Enhancements for Real-Time Monte-Carlo Tree Search in General Video Game Playing

Dennis J. N. J. Soemers, Chiara F. Sironi, Torsten Schuster, and Mark H. M. Winands

Department of Data Science and Knowledge Engineering, Maastricht University

d.soemers@gmail.com, t.schuster@student.maastrichtuniversity.nl, {c.sironi,m.winands}@maastrichtuniversity.nl

Abstract—General Video Game Playing (GVGP) is a field of Artificial Intelligence where agents play a variety of real-time video games that are unknown in advance. This limits the use of domain-specific heuristics. Monte-Carlo Tree Search (MCTS) is a search technique for game playing that does not rely on domain-specific knowledge. This paper discusses eight enhancements for MCTS in GVGP; Progressive History, N-Gram Selection Technique, Tree Reuse, Breadth-First Tree Initialization, Loss Avoidance, Novelty-Based Pruning, Knowledge-Based Evaluations, and Deterministic Game Detection. Some of these are known from existing literature, and are either extended or introduced in the context of GVGP, and some are novel enhancements for MCTS. Most enhancements are shown to provide statistically significant increases in win percentages when applied individually. When combined, they increase the average win percentage over sixty different games from 31.0% to 48.4% in comparison to a vanilla MCTS implementation, approaching a level that is competitive with the best agents of the GVG-AI competition in 2015.

I. Introduction

General Video Game Playing (GVGP) [1] is a field of Artificial Intelligence in games where the goal is to develop agents that are able to play a variety of real-time video games that are unknown in advance. It is closely related to General Game Playing (GGP) [2], which focuses on abstract games instead of video games. The wide variety of games in GGP and GVGP makes it difficult to use domain-specific knowledge, and promotes the use of generally applicable techniques.

There are two main frameworks for GVGP. The first framework is the Arcade Learning Environment (ALE) [3] for developing agents that can play games of the Atari 2600 console. The second framework is GVG-AI [4], which can run any real-time video game described in a Video Game Description Language [5], [6]. This paper focuses on the GVG-AI framework.

The GVG-AI framework is used in the GVG-AI Competition [4], [7]. Past competitions only ran a *Planning Track*, where agents were ranked based on their performance in single-player games. In 2016, it is planned to extend this with a *2/N-Player Track*, a *Learning Track*, and a *Procedural Content Generation Track*. This paper focuses on the Planning Track.

Monte-Carlo Tree Search (MCTS) [8], [9] is a popular technique in GGP [10] because it does not rely on domain-specific knowledge. MCTS has also performed well in GVGP in 2014 [4], which was the first year of the GVG-AI competition, but was less dominant in 2015 [7]. This paper discusses and evaluates eight enhancements for MCTS to improve its

performance in GVGP: *Progressive History*, *N-Gram Selection Technique*, *Tree Reuse*, *Breadth-First Tree Initialization*, *Loss Avoidance*, *Novelty-Based Pruning*, *Knowledge-Based Evaluations* and *Deterministic Game Detection*.

The remainder of the paper is structured as follows. Section II provides background information on the GVG-AI framework and the GVG-AI competition. MCTS is discussed in Section III. In Section IV, the enhancements for MCTS in GVGP are explained. Section V describes the experiments to assess the enhancements. Finally, the paper is concluded in Section VI and ideas for future research are discussed.

II. GVG-AI Framework and Competition

In the GVG-AI competition [4], [7], agents play a variety of games that are unknown in advance. Agents are given 1 second of processing time at the start of every game, and 40 milliseconds of processing time per *tick*. A tick can be thought of as a turn in an abstract game. Every tick, the agent can choose an action to play, and at the end of the tick the chosen action is played and the game state progresses. Every game has a duration of at most 2000 ticks, after which the game is a loss. Other than that, different games have different termination conditions, which define when the agent wins or loses. Every game in GVG-AI contains at least an *avatar* object, which is the "character" controlled by the agent. Games can also contain many other types of objects. Games in GVG-AI are *fully observable* and can be *nondeterministic*.

Agents can perform searches and attempt to learn which actions are good using the *Forward Model*, consisting of two important functions; *advance* and *copy*. Given a game state s_t, the *advance(a)* function can be used to generate a successor state s_{t+1}, which represents one of the possible states that can be reached by playing an action a. In deterministic games, there is only one such state s_{t+1} for every action a, but in nondeterministic games there can be more than one. The *copy(s_t)* function creates a copy of s_t. This function is required when it is desirable to generate multiple possible successors of s_t, because every call to *advance* modifies the original state, and there is no *undo* function. Because the framework supports a wide variety of different games, it is not optimized as well as any framework dedicated to a specific game would be. This means that the *advance* and *copy* operations tend to be significantly slower than equivalent functions in individual game implementations.
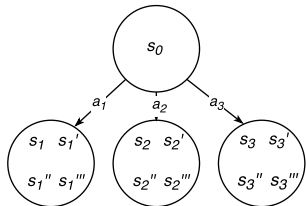
Fig. 1. Example open-loop game tree. Nodes other than the root node can represent multiple possible states in nondeterministic games.

Fig. 2. The four steps of an MCTS simulation. Adapted from [12].

III. MONTE-CARLO TREE SEARCH

Monte-Carlo Tree Search (MCTS) [8], [9] is a best-first search algorithm that gradually builds up a search tree and uses Monte-Carlo simulations to approximate the value of game states. To handle nondeterministic games with probabilistic models that are not exposed to the agent, an *"open-loop"* [11] implementation of MCTS is used. In an open-loop approach, the root node represents the current game state (s_0), every edge represents an action, and every other node n represents the set of game states that can be reached by playing the sequence of actions corresponding to the path from the root node to n, starting from s_0. See Figure 1 for an example.

MCTS is initialized with only the root node. Next, until some computational budget expires, the algorithm repeatedly executes simulations. Every simulation consists of the following four steps [12], depicted in Figure 2.

In the *Selection* step, a *selection policy* is applied recursively, starting from the root node, until a node is reached that is not yet fully expanded (meaning that it currently has fewer successors than available actions). The selection policy determines which part of the tree built up so far is evaluated in more detail. It should provide a balance between *exploitation* of parts of the search tree that are estimated to have a high value so far, and *exploration* of parts of the tree that have not yet been visited frequently. The most commonly implemented selection policy is *UCB1* [8], [13], which selects the successor S_i of the current node P that maximizes Equation 1. S_i and P are nodes, which can represent sets of states.

$$UCB1(S_i) = \overline{Q}(S_i) + C \times \sqrt{\frac{ln(n_P)}{n_i}} \qquad (1)$$

$\overline{Q}(S_i) \in [0, 1]$ denotes the normalized average score backpropagated through S_i so far (as described below), C is a parameter where higher values lead to more exploration, and n_P and n_i denote the visit counts of P and S_i, respectively.

In the *Play-out* step, the simulation is continued, starting from the last state encountered in the selection step, using a (semi-)random *play-out policy*. The most straightforward implementation is to randomly draw actions to play from a uniform distribution until a terminal game state is reached. In GVGP, this is typically not feasible, and a maximum play-out depth is used to end play-outs early.

In the *Expansion* step, the tree is expanded by adding one or more nodes. The most common implementation adds one node to the tree per simulation; the node corresponding to the first action played in the play-out step. In this paper, the
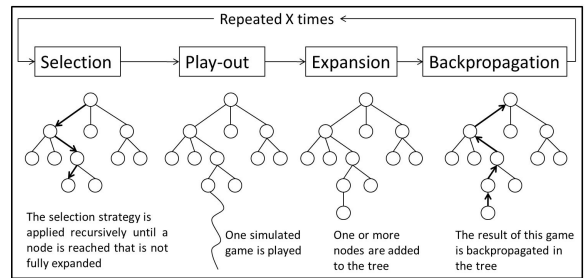
tree is simply expanded by adding the whole play-out to the tree. The number of simulations per tick tends to be low enough in GVG-AI that there is no risk of running out of memory. Therefore, to keep all information gathered, all nodes are stored in memory.

In the *Backpropagation* step, the outcome of the final state of the simulation is backpropagated through the tree. Let s_T be the final state of the simulation. Next, an evaluation $X(s_T)$ of the state is added to a sum of scores stored in every node on the path from the root node to the final node of the simulation, and the visit counts of the same nodes are incremented. Because it is not feasible to let all simulations continue until terminal states are reached in GVG-AI, it is necessary to use some evaluation function for non-terminal states. A basic evaluation function that is also used by the sample MCTS controllers included in the GVG-AI framework is given by Equation 2.

$$X(s_T) = \begin{cases} 10^7 + score(s_T) & \text{if } s_T \text{ is a winning state} \\ -10^7 + score(s_T) & \text{if } s_T \text{ is a losing state} \\ score(s_T) & \text{if } s_T \text{ is a non-terminal state} \end{cases}$$
$$(2)$$

$score(s_T)$ is the game score value of a state s_T in GVG-AI. In some games a high game score value can indicate that the agent is playing well, but this is not guaranteed in all games.

Finally, the action leading to the node with the highest average score is played when the computational budget expires.

IV. MCTS ENHANCEMENTS FOR GVGP

There is a wide variety of existing enhancements for the MCTS algorithm, many of which are described in [14]. This section discusses a number of enhancements that have been evaluated in GVGP; Progressive History, N-Gram Selection Technique, Tree Reuse, Breadth-First Tree Initialization, Loss Avoidance, Novelty-Based Pruning, Knowledge-Based Evaluations, and Deterministic Game Detection. Some are known from existing research, and some are new.

A. Progressive History and N-Gram Selection Technique

Progressive History (PH) [15] and *N-Gram Selection Technique* (NST) [16] are two existing enhancements for the selection and play-out steps of MCTS, respectively. The basic idea of PH and NST is to introduce a bias in the respective steps towards playing actions, or sequences of actions, that performed well in earlier simulations. Because the value of playing an action in GVG-AI typically depends greatly on the current position of the avatar, this position is also
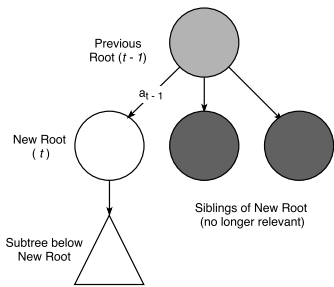
Fig. 3. Tree Reuse in MCTS.

Fig. 4. Example search tree. Dark nodes represent losing game states, and white nodes represent winning or neutral game states.

taken into account when storing data concerning the previous performance of actions. For a detailed description of these enhancements we refer to the original publications [15], [16].

B. Tree Reuse

Suppose that a search tree was built up by MCTS in a previous game tick $t - 1 \geq 0$, and an action a_{t-1} was played. The entire subtree rooted in the node corresponding to that action can still be considered to be relevant for the new search process in the current tick t. Therefore, instead of initializing MCTS with only a root node, it can be initialized with a part of the tree built in the previous tick, as depicted in Figure 3. This was previously found to be useful in the real-time game of *Ms Pac-Man* [17]. This idea has also previously been suggested in the context of GVGP [11], but, to the best of our knowledge, the effect of this enhancement on the performance of MCTS in GVGP has not yet been evaluated.

In nondeterministic games, it is possible that the new root (which was previously a direct successor of the previous root) represented more than one possible game state. In the current tick, it is known exactly which of those possible states has been reached. Therefore, some of the old results in this tree are no longer relevant. For this reason, all the scores and visit counts in the tree are decayed by multiplying them by a decay factor $\gamma \in [0, 1]$ before starting the next MCTS procedure. Tree Reuse (TR) with $\gamma = 0$ completely resets the accumulated scores and visit counts of nodes (but still retains the nodes, and therefore the structure of the generated tree), and TR with $\gamma = 1$ does not decay old results.

C. Breadth-First Tree Initialization and Safety Prepruning

In some of the games supported by the GVG-AI framework, the number of MCTS simulations that can be executed in a single tick can be very small; sometimes smaller than the number of available actions. In such a situation, MCTS behaves nearly randomly, and is susceptible to playing actions that lead to a direct loss, even when there are actions available that do not directly lose the game.

Theoretically this problem could be avoided by adjusting the limit of the play-out depth of MCTS to ensure that a sufficient number of simulations can be done. In practice, this can be problematic because it requires a low initial depth limit to ensure that it is not too high at the start of a game, and this can in turn be detrimental in games where it *is* feasible and beneficial to run a larger number of longer play-outs.
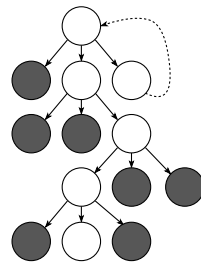
We propose to handle this problem using Breadth-First Tree Initialization. The idea is straightforward; before starting MCTS, the direct successors of the root node are generated by a 1-ply Breadth-First Search. Every action available in the root state is executed up to a number M times to deal with nondeterminism, and the resulting states are evaluated. The average of these M evaluations is backpropagated for every successor with a weight equal to a single MCTS simulation. MCTS is only started after this process. When MCTS starts, every direct successor of the root node already has a prior evaluation that can be used to avoid playing randomly in cases with an extremely small number of simulations. The M states generated for every successor are cached in the corresponding nodes, so that they can be re-used in the subsequent MCTS process. This reduces the computational overhead of the enhancement.

Safety prepruning, originally used in an algorithm called Iterated Width [18], has been integrated in this process. The idea of safety prepruning is to count the number of immediate game losses among the M generated states for each action, and only keep the actions leading to nodes with the minimum observed number of losses. All other actions are pruned.

D. Loss Avoidance

In GVGP, many games have a high number of losing game states that are relatively easy to avoid. An example of such a game is *Frogs*, where the avatar is a frog that should cross a road and a river. The road contains trucks that cause a loss upon collision, but can easily be avoided because they move at a constant speed. The river contains logs that also move at a constant speed, which the frog should jump on in order to safely cross the river.

An example of a search tree with many losing states is depicted in Figure 4. In this example, the rightmost action in the root node is an action that brings the agent back to a similar state as in the root node. In the *Frogs* game, this could be an action where the frog stays close to the initial position, and does not move towards the road.

The (semi-)random play used in the play-out step of MCTS is likely to frequently run into losing game states in situations like this. This leads to a negative evaluation of nodes that do in fact lead to a winning position. This is only corrected when sufficient simulations have been run such that the selection step of MCTS correctly biases the majority of the simulations towards a winning node. With a low simulation count in GVG-
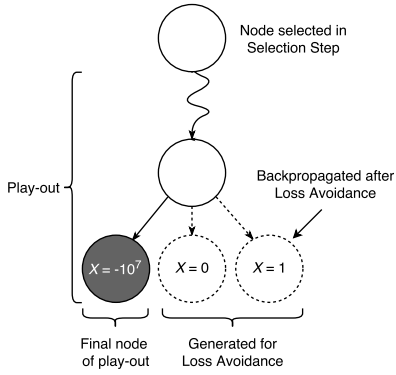
Fig. 5. Example MCTS simulation with Loss Avoidance. The X values in the last three nodes are evaluations of game states in those nodes. The dark node is a losing node.

AI, MCTS is likely to repeatedly play the rightmost action in Figure 4, which only delays the game until it is lost due to reaching the maximum game duration.

This problem is similar to the problem of *traps* [19] or *optimistic moves* [20] in (two-player) adversarial games. In those cases, MCTS has an overly optimistic evaluation of some states, whereas in the cases discussed here it has an overly pessimistic evaluation of some states. In [21], it was proposed to integrate shallow minimax searches inside some of the steps of MCTS to improve its performance in game trees with *traps* or *optimistic moves*. Using minimax searches to *prove* wins or losses is difficult in GVGP because games can be nondeterministic, but a similar idea can be used to get less pessimistic evaluations.

In this paper, an idea named *Loss Avoidance* (LA) is proposed for GVGP. The idea of LA is to try to ignore losses by immediately searching for a better alternative whenever a loss is encountered the first time a node is visited. An example is depicted in Figure 5. Whenever the play-out step of MCTS ends in a losing game state, that result is not backpropagated as would commonly be done in MCTS. Instead, one state is generated for every sibling of the last node, and only the evaluation of the node with the highest evaluation is backpropagated. All generated nodes are still added to the tree, and store their own evaluation in memory.

LA causes MCTS to keep an optimistic initial view of the value of nodes. This tends to work well in the single-player games of GVG-AI, where it is often possible to reactively get out of dangerous situations. It is unlikely to work well in, for instance, adversarial games, where a high concentration of losses in a subtree typically indicates that an opposing player has more options to win and is likely in a stronger position.

In an open-loop implementation of MCTS, LA can have a significant amount of computational overhead in game trees with many losses. For instance, in the *Frogs* game it roughly halves the average number of MCTS simulations per tick. This is because the node prior to the node with the losing game state does not store the corresponding game state in memory, which means that *all* states generated in the selection and play-out steps need to be re-generated by playing the same action sequence from the root node. In nondeterministic games this

process can also lead to finding a terminal state before the full action sequence has been executed again. To prevent spending too much time in the same simulation, the LA process is not started again, but the outcome of that state is backpropagated.

E. Novelty-Based Pruning

The concept of *novelty tests* was first introduced in the Iterated Width algorithm (IW) [18], [22]. In IW, novelty tests are used for pruning in Breadth-First Search (BrFS). Whenever a state s is generated in a BrFS, a novelty measure (described in more detail below) $nov(s)$ is computed. This is a measure of the extent to which s is "new" with respect to all previously generated states. States with a lower measure are "more novel" than states with a higher measure [22]. The original IW algorithm consists of a sequence of calls to IW(0), IW(1), etc., where IW(i) is a BrFS that prunes a state s if $nov(s) > i$. In GVGP, it was found that it is only feasible to run a single IW(i) iteration [18]. The best results were obtained with IW(1), and a variant named IW($\frac{3}{2}$) (see [18] for details).

The definition of the novelty measure $nov(s)$ of a state s requires s to be defined in terms of a set of boolean features. An example of a boolean feature that can be a part of a state is a predicate *at(cell, type)*, which is true in s if and only if there is an object of the given type in the given cell in s. Then, $nov(s)$ is defined as the size of the smallest tuple of features that are all true in s, and not all true in any other state generated previously in the same search process. If there is no such tuple, s must be an exact copy of some previously generated state, and $nov(s)$ is defined as $n + 1$, where n is the number of features that are defined. For example, suppose that in s, $at((x, y), i) = true$, and in all previously generated states, $at((x, y), i) = false$. Then, $nov(s) = 1$, because there is a tuple of size 1 of features that were not all true in any previously generated state.

IW(1) prunes any state s with $nov(s) > 1$. In this paper, *Novelty-Based Pruning* (NBP) is proposed as an idea to prune nodes based on novelty tests in MCTS. The goal is not to prune *bad* lines of play, but to prune *redundant* lines of play.

MCTS often generates states deep in the tree before other states close to the root. For instance, the last state of the first play-out is much deeper in the tree than the first state of the second play-out. This is an important difference with the BrFS used by IW. It means that the novelty measure $nov(s)$ of a state s should be redefined in such a way that it not necessarily uses *all* previously generated states, but only a specific set of states, referred to as the *neighborhood* $N(s)$ of s.

$N(s)$ is the union of four sets of states. The first set consists of the siblings on the "left" side of s. The ordering of the states matters, but can be arbitrary (as in a BrFS). The second set contains only the parent $p(s)$ of s. The third set consists of *all* siblings of $p(s)$. The fourth set is the neighborhood of $p(s)$. More formally, let s_i denote the ith successor of a parent $p(s_i)$. Then, $N(s_i)$ is defined as $N(s_i) = \{s_1, s_2, \ldots, s_{i-1}\} \cup \{p(s_i)\} \cup Sib(p(s_i)) \cup N(p(s_i))$, where $Sib(p(s_i))$ denotes the set of siblings of $p(s_i)$. For the root state r, $N(r) = Sib(r) = \varnothing$. An example is depicted in Figure 6.
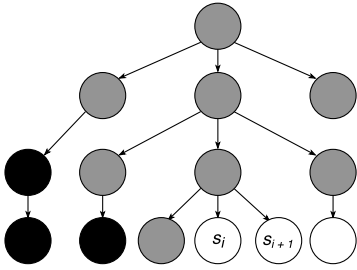
Fig. 6. States used for NBP in MCTS. The grey states are the neighborhood of s_i in MCTS. For s_{i+1}, s_i is also included. The black states would be included for the novelty tests in IW, but not in MCTS.

Using the above definition of $N(s)$, $nov(s, N(s))$ is defined as the size of the smallest tuple of features that are all true in s, and not all true in any other state in the set $N(s)$. The novelty tests are used in MCTS as follows. Let n be a node with a list of successors $Succ(n)$. The first time that the selection step reaches n when it is fully expanded, all successors $Succ(n)$ are novelty tested based on a single state generated per node, using a threshold of 1 for the novelty tests (as in IW(1)). The same boolean features are used to define states in GVG-AI as described in [18]. Nodes are marked as not being novel if they fail the novelty test. Whenever *all* successors of a node are marked as not novel, that node itself is also marked as not novel. There are a few exceptions where nodes are not marked. If a state has a higher game score than the parent, it is always considered to be novel. Additionally, states transitioned into by playing a movement action are always considered to be novel in games where either only horizontal, or only vertical movement is available (because these games often require moving back and forth which can get incorrectly pruned by NBP otherwise), and in games where the avatar has a movement speed ≤ 0.5 (because slow movement does not result in the avatar reaching a new cell every tick, and is therefore not detected by the cell-based boolean features).

In the selection step of MCTS, when one of the successors $Succ(n)$ of n should be selected, any successor $n' \in Succ(n)$ is ignored if it is marked as not novel, unless the average normalized score $\overline{Q}(n) < 0.5$. In such cases, the situation is considered to be dangerous and all alternatives should be considered to see if a better position can be found. For the final selection of the move to play in the real game, non-novel nodes are also only considered if the best novel alternative has a normalized average score < 0.5.

When the successors $Succ(n)$ have been novelty tested, every node $n_i \in Succ(n)$ stores a set of tuples of features that were all true in the states generated for the purpose of novelty testing for the nodes $\{n\} \cup Succ(n)$. This means that the tuples of features that are true in the neighborhood $N(s)$ of a state s can be reconstructed relatively efficiently by traversing the path from s back to the root, and collecting the tuples in the stored sets. This is the main reason for defining $N(s)$ as described above. Including more states (for instance, the black states in Figure 6) would require also traversing back down the tree to collect more sets of tuples. This could increase the number of nodes that NBP marks as not being novel, but

would also be more expensive computationally. This is not a problem in the BrFS of IW, because it can simply store all tuples of features that are all true in any generated state in the same set for the entire search process.

Novelty measures are assigned to nodes based on only one state per node. Therefore, given two identical open-loop game trees in nondeterministic games, it is possible that a node in one tree is pruned and the equivalent node in the other tree is not pruned. For this reason, when combining NBP with Tree Reuse, the results of novelty tests on nodes in the first ply below the new root node are reset when reusing the previous tree. This does not entirely remove the influence of nondeterminism on NBP, but close to the root that influence is at least reduced.

F. Knowledge-Based Evaluations

An important problem with MCTS in GVG-AI is that it is often infeasible to find any terminal states, or even states with a change in game score. This means that the evaluation function in Equation 2 often returns the same value for all states generated in the same tick, and MCTS explores the search space and behaves randomly. In this paper, a heuristic evaluation function is proposed that uses knowledge collected during simulations, and distances to objects that could potentially be interesting, to distinguish between states that have identical evaluations according to Equation 2. The basic idea is not new; some agents in the competition of 2014 used distance-based evaluation functions [4]. A similar idea is also described in [23], and extended in [24]. The idea discussed here is based on the same intuition, but a number of implementation details are different. Another related idea is described in [25], where MCTS is used to learn which objects are interesting, and a pathfinding algorithm is used to move towards a selected goal.

Let $X(s_0)$ denote the evaluation of the current game state s_0, and let $X(s_T)$ denote the evaluation of the final state s_T of a play-out. If $X(s_T) = X(s_0)$, a heuristic evaluation $Eval_{KB}(s_T)$ is computed and added to $X(s_T)$. For every object type i observed in a game, let $d_0(i)$ denote the distance from the avatar to the closest object of type i in s_0, and let $d_T(i)$ denote the distance from the avatar to the closest object of type i in s_T. These distances are computed using the A* pathfinding algorithm [26]. The pathfinding algorithm takes objects of the *wall* type into account as obstacles. Many games can also contain other objects that block movement, or portals that can be used for teleportation. These objects are not taken into account, because the agent would first need to learn how these objects influence pathfinding. For every object type i, a weight w_i is used to reward or punish the agent for moving to objects of that type. This is done by computing $Eval_{KB}(s_T)$ as given by Equation 3, normalizing it to lie in $[0, 0.5]$, and adding it to $X(s_T)$ if otherwise $X(s_T) = X(s_0)$.

$$Eval_{KB}(s_T) = \sum_i w_i \times (d_0(i) - d_T(i)) \qquad (3)$$

Object types i with a small absolute weight ($|w_i| < 10^{-4}$) are ignored, to save the computational cost of pathfinding.

The weights w_i are determined as follows. To motivate exploration, all weights are initialized with positive values (0.1 for NPCs, 0.25 for Movables, and 1 for Resources and Portals), and incremented by 10^{-4} every game tick. States s_t generated during the selection or play-out steps of MCTS are used to adjust these weights. Let s_{t-1} denote the predecessor of s_t. Whenever such a state s_t is generated, it is used to update some of the weights w_i. The intuition is that, if $X(s_t) \neq X(s_{t-1})$, it is likely that some interesting collision event occurred in the transition from s_{t-1} to s_t that caused the change in score. The framework provides access to a set $E(s_t)$ of collision events that occurred in that transition. Every event $e \in E(s_t)$ is a collision event between two objects, where one object is either the avatar, or an object created by the avatar (for instance, a missile fired by the avatar), and the other object is of some type i. Let $\Delta = X(s_t) - X(s_{t-1})$ denote the observed change in score. For every object type i, a sum Δ_i is kept of all changes in scores observed in state transitions where collision events with objects of type i occurred. Additionally, a counter n_i of event occurrences is kept for every type i, such that the average change in score $\overline{\Delta}_i = \frac{\Delta_i}{n_i}$ for collisions with every type can be computed. Whenever an event with an object of type i is observed, w_i is updated as given by Formula 4.

$$w_i \leftarrow w_i + (\overline{\Delta}_i - w_i) \times \alpha_i \qquad (4)$$

α_i is a learning rate that is initialized to 0.8 for every type, and updated as given by Formula 5 after updating w_i.

$$\alpha_i \leftarrow \max(0.1, 0.75 \times \alpha_i) \qquad (5)$$

This idea is similar to using gradient descent for minimizing $|\overline{\Delta}_i - w_i|$. The main reason for not simply using $\overline{\Delta}_i$ directly is to avoid relying too much on the knowledge obtained from a low number of observed events.

G. Deterministic Game Detection

The idea of *Deterministic Game Detection* (DGD) is to detect when a game is likely to be deterministic, and treat deterministic games differently from nondeterministic games. At the start of every game, M random sequences of actions of length N are generated. Each of the M sequences is used to advance a copy of the initial game state s_0, with R repetitions per sequence. If any of the M action sequences did not result in equivalent states among the R repetitions for that sequence, the game is classified as nondeterministic. Additionally, any game in which NPCs are observed is immediately classified as nondeterministic. Any other game is classified as deterministic. In this paper, $M = N = 5$ and $R = 3$.

Many participants in previous GVG-AI competitions [7] used a similar idea to switch to a different algorithm for deterministic games (for instance, using Breadth-First Search in deterministic games and MCTS in nondeterministic games). In this paper, DGD is only used to modify MCTS and the TR and NBP enhancements in deterministic games. The $\overline{Q}(S_i)$ term in Equation 1 (or the equivalent term in the formula of PH) is replaced by $\frac{3}{4} \times \overline{Q}(S_i) + \frac{1}{4} \times \hat{Q}_{max}(S_i)$, where $\hat{Q}_{max}(S_i)$ is the maximum score observed in the subtree rooted in S_i.

TABLE I
WIN PERCENTAGES (BENCHMARK AGENTS, 1000 RUNS PER SET)

Sets	SOLMCTS	MCTS	IW(1)	YBCRIBER
Set 1	34.5 ± 2.9	41.7 ± 3.1	55.8 ± 3.1	68.8 ± 2.9
Set 2	33.4 ± 2.9	33.5 ± 2.9	47.0 ± 3.1	65.0 ± 3.0
Set 3	13.2 ± 2.1	23.0 ± 2.6	17.8 ± 2.4	40.3 ± 3.0
Set 4	28.3 ± 2.8	30.5 ± 2.9	30.6 ± 2.9	43.5 ± 3.1
Set 5	19.7 ± 2.5	28.9 ± 2.8	17.5 ± 2.4	42.6 ± 3.1
Set 6	30.1 ± 2.8	28.6 ± 2.8	32.8 ± 2.9	54.4 ± 3.1
Total	26.5 ± 1.1	31.0 ± 1.2	33.6 ± 1.2	52.4 ± 1.3

This is referred to as *mixmax* [27], [28]. Additionally, TR and NBP are modified to no longer decay or reset any old results.

V. EXPERIMENTS

A. Setup

The enhancements discussed in this paper have been experimentally evaluated using the following setup. Every experiment was run using six sets that are available in the framework, of ten games each, for a total of sixty different games per experiment. Table VI lists the names of the games for every set. Average results are presented for every set of games, and for the total of all sixty games combined. For every game, five different levels were used, with a minimum of fifteen repetitions per level per experiment (leading to a minimum of 750 runs per set). 95% confidence intervals are presented for all results. All games were played according to the GVG-AI competition rules[1], on a CentOS Linux server consisting of four AMD Twelve-Core OpteronT 6174 processors (2.2 GHz).

B. Results

In the first experiment, the following benchmark agents are compared to each other; SOLMCTS, MCTS, IW(1), and YBCRIBER. SOLMCTS is the Sample Open Loop MCTS controller included in the framework. MCTS is our baseline implementation of MCTS, based on the MAASTCTS [29] agent, which has a number of differences in comparison to SOLMCTS. MCTS expands all nodes for states generated in simulations (as opposed to one node per simulation), C is set to 0.6 in the UCB1 equation (as opposed to $C = \sqrt{2}$), it simulates up to ten actions after the selection step (as opposed to ten steps from the root node), it uses the 1 second of initialization time for running the algorithm (as opposed to not using that time), and it plays the action with the maximum average score (as opposed to the maximum visit count). IW(1) is the Iterated Width-based agent, as described in [18]. YBCRIBER is an IW-based agent with a number of other features, which won the GVG-AI competition at the IEEE CEEC 2015 conference. The results are given in Table I. The experimental data reveals that the baseline MCTS agent outperforms SOLMCTS. IW(1) performs slightly better than MCTS overall, and YBCRIBER performs much better than the other benchmark agents.

In Table II, our MCTS implementation with Breadth-First Tree Initialization and Safety Prepruning (BFTI) is compared

---

[1]Revision 24b11aea75722ab02954c326357949b97efb7789 of the GVG-AI framework (https://github.com/EssexUniversityMCTS/gvgai) was used.

TABLE II
BREADTH-FIRST TREE INITIALIZATION WITH SAFETY PREPRUNING

Sets	Win Percentage		% of Losses $t < 2000$	
	BFTI	MCTS	BFTI	MCTS
Set 1	43.3 ± 3.5	41.7 ± 3.1	42.6 ± 4.7	52.8 ± 4.1
Set 2	33.1 ± 3.4	33.5 ± 2.9	50.8 ± 4.4	51.1 ± 3.8
Set 3	21.2 ± 2.9	23.0 ± 2.6	0.0 ± 0.0	16.1 ± 2.6
Set 4	30.3 ± 3.3	30.5 ± 2.9	73.4 ± 3.8	76.8 ± 3.1
Set 5	23.1 ± 3.0	28.9 ± 2.8	72.4 ± 3.6	73.7 ± 3.2
Set 6	29.2 ± 3.3	28.6 ± 2.8	69.3 ± 3.9	72.3 ± 3.3
Total	30.0 ± 1.3	31.0 ± 1.2	51.0 ± 1.7	56.7 ± 1.5

TABLE III
WIN PERCENTAGES (PH AND NST, 750 RUNS PER SET)

Sets	BFTI	PH	NST	NST+PH
Set 1	43.3 ± 3.5	43.2 ± 3.5	45.1 ± 3.6	43.5 ± 3.5
Set 2	33.1 ± 3.4	34.5 ± 3.4	36.5 ± 3.4	38.0 ± 3.5
Set 3	21.2 ± 2.9	23.3 ± 3.0	23.1 ± 3.0	24.1 ± 3.1
Set 4	30.3 ± 3.3	29.5 ± 3.3	29.7 ± 3.3	32.3 ± 3.3
Set 5	23.1 ± 3.0	23.9 ± 3.1	30.0 ± 3.3	28.0 ± 3.2
Set 6	29.2 ± 3.3	30.0 ± 3.3	31.1 ± 3.3	33.1 ± 3.4
Total	30.0 ± 1.3	30.7 ± 1.3	32.6 ± 1.4	33.2 ± 1.4

to the MCTS implementation without BFTI. The results for MCTS are based on 1000 runs per set, and the results for BFTI on 750 runs per set. BFTI appears to lower the win percentage slightly, but the 95% confidence intervals overlap. The two columns on the right-hand side show the percentage of lost games where the game was terminated before $t = 2000$ (where $t = 2000$ is the maximum duration of a game in GVG-AI). BFTI reduces this percentage significantly. Even though it may slightly decrease win percentages, the quality of play in lost games can be considered to be improved; the agent delays a significant number of losses. This may leave more time for other enhancements to find wins. Therefore, BFTI is included in the baseline MCTS agent for the following experiments that evaluate other enhancements individually. This is followed by an experiment with more enhancements combined.

Table III shows the win percentages obtained by adding Progressive History (PH), N-Gram Selection Technique (NST), or both to the BFTI agent. PH and NST appear to increase the average win percentage, but the confidence intervals overlap. The two combined result in a statistically significant increase.

Figure 7 depicts 95% confidence intervals for the win percentage of the BFTI agent with Tree Reuse (TR), for six different values of the decay factor γ. The confidence interval for BFTI is shaded in grey. TR with $\gamma \in \{0.4, 0.6, 1.0\}$ significantly improves the win percentage of BFTI.

Table IV shows the win percentages of adding either Knowledge-Based Evaluations (KBE), Loss Avoidance (LA) or Novelty-Based Pruning (NBP) to the BFTI agent. All three individually show an increase in the average win percentage over BFTI, with KBE giving the largest increase.

Table V shows the win percentages of a number of variants of MCTS with multiple enhancements combined. "No DGD" is an agent with all enhancements discussed in this paper, except for Deterministic Game Detection (DGD). "No BFTI" is an agent with all enhancements except for BFTI. This is added to test the assumption made earlier that the ability of BFTI
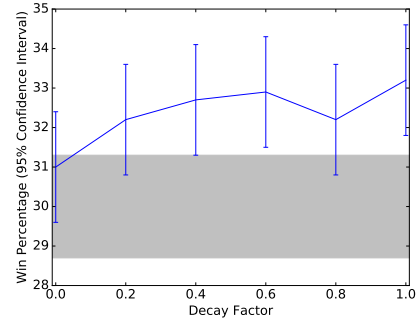

Fig. 7. 95% confidence intervals for win percentages of BFTI with Tree Reuse (TR) for different values of the decay factor γ. The area shaded in grey is the confidence interval for the win percentage of BFTI without TR.

TABLE IV
WIN PERCENTAGES (KBE, LA AND NBP, 750 RUNS PER SET)

Sets	BFTI	KBE	LA	NBP
Set 1	43.3 ± 3.5	50.4 ± 3.6	52.0 ± 3.6	49.6 ± 3.6
Set 2	33.1 ± 3.4	52.3 ± 3.6	34.0 ± 3.4	34.5 ± 3.4
Set 3	21.2 ± 2.9	19.1 ± 2.8	23.3 ± 3.0	23.5 ± 3.0
Set 4	30.3 ± 3.3	30.1 ± 3.3	29.6 ± 3.3	32.0 ± 3.3
Set 5	23.1 ± 3.0	31.3 ± 3.3	31.9 ± 3.3	23.9 ± 3.1
Set 6	29.2 ± 3.3	33.2 ± 3.4	28.8 ± 3.2	34.8 ± 3.4
Total	30.0 ± 1.3	36.1 ± 1.4	33.3 ± 1.4	33.0 ± 1.4

to delay games may enable other enhancements to find more wins. The last agent contains all enhancements. In combination with all the other enhancements, DGD significantly improves the win percentage. DGD was found not to provide a significant increase in win percentage when applied to the BFTI, TR ($\gamma = 0.6$) or NBP agents without other enhancements (those results have been omitted to save space). Additionally, BFTI appears to increase the win percentage in combination with all other enhancements, whereas Table II shows it appears to *decrease* the win percentage when other enhancements are absent, but these differences are not statistically significant.

VI. CONCLUSION AND FUTURE WORK

Eight enhancements for Monte-Carlo Tree Search (MCTS) in General Video Game Playing (GVGP) have been discussed and evaluated. Most of them have been shown to significantly (95% confidence) increase the average win percentage over sixty different games when added individually to MCTS. All the enhancements combined increase the win percentage of our basic MCTS implementation from 31.0 ± 1.2 to 48.4 ± 1.5. This final performance is relatively close to the win percentage of the winner of the IEEE CEEC 2015 conference; YBCRIBER, with a win percentage of 52.4 ± 1.3.

Many of the discussed enhancements have parameters, which so far have only been tuned according to short, preliminary experiments. These parameters can likely be tuned better in future work to improve the performance. Loss Avoidance (LA) and Novelty-Based Pruning (NBP) as proposed in this paper have binary effects, in that LA backpropagates only one result from multiple generated siblings and NBP classifies nodes as either novel or not novel. Perhaps these can be improved by making them less binary. The overall performance of the agent can also likely be improved by incorporating more features that are commonly seen among the top entries in past

TABLE V
WIN PERCENTAGES (ENHANCEMENTS COMBINED, 750 RUNS PER SET)

Sets	BFTI	No DGD	No BFTI	All Enhanc.
Set 1	43.3 ± 3.5	62.7 ± 3.5	62.7 ± 3.5	62.8 ± 3.5
Set 2	33.1 ± 3.4	56.4 ± 3.5	55.7 ± 3.6	59.3 ± 3.6
Set 3	21.2 ± 2.9	22.1 ± 3.0	28.5 ± 3.2	28.7 ± 3.2
Set 4	30.3 ± 3.3	32.7 ± 3.4	47.1 ± 3.6	48.1 ± 3.6
Set 5	23.1 ± 3.0	37.2 ± 3.5	39.6 ± 3.5	42.1 ± 3.5
Set 6	29.2 ± 3.3	38.3 ± 3.5	49.2 ± 3.6	49.1 ± 3.6
Total	30.0 ± 1.3	41.6 ± 1.4	47.1 ± 1.5	48.4 ± 1.5

TABLE VI
NAMES OF THE GAMES IN EVERY SET

Set 1	*Aliens, Boulderdash, Butterflies, Chase, Frogs, Missile Command, Portals, Sokoban, Survive Zombies, Zelda*
Set 2	*Camel Race, Digdug, Firestorms, Infection, Firecaster, Overload, Pacman, Seaquest, Whackamole, Eggomania*
Set 3	*Bait, BoloAdventures, BrainMan, ChipsChallenge, Modality, Painter, RealPortals, RealSokoban, TheCitadel, ZenPuzzle*
Set 4	*Roguelike, Surround, Catapults, Plants, Plaque-Attack, Jaws, Labyrinth, Boulderchase, Escape, Lemmings*
Set 5	*Solarfox, Defender, Enemy Citadel, Crossfire, Lasers, Sheriff, Chopper, Superman, WaitForBreakfast, CakyBaky*
Set 6	*Lasers 2, Hungry Birds, Cook me Pasta, Factory Manager, Race Bet 2, Intersection, Black Smoke, Ice and Fire, Gymkhana, Tercio*

competitions, such as the use of influence maps [30]. Finally, some of the new enhancements for MCTS, such as LA and NBP, can be evaluated in domains other than GVG-AI.

ACKNOWLEDGEMENT

REFERENCES

[1] J. Levine, C. B. Congdon, M. Ebner, G. Kendall, S. M. Lucas, R. M. anad T. Schaul, and T. Thompson, "General Video Game Playing," in *Artif. and Comput. Intell. in Games*, ser. Dagstuhl Follow-Ups, S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, Eds. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, vol. 6, pp. 77–83.

[2] M. Genesereth, N. Love, and B. Pell, "General Game Playing: Overview of the AAAI Competition," *AI Magazine*, vol. 26, no. 2, pp. 62–72, 2005.

[3] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents," *Journal of Artif. Intell. Research*, vol. 47, pp. 253–279, 2013.

[4] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, A. Couëtoux, J. Lee, C.-U. Lim, and T. Thompson, "The 2014 General Video Game Playing Competition," *IEEE Trans. on Comput. Intell. and AI in Games*, 2016, To appear.

[5] M. Ebner, J. Levine, S. M. Lucas, T. Schaul, T. Thompson, and J. Togelius, "Towards a Video Game Description Language," in *Artif. and Comput. Intell. in Games*, ser. Dagstuhl Follow-Ups, S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, Eds. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, vol. 6, pp. 85–100.

[6] T. Schaul, "A Video Game Description Language for Model-based or Interactive Learning," in *Proc. of the IEEE Conf. on Comput. Intell. in Games*. Niagara Falls: IEEE Press, 2013, pp. 193–200.

[7] D. Perez-Liebana, S. Samothrakis, J. Togelius, S. M. Lucas, and T. Schaul, "General Video Game AI: Competition, Challenges and Opportunities," in *Proc. of the Thirtieth AAAI Conf. on Artif. Intell.* AAAI Press, 2016, pp. 4335–4337.

[8] L. Kocsis and C. Szepesvári, "Bandit Based Monte-Carlo Planning," in *Machine Learning: ECML 2006*, ser. LNCS, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds. Springer Berlin Heidelberg, 2006, vol. 4212, pp. 282–293.

[9] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *Computers and Games*, ser. LNCS, H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, Eds., vol. 4630. Springer Berlin Heidelberg, 2007, pp. 72–83.

[10] Y. Björnsson and H. Finnsson, "CadiaPlayer: A Simulation-Based General Game Player," *IEEE Trans. on Comput. Intell. and AI in Games*, vol. 1, no. 1, pp. 4–15, 2009.

[11] D. Perez, J. Dieskau, M. Hünermund, S. Mostaghim, and S. M. Lucas, "Open Loop Search for General Video Game Playing," in *Proc. of the Genetic and Evol. Computation Conf.* ACM, 2015, pp. 337–344.

[12] G. M. J.-B. Chaslot, M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and B. Bouzy, "Progressive Strategies for Monte-Carlo Tree Search," *New Mathematics and Natural Computation*, vol. 4, no. 3, pp. 343–357, 2008.

[13] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time Analysis of the Multiarmed Bandit Problem," *Machine Learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[14] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlf-shagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *IEEE Trans. on Comput. Intell. and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[15] J. A. M. Nijssen and M. H. M. Winands, "Enhancements for Multi-Player Monte-Carlo Tree Search," in *Computers and Games (CG 2010)*, ser. LNCS, H. J. van den Herik, H. Iida, and A. Plaat, Eds. Springer Berlin Heidelberg, 2011, vol. 6515, pp. 238–249.

[16] M. J. W. Tak, M. H. M. Winands, and Y. Björnsson, "N-Grams and the Last-Good-Reply Policy Applied in General Game Playing," *IEEE Trans. on Comput. Intell. and AI in Games*, vol. 4, no. 2, pp. 73–83, 2012.

[17] T. Pepels, M. H. M. Winands, and M. Lanctot, "Real-Time Monte Carlo Tree Search in Ms Pac-Man," *IEEE Trans. on Comput. Intell. and AI in Games*, vol. 6, no. 3, pp. 245–257, 2014.

[18] T. Geffner and H. Geffner, "Width-Based Planning for General Video-Game Playing," in *Proc. of the Eleventh Artif. Intell. and Interactive Digital Entertainment International Conf.*, A. Jhala and N. Sturtevant, Eds. AAAI Press, 2015, pp. 23–29.

[19] R. Ramanujan, A. Sabharwal, and B. Selman, "On Adversarial Search Spaces and Sampling-Based Planning," in *20th International Conf. on Automated Planning and Scheduling*, R. I. Brafman, H. Geffner, J. Hoffmann, and H. A. Kautz, Eds. AAAI, 2010, pp. 242–245.

[20] H. Finnsson and Y. Björnsson, "Game-Tree Properties and MCTS Performance," in *IJCAI'11 Workshop on General Intelligence in Game Playing Agents (GIGA'11)*, Y. Björnsson, N. Sturtevant, and M. Thielscher, Eds., 2011, pp. 23–30.

[21] H. Baier and M. H. M. Winands, "MCTS-Minimax Hybrids," *IEEE Trans. on Comput. Intell. and AI in Games*, vol. 7, no. 2, pp. 167–179, 2015.

[22] N. Lipovetzky and H. Geffner, "Width and Serialization of Classical Planning Problems," in *Proc. of the Twentieth European Conf. on Artif. Intell. (ECAI 2012)*, L. De Raedt, C. Bessiere, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, and P. Lucas, Eds. IOS Press, 2012, pp. 540–545.

[23] D. Perez, S. Samothrakis, and S. Lucas, "Knowledge-based Fast Evolutionary MCTS for General Video Game Playing," in *Proc. of the IEEE Conf. on Comput. Intell. and Games.* IEEE, 2014, pp. 68–75.

[24] J. van Eeden, "Analysing and Improving the Knowledge-based Fast Evolutionary MCTS Algorithm," Master's thesis, Utrecht University, Utrecht, the Netherlands, 2015.

[25] C. Y. Chu, H. Hashizume, Z. Guo, T. Harada, and R. Thawonmas, "Combining Pathfinding Algorithm with Knowledge-based Monte-Carlo Tree Search in General Video Game Playing," in *Proc. of the IEEE Conf. on Comput. Intell. and Games.* IEEE, 2015, pp. 523–529.

[26] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100–107, 1968.

[27] E. J. Jacobsen, R. Greve, and J. Togelius, "Monte Mario: Platforming with MCTS," in *Proc. of the 2014 Conf. on Genetic and Evolutionary Computation.* ACM, 2014, pp. 293–300.

[28] F. Frydenberg, K. R. Andersen, S. Risi, and J. Togelius, "Investigating MCTS Modifications in General Video Game Playing," in *Proc. of the IEEE Conf. on Comput. Intell. and Games.* IEEE, 2015, pp. 107–113.

[29] T. Schuster, "MCTS Based Agent for General Video Games," Master's thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, the Netherlands, 2015.

[30] I. Millington and J. Funge, *Artificial Intelligence for Games*, 2nd ed. Morgan Kaufmann, 2009.