# 28 Pitfalls and Solutions When Using Monte Carlo Tree Search for Strategy and Tactical Games

*Gijs-Jan Roelofs*

## 28.1 Introduction

The biggest challenge when building AI for modern strategy games is dealing with their complexity. Utility- or expert-based techniques provide good bang for their buck in the sense that they get the AI to a reasonable, even decent level of play quickly. Problems start to arise, however, when an AI programmer is faced with situations that are not straightforward or too time-consuming to encode in heuristics. Or when he or she is faced with above average gamers who demand a good opponent and are quick to find any exploits in an AI.

Search techniques seem like an interesting solution to provide adaptable and more dynamic AI. Traditional search techniques, like minimax, iterate over all possible moves, evaluate each resulting state, and then return the best move found. This approach does not work in strategy games because there are simply too many moves to explore.

MCTS (*Monte Carlo Tree Search*) is a new and increasingly popular technique that has shown it is capable of dealing with more complex games. This chapter outlines solutions to common pitfalls when using MCTS and shows how to adapt the algorithm to work with strategy games.

An example of one such pitfall is that implementations of search techniques in game AI tend to work by limiting the search to a subset of moves deemed interesting. This approach can lead to an AI that can easily be exploited because it literally does not see the pruned moves coming. Techniques outlined within this chapter will give MCTS the opportunity to discover these moves while still ensuring a good base level of play in those worst-case situations where it does not find them.

Throughout the chapter we showcase how these techniques were implemented using two vastly different strategy games: *Berlin*, an online Risk-like game in which the player commands hundreds of units in an effort to capture all regions, and *Xenonauts 2*, a tactical game akin to *X-Com* developed by Goldhawk Interactive in which the player commands a squad of 12 soldiers. In *Berlin*, the introduction of the core techniques outlined in this chapter led to a 73% win and a 15% draw rate over the best hierarchical AI based implementation (utility-based driven by behavior tree adjusted weights). The actual gameplay shown by MCTS was adaptive to new player strategies without requiring any input or rebalancing (Roelofs 2015).

This chapter delves into details which assume a basic understanding of MCTS. Those unfamiliar with this technique should start by reading the introduction provided in *Game AI Pro 2* (Sturtevant 2015). For this chapter, we use the most common and battle tested variant of MCTS: *UCT* (*Upper Confidence Bounds Applied to Trees*) (Chaslot 2010).

To ease understanding, the sections of this chapter are grouped into two categories: design principles and implementation tricks. The design principles outline general principles which need to be adapted to each game to which they are applied, and require full understanding to apply correctly. Implementation tricks are general ways to improve any MCTS implementation.

In the next section, we give an overview of the problem. The sections beyond that are structured according to the four major phases of MCTS (shown in Figure 28.1): *Selection*, *Expansion*, *Simulation*, and *Backpropagation*.

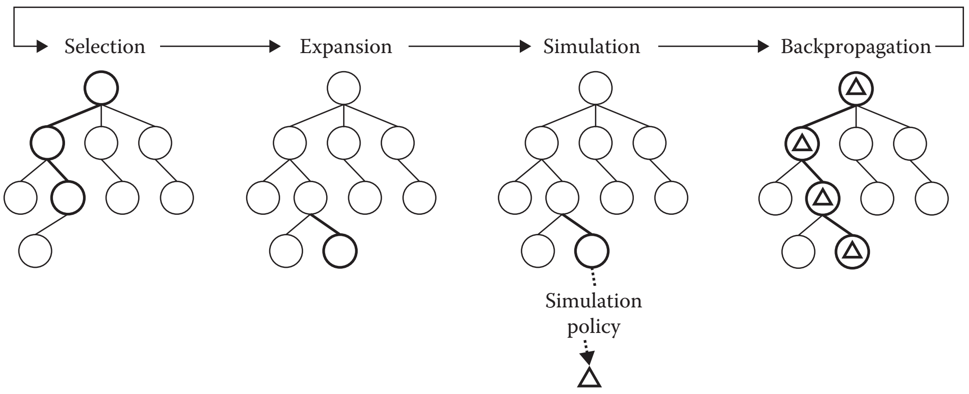

Figure 28.1

The *Monte Carlo Tree Search* algorithm.

## 28.2 The Problem

The strength of MCTS lies in its ability to cope with large, complex problems in which the long-term ramifications of a move are generally more important than its short-term effects. In other words, in problems where the actual effects only become apparent after executing long sequences of moves, that is, a *deep* search space. It excels in areas where alternative techniques, such as minimax, given ample depth (6–12 ply) still do not return useful strategies.

In general, performance for search techniques starts to falter when the number of choices possible per level of the tree starts to increase, that is, a *wide* search space. Although MCTS tends to cope better with this than many other techniques, increasing the width is still the quickest way to drastically reduce performance (Roelofs 2015).

For strategy games, one common problem is when a single move for a player can be broken down into several actions (e.g., many orders to individual units). If each of these actions affects the outcome of the sequence as a whole, we need to examine all possible combinations of them, resulting in search spaces of often insurmountable width and subsequently, size (Churchill and Buro 2015, Roelofs 2015).

For example, in *Xenonauts 2* a unit can move to upwards of 50 different locations and select between 4 different weapons to attack 1 of 16 different targets, not to mention additional unit abilities. Even after pruning those locations, weapons, and targets which are not interesting we are still left with about 12 options per unit. Given 16 different units, this results in $12^{16}$ different combinations of actions to search through per turn. No amount of calculation time will ensure that the search returns with good results if we try to look at all combinations of actions.

Luckily, more often than not these actions do not influence each other that much, or if they do, there are specific actions which have large ramifications and thus stand out. Often, the AI programmer might not know which particular action would work well, but he or she has an understanding that certain types of actions have the potential to perform far better than others. This is better explained with some concrete examples:

In *Xenonauts 2*, some units had the ability to mind control an enemy unit. This ability tended to have a far bigger effect on the outcome of a mission than other actions. By creating a heuristic function based on the worth (objectives or worth of the units to the player) and potential for the AI (weapons, abilities, and position) we were quickly able to get the AI to function at a decent level of play. However, throughout development we kept discovering that the AI missed crucial moves which were extremely situational and nightmarish to encode through heuristics. Moves which were the result of combinations of abilities or weaknesses on units in both the player and AI squad, objectives which invalidated or revalidated certain tactics, or moves which resulted in wins further in the future through sacrifice of units.

In *Berlin*, an example of such an action would be the capture of a region that would not be deemed valuable using common heuristics, but would provide a crucial vector of attack in subsequent turns giving the AI an advantage in the long run.

These are actions that if done correctly would have given the player pause but if done incorrectly would be glaring mistakes. In short, actions that are not easily found through the use of heuristics or for which the actual effects several turns from now need to be simulated to see their actual worth. The essence of the solution we propose is to enable

MCTS to discover and subsequently exploit these actions—that is, actions that have a large impact on the overall outcome of a move—in an effort to exploit those moves that contain them.

Before delving into the details and actual approach, we reiterate and define the key terms used throughout the chapter:

- *Move*: A sequence of independent actions constituting a full player turn.
- For example: All units have received orders for the turn.
- *Action*: An independent part of a move.
- For example: The orders to be sent to a single unit, attack enemy X, or move to location A.
- *Action set*: All of the possible values for a particular action.
- For example: A unit can attack {X, Y, or Z}, move to {A, B, or C}, or use special ability {J, K, or L}.

## 28.3 Expansion

This section outlines how to restructure the search such that MCTS will be able to identify good actions and subsequently use exploitation to ensure it can spend more time on moves that contain them and less on moves that do not. The key to this is to first expand into those actions which we expect to perform well. This provides good early estimates that later help the exploitation process of MCTS. Which actions we expand into can be specified through heuristics, learned using entropy learning or by restructuring the expansion such that MCTS learns it during search (Roelofs 2015). Furthermore, splitting up the move also allows us to prune actions or options in an action set which are not of interest, or simply select without search those for which we know a good solution exists through heuristics.

### 28.3.1 Restructuring Your Search Space (Design Principle)

By default, at each expansion step MCTS expands into all possible moves from the current node. The problem with this type of expansion is that there is no transfer of information between moves at the same level. MCTS simply cannot exploit information gained from one move to prefer one over another and needs to inspect each move individually.

By dividing the move into actions we can enable MCTS to understand that move B, which is a minor variation of a good move A that it explored earlier, might be a good move as well. Instead of expanding over all possible moves at each expansion step, we select a specific action within the current move and expand into the actions defined in its action set. We can further enhance performance by carefully pruning or ordering the action set of an action before the start of the search, or during it based on the actions already explored in the current move. To distinguish from the default expansion strategy, we term *hierarchical expansion* as expanding into actions as opposed to moves.

In *Xenonauts 2*, we carved up a move into the orders per unit, each action being a single order assigned to a unit. Each unit could have multiple orders assigned to it until a move consisted of actions in which it had spent all of its time units or was put on overwatch. The reasoning behind this was that often the outcome of a mission could be swayed by the actions of a single unit, and thus the selection between individual actions was crucial.

This was helped by the fact that we limited the search to those units in engagement with the player and let overall positioning of nonengaging squads be done by simple heuristics.

In *Berlin*, the move was carved up into a single action per region, in which we decided how soldiers would attack or reinforce adjacent regions. The reasoning behind this was that we only needed to hone in on the decisions of specific regions. Looking at the soldier level would result an in excessive increase in size of the search space.

Figure 28.2 gives a graphical example of hierarchical expansion given three action sets, each with two actions ({*A*, *B*}, {*C*, *D*}, {*E*, *F*}). In this example, as the search starts MCTS will explore the actions *A* or *B* and subsequently all moves which contain those actions. As it becomes apparent that one action results in significantly better results, it will avoid the other action, and subsequently all moves that contain it. The core idea being that if action *A* or *B* is truly better than the other, it will show through the simulations, regardless of which move it was played in.

Herein lies the drawback of this approach. If we start with an action set which has no apparent good move, or if a good move happens further down the line, we may lose crucial calculation time. MCTS will keep switching between the nodes as it is unapparent which sequence of actions is good, and the quality of the result suffers. Another issue is that this technique works only when the search space starts becoming large. If it is not, we are simply wasting time by adding in extra nodes and splitting the tree. As a general rule, once we start dealing with games in which a single move can be broken down into actions, and a combinatorial number of moves exist, the technique can be applied (Roelofs 2015).

## 28.3.2 Expansion Order of Actions (Design Principle)

The key understanding in all this is that we stimulate MCTS to exploit the value found for an action, and thus avoid those moves which do not contain any interesting actions.

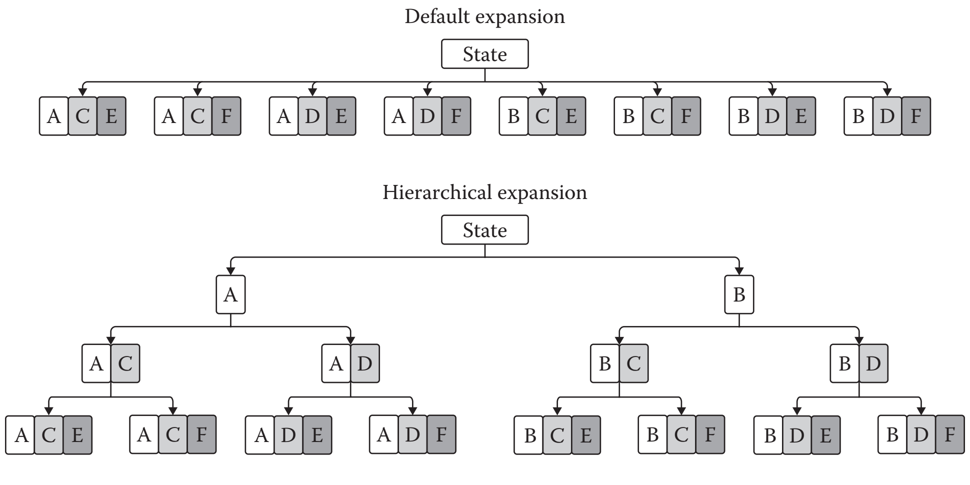

Figure 28.2

Hierarchical expansion for three action sets, each with two actions, resulting in one level for normal expansion, and three levels for hierarchical. (*Note:* As MCTS is a best-first search algorithm, more interesting lines of the tree will be explored in favor of others. The above figure of a balanced tree is to illustrate how expansion happens in a fully explored tree.)

If we can ensure that action sets with apparent good choices are explored first, MCTS will quickly converge and exploit the moves which contain them, avoiding large swathes of uninteresting search space. This basic understanding is why the algorithm performs well while we are counterintuitively increasing the number of nodes in the tree. This approach also has the added benefit of ensuring that most time is spent on those actions which are important as they are simply considered first.

Per example, in *Xenonauts 2* we first expand into orders for enemy units nearer to the player under the assumption that they would provide good opportunities of attack. The action set itself was ordered to prefer attack and ability actions, followed by movement. This ensured that the algorithm was unlikely to miss any crucial attacks on player units. The AI played stronger, and players were quick to label the AI as "stupid" if it missed obvious strong ways to attack thus necessitating the focus on attacks.

In *Berlin* we expanded into regions closer to enemy regions first, under the assumption that crucial troop movement would occur there.

Even if the value of an action is not apparent it is often relatively intuitive to provide an ordering on which actions to explore first. From our earlier example: We might not know the heuristic value of the mind control ability, but we know it tends to be more valuable than any other action. Even if it is not effective in the current situation it allows us to exclude those moves that do not contain it.

To test the overall strength of a given ordering, or find out an ordering if no heuristic is known, we can use two alternative strategies to provide one:

1. *Entropy-based*: The *entropy* of an action set represents the ease with which MCTS can make a decision between the actions in it. If the entropy is low, and thus all values are nearly equal to each other, MCTS will often find it hard to converge on a specific action. If the entropy is high, values of actions differ wildly and MCTS will be quicker to converge. We can simply calculate the entropy of an action set by using the ratio of the number of times MCTS has visited an action compared to its parent as *P(x)* in *Shannon Entropy:* $-\sum_{i=1}^{n} P(x_i)\log P(x_i)$
2. This technique is mostly used to either discover an ordering we can use as a heuristic or to check whether the heuristic ordering you have defined is good enough.
3. *Dynamic ordering*: Normally, we select a specific ordering to the actions at the start of the search, and which action is expanded is determined by the layer of the tree we are in. Looking at Figure 28.2 we can see that we first expand into $\{A, B\}$, followed by $\{C, D\}$ and finally $\{E, F\}$. If we select a random action set to expand into for *each expansion of an action*, we essentially allow MCTS to exploit the ordering itself during the search. To better explain this concept using Figure 28.2: After action $A$ we could expand into $\{E, F\}$; after action $B$ we could expand into $\{C, D\}$. This strategy works best if actions are truly without sequence. It works because if an action set truly has an important choice, the search will gravitate toward where this action set exists nearest to the root. This strategy tends to give far better results than just choosing an arbitrary ordering as there we can have the bad luck that the action sets with interesting choices exist too deep in the tree, with the search never reaching them (Roelofs 2015).

### 28.3.3 Dealing with Partial Results (Design Principle)

Using default expansion, only complete moves are explored in MCTS. The end result of a search is simply the best node at the root of the tree: a move that can be applied immediately. However, if we split up the move, the actions at the root are just part of the answer. We need to reconstruct the complete action by iteratively applying the *final node selection* strategy up the tree gathering the resulting actions and reconstructing the move.

When using the above approach, MCTS will not always be able to return a move if the actions in the tree are not sufficient to create a complete move. We used two different strategies to complete a so-called partial move, in order of performance:

1. *Estimated values*: During the backpropagation phase of MCTS, maintain an average value of the actions used in the playout. When completing a partial move, we simply select the action that scored the best on average through all playouts. This technique worked well when the playout itself was guided by heuristics as well, or even better, a portfolio of heuristics. The actions which returned then would always be at least actions we approved by heuristics, but evaluated in context. This would allow us to reclaim information lost due to MCTS switching between branches of actions.
2. *Heuristics*: Apply the strategies used in the playouts to complete the action, assuming they are using heuristics. This tends to lead to better performance than just randomly completing the action as we can then ensure at least a base level of play.

### 28.3.4 Pruning Action Sets (Design Principle)

The key concept is that we only explore those actions in MCTS which we think actually need exploration. If a solution is apparent for a certain action set, then there is no reason to add these to the search. By pruning those action sets from expansion all together and letting them be resolved through heuristics we can improve on the quality of the search as it can spend more time on action sets which actually require it.

For example, in *Xenonauts* any unit far away from the area of conflict would either just go to the nearest applicable cover, or move toward the conflict if so requested by any unit in it. We do not need to explore the actions of these units as the choice is rather apparent.

For *Berlin* we determined that regions some distance away from the front should move their units toward the front, as this was the decision returned often by the search.

When applying MCTS to multiplayer games with more than two players, this concept can be applied by only focusing on the strongest player. At expansion into the moves of other players, simply determine the strongest opponent and ignore moves from others (Schadd and Winands 2011).

### 28.3.5 Iterator-Based Expansion (Implementation)

The default implementation of the expansion phase is to create all actions, adding them as child actions as shown in Figure 28.1. However, if there are many possible moves this tends to be quite wasteful as MCTS might decide that the parent action is no longer interesting after a single iteration. By using an *iterator pattern* and only adding a single action per expansion we can avoid a lot of resource waste resulting in a significant performance boost.

The order in which actions are created by this pattern affects the performance of the algorithm. If the sequence is always the same, the first actions returned will be explored far more often in the overall search. When the first action returned is a badly performing action the node will be undervalued in its first iteration and it might take the algorithm a while before revisiting it, if at all. Returning a random ordering of actions, or providing the best actions according to some heuristic first thus tends to improve the quality of the result.

## 28.4 Simulation

Complexity rears its ugly head again in the simulation phase of MCTS. In the following sections we will explore how judiciously applying heuristics to guide the playout can increase its value and how abstracting over the simulation can lead to more and better iterations, and thus a better result.

### 28.4.1 Better Information (Design Principle)

For strategy games the traditional approach, in which we simulate the move chosen by the search by playing random moves until the game ends, tends to give little useful information. Due to the enormous size of the search space, playing random moves rarely gives any interesting results or represents good play. Using heuristic-based strategies tends to increase performance of the algorithm as the information provided by a single playout increases. However, this does come at an expense: It will ensure that certain actions are overlooked in the simulation and that the AI works with partial information. This in turn can result in exploitable behavior as the AI becomes blind to certain actions, or even lower performance if the AI misses crucial actions.

Heuristics strategies can further decrease performance if they are expensive to compute. Increasing the time in simulation reduces the number of iterations of MCTS, in turn reducing performance. On the other hand, heuristic strategies can actually speed up the simulation by ensuring that a game ends quickly in the simulation phase. A game played randomly can take quite a lot longer before it ends as opposed to a game played with a valid strategy.

The best of both worlds is to use a mixed strategy approach, often called an epsilon-greedy playout, resulting in far stronger play than a pure random or heuristic strategy. This can be achieved by constructing a weighted portfolio of strategies to use in the playout, including the default random playout, and selecting an action from this portfolio (Roelofs 2015).

### 28.4.2 Abstract Your Simulation (Design Principle)

Most often the simulation phase uses the actual game rules as they are readily available and correct. However due to the fact that by their very nature strategy games tend to be complex problems, it is advisable to use an abstraction in the simulation phase. Instead of actually calculating the results of sending your units and simulating the full conflict using game logic, build an abstraction of the conflict and define a function that given a setup quickly calculates whether the conflict is a loss or victory. The key to this approach is that the actions explored within the tree are still legal moves in your game.

In *Xenonauts* using the actual game logic was infeasible as it was tied to too many systems outside the AI and performance-wise was never intended to be used in simulations. We therefore created a model in which soldiers were represented solely by their computed strength (based on their health, inventory, and abilities), and position (to calculate cover and LOS). The outcome of confrontations then became a simple computation that could be iterated over much more quickly. Actions executed during the expansion and selection phases, and thus not in simulation, would be executed using actual logic to ensure they would be correct. The moment the search entered simulation, the current values of the state would be transferred to the constructed model.

## 28.5 Backpropagation

Game AI programmers have ample experience in constructing evaluation functions in order to enact specific behavior on a character. Modeling an evaluation function for a search technique has some minor, albeit very important, distinctions. The primary difference is that how we structure our evaluation function influences not only the results found (and thus the behavior of the character), but also influences the *behavior of the search* as it executes.

MCTS has a poor track record for tactical decision-making. More accurately, it struggles with problems that have a very narrow path of victory or success. The game Tic Tac Toe is a prime example of this: Making a single mistake will lead to a loss. A simple evaluation function that returns –1 for losses and that returns +1 for wins will take quite a while to converge on the proper set of moves. However, if we severely punish the AI for losses with –100, MCTS displays a very paranoid behavior in which it will only select nodes that give the enemy no chance of a win. The end result is a search that converges to the correct move far sooner.

Adjusting the evaluation function in this way causes MCTS to quickly move away from nodes that lead to negative reinforcement, or alternatively make it so that it is eager to explore nodes that have not encountered any losses.

Proper evaluation functions, such as evaluating the margin of victory, that give continuous values are preferred. However, these functions run the risk of diluting which actions are better from the perspective of the algorithm. We would rather have that the search explores one action deeper than keep switching between two actions of similar value. Changing the range of your function to make the differences in actions more apparent often leads to better results.

## 28.6 Selection

The goal of the selection phase is to determine the “best” node according to some strategy, often *UCT*. The optimization in this section is based on the fact that only a single node per layer gets its value updated during backpropagation. Also, as MCTS starts to converge, most nodes do not, or barely, change their position in this order. So, instead of repeatedly recomputing the rank of each node during selection, an optimized variation of insertion sort can be used to cache the rank of each node and store which node should be selected next. This becomes particularly important when the number of moves or actions increases. A code example of this is provided in Listing 28.1. In a single thread environment, or in *playout* parallelization, this solution will be sufficient. If a *root* parallelization technique is chosen, the update of this order should be moved to the backpropagation phase, in order to more easily lock the node in question (Chaslot et al. 2008).

Listing 28.1 also showcases an often overlooked requirement: The *selection* phase of MCTS should ensure that all child nodes have received some minimum number of visits $T$ (often $T = 20$) before actively selecting the best child. This ensures that the value of the node is somewhat reliable, and not just the result of a single lucky playout.

**Listing 28.1.** Returns the next node to visit, after all children have been visited a minimum number of times.

```
public Node selectNextNode(Node node, int minVisits)
{
     Array<TreeSearchNode> children = node.getChildren();
     int childrenNo = children.size;
     int minVisitsOnParent = minVisits * childrenNo;

     // If the parent node hasn't been visited a minimum
     // number of times, select the next appropriate child.
     if(minVisitsOnParent > node.visits) {
          return children.get(node.visits % childrenNo);
     }
     else if(minVisitsOnParent == node.visits) {
          // Sort all children once, the sorted state
          // which we'll keep updated afterwards with
          // minimal effort
          children.sort(nodeComparator);
     }
     else {
          // The first child is always the one updated;
          // so it is the only node we need to sort to
          // a new location.
          Node frontChild = children.first();

          // Determine its new location, often very near
          // to it's previous location.
          int i = 1;
          for ( ; i < children.size; i++){
               if(frontChild.score >= children.get(i).score)
                    break;
          }

          i--;

          // Move everyone by one, and set the child
          // at its newest index.
          if(i > 0) {
               if(i == 1) {
```

*(Continued)*

```
                    // Special case where we optimize for
                    // when we are just better than the
                    // second item. (often)
                    Object[] items = children.items;
                    items[0] = items[1];
                    items[1] = frontChild;
                }
                else {
                    Object[] items = children.items;
                    System.arraycopy(items, 1, items, 0, i);
                    items[i] = frontChild;
                }
            }
            else {
                  return frontChild;
            }
        }
return children.first();
}
```

## 28.7 Conclusion

This chapter introduced new insights and optimizations to tackle large and complex problems using MCTS. Any reader implementing MCTS is recommended to skim through the sections marked as implementation to find optimizations to the algorithm that can be used in any situation.

The main technique explored to deal with complexity is to break up a move into smaller actions, and provide these in order of apparent importance such that the strengths of MCTS may be used to explore and exploit the search space, judicially applying heuristics to various aspects of the algorithm to make it feasible.

Readers interested in the details of hierarchical expansion and concrete results on the comparison of the various techniques outlined in this chapter are referred to (Roelofs 2015), where a complete analysis is presented using the game of *Berlin*.

## References

Chaslot, G. Monte-Carlo Tree Search. Maastricht University, 2010.

Chaslot, G. M. J.-B., M. H. M. Winands, and H. J. van den Herik. Parallel Monte-Carlo Tree Search. In *Computers and Games*, eds. H. Jaap van den Herik, X. Xu, Z. Ma, and M. H. M. Winands. Berlin, Germany: Springer, pp. 60–71, 2008.

Churchill, D., and M. Buro. 2015. Hierarchical Portfolio Search: Prismata's Robust AI Architecture for Games with Large Search Spaces. *Proceedings of the Artificial Intelligence in Interactive Digital Entertainment Conference*. University of California, Santa Cruz, 2015.

Roelofs, G. Action space representation in combinatorial multi-armed bandits. Maastricht University, 2015.

Schadd, M. P. D., and M. H. M. Winands. Best reply search for multiplayer games. *IEEE Transactions on Computational Intelligence and AI in Games* 3(1): 57–66, 2011

Sturtevant, N. R. Monte Carlo tree search and related algorithms for games. In *Game AI Pro 2: Collected Wisdom of Game AI Professionals*, ed. S. Rabin. Boca Raton, FL: CRC Press, 2015, pp. 265–281.