Proceedings, The Thirteenth AAAI Conference on
Artificial Intelligence and Interactive Digital Entertainment (AIIDE-17)

Memory Bounded Monte Carlo Tree Search

Edward J. Powley
The MetaMakers Institute
Games Academy
Falmouth University, UK

Peter I. Cowling
Digital Creativity Labs
University of York, UK

Daniel Whitehouse
Department of Computer Science
University of York, UK

Abstract

Monte Carlo Tree Search (MCTS) is an effective decision making algorithm that often works well without domain knowledge, finding an increasing application in commercial mobile and video games. A promising application of MCTS is creating AI opponents for board and card games, where *Information Set* MCTS (*IS*MCTS) can provide a challenging opponent and reduces the cost of creating game-specific AI opponents. Most research to date has aimed at improving the quality of decision making by (IS)MCTS, with respect to time usage. Memory usage is also an important constraint in commercial applications, particularly on mobile platforms or when there are many AI agents. This paper presents the first systematic study of memory bounding techniques for (IS)MCTS. (IS)MCTS is well known to be an *anytime* algorithm. We also introduce an *anyspace* version of (IS)MCTS which can make effective use of any pre-specified amount of memory. This algorithm has been implemented in a commercial version of the card game Spades downloaded more than 6 million times. We find that for games of imperfect information high quality decisions can be made with rather small memory footprints, making (IS)MCTS an even more attractive algorithm for commercial game implementations.

Introduction

Monte Carlo Tree Search (MCTS) is a family of decision tree search algorithms that have produced breakthroughs in AI for several domains (Browne et al. 2012), notably Computer Go (Lee, Müller, and Teytaud 2010; Silver et al. 2016) and General Game Playing (Björnsson and Finnsson 2009; Perez et al. 2015). MCTS was discovered in 2006 (Coulom 2007), building on previous work on Monte Carlo techniques for Go (Chaslot et al. 2006). Most current implementations of MCTS are based on the UCT algorithm (Kocsis and Szepesvári 2006). MCTS algorithms build an asymmetric partial search tree, evaluating nonterminal nodes by random playouts and preferring to expand regions of the tree that are either promising or unexplored. By treating the tree as a hierarchy of multi-armed bandits, MCTS strikes a balance between exploiting known good regions of the tree and exploring untried regions. MCTS often works well without heuristic evaluation functions or any other domain-specific expert knowledge, but can be enhanced if such knowledge is available. MCTS is an *anytime* algorithm: it can make effective use of any number of iterations, and running for longer generally yields better decisions.

MCTS is used in commercial games such as the mobile card game Spades by AI Factory (Whitehouse et al. 2013), the Total War series of strategy games by Creative Assembly (Champandard 2014), and the role-playing game Fable Legends by Lionhead (Mountain 2015). MCTS can make developing AI easier than more traditional knowledge-based methods such as heuristically-guided search, and the generality of MCTS allows for the development of reusable AI modules. MCTS shows particular promise in situations where authoring of heuristics or rules for techniques such as A* search and behaviour trees is time-intensive, error-prone and gives unsatisfactory results.

In commercial games, particularly those developed for mobile devices, the AI must adhere to a limited memory budget. Even on a modern games console with gigabytes of memory, the memory budget for the AI code may be relatively small compared to other game subsystems such as graphics, audio and physics simulation. There are also significant performance benefits to fitting the AI's data structures within the cache memory of a single processing unit (Fauerby 2012). On mobile devices, conserving memory helps to maintain battery life and multitasking performance, and ensures the game works well across a broad spectrum of devices. Avoiding dynamic memory allocation also allows programs to be written in such a way that memory leaks are impossible, and the memory usage of the program is static and thus known in advance.

The anytime property of MCTS allows the decision time of the algorithm to be constrained whilst ensuring all available time for decision making is fully utilised. However, performing more MCTS iterations causes more memory to be used by the algorithm, since a new node is usually added on each iteration. As the number of MCTS iterations increases, the memory usage of the algorithm is bounded only by the (combinatorially large) size of the game tree. Several methods have been proposed to limit the memory used by MCTS, however most works treat this as an implementation detail, e.g. (Coulom 2007; Enzenberger et al. 2010). This paper is the first systematic study of memory bounding techniques for MCTS, and introduces a new method which

significantly outperforms existing methods in terms of playing strength, making MCTS an *anyspace* algorithm. In addition we outline a practical implementation with very little computational overhead compared to the standard (memory unbounded) MCTS algorithm.

The most commonly used memory bounding technique for search-based AI methods is depth limiting: cut off the search tree at some constant depth, chosen to be shallow enough that the entire game tree down to that depth will fit into memory. Closely related is the idea of iterative deepening search (Korf 1985), which can be applied to depth-first, minimax and A* search among others. However depth limiting is a blunt instrument for bounding memory usage: the difference in memory between depth d and depth $d+1$ may be large, and grows exponentially larger as d increases. Also, iterative deepening methods often do not make the most efficient use of a given time or memory budget: if the algorithm is halted when the budget is exhausted, and this happens midway through the search at a particular depth, all effort spent so far at that depth may be wasted. The new memory bounding method presented in this paper limits the number of nodes in the tree rather than the depth, allowing much finer grained control over memory usage. Thus it has some similarity to methods such as SMA* (Russell 1992) and replacement schemes for transposition tables (Breuker, Uiterwijk, and van den Herik 1994): the memory bound is expressed as a maximum number of objects that can be stored at once, and this bound is enforced by removing "stale" objects to make way for "fresh" ones.

In this paper, we apply memory bounding to the ISMCTS algorithm (Cowling, Powley, and Whitehouse 2012) for decision making in games with hidden information and uncertainty (commercially a much larger sector than perfect information games). For such games we observe that reducing the memory budget has comparatively little impact on the playing strength of the algorithm. Reducing the memory budget causes MCTS to store statistics for fewer future states. This suggests that when there is hidden information and randomness, analysing decisions in the distant future is less beneficial than for games with perfect information, so long as we collect sufficient information about states higher in the decision tree.

Background

Monte Carlo Tree Search (MCTS)

This section outlines the basic operation of MCTS, introducing the notation and terminology we use when discussing modifications to the base algorithm. MCTS iteratively builds a search tree, where one new node is added to the tree on each iteration. In addition to its position within the tree, each node v has four pieces of information associated with it: $s(v)$, the game state corresponding to the node; $a(v)$, the action that immediately preceded state $s(v)$; $Q(v)$, the total reward for all playouts that have passed through v; and $N(v)$, the number of playouts that have passed through v. Each MCTS iteration peforms a *playout* from the current position to a terminal state, with each playout consisting of a *selection* phase and a *simulation* phase. The selection phase corresponds to positions visited which are in the MCTS search tree and actions are selected according to the *tree policy*. The tree policy used in the UCT algorithm is the UCB1 bandit algorithm (Auer, Cesa-Bianchi, and Fischer 2002). If the current node is v, then the tree policy selects a child v' of v, and corresponding action $a(v')$, maximizing $\frac{Q(v')}{N(v')} + K\sqrt{\frac{\ln N(v)}{N(v')}}$ where K is a constant tuned to balance exploration with exploitation. In this paper we use $K = 0.7$, since $K \approx \frac{\sqrt{2}}{2}$ is a known good default setting when the terminal state rewards are in the range $[0, 1]$ (Kocsis and Szepesvári 2006). Ties in the choice of v' are resolved uniformly at random. When a state reached by the tree policy does not have a corresponding node in the MCTS tree, a new node is *expanded* (added to the tree) and the rest of the game is played out according to the *default policy* (called the *simulation policy* in some literature). The default policy for the UCT algorithm is to select actions uniformly at random. The final step is to *back-propagate* the result through the tree, updating the Q and N values of the newly expanded node and its ancestors up to the root node.

Information Set MCTS

Information Set MCTS (ISMCTS) is a modification of MCTS which enables MCTS to be applied to games of *imperfect information*, specifically games with *hidden information* or *partially observable moves* (Cowling, Powley, and Whitehouse 2012; Whitehouse, Powley, and Cowling 2011). In such a game, the current state is generally not known. On each iteration, ISMCTS samples a *determinization*: a state which could possibly be the current state, given the information available to the player. This is analogous to "guessing" information. For example a determinization in a card game is obtained by guessing the hidden cards in the hands of other players. The determinization is then used in place of the current game state for the current MCTS iteration. Subsequent iterations sample different determinizations but still operate on the same tree, thus the tree collects statistics relevant to all sampled determinizations. The above algorithm is referred to as *Single-Observer ISMCTS (SO-ISMCTS)*.

In some games, moves are *partially observable*. For example an opponent might choose a card and play it face down; the player can observe that a card was played, but not which card it is. ISMCTS can be extended to deal with such situations. *Multi-Observer ISMCTS (MO-ISMCTS)* works similarly to ISMCTS, but constructs a separate tree for each player. Each iteration descends all trees in parallel. To choose a move for player i, the tree policy uses player i's tree. Player i's tree has a branch for each available action, whereas player $j \neq i$'s tree merges the branches for moves which are indistinguishable. The extra trees mean that MO-ISMCTS uses more memory than SO-ISMCTS, but this is necessary to correctly model the situation in some games.

In the remainder of this paper, we write *(IS)MCTS* in situations where the discussion holds true both for standard MCTS and for ISMCTS.

(IS)MCTS with memory bounds

Since the MCTS and SO-ISMCTS algorithms create a new node on each iteration, the memory usage increases linearly as more iterations are executed. MO-ISMCTS creates a new node for each player at each iteration, still giving a linear increase of memory use with iteration count. In this section we describe several ways of modifying MCTS and ISMCTS to work within a fixed memory bound: some simple, some from the literature, and a new method. Throughout this section when we refer to MCTS we mean all of MCTS, SO-ISMCTS and MO-ISMCTS (with suitable minor modifications for the multiple trees of MO-ISCTS).

Stopping. The simplest way to limit the memory used by MCTS is simply to halt the algorithm once the memory limit is reached, even if time budget remains. If each iteration adds one node to the tree, then a memory budget of n nodes means that at most n MCTS iterations will be executed. If the time budget allows for many more than n iterations then this is wasteful, but we include it as a baseline.

Stunting. A slightly more advanced strategy than stopping is to continue searching once the memory limit is reached, but halt the growth of the tree at this point. Once memory is full, the selection, simulation and backpropagation steps continue to be executed, but the expansion step is skipped. Thus extra iterations are performed, but serve only to refine the bandit statistics in the tree that has already been built.

Ensemble MCTS (Fern and Lewis 2011) works by executing several independent instances of MCTS from the same root state, and combining the statistics on moves from the root in order to make the final decision. In implementations where the independent searches execute concurrently, this is known as *root parallelisation* (Cazenave and Jouandeau 2007; Chaslot, Winands, and van den Herik 2008), which has also been applied to ISMCTS (Sephton et al. 2014). The idea is similar to *determinized MCTS* for games of imperfect information, where states are sampled from the current information set and searched independently (Bjarnason, Fern, and Tadepalli 2009; Powley, Whitehouse, and Cowling 2011; Cowling, Ward, and Powley 2012)

Ensemble MCTS can also be seen as a memory bounding method if the MCTS instances are executed sequentially rather than concurrently. As with stopping, we execute MCTS until it exhausts the memory budget. At this point we discard the MCTS tree and commence an entirely new MCTS search from scratch, but first store the visit counts of all moves from the root in a separate data structure. Repeat this process until the time budget is reached, accumulating the visit counts for root actions. Finally choose a move from the root state to maximise the sum of recorded visit counts. This is algorithmically equivalent to the ensemble MCTS of Fern and Lewis (Fern and Lewis 2011) when the node budget divides exactly the number of iterations. However unlike the concurrent implementations of root parallelisation, in our implementation only one MCTS tree is stored in memory at once so as not to exceed the memory bound.

Fern and Lewis (Fern and Lewis 2011) study the "single-core memory advantage" of ensemble MCTS: in the terminology of this paper, this is the benefit of ensemble MCTS versus stopping for a fixed memory budget. Ensemble MCTS is found to significantly outperform stopping across all tested domains.

Tree flattening. Every time it discards the tree, ensemble MCTS must relearn the values of moves from the root in order to determine which to exploit and which to ignore. Instead of discarding the entire tree, *tree flattening* keeps the first ply of the tree and discards the rest. The reward and visit information for moves from the root is retained, allowing the next MCTS instance to continue exploiting and exploring according to statistics already gathered. Thus tree flattening is to ensemble MCTS as HOP-UCT is to determinized MCTS (Bjarnason, Fern, and Tadepalli 2009). Unlike ensemble MCTS there is no need to record visit counts outside the tree, as they are retained in the flattened tree.

Node recycling. Ensemble MCTS and tree flattening both have the weakness that they repeatedly throw away most or all of the policy information learned by MCTS before the memory bound was reached. Here we introduce a new method, *node recycling*, which aims to throw away only the information that is least relevant to the search, retaining as much useful information as possible in the remaining tree. The idea of node recycling is to remove nodes from unpromising areas of the tree and recycle the freed memory to build more promising areas. The behaviour of MCTS is to exploit, i.e. repeatedly visit, the most promising areas of the search tree, whilst infrequently exploring less promising areas. Thus it makes sense to recycle areas of the tree that have not recently been accessed, as those areas have a low priority for being exploited.

The overall idea of node recycling is as follows. Rather than creating a new node upon each expansion, a fixed pool Φ of nodes is allocated up front (the size of Φ is determined by the memory budget). We use nodes from Φ until it is exhausted, at which point we begin recycling nodes from the tree. The node to be recycled is the leaf node whose statistics have least recently been *accessed*, i.e. for which the UCB1 value has least recently been computed. We recycle based on accesses rather than *visits*, as otherwise a rarely visited child of a frequently visited node may be recycled and then immediately re-expanded and re-explored. Even if a node is rarely visited by playouts, it may frequently be accessed and rejected by the selection policy.

Finding the least recently accessed leaf node by searching over the entire tree would imply a time complexity $O(|\Phi|)$ for the above algorithm. However, using appropriate data structures and a small amount of bookkeeping during MCTS iterations we achieve $O(1)$ time complexity (i.e. the time required does not depend on $|\Phi|$). Essentially the nodes are managed using a least recently used (LRU) cache implemented as a first-in first-out (FIFO) queue. Whenever a node is accessed it is removed from its current position in the queue and pushed to the back. When the memory bound is reached, the front node in the queue (i.e. the least recently accessed node) is recycled.

Our implementation introduces no measurable difference in computation time per decision compared to MCTS with

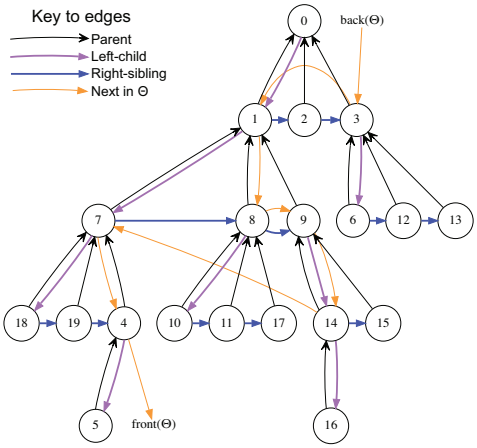

Figure 1: Illustration of node pointers for node recycling MCTS, showing the left-child right-sibling tree representation and the queue Θ of visited nodes. Nodes are numbered with their indices in the array Φ. The "previous node" pointers in the doubly linked list Θ are omitted for clarity.

unbounded memory. Indeed it is difficult to say whether the bounded algorithm is slower or faster than the unbounded algorithm, i.e. whether the bookkeeping overhead is less or greater than the time saved by performing fewer memory allocations and causing fewer cache misses when accessing memory. In our implementation, the difference in decision times for a fixed number of iterations is less than 1%, which is within experimental error.

Our MCTS implementation uses a *left-child right-sibling* representation (Cormen et al. 2001) for the search tree. In this representation, each node stores pointers to three other nodes: its parent, its leftmost child, and its sibling immediately to the right. This means that the children of a node are stored as a singly linked list. Figure 1 illustrates this.

In MCTS, backpropagation proceeds up the tree, from leaf to root. This means that backpropagation updates each node before its parent. Let v be the non-leaf node for which the time since last update is maximal. We know that v has only leaf nodes as children: if v had a non-leaf child v' then the last update at v' would necessarily be before the last update at v, violating the requirement on the choice of v. Furthermore, the leaf node whose statistics were accessed least recently must be a child of v: if it were not, then the leaf's parent would necessarily have been updated less recently than v. Thus if we can find v, we can find the next leaf node to recycle.

To rapidly determine the least recently updated node v, we maintain a queue Θ of nodes. Non-leaf nodes are removed from Θ when visited during playout, and pushed to the back of Θ when they are updated. Leaf nodes are never present in Θ. Thus the node at the front of Θ is the least recently updated non-leaf node, i.e. the node v sought in the previous paragraph. The queue is implemented as a doubly-linked list implemented by storing pointers to the previous and next queue elements in the node class itself. This guarantees that removing a node from its current queue position, pushing a node to the back, and popping a node from the front all execute in constant time ($O(1)$ with respect to the node pool size). Figure 1 illustrates the tree structure used by node recycling MCTS, showing the pointers used by each node to store both its position within the tree and its position within Θ. From front to back, the nodes in Θ are $\langle 4, 7, 14, 9, 8, 1, 3 \rangle$, thus the child of node 4, i.e. node 5, is the next node to be recycled.

Using a FIFO queue to keep track of the least recently visited node gives a good combination of sensible node choice and low computational overhead, as there is no need to traverse the tree to find a node to recycle. There is a memory overhead associated with storing two extra pointers per node (the previous and next nodes in Θ); in our C++ implementation this increases the memory usage from 48 to 56 bytes per node, an increase of 16.7%.

Fuego-like garbage collection. The Go program Fuego (Enzenberger et al. 2010) uses a "garbage collection" strategy when the search reaches its memory limit. All nodes which have been visited less than some fixed threshold are removed, and the search continues as normal.

The conceptual goal is similar to that of our node recycling scheme: discard potentially unimportant areas of the tree to make way for more promising ones. The key difference is in the assessment of "potentially unimportant": for node recycling this means areas of the tree which have not been visited recently, whereas for Fuego-like garbage collection it means areas which have only rarely been visited over the entire course of the search. Notice that once a node has received more than the threshold number of visits, it can never be deleted. This contrasts to node recycling where a region of the tree visited intensively early in the search, which turns out to be unpromising, will eventually be deleted.

In (Enzenberger et al. 2010) the node visit threshold is set to 16. However the optimal setting is likely to depend on the number of iterations, which is orders of magnitude larger for Fuego than for the experiments in this paper. We found that a threshold of 4 gave the best performance overall across the domains and parameter settings studied in this paper.

Experiments

In this section we investigate the performance of the memory bounding methods on three very different games of imperfect information: a partnership card game with fully observable moves; a card game without partnerships, with both hidden and fully observable moves; and a 2-player board game where both moves and position information are hidden.

Domains

Spades is a four-player trick taking card game (Pagat 2017b). The players are paired into two opposing partnerships: North and South versus East and West. A version of our ISMCTS player enhanced with heuristic knowledge and our new node recycling algorithm is currently deployed in a popular implementation of Spades for Android mobile phones (Whitehouse et al. 2013; Cowling et al. 2015).

The heuristic knowledge is primarily used to alter the playing style for a more enjoyable game; in terms of playing strength, SO-ISMCTS is strong without knowledge. The version of the ISMCTS player used in this paper does not use heuristic knowledge, other than the end-of-round simulation cutoff described in (Whitehouse et al. 2013).

Hearts is a four-player trick taking card game (Pagat 2017a). There are no partnerships: all four players are in competition with each other. Each round begins with players simultaneously passing cards to each other. These are partially observable moves, so we use MO-ISMCTS. As with Spades, the ISMCTS implementation for Hearts simulates to the end of the current round rather than the end of the entire game. A version of Hearts created by Microsoft is included with some versions of Windows; we have performed AI-versus-AI tests of our ISMCTS player against the AI opponent provided with the Windows 7 version, and found it to be on a par in terms of playing strength.

Lord of the Rings: The Confrontation (LOTR:C) is a two-player strategic board game designed by Reiner Knizia (BoardGameGeek 2017). The two players are designated the *Good* player and the *Dark* player. The game has hidden information: the pieces have different strength values and special abilities, but these attributes are hidden from the opponent. Our previous work on LOTR:C showed that MO-ISMCTS achieves a level of play equal to that of strong human players (Cowling, Powley, and Whitehouse 2012).

Comparison of memory bounding methods

We test the relative playing strength of the memory bounded MCTS methods described in the previous section. For Hearts and LOTR:C we play one instance of memory bounded MCTS against $\kappa - 1$ instances of MCTS with unbounded memory, where κ is the number of players. For Spades, we test a partnership of two memory bounded players against a partnership of two unbounded MCTS opponents. In LOTR:C, due to the highly asymmetric nature of the game (i.e. different abilities and win conditions for each player), we conduct separate experiments for Good and Dark players. In Hearts and Spades the starting player is determined randomly according to the game rules, and so the player number assigned to the memory bounded MCTS player has no bearing on the result averaged over many games. We vary the size of the node pool for each variant of memory bounded MCTS, playing $1\,000$ games for each setting and using $5\,000$ iterations per decision. This experiment represents around 4 CPU-months of computation in total, and was performed on a cluster of 72 processor cores. Results of this experiment are shown in Figure 2.

For large memory budgets, memory bounding has little or no effect on the operation of MCTS, since the search does not have chance to exhaust the available memory. The most pertinent region of the graphs is thus at the low and middle parts of the x-axis range (20–500).

The most effective methods overall (node recycling and Fuego-like garbage collection) are the only methods which keep frequently visited lines of play in the tree for the duration of the search at the expense of less frequently visited lines, suggesting that retaining these nodes is important to the strength of MCTS. Stunting, stopping and ensemble MCTS do not have this property. Furthermore, stopping and ensemble MCTS bound the number of UCB1 iterations at the root whereas stunting does not. Comparing these algorithms indicates that using more UCB1 iterations at the root node of a simple tree gives better performance, even if the tree structure below the root remains static.

For Spades and Hearts, the best algorithms perform well with as few as 20 nodes. For these games, 20 nodes do not allow more than one or two plies in the tree to be built, which suggests that exhaustive search of the game tree beyond a few moves is not necessary for good playing strength. In many cases 20 nodes are enough to cover or nearly cover the current trick, so this suggests diminishing returns from building the tree beyond the current trick. For LOTR:C, there is a clearer (though still not catastrophic) disadvantage to smaller node pool sizes. In all cases the slope of the other lines is much shallower than for stopping, showing that, if an appropriate memory bounding technique is used, reducing the memory budget has much less impact on the performance of MCTS than reducing the time budget.

In Spades, stunting with a memory budget of 20 nodes is significantly better than for 50 or 100 nodes. This result is highly statistically significant ($p < 10^{-89}$), so cannot be dismissed as a statistical anomaly. In the stunted decision tree for Spades, budgets of 50 or 100 may bias the search by ending the tree part way through considering the moves of a particular player, whereas 20 nodes allows for one complete ply of the tree and little else.

A smaller memory budget restricts the search to consider only a few long lines of play. This seems to work fine for Hearts and Spades, where there is a good deal of uncertainty, but not so well for LOTR:C which requires continuous comparison of deeper long-term strategies.

Conclusion

MCTS and ISMCTS are powerful search algorithms with many promising applications in commercial games, which are only just being explored. Many of these applications restrict the memory available for search, for example on mobile platforms or in a multi-agent setting. The memory usage of (IS)MCTS grows linearly with respect to the number of iterations used. This paper proposes a number of variants of (IS)MCTS which allow a predetermined bound to be placed upon the amount of memory used by the algorithm. We have demonstrated that the playing strength of (IS)MCTS continues to increase with respect to the number of iterations used when the memory is bounded. We have introduced a *node recycling* algorithm which transforms (IS)MCTS into an *anyspace* algorithm, well suited to applications where both time and space are limited. Node recycling can be implemented efficiently and achieved the best performance given a memory bound in all our test domains.

Since (IS)MCTS explores a game tree asymmetrically, when using node recycling the memory budget is used to store nodes from the subtree currently being explored. If another subtree is explored, information near the leaves of the original subtree will be forgotten. We have shown that memory bounding in this way *does* degrade the performance of

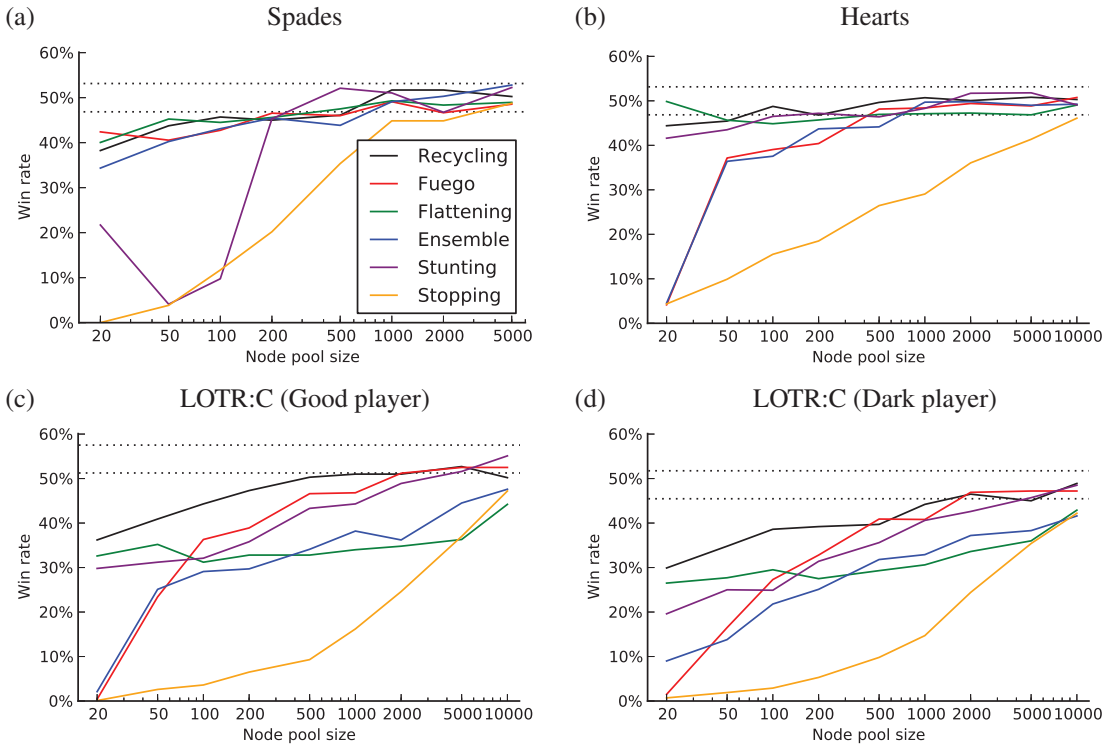

Figure 2: Effect of node pool size on playing strength for memory bounded MCTS. The x-axis (node pool size) is logarithmic. In Hearts and LOTR:C one instance of memory bounded MCTS plays against MCTS opponents without memory bounds; in Spades a partnership of two memory bounded MCTS agents play against two unbounded MCTS players. All players use 5000 iterations per decision. Each data point is based on 1000 games. The horizontal dotted lines indicate the 95% confidence interval of the baseline win rate for a player identical to its opponents (i.e. without memory bounds). Confidence intervals are otherwise omitted for clarity, but are of similar size to those for the baseline. For Hearts, finishing in second or third place counts as half a win, hence the expected "win rate" amongst evenly matched players is 50%.

(IS)MCTS, but some games (particularly those with imperfect information) are not very sensitive to memory bounding. It is notable in Figure 2 that a node pool size of 1000 is not significantly worse than unbounded MCTS in any of the games tested here, with 5000 simulations. This player requires only $\frac{1}{5}$ of the memory used by unbounded MCTS in the perfect information and SO-ISMCTS cases, and around $\frac{1}{5\kappa}$ of the memory for unbounded MO-ISMCTS in a game with κ players. Crucially, the node recycling method is able to make full use of any available time and memory budget — an *anytime, anyspace* algorithm.

The node recycling scheme proposed here is inspired by the exploitative behaviour of (IS)MCTS: if a region of the tree was promising then it would have been visited recently, therefore unvisited regions of the tree must be relatively unpromising and can be pruned with minimal effect on the performance of the algorithm. However there may be cases where keeping an unpromising region of the tree is beneficial. For example, consider a move which initially appears good, but a line of play exists that shows it to be poor. The region containing this line of play will be infrequently visited, but if it is deleted then the move will begin to appear good again. The effect of this is that the tree below this move is constantly deleted and rebuilt. A more informed recycling scheme, that somehow detects and keeps such refutations of initially promising moves, could avoid this problem. However, the node recycling scheme we have presented works well in the test domains in this paper (and several others).

We have successfully demonstrated that (IS)MCTS can be modified to work within a fixed memory budget, with relatively little degradation in performance. In recent years (IS)MCTS has comprehensively proved itself in empirical work and competitions, where the primary constraint is decision time. Memory bounding paves the way for practical deployment of (IS)MCTS in domains where memory is constrained also, including video games as well as embedded and mobile applications.

Acknowledgments

We thank Jeff Rollason of AI Factory Ltd for invaluable input to our work and for providing a software framework for our Spades experiments. This work is funded by grant EP/H049061/1 of the UK Engineering and Physical Sciences Research Council (EPSRC).

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Mach. Learn.* 47(2):235–256.

Bjarnason, R.; Fern, A.; and Tadepalli, P. 2009. Lower Bounding Klondike Solitaire with Monte-Carlo Planning. In *Proc. 19th Int. Conf. Automat. Plan. Sched.*, 26–33.

Björnsson, Y., and Finnsson, H. 2009. CadiaPlayer: A Simulation-Based General Game Player. *IEEE Trans. Comp. Intell. AI Games* 1(1):4–15.

BoardGameGeek. 2017. Lord of the Rings: The Confrontation. http://boardgamegeek.com/boardgame/3201/lord-of-the-rings-the-confrontation.

Breuker, D. M.; Uiterwijk, J. W. H. M.; and van den Herik, H. J. 1994. Replacement schemes for transposition tables. *ICCA Journal* 17(4):183–193.

Browne, C.; Powley, E. J.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Trans. Comp. Intell. AI Games* 4(1):1–43.

Cazenave, T., and Jouandeau, N. 2007. On the Parallelization of UCT. In *Proc. Comput. Games Workshop*, 93–101.

Champandard, A. J. 2014. Monte-Carlo Tree Search in Total War: Rome II's Campaign AI. http://aigamedev.com/open/coverage/mcts-rome-ii/.

Chaslot, G. M. J.-B.; Saito, J.-T.; Bouzy, B.; Uiterwijk, J. W. H. M.; and van den Herik, H. J. 2006. Monte-Carlo Strategies for Computer Go. In *Proc. BeNeLux Conf. Artif. Intell.*, 83–91.

Chaslot, G. M. J.-B.; Winands, M. H. M.; and van den Herik, H. J. 2008. Parallel Monte-Carlo Tree Search. In *Proc. Comput. and Games, LNCS 5131*, 60–71.

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. 2001. *Introduction to Algorithms*. MIT Press, second edition.

Coulom, R. 2007. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Proc. 5th Int. Conf. Comput. and Games, LNCS 4630*, 72–83.

Cowling, P. I.; Devlin, S.; Powley, E. J.; Whitehouse, D.; and Rollason, J. 2015. Player Preference and Style in a Leading Mobile Card Game. *IEEE Trans. Comp. Intell. AI Games*.

Cowling, P. I.; Powley, E. J.; and Whitehouse, D. 2012. Information Set Monte Carlo Tree Search. *IEEE Trans. Comp. Intell. AI Games* 4(2):120–143.

Cowling, P. I.; Ward, C. D.; and Powley, E. J. 2012. Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering. *IEEE Trans. Comp. Intell. AI Games* 4(4):241–257.

Enzenberger, M.; Müller, M.; Arneson, B.; and Segal, R. B. 2010. Fuego - An Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search. *IEEE Trans. Comp. Intell. AI Games* 2(4):259–270.

Fauerby, K. 2012. Crowds in Hitman: Absolution. http://files.aigamedev.com/coverage/GAIC12_CrowdsInHitmanAbsolution.pdf.

Fern, A., and Lewis, P. 2011. Ensemble Monte-Carlo Planning: An Empirical Study. In *Proc. 21st Int. Conf. Automat. Plan. Sched.*, 58–65.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo Planning. In Fürnkranz, J.; Scheffer, T.; and Spiliopoulou, M., eds., *Euro. Conf. Mach. Learn.*, 282–293. Berlin, Germany: Springer.

Korf, R. E. 1985. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.* 27(1):97–109.

Lee, C.-S.; Müller, M.; and Teytaud, O. 2010. Guest Editorial: Special Issue on Monte Carlo Techniques and Computer Go. *IEEE Trans. Comp. Intell. AI Games* 2(4):225–228.

Mountain, G. 2015. Tactical planning and real-time MCTS in Fable Legends. https://archives.nucl.ai/recording/tactical-planning-and-real-time-mcts-in-fable-legends/.

Pagat. 2017a. Hearts. http://www.pagat.com/reverse/hearts.html.

Pagat. 2017b. Spades. http://www.pagat.com/boston/spades.html.

Perez, D.; Samothrakis, S.; Togelius, J.; Schaul, T.; Lucas, S.; Couetoux, A.; Lee, J.; Lim, C.-U.; and Thompson, T. 2015. The 2014 general game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games*.

Powley, E. J.; Whitehouse, D.; and Cowling, P. I. 2011. Determinization in Monte-Carlo Tree Search for the card game Dou Di Zhu. In *Proc. Artif. Intell. Simul. Behav.*, 17–24.

Russell, S. 1992. Efficient memory-bounded search methods. In *Euro. Conf. Artif. Intell.*, 1–5.

Sephton, N.; Cowling, P. I.; Powley, E. J.; Whitehouse, D.; and Slaven, N. H. 2014. Parallelization of information set monte carlo tree search. In *Proc. IEEE Congress on Evolutionary Computation*, 2290–2297.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529:484–489.

Whitehouse, D.; Cowling, P. I.; Powley, E. J.; and Rollason, J. 2013. Integrating Monte Carlo Tree Search with Knowledge-Based Methods to Create Engaging Play in a Commercial Mobile Game. In *Proc. Artif. Intell. Interact. Digital Entert. Conf.*, 100–106.

Whitehouse, D.; Powley, E. J.; and Cowling, P. I. 2011. Determinization and Information Set Monte Carlo Tree Search for the Card Game Dou Di Zhu. In *Proc. IEEE Conf. Comput. Intell. Games*, 87–94.