages 1–8, 2013.

[Yoon *et al.*, 2007] S. Yoon, A. Fern, and R. Givan. FF-replan: A baseline for probabilistic planning. In *Proc. ICAPS-07*, pages 352–359, 2007.

[Younes *et al.*, 2005] H. Younes, M. Littman, D. Weissman, and J. Asmuth. The first probabilistic track of the international planning competition. *J. Artif. Intell. Res.(JAIR)*, 24:851–887, 2005.