Monte Mario: Platforming with MCTS

Emil Juul Jacobsen
IT University of Copenhagen
Rued Langgaards Vej 7
Copenhagen, Denmark
ejuu@itu.dk

Rasmus Greve
IT University of Copenhagen
Rued Langgaards Vej 7
Copenhagen, Denmark
ragr@itu.dk

Julian Togelius
IT University of Copenhagen
Rued Langgaards Vej 7
Copenhagen, Denmark
julian@togelius.com

ABSTRACT

Monte Carlo Tree Search (MCTS) is applied to control the player character in a clone of the popular platform game Super Mario Bros. Standard MCTS is applied through search in state space with the goal of moving the furthest to the right as quickly as possible. Despite parameter tuning, only moderate success is reached. Several modifications to the algorithm are then introduced specifically to deal with the behavioural pathologies that were observed. Two of the modifications are to our best knowledge novel. A combination of these modifications is found to lead to almost perfect play on linear levels. Furthermore, when adding noise to the benchmark, MCTS outperforms the best known algorithm for these levels. The analysis and algorithmic innovations in this paper are likely to be useful when applying MCTS to other video games.

1. INTRODUCTION

Monte Carlo Tree Search (MCTS) is a relatively recently devised statistical tree search method [2] that has proven to be remarkably good at playing classic board games such as *Go* [8]. MCTS has also shown promise when it comes to playing unseen games such as those in the *General Game Playing Competition* [6], partly due to not relying on a state evaluation function. In both of these domains, MCTS is arguably the best method available, as variations of this algorithm tops the leagues for computer players. Naturally, it has therefore been suggested that MCTS has much broader applicability than this, presenting a viable approach to all of game AI or perhaps even all of AI. However, in domains such as Go and general game playing[1] games are played in a relatively small number of discrete turns and the state of the game is discrete and can be described using a relatively small number of variables. In other words, not only are they discrete-time and discrete-state, both the time and state space are highly constrained.

Most games that we think of as "video games" are completely different from this. It is very common to emulate some of the physical aspects of the "real world", which at a human scale is continuous-time and continuous-space. While digital computers are by their nature discrete state machines, the state space of a computer is vast, and continuous state and time in a game world can be approximated using floating point numbers and multiple frames per second. Thus, first-person shooter games such as *Halo* or *Bioshock*, real-time strategy games such as *StarCraft* or *Age of Empires* and action adventure games such as *Tomb Raider* and *Skyrim* typically have game states that update 30 or 60 times per second, and where the position of each object or character is denoted by a 32 or 64 bit floating point value per dimension. These circumstances would seem to pose significant obstacles for any method based on evaluating all possible actions (moves) at each time step (turn). Therefore, we feel it is an open question whether MCTS can be made to perform well in pseudo-continuous game domains, and how.

There have been a few recent attempts to apply MCTS to such domains. In particular, there has been considerable work on applying MCTS to the Physical Travelling Salesman Problem (PTSP), which can be described as a mix between the classic TSP problem and a racing game. This game has pseudo-continuous space, but very few actions available per time step and a static game world. It was found that MCTS-based agents exhibit human-competitive performance, but only after several modifications, including macro-actions that constitute several smaller actions [10, 11]. MCTS has also been applied to small-scale combat scenarios in StarCraft with moderate success [4]. Multiple attempts has been made at applying MCTS to *Ms. Pac-Man* both for controlling the Pac-Man agent as well as for ghost enemy control with great success [12], [9].

In this paper, we consider the problem of playing *Super Mario Bros*, or more accurately the *Mario AI Benchmark*, which is based on a clone of that game, using MCTS. This is not only an important game representative of a very popular genre, but also provides a different kind of challenge than anything that has been attempted with MCTS so far. Though it shares some properties with Ms. Pac-Man as both games require realtime decision making and give the player the option of allowing the agent to "procrastinate" by not making any progress towards an end state. In the next

[1]With "general game playing", we are here referring to the type of games used in the existing GGP competition and expressible in that language, elsewhere referred to as "Stanford GDL"; other types are certainly possible and even proposed [5].

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.

ACM 978-1-4503-2662-9/14/07 ...$15.00.
http://dx.doi.org/10.1145/2576768.2598392.

two sections we give a brief background on both MCTS and the Mario AI Benchmark. We then describe the particular way in which we applied MCTS to the Mario AI Benchmark, and describe the unsatisfactory behaviour of "vanilla" MCTS. The next section describes a number of modifications to the algorithm that we developed specifically to counter the identified failures of MCTS on this problem. The results section details the quantitative performance of MCTS with and without the modifications, and compares it to A* search in state space, which has performed well on these levels. The results analysis section qualitatively discusses the behaviour of the algorithm for selected combinations of modifications. After noting that the deterministic nature of the benchmark does not exploit the strength of MCTS in handling unpredictable behaviour, we produce a noisy version of the benchmark, and again compare its performance to A*.

In the result tables in this paper, we compare each result with a specified other result and show how statistically significantly different they are. Statistical significance levels are denoted by stars (*) or daggers (†) and the levels are: * means $p \leq 0.05$, ** means $p \leq 0.01$ and *** means $p \leq 0.0005$,

2. MONTE CARLO TREE SEARCH

MCTS is a tree search algorithm in which evaluation of a node is done by performing random actions from the decision space until an outcome can be determined. MCTS is an *anytime* algorithm, meaning that it can be halted when a time limit expires and give the result that looks the most promising at the given time. Furthermore it often requires little, if any, domain knowledge because a basic implementation only requires knowledge of the action space and a means of simulating the outcome of an action.

Searching using MCTS is done by iteratively building a search tree where the nodes are different game states, and the edges are the actions leading to one state from another. A node is added to the tree during each iteration and recursively, based on the reward of the new node, the reward values of parent nodes are updated. A single iteration of the MCTS building process consists of these four steps:

1. Tree Policy (A node to be expanded is chosen)
2. Expansion (The node is expanded by simulating the associated action)
3. Default Policy (The game is simulated following a random path until a terminal node is reached)
4. Backpropagation (The result propagates up through the tree)

Upper Confidence Bound for Trees (UCT) is a bandit based approach to choosing the most urgent node to expand. The benefit from UCT is that it allows for prioritizing between exploitation of seemingly promising nodes (first term) and exploration of less tried nodes (second term).

$$UCB_j = \overline{X}_j + C_p \cdot \sqrt{\frac{2 \cdot \ln(n)}{n_j}} \quad (1)$$

Equation 1 is used for calculating the Upper Confidence Bound for node j. Here $\overline{X}_j$ is the average (i.e. expected) reward over the times (n_j) node j has been visited and C_p is a constant for adjusting the weight of the exploration term. n is the total number of expansions and n_j is the number of times node j was selected. Since the exploration term of the equation depends on how explored the node is compared to the parent, the confidence in a node will increase steadily until the node is eventually explored. The exploitation term will, however, make sure that good nodes will be explored more frequently than less promising ones.

3. THE MARIO AI BENCHMARK

Super Mario Bros (SMB), published by Nintendo in 1985, is one of the world's best-selling games and certainly one of the most influential. It introduced the canonical form of side-scrolling platform games, and its design has inspired countless games since. *Infinite Mario Bros* (IMB) is a Java-based clone of SMB by Notch. IMB is the basis for the Mario AI Benchmark, a benchmark for artificial intelligence that has been used in a series of international competitions [7].

In IMB, the player controls Mario, who can be made to walk or run left or right, jump, and can be upgraded to jump through bricks and spit fire. The main objective in each level is to reach the end (the rightmost point) of the level, which involves dodging or killing enemies and not falling into gaps. The original game is controlled with a joypad and two buttons; in the Mario AI Benchmark, an action is defined as 5 bits (buttons: left, right, down, a, b), yielding 32 possible actions. 25 times per second, the controller is presented with a representation of the area surrounding Mario (one screen) and asked to return what action to take.

The first Mario AI competition, which ran in 2009, was won by Robin Baumgarten who implemented an agent based on A* search in state space [14]. At every time step, the agent tried to find the shortest path towards the right edge of the current screen (that is, in the direction of the end of the level) using a simple heuristic based on horizontal position. This agent managed to consistenly clear the linear levels that was used in the 2009 competition, but lost its top position when levels that require backtracking were introduced in the 2010 competition [1].

4. MCTS IN MARIO

When searching the action space using MCTS we found the heaviest computation to be the simulation of actions on a given state. This makes it practically impossible to perform the rollout to termination and even in that case it would practically always be a death.

This led us to experiment with several different evaluation functions for calculating the reward value of a non-terminal state. The first logical solution would be to use the x-distance to the right edge of the screen (as far as Mario can see) and let the value be the fraction between the initial position (for this tick) and the edge of the screen. This, however, gave very bad results with Mario unable to clear even small obstacles and consistently running backwards when facing Bullet Bills flying towards him.

Another approach was to make the value relative to only the position of the parent node and let the maximum reward be the furthest you could potentially go by running directly to the right. This evaluation function shows much better performance and is what we have chosen to use.

$$reward_j = \frac{1}{2} + \frac{1}{2} \cdot \frac{x_j - x_p}{11 \cdot (1 + n_R)} \quad (2)$$

In Equation 2 x_j is the current node's final x-coordinate and x_p is the parent's. The denominator of the right term is the maximum x-distance Mario can run in one tick where n_R is the chosen rollout length cap. This value is always within the range of 0 to 1.

Since the levels generated in this version of Super Mario did not require the use of the "down" key, and including it among the possible actions degraded performance greatly, we removed that key from the action set.

For the simulation of future game states we use the LevelScene model by Robin Baumgarten [14] which was originally extracted from the game engine itself in order to obtain the correct behaviour of monsters and objects. This model is provided with the environment information given to the agent to start the simulation from the current game state. Each node in the search tree therefore contains the LevelScene as its state and the different childrens' states are the result of performing a specific action on the current node's state. Each node has capacity for a child corresponding to each possible action for the possibility to explore in all possible directions.

4.1 Parameters

Since we had to decide how long the roll out phase could continue for we had to determine the best rollout depth cap for our basic MCTS implementation. To do this we have performed a series of benchmarks. The results can be seen in Table 1 where we have highlighted the most promising value. The mean score in our benchmarking is the mean of x-distance traveled in each of the 100 levels. Due to how the levels are randomly generated their length can vary slightly but are generally around 4300 pixels.

The roll out depth cap we chose did not give the highest value, but has a low average completion time and has a score very close to the highest valued one.

We have done the same kind of parameter tuning for the C_p value (for UCB calculation). The results can be found in table 2. We chose the value of 0.25 for C_p since it has shown good results and behaviour on manual tests and the highest score is not significantly different. The rollout cap and C_p values chosen here have been used in all further tests and modifications.

5. MODIFICATIONS TO MCTS

The performance of the MCTS-based agent was not dreadfully bad, but not satisfactory, and far from the elegance of Baumgarten's A* agent. There were several failures: Mario was sometimes unable to jump over tall cannons, was very reluctant to jump over gaps, sometimes fell into gaps and ran straight into enemies. Mario seemed shortsighted and cowardly. Based on this qualitative analysis, we suggested the following improvements.

5.1 Mixmax rewards

When calculating the confidence of a node by the average value of its children, the resulting behaviour of Super Mario will be quite defensive. This is because a single good path among many dangerous ones will only increase the average value slightly.

Rollout Cap	Mean Score	Avg. Time Left
0	1220***	-
1	1430***	116
2	1790***	138
4	2684***	134
6	**3861**	**134**
8	3843	120
10	2930***	93
12	1724***	71
16	922***	11
$1 * 10^6$	248***	-

Table 1: Results of vanilla MCTS with various rollout depth cap. These calculations are performed with a C_p-value of 0.25 on 100 levels. The results are the mean score of all levels and the average time left from completed levels. The stars denote the statistical significance of difference from the results of all measurements and the result from using a rollout cap of 6.

C_p-value	Mean Score	Avg. Time Left
0	3319***	131
0.188 (1.5/8)	3993	134
0.25	**3950**	**131**
0.333	3868	132
0.5	3859	132
0.707 $(1/\sqrt{2})$	3815	129
2	3886	132
5	3791	130
10	3868	130

Table 2: Results of vanilla MCTS with various C_p values with rollout depth cap of 6. These calculations are performed with a rollout depth cap of 6 on 100 levels. The results are the mean score of all levels and the average left over time from completed levels. The stars denote the statistical significance of difference from the results of all measurements and the result from using a C_p value of 0.25.

Mixmax Q-value	Mean Score	Avg. Time Left
0 (Avg.)	4029	131
0.125	**4103**	**146**
0.25	4093	149
0.375	3846**	151
0.5	4007	152
0.75	3753**	153
1 (Max)	2098***	153

Table 3: Results of using Mixmax backup with different Q values on 100 levels. The results are the mean score of all levels and the average time left from completed levels. The stars denote the statistical significance of the difference from each measurement to the result of using $Q = 0.125$

This can be alleviated by calculating the total reward as a mix between the average value and the maximum value between the children. Equation 3 show the use of the constant Q as a factor between the two values. This value will then be used for the UCB calculation (in Equation 1 replacing $\overline{X}_j$).

$$exploitation = Q \cdot max + (1 - Q) \cdot \overline{X}_j \tag{3}$$

Through experiments (See table 3) we have observed that a Q-value of 0.125 yields the best results and is better than both average ($Q = 0$) and maximum ($Q = 1$) while having a much better average completion time.

5.2 Macro-actions

In Super Mario, the pace at which new moves are to be decided is so fast that the search tree rarely gets deeper than 4-5 levels. This results in Mario making decisions based purely on the very nearest surroundings, and possibly taking a bad route on the macroscopic level. (E.g. choosing to jump into a pack of monsters instead of taking a peaceful route above them on another path). Because Macro Actions have been previously shown to be a good enhancement for MCTS allowing a tradeoff between precision and strategic quality in a realtime continous domain [11], it is interesting to see how it changes the performance of MCTS in Super Mario.

Macro Actions allows us to make better use of the limited search depth by simulating further into the level via a coarse path. This is achieved by modifying the expansion process such that instead of performing the action just once, it is repeated a predefined number of times before a child node is created. After an action has been decided upon, it is performed this same number of repetitions, thus leaving more time to expand on the same search tree.

In some cases the coarse route and the commitment to repeating an action a certain number of times is, however, not optimal. In particular, close to monsters or gaps there is a need for micro-planning where every action taken can be different. When close to monsters, the monsters must be avoided or hit correctly in order to stay alive, and when close to gaps, the best way to jump over them is to move close to the edge before jumping. Consequently our implementation switches to the normal action size of 1 when in danger, which is determined by being either close to a monster or a gap. The distance thresholds for being close have been determined by just a few loose experiments as this enhancement alone is not the focus of the study. This does include some domain-knowledge into the implementation of macro-actions.

5.3 Partial Expansion

Simulating a move in Super Mario has proven to be a relatively time consuming operation. Since about half the possible actions lead in the wrong direction with respect to where the finish line is, it seems superfluous to simulate them if it can be avoided.

The strategy "progressive unpruning" deals with the issue of having a large branching factor and little time, by first reducing the branching factor artificially, and then unprune actions when more time is available [3]. In Partial Expansion we follow this idea, but instead of reducing the branching factor, we simply add the option of traversing partially expanded nodes. In the tree policy step of MCTS the urgency of expanding a new child is calculated and compared to the confidences for each existing child. Only if the urgency for creating a new child exceeds any of the childrens' confidences a new child is created. With this change UCT can either choose to exploit a promising path, or expand another unexplored action.

$$UCB_c = k + C_p \cdot \sqrt{\frac{2 \cdot \ln(n)}{1 + c_n}} \tag{4}$$

Equation 4 calculates the urgency of creating a new child and is quite similar to Equation 1, but with some differences, since the non-existent nodes don't have a calculated reward and have never been visited. The k-value is the standard reward for non-existent children, we set this constant to 0.5. c_n is the number of expanded children on the current node. This way nodes will be gradually expanded with nodes higher in the tree being expanded more than lower ones.

This results in a deeper tree since the average branching factor of the nodes is significantly reduced, but when selecting the child to expand randomly you risk missing promising paths. The higher branches of the tree will, however, be expanded more than the lower, and since the deeper levels of the tree have less immediate effect on the state this risk isn't very high.

5.4 Roulette wheel selection

Our implementation of MCTS expand nodes through multiple iterations, and not all children at once. This requires that we decide an order in which children are to be generated. Normally this is chosen at random, but by introducing some domain knowledge, the generally most beneficial actions can be chosen for expansion first. Expanding some actions before others does not have a great impact on its own, but it does skew the search tree in a desirable direction, and is interesting in combination with other enhancements.

The method of Roulette Wheel Selection is implemented by associating a weight with each possible action. These weights are determined from prior observations of the algorithm playing Super Mario levels by recording which actions where used the most in successful playthroughs. When selecting which child to create while expanding a node, the weights are used in the random selection as tickets in a lottery, making actions with high weights more likely to be selected than actions with low weights - this without removing the possibility of any action, good or bad, being selected first.

Using Roulette Wheel Selection with MCTS has previously shown good results in determining what child to expand next [13].

5.5 Domain knowledge

Although MCTS can get a long way in Super Mario without introducing much domain knowledge, some challenges of the game are better tackled with some knowledge of the game. Since we had to ignore the down key, the action space consist of 16 actions. By removing actions that either don't add any new options (e.g. left + right + jump is equal to just jump) or that we deemed unnecessary we managed to reduce the set of actions to 10. This decreased the branching factor which lead to increasing the possible search depth in the same amount of time.

A relatively frequent challenge in Super Mario levels is gaps. Gaps can be of varying depth and if Mario touches the bottom he dies. The challenge with gaps is that their deadly property is only discovered if the algorithm is able to simulate steps all they way to the bottom. Deep gaps pose a problem as the shortsightedness of unmodified MCTS (Figure 1, yellow line) results in Mario going down the gap and ending up walljumping back and forth near the bottom of the gap if not dying straight away. By detecting when a position is inside a gap we can avoid them by drastically reducing the reward of positions inside gaps.

6. RESULTS

6.1 Individual modifications

Figure 1: Illustrates the different search depths for three individual modifications. Red: Macro Actions, Yellow: Vanilla MCTS, Blue: Best Combination

Our experiment consist of 6 different enhancements, all of which are tested in combination with each other giving 64 different agents to test.

The agents are tested on the same 100 randomly generated levels on the hardest difficulty the Mario AI Framework allows. We use the "Wilcoxon signed-ranks test" to calculate p, and require $p < 0.05$ for the difference to be significant.

The calculated final x-position of mario in each level is the level score, and the mean of all levels is the score that the combination receives. Table 5 show the results of these tests.

Modification	Mean Score	Avg. T Left
Vanilla MCTS (Avg.)	3918	131
Vanilla MCTS (Max)	2098***	153
Mixmax (0.125)	4093	147
Macro Actions	3869	142
Partial Expansion	3928	134
Roulette Wheel Selection	4032	139
Hole Detection	4196**	134
Limited Actions	4141*	137
(Robin Baumgarten's A*)	4289***	169

Table 4: Results of each individual modification on 100 levels. The stars denote the statistical significance in difference from each result to unmodified MCTS. We include results from running Baumgarten's A* implementation as reference.

In this section the behaviour of the individual and the most interesting combinations is described. Results from testing the individual enhancements can be found in table 4.

6.2 Vanilla MCTS

When adjusted with the optimal values for the constants C_p and rollout depth cap, the basic MCTS algorithm actually performs rather well. It completes 80 out of 100 levels. The main problems with this agent is the very limited search depth which is never more than 5 actions, constituting only 200ms ahead (See Figure 1, yellow line). This makes the agent vulnerable towards monsters falling from above and makes it hard to scale the tall cannon towers.

6.2.1 *Mixmax*

The higher weight of the largest reward beneath each node reduces the cowardness of Mario greatly. This change can, however, also in some cases result in reckless and dangerous behaviour, sometimes resulting in certain death. That is alleviated by only letting the maximum reward have a relatively low influence ($\frac{1}{8}$) compared to the average reward ($\frac{7}{8}$). This skews the simulation tree further towards the larger values exploring those nodes further while making it more certain that they are safe.

The Mixmax modification on its own doesn't show a significant difference from Vanilla MCTS as can be seen in table 4.

6.2.2 *Macro Actions*

The increased action size of Macro Actions enables the search to go further into the level without having more iterations (Figure 1, red line). This not only makes Mario take routes with less monsters, but also enables the agent to quickly scale tall cannons, which seems to be a challenge. This reduces the wasted time (and reduce timeout related losses), but since Mario is exposed to monsters when trying to find a way over the canon it also reduces the risk of dying from monsters.

6.2.3 *Partial Expansion*

Like the Macro Actions enhancement, using Partial Expansion enables the agent to search to a greater depth. This

Method	Score	T	Method	Score	T
----	3918 †	131	---L	4141* †	137
X---	4093 †	147	X---L	4152* †	147
-M---	3869 †	142	-M--L	4025 †	143
XM---	3922 †	146	XM--L	4043 †	147
-P--	3928 †	146	-P-L	4214* †	146
X-P--	4109* †	140	X-P-L	4278*	150
-MP--	3997 †	134	-MP-L	4156* †	135
XMP--	4166* †	139	XMP-L	4220* †	143
--R-	4032 †	139	--R-L	4132* †	142
X-R-	3786 †	149	X-R-L	4134* †	150
-M-R-	3956 †	145	-M-R-L	4063 †	145
XM-R-	4088 †	146	XM-R-L	4031 †	150
-PR-	4271*	149	-PR-L	4275*	153
X-PR-	4281*	154	X-PR-L	4260*	156
-MPR-	4165* †	135	-MPR-L	3955 †	136
XMPR-	4182* †	141	XMPR-L	4145* †	140
---H-	4196* †	134	---HL	4161* †	139
X--H-	4221* †	145	X--HL	4281*	147
-M-H-	4182* †	142	-M-HL	4251*	141
XM-H-	4197* †	146	XM-HL	4260*	146
-P-H-	2679*†	96	-P-HL	4277* †	138
X-P-H-	3656* †	105	X-P-HL	4277*	147
-MP-H-	2692*†	112	-MP-HL	3970 †	125
XMP-H-	3583* †	105	XMP-HL	4237* †	138
--RH-	4212* †	139	--RHL	4211* †	142
X-RH-	4206* †	148	X-RHL	4195* †	150
-M-RH-	4189* †	143	-M-RHL	4204* †	141
XM-RH-	4240*†	145	XM-RHL	4248*†	147
-PRH-	4268*	148	-PRHL	4284*	152
X-PRH-	4274*	153	X-PRHL	4272*	155
-MPRH-	4071 †	126	-MPRHL	3692 †	126
XMPRH-	4189* †	132	XMPRHL	4061 †	131

Table 5: Results of all enhancement combinations. X: Mixmax, M: Macro Actions, P: Partial Expansion, R: Roulette Wheel Selection, H: Hole Detection, L: Limited Actions. The results are the mean score of all levels and the T value is the average time left on completed levels. Stars denote statistical significance in difference from unmodified MCTS. Daggers denote statistical significance in difference from a perfect score of 4289.

enables Mario to quickly and safely jump over gaps, and choose safer routes. The greater depth is obtained when some actions are clearly better than others (in terms of how much to the right Mario gets). This unfortunately means that Partial Expansion doesn't help in the case where Mario gets stuck in front of cannons, as none of the immediate moves give a high reward and thus exploration evens out the tree.

6.2.4 *Roulette Wheel Selection*

On its own Roulette Wheel selection doesn't make much of a difference to the behaviour. This is because all children of a node must be simulated before the next depth can be searched, thus the order in which children are created only makes a difference in the last level of expansion where not all children are created. Unless the values of the roulette are chosen badly this modification should give atleast the same results as without, since exploring more promising actions earlier should seem safer.

6.2.5 *Hole Detection*

This enhancement very clearly has a great effect on the ability to avoid going into gaps, which in turn reduces the number of gap-related deaths.

6.2.6 *Limited Actions*

Limiting the action space reduces the branching factor which increases the search depth. Since we had to remove the down key in our base algorithm the reduction isn't that big, and the effect of limited actions isn't very noticeable on its own.

6.3 No domain knowledge

A particularly interesting combination is Mixmax and Partial Expansion. This combination is the best agent consisting only of enhancements that do not add any domain knowledge to the agent. It performs better than Vanilla MCTS with a significant difference, but is still some way from a winning score. It seems that some domain knowledge is needed in order to really improve the performance of MCTS.

6.4 Partial Expansion + Roulette Wheel Selection

The combination of Partial Expansion with Roulette Wheel Selection has shown very good performance. The two enhancements separately only shows decent results, but in conjunction the algorithm is able to search a great distance into the level, which allows for planning on the macroscopic level, but with great detail in movement when need be. This combination in able to detect and avoid gaps by searching far enough to reach the opposite edge. They complement each other well since Roulette Wheel Selection is a solution the the problem that Partial Expansion wants to explore the most promising actions first and leave the rest for later.

6.5 Best combination

The combination that gave the best results was the one using all enhancements except Macro Actions. It is not the combination with the highest score in the benchmarking but was fast and more robust than other candidates. The feature that Macro Actions would give is the ability to search far into the level at the cost of presicion, but when using Partial

Expansion and Roulette Wheel Selection in conjunction this is achieved, even without losing the precision as is the case with Macro Actions (Figure 1, blue line). Hole Detection makes the search more focused on actions not leading into gaps, Limited Actions reduces the action space and Mixmax reduces the cowardly behaviour. This combination enables the agent to often search to the end of the screen. It is evident that this combination still can fail in certain difficult levels, but even here it rarely happens.

The variation observed is due to the fact that MCTS is not deterministic in that the chosen action can vary depending on what order the actions of a state are explored. Additionally the scheduling of the agentâĂŹs thread can vary, possibly yielding a different number of iterations of the algorithm, and thus possibly changing the choice of actions.

7. ANALYSIS OF RESULTS

The cowardly behaviour of Mario when using unaugmented MCTS is very interesting, as it points to a problem that is likely to recur in other video game domains. Essentially, the agent became very risk-averse: it refused to take actions that might lead to positive rewards (e.g. jumping over a gap) because most of the random rollouts would lead to substantial negative rewards (e.g. falling into the gap). To our best knowledge and mild surprise, this has not been identified as a problem with MCTS before. To understand why, it is instructive to look at the differences between the types of games to which MCTS is commonly applied and to games like SMB. In SMB, it is possible to "procrastinate" by standing still or moving back and forth. This is because no forward movement is enforced (the screen follows the player character) and there's no adversary to race or combat. The negative consequences of procrastinating – running out of time – only show up hundreds of seconds (tens of thousands of time steps) later, which is beyond the rollout depth of any MCTS agent. In contrast, board games such as Go necessarily approach the end of the game with every move, and a rollout of a manageable length will always reach an end state.

Given that many video games have more in common with Super Mario Bros than with Go, it stands to reason that the same modifications to MCTS that helped overcome the pathological behaviour of unaugmented MCTS on this problem would improve the efficiency of this algorithm on other problems. While it is always possible to incorporate domain knowledge in various forms, it is interesting to note that it was possible to achieve near-perfect play with only knowledge-free modifications. It is entirely plausible that Mixmax backups, partial expansion or roulette wheel selection would make MCTS work well in a first-person shooter or an open-world racing game.

The most surprising thing about the Mixmax backup modification is that we have not seen it described in the literature before. It is a very simple idea, which essentially gives us a "testosterone gauge" for Mario: the more of the max reward we mix in, the more risks Mario takes. It is possible that such a modification would never be useful in a game such s Go or Chess, which points to the influence and importance of the choice of benchmark problem when developing an algorithm.

Note that the modified MCTS agent still does not play better than the A* agent. The most likely reason for this is that it is not possible to play that set of levels better. As stated above, the main objective of the paper is not to develop a better-playing Mario agent, but rather to understand and improve the behaviour of the very promising MCTS algorithm on continuous-state continuous-time videogames without mandatory progress, a large category of games of which Mario is a good example. For clarity, it should be pointed out that a best-first search algorithm like A* would not perform well on playing any game which did not have a very fine-grained heuristic function or which was adversarial.

Still, it would be interesting to see if there was some version of the Mario AI benchmark where our improved MCTS algorithms would outperform A*. The stochastic nature of MCTS would seem to make it more adept at handling unpredictability, not only due to an opposing player but also due to the environment. A study of the paths the A* agent reveals that they depend on the environment being absolute predictable: Mario, as controlled by A*, jumps at the very last pixel before falling into a gap, and passes enemies with absolutely no margin. Such solutions would be very brittle in the face of any unreliability in the agent-environment coupling. We therefore undertook to investigate the effects of introducing such unreliability.

8. SECOND TEST: NOISY MARIO

In order to test our implementationâĂŹs rigidity against noise, we created a test scenario where 20% of an agentâĂŹs moves would be random regardless of what it actually wanted to do. This was implemented as a wrapper around the agent code, where in every interaction cycle there was a 0.2 chance that the command from the agent would be replaced with a random command. With these settings our best combination (all enhancements except macro actions) was benchmarked against the A* implementation on the same 100 levels as previously and the results are compared. During testing the A* implementation crashed occasionally in some levels, these two levels have been removed from both samples for fairness yielding a basis of 98 levels.

The results from the noisy action test reveals that our MCTS implementation performs significantly better than A* under these conditions. MCTS cleared three levels while A* died in all. Across the 98 levels MCTS achieved a mean of 1770 points while A* reached 1342 (see table 6). This difference suggests that the MCTS implementation is coping much better with the uncertainty under these conditions than A*. Both controllers performed much worse than either controller did in the noiseless case, which is because the noisy version of the problem is much more challenging.

AI	**Mean Score**
MCTS (X-PRHL)	1770
A* agent	1342**

Table 6: Results of evaluating the MCTS and A* agents with a 20% chance of a random action being performed instead of the selected one. The stars denote the statistical significance in difference from the selected variant of MCTS (X-PRHL)

It is interesting to note that the characteristics of the noisy version of the Mario AI Benchmark are almost diametrically opposed to those of the game which MCTS has had most pronounced success on, namely Go. Whereas Go has a high branching factor, relatively short average and maximum

game length and is deterministic, the Mario AI Benchmark has very low branching factor, long average game length and very long maximum game length (if you do nothing, you have to wait tens of thousands of ticks to time out). Further, the noisy version of the benchmark is nondeterministic. The relative success of MCTS under these conditions points to that the core algorithm, with suitably chosen extensions, can provide good performance in conditions which are very different from those which it was originally designed for and tested on. It should be investigated further and more systematically which kinds of games MCTS can work well on.

9. CONCLUSION

In this paper, we investigated the performance and behaviour of MCTS on the Mario AI benchmark, and found that standard MCTS performed relatively badly. Based on the problems we identified with MCTS' behaviour on this benchmark, we then proposed a number of modifications, and tested all combinations of those. It was found that a judicious combination of these modifications yields an agent that plays near optimally. Two of the modifications are novel in the sense that we invented them and have not found them in the literature: mixmax backups and partial expansion. We then produced a noisy version of the Mario AI benchmark, and tested the algorithms on this benchmark. It was found that the improved MCTS controller clearly outperforms A* on this version of the benchmark. We hope to have helped clarify the role MCTS can play in video games, and suggested a few useful techniques for conquering continuous time and space.

10. REFERENCES

[1] S. Bojarski and C. Congdon. Realm: A rule-based evolutionary computation agent that learns to play mario. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium*, pages 83–90, 2010.

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

[3] G. M. J. Chaslot, M. H. Winands, H. J. V. D. HERIK, J. W. Uiterwijk, and B. Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008.

[4] D. Churchill, A. Saffidine, and M. Buro. Fast heuristic search for rts game combat scenarios. *Proceedings of AIIDE*, 2012.

[5] M. Ebner, J. Levine, S. Lucas, T. Schaul, T. Thompson, and J. Togelius. Towards a video game description language. *Dagstuhl Follow-up. To appear., Preprint available at http://www. idsia. ch/˜ tom/publications/dagstuhl-vgdl. pdf*, 2013.

[6] M. Genesereth, N. Love, and B. Pell. General game playing: Overview of the aaai competition. *AI magazine*, 26(2):62, 2005.

[7] S. Karakovskiy and J. Togelius. The mario ai benchmark and competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, 2012.

[8] C.-S. Lee, M.-H. Wang, G. Chaslot, J.-B. Hoock, A. Rimmel, O. Teytaud, S.-R. Tsai, S.-C. Hsu, and T.-P. Hong. The computational intelligence of mogo revealed in taiwan's computer go tournaments. *Computational Intelligence and AI in Games, IEEE Transactions on*, 1(1):73–89, 2009.

[9] T. Pepels and M. H. Winands. Enhancements for monte-carlo tree search in ms pac-man. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 265–272. IEEE, 2012.

[10] D. Perez, P. Rohlfshagen, and S. M. Lucas. Monte-carlo tree search for the physical travelling salesman problem. In *Applications of Evolutionary Computation*, pages 255–264. Springer, 2012.

[11] E. Powley, D. Whitehouse, and P. Cowling. Monte carlo tree search with macro-actions and heuristic route planning for the physical travelling salesman problem. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 234–241, 2012.

[12] S. Samothrakis, D. Robles, and S. Lucas. Fast approximate max-n monte carlo tree search for ms pac-man. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(2):142–154, 2011.

[13] F. W. Takes and W. A. Kosters. Solving samegame and its chessboard variant. In *Proceedings of the 21st Benelux Conference on Artificial Intelligence (BNAIC?09)(eds. T. Calders, K. Tuyls, and M. Pechenizkiy)*, pages 249–256, 2009.

[14] J. Togelius, S. Karakovskiy, and R. Baumgarten. The 2009 mario ai competition. In *Proceedings of the IEEE Congress on Evolutionary Computation*, 2010.