Monte Carlo Search for a real-time arcade game, *Puyo-Puyo*

Paul Hanson and **David C. Moffat**[1]

Abstract.

Monte Carlo Tree Search (MCTS) and other Monte Carlo Search (MCS) algorithms have been successful in making computers able to play Go and similar games. This paper reports an attempt to apply MCS to an arcade game *Puyo-Puyo* which requires good real-time performance. The game *Puyo-Puyo* is a falling blocks game similar to *Tetris*, and has a large branching factor that makes it less tractable for more traditional techniques used for AI game players. In this experiment a simple Monte Carlo algorithm was implemented to play *Puyo-Puyo*, and its performance evaluated against Depth First and Breadth First search algorithms (BFS and DFS), for realistic and optimal comparisons.

Results show that the MCS algorithm had better performance, and it could play the game in real-time, about as well as a fair human player. This was achieved as long as its simulation depth was limited appropriately.

It is thus demonstrated that real-time AI decisions can be made without heuristic knowledge, and that MCS has potentially useful application to a wider range of games.

Through investigation of the varying depth limit to the MCS algorithm, it appears that such algorithms need to be tuned to the game, at least for some kinds of game or application, before determining whether they can be successful.

1 Motivation

Artificial Intelligence (AI) plays a key role in many video games. Games and AI have had a long standing interaction; from the early days of gaming and titles such as PacMan and SpaceInvaders, right up to modern day releases such as *Half Life*, the *Total War* series, *The Sims*; and other first person shooters, real-time strategy games, and social simulation games.

Some of the more common AI techniques and algorithms used include A* search for path finding, finite state machines for simple decision making, and various adaptations of Reynolds' ([9]) steering behaviour for believably realistic autonomous character movement. These simpler techniques work more on the illusion of intelligence rather than the demonstration of categorically intelligent behaviours ([8]). So far, this has been an adequate approach for the majority of games, but as games become more complex and players' expectations increase, so must the techniques used in AI evolve to meet the higher demands.

[1] Department of Computing, Glasgow Caledonian University, UK. email:D.C.Moffat@gcu.ac.uk

1.1 MCS and MCTS for game AI

In recent years there has been a substantial development in ComputerGo AI using Monte Carlo search algorithms. A number of successful programs based on this technique have been developed including *Gobble*, *Crazystone* and *Fuego*. In general, a Monte Carlo Search (MCS) algorithm determines the best move for any given situation by simulating a series of random moves. By using the results gathered from playing multiple simulated games, a Monte Carlo algorithm can make move decisions using very little given domain knowledge [4].

The success of Monte Carlo techniques has lead to variations being used in other games. They were implemented in Battleships to determine the optimum position to place ships [5], and even used as a General Game Player (GGP) agent in CadiaPlayer to determine which actions to take [1].

Because Monte Carlo Tree Search (MCTS), in particular, has naturally been tried first as a replacement for alpha-beta search, it has been used in two-player perfect information games. Such games (like Chess and Go) have two players taking turns to move, with no moves played by any simulated world (or "nature"). Most video games do simulate a virtual world, however, and require players to react in real time. This raises the question whether Monte Carlo methods can be used when only a finite amount of time is available to the algorithm.

Falling block puzzle video games such as *Tetris* appear to be ideal candidates to test the use of MCTS in video games. for a number of reasons. Firstly, titles which support simultaneous play between two opposing players will allow Monte Carlo to be tested against an opponent in a non- sequential manner. The game *Puyo-Puyo*, chosen for our experiment, is one such game, although we initially try it in a single-player mode.

Secondly, The theoretical number of moves in a game like Tetris is infinite, which will allow for the exploitation of the exploration of random moves by a Monte Carlo method. Thirdly, each move has a time constraint which is dictated by the drop rate of the block, providing a constraint of real time gameplay that is not related to frames per second.

Since AI has been relatively successful in Tetris [7] [2], another popular falling block puzzle game called *Puyo-Puyo* has been chosen as the test game. The gameplay of *Puyo-Puyo* contains the necessary requirements previously identified for this experiment: a large game tree with many possible placement options per move, a relatively large but constrained period of time for each move and two player simultaneous gameplay. Unlike Tetris however, PuyoPuyo does not have a recognised algorithm for making the best move and may be suited, in that sense, to the particular strengths of MCS. The MCS family of algorithms may represent one of the few chances we have to play such games as *Puyo-Puyo*, which is therefore a good candi-

date game for MCS research.

2 The game *Puyo-Puyo*

PuyoPuyo is played at the same time by two opposing players, where each player has a game board of 6x13 spaces. Pairs of different coloured puyo are dropped onto each player's board, which must then be positioned to avoid filling up the game board. The player who fills up their entire game board first, loses [10]. Spaces on the board can be cleared by popping four or more puyo of the same colour, which in turn sends garbage puyo to the opponent's field. These garbage puyo help to fill the opponent's board faster and can only be cleared by popping nearby puyo. The number of garbage puyo sent over to the opponent's field is dependant on the number of puyos popped as well as the number of chains in the sequence. A chain starts when four or more puyos are popped and stops when the next block falls down.

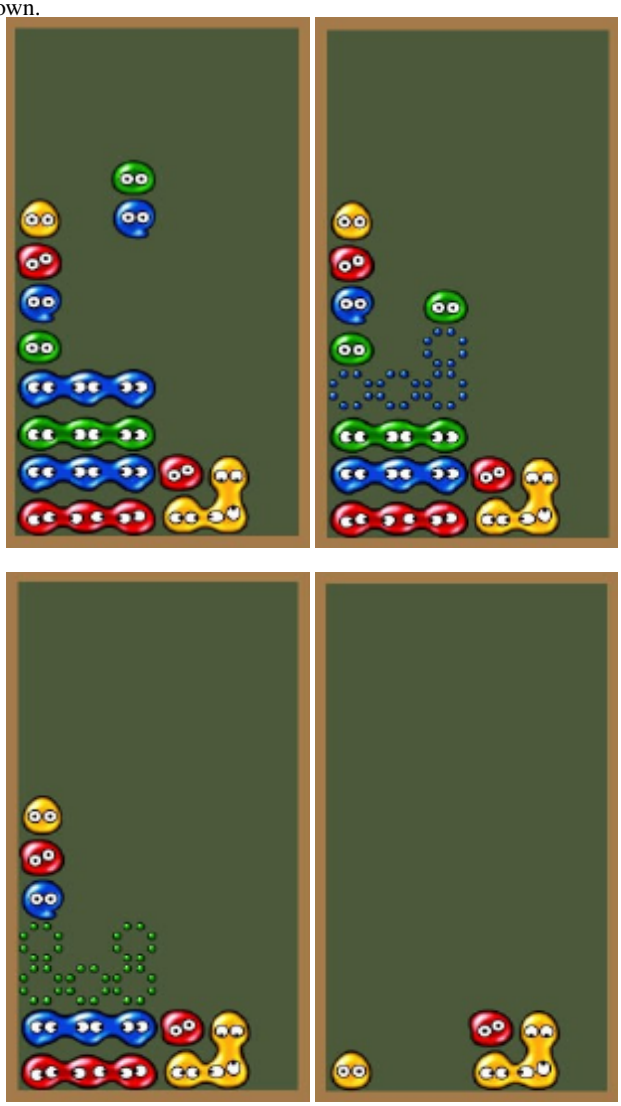

Figure 1. Screenshots of *PuyoPuyo*. A green/blue pair falls, to make a chain of four blues, which clear away; then the five greens go; then the next four blues; and finally four reds.

In the example a blue and green puyo pair is dropped. The blue one completes a chain, which then disappears to let the higher puyos fall. A green chain is thereby completed, which also disappears. This causes a blue chain and then a red chain to go as well, leaving only five puyos at the end of the move. Because four chains have gone in a single move, it has got a high score, and created more garbage puyo for the opponent's board (not shown here). To build up the potential for such "cascades" of chains is thus the strategic aim of the game.

While the strategic aim may be clear, it is unclear how to achieve it. In other games like *Tetris*, one can postulate general rules or heuristics about good ways to play the game. Players soon learn to fit blocks in patterns and wait or hope for blocks to fall that exactly fit into the gaps, so that they can clear lines. But in *PuyoPuyo* there is not such an obvious strategic method. We can fall back on brute search methods, which are not guided by expert heuristics, but then we should make the algorithms as efficient as we can because the search space is enormous. It is for these reasons that MC algorithms are interesting to investigate, and hence this study.

3 Monte Carlo Search (MCS) for *PuyoPuyo*

Monte Carlo (MC) simulations are predominantly used as a substitute to evaluation functions in game AI — the advantage being that a value in a search tree can be determined without the need for complex, time consuming expert knowledge based functions. In order to capitalize on the advantages of MC simulations and to improve the search capabilities of it, the concept of MC evolved into what is known as the Monte Carlo Tree Search (MCTS).

3.1 Monte Carlo Tree Search (MCTS)

The MCTS is a form of best-first search which uses the results of multiple MC simulations to determine which path to take [6] This form of search was used by Bouzy [3] who combined a standard MC search, with a shallow and selective global tree search. His ComputerGo programs Indigo and Olga used MCTS and were tested against the expert-knowledge based program GNUGo 3.2. It was found that increasing the depth and width of the search tree yielded better results, however, as the depth of the tree increased, the time it took to perform the simulations also increased [3].

While MCTS has been very successful in playing some sorts of game and is currently an active research area, in this study we investigate the possible value of a simpler MC algorithm, because it may help to see which components of more complex algorithms are most fruitful. It may also help to set a lower bound on what future improvements might be made.

3.2 A simple MCS algorithm to play *PuyoPuyo*

As the game *PuyoPuyo* has a large branching factor (of 22 children for each node that represents a drop, or move played), it becomes computationally difficult to exhaustively search the tree more than a few layers deep. In this study the MCS algorithm only searches the first layer exhaustively, by visiting each of the 22 children in turn, in simple round-robin fashion. None of the nodes are favoured either; in contrast to what happens with many MCTS algorithms, where more attractive looking nodes are visited more often under the assumption that the best solutions may be found under those ones. These two simplifications of the more complex and advance MCTS family of algorithms make this MCS algorithm much shorter to implement in code.

Below the first layer, the MCS algorithm switches into random search mode in which it simulates further plays of the game, by visiting and expanding only one child at random in each layer. However, it is not possible in *PuyoPuyo* to complete each so-called *playout* to the end of the game, until a terminal node is reached with a win-lose result. This is because the game can only terminate in a loss for the player, whose aim is to survive as long as possible, just as in other falling-block games like *Tetris*; and that loss should be far into the future, and well beyond the depth that could guarantee to be reasonably searched in a fixed time limit.

In this implementation of MCS therefore, we impose a depth limit to the search and take the increment in score when the limit is reached as our indication of how good the partial playout (the simulated random play to that depth) was. This score is then used to update the estimated value of the initial move at the first layer that preceded the random playout stage. Each of the 22 first-level nodes (chilredn of the root node) is annotated with its own "best so far" score, from all the playouts that have yet originated from it. In this way the algorithm may be interrupted at any time, and the best of all those first-level nodes can be selected as the best move to play. As the game is to be played in real-time, the MCS algorithm is allowed to run for as long as it can, until the blocks have fallen so far that the AI player must now commit itself to one of the 22 possibile moves, and play it.

There remains the matter of how deep we should allow the algorithm to go in its playout simulations, This is the second of the questions we had about how well the algorithm could perform: what would be the optimal search-depth limit for the MCS, and how would its performance turn out at either higher or lower limits?

4 Evaluation scheme

The performance of the MCS algorithm was compared with that of DFS as a commonly used search algorithm that has minimal memory requirements and makes no use of expert or heuristic knowledge. Performance of BFS was also compared, both in real-time and then once with unlimited time. The real-time limit was imposed to make a fair comparison with the other algorithms under realistic gameplay conditions. The unlimited time execution was done by allowing the BFS to finish exploring the whole tree exhaustively to the given depth limit. By this means the theoretical optimum move is found for the tree at that depth, and the other algorithms can be compared with this best possible performance. Finally there was a random search algorithm, in which one of the children of each visited node was chosen at random to explore further. This

All the algorithms were tried for depth-limits of 1, 2 and 3 layers deep, corresponding to the *PuyoPuyo* game situation in which players can see the current block to fall, as well as the next two ones that will follow it. For each depth-limit, ten different games were played (with ten different starting positions). The results of all ten games were averaged for each of the algorithms and depth-limits, so that a fair comparison could be drawn between them, and any differences would not be attributable to any effect of the random starting position. Each game was played to a maximum length of 150 moves, or blocks to be dropped. This is because the games could otherwise go on for an arbitrarily long time, which would make the experiment and unpredictable duration. The 150 drop limit was the maximum, but not all the games went on to that limit. In particular, in the weaker or underperforming cases, the games could terminate well short of 150 drops, as the AI player would sometimes lose quickly when its algorithm could not cope with the complexity under the real-time constraint.

Finally, the MCS algorithm was explored further by allowing it to search deeper in the tree, to depth-limits of 4, 5 and 6. This was to investigate when and how its performance would degrade under real-time demands, as the tree deepend. Although players can normally only see the current block and the next two (equivalent to three layers below the root node), the MCS algorithm was allowed to see all the next coming moves down to the sixth layer. If its performance can manage to profit from that extra information, then we may expect that it should be able to achieve remarkably high scores, as any human player would find it very difficult to plan so many drops ahead, especially under time pressure, and with the limitations on human working memory. It was no doubt because of these natural human cognitive limitations that the game *PuyoPuyo* shows only three moves ahead, including the current block.

5 Results

The exhaustive search sets the maximum score achievable to the other algorithms, to give a fair impression of how good they are.

Figure 2 shows the scores that it can reach as the game continues, up to the maximum of 150 drops (moves). At every ten drops, the lines show the range from lowest to highest scores over all the ten games that are played; and the dots in the middle show the mean (average) over all ten games. The best played game attains a score of more than 100,000 but the average score at the end of all ten games is also high at nearly 70,000.

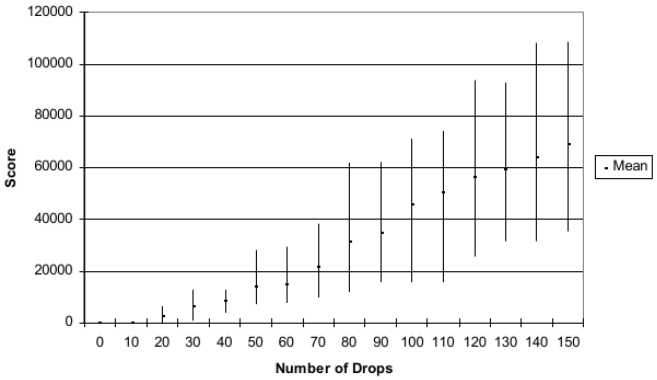

Figure 2. Exhaustive search score range

The scores for BFS (in Figure 3) are also steadily increasing, and the algorithm performs quite well, but only scores about half of the optimal scores possible, because it runs out of time.

The performance of all the algorithms is shown, in averages only, in Figure 4. Here we can see that all the algorithms perform about equally well, except for DFS which fails miserably, with its average scores never getting far above zero, and dropping as the games continue. The reason its scores fall for higher numbers of drops is because it does so poorly that it loses many of the games well before the 150 drop limit; as those games contribute nothing to the score the average goes down.

However the other three algorithms perform similarly, and it may come as a surprise to see how close they are. Nevertheless, there is a lot of room for improvement as the exhaustive search, which is not time-limited, eventually achieves scores significantly higher. Another way to put this is to say that the real-time interruptable algorithms (BFS, Random search, and MCS) all start strongly, and can plan their moves very well; but as the game progresses their small

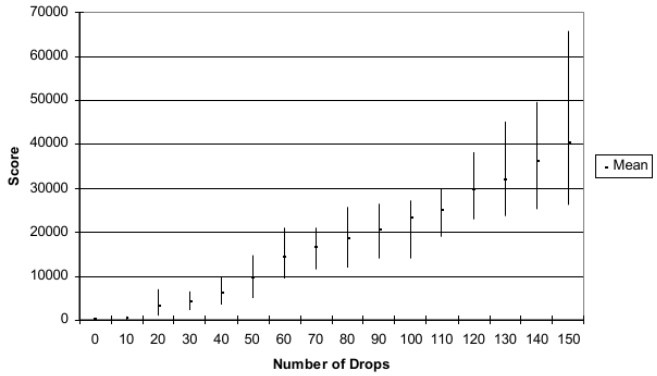

Figure 3. BFS score range

mistakes (or sub-optimal moves) begin to accumulate. Rather like *Tetris*, the game changes character as it goes on, from scoring well by clearing blocks, to a struggle to survive as the board fills up.

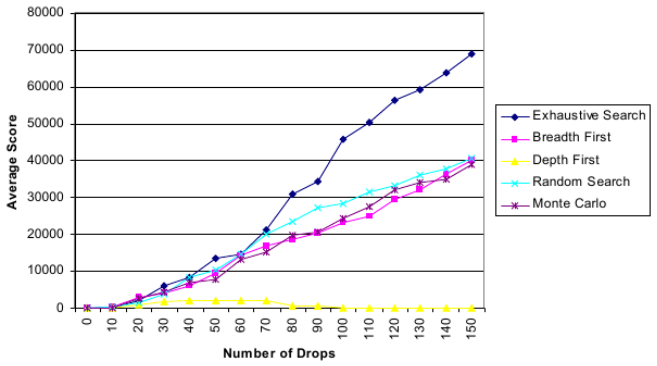

Figure 4. Scores for all algorithms

Having compared the different algorithms, we now move to our second question, regarding how well the performance of the MCS holds up as it is set to search deeper in the tree. Of course if it can manage to search deeper then its scores should increase, but it might not be able to cope, and consequently degrade in performance.

We see from Figure 5 that the MCS does much better in this game if it searches to depth 4, ending with an average score of more than 50,000; however that is still not as high as the exhaustive search can achieve at only depth 3.

It would have been interesting to compare the exhaustive and the BFS (or Random search) algorithms at this size of tree as well, but unfortunately this was not easy to test because of the way the search algorithms were implemented. (The BFS algorithm is hard-coded to search the tree to depth 3 in order to pre-allocate memory.)

However, the MCS has no such coding limit, and so we investigate how its performance fares at deeper levels, in Figure 6. It is clear that depth 4 is optimal for the MCS in this case. Searching down to depth 5 results in a final score that is actually lower than that achieved at depth 3. In fact of the ten games, this version only manages to stay the distance to 150 drops on half of them. The search to depth 6 is even weaker still, with the game ending prematurely in most of the games (8 of the 10). The search to depth 4 does play all games to the limit of 150 drops, however, showing that it always has enough time to at least keep the board fairly clear of blocks, even if not to reach a top score.

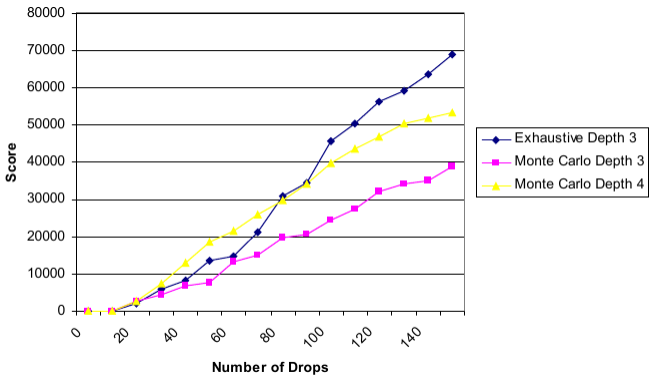

Figure 5. Scores for MCS at depth 4, better than depth 3

The reason that deeper searches yield poorer performance is that searching the lower layers takes time away from the higher layers. This means that, although the estimates arrived at on some of the first-level nodes are more accurate (because playout out for longer), the estimates are on the whole underexplored, with first-level nodes only holding data from a small number of simulated playouts. The estimates are therefore less representative and reliable, and the best moves are easily missed.

We conclude that it is crucial in real-time applications to test the MC family algorithms at multiple levels, to find the best trade-off between searching deep and searching often. Arriving at such an optimum by mathematical analysis alone would be challenging, so tuning the algorithm by trial and error is the way to go.

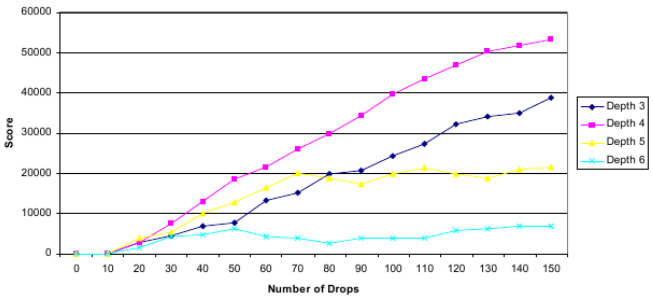

Figure 6. Scores for MCS at lower depths — algorithm fails below level 4

6 Conclusion

A basic MCS algorithm was applied to a falling-blocks puzzle game that challenges other AI techniques because it has a large search-space, no clear strategy that can be expressed heuristically, and no winning end state. The algorithm combined a uniform round-robin search, of the top layer of the tree, with a Monte Carlo search of child nodes down to a depth limit. It accumulated the leaf node scores and kept the best ones to annotate the first-level nodes, so that the algorithm could be interrupted at any time to select the best initial move found so far.

Other standard search algorithms were compared, for the same depth of tree down to the third layer, including an exhaustive search that set the upper bound on what the best possible algorithm could achieve. All the algorithms did fairly well except for DFS, which was unable to cope with this game in real-time conditions, due to its inherent bias in searching child nodes in its own arbitrary order.

Even the algorithms that did well showed that they paid a large cost however, due to the time-pressure of the game.

While the MCS did not outperform BFS in that test, it did however at least perform as well, showing that the MC approach, although in a very simple form here, is an able alternative to other possible search algorithms. With further research it should be possible to find specific improvements to the MCS that can enable it to outperform the other algorithms here.

A second question was how the MCS would perform at deeper search limits. Searching down to depth 4 allowed it to attain a much higher score, and it was still able to keep the games going for as long as 150 moves, which was the maximum allowed in our experiment. However deepening the search to limits of 5 and 6 gave much worse performance than the previous trials at depth limits 4 and even 3. The search a depth limit 6 was practically useless, not even managing to finish most of the games. We conclude that it is necessary to tune MC algorithms to the particular game that they are intended to play. Otherwise it is possible to be mislead into thinking that the method is not appropriate, only because it has not been given a fair trial with different search parameters like depth, in order to find its "sweet spot" where it can perform well.

Monte Carlo methods in general, therefore, should probably be experimented with for any game or application, before judging whether they are suited or not. In the future however, it would be nice to see more analytical developments that would give us a better idea of how successful MC algorithms might be in some application, without our having to fall back all the time on trial and error.

REFERENCES

[1] Yngvi Björnsson and Hilmar Finnsson, 'Cadiaplayer: A simulation-based general game player.', *IEEE Trans. Comput. Intellig. and AI in Games*, **1**(1), 4–15, (2009).

[2] Niko Böhm, Gabriella Kókai, and Stefan Mandl, 'An evolutionary approach to tetris', in *6th Metaheuristics International Conference (6th Metaheuristics International Conference Wien August 22-26, 2005)*, (2005).

[3] Bruno Bouzy, 'Associating Shallow and Global Tree Search with Monte Carlo for 9x9 Go', in *Proc. Int. Conf. Comput. and Games, LNCS 3846*, pp. 67–80, Ramat-Gan, Israel, (2004).

[4] Bruno Bouzy and Bernard Helmstetter, 'Monte-carlo go developments', in *Advances in computer games*, 159–174, Springer, (2004).

[5] Jeremy G Bridon, Zachary A Correll, Craig R Dubler, and Zachary K Gotsch, 'An artificially intelligent battleship player utilizing adaptive firing and placement strategies', *The Pennsylvania State University, State College, PA*, **16802**, (2009).

[6] Guillaume Maurice Jean-Bernard Chaslot, Mark H. M. Winands, H. Jaap van den Herik, Jos W. H. M. Uiterwijk, and Bruno Bouzy, 'Progressive Strategies for Monte-Carlo Tree Search', *New Math. Nat. Comput.*, **4**(3), 343–357, (2008).

[7] Colin Fahey. Tetris, 2003.

[8] Alexander Nareyek, 'Ai in computer games', *Queue*, **1**(10), 58–65, (February 2004).

[9] Craig W Reynolds, 'Steering behaviors for autonomous characters', in *Game developers conference*, volume 1999, pp. 763–782, (1999).

[10] Sonic Team. Puyopuyo, 2009.