Hierarchical Monte Carlo Tree Search Applied to a Turn-Based Multi-Agent Strategy Game

A.J.J. Valkenberg*
* *Department of Data Science & Knowledge Engineering*
Maastricht University
Email: a.valkenberg@student.maastrichtuniversity.nl

Abstract—**Games that have a combinatorial action space are a challenge for current search techniques. The Hierarchical Expansion (HE) algorithm is designed to deal with a combinatorial action space, but currently has two open problems. As a solution to HE's Partial Move Completion problem this report proposes the Side Informed Move Completion (SIMC) technique. To deal with the Dimensional Ordering problem in HE we present three heuristic techniques, and one online learning technique that uses the Side Information gathered by SIMC. The experiments show HE using SIMC to perform better than existing algorithms, within the domain of HunterKiller, a Turn-Based Multi-Agent Strategy game. We also show that for Dimensional Ordering the online learning technique outperforms all but one of the heuristic strategies.**

1. Introduction

When playing a game, players are predominantly concerned with finding a way to win. This can be described as a strategy: a plan of how to progress the game towards an end state that is beneficial to the player. For a game with perfect information it is possible to construct the entire state space and find a path to such an end state [1]. However, a game's state space can become too large to fully explore with current day hardware. When some aspects of a game's state are unknown, such as hidden cards in an opponent's hand or enemy units that are not visible to the player, assumptions and predictions have to be made regarding current and future states. This imperfect information increases a game's complexity, making the state space more difficult to explore. Poker is an example of a game with imperfect information where Neural Networks and Deep Learning strategies have been successfully applied [2]. Another example of state spaces that are difficult to explore are those that have a high branching factor. Where some games give the player the option to play a couple different moves each turn, some games allow hundreds of moves. This increases the number of states that can be reached from a certain game state. To combat state spaces with a high branching factor, a recent strategy used an Evolutionary Algorithm [3]. Search techniques attempt to focus on specific parts of the state space while exploring it; they evaluate states using a predetermined policy and ignore parts of the space that lead to suboptimal states [4]. Considerable progress has recently been made in the game of Go by using a combination of different strategies; Monte Carlo Tree Search (MCTS) aided by Deep Neural Networks [5].

Improvements to MCTS are still being researched, such as Hierarchical Expansion (HE) [6], specifically for domains that present a combinatorial action space. The Combinatorial Multi-Armed Bandit (CMAB) problem is an example of such a domain [7]. This problem presents the player with actions that are *combined-actions*: they consist of one or more *partial-actions*. Domains that can be formulated as a CMAB have multiple dimensions for which one or more options can be chosen each turn, leading to a combinatorial action space. An open problem with HE is that of Partial Move Completion: constructing a valid combined-action when a partial-action was selected for only a sub-set of all dimensions. To deal with this problem, we propose a new technique called Side Informed Move Completion (SIMC), which captures Side Information on partial-actions during the playout phase of MCTS and uses this information during Move Completion. Another open problem of HE is the order in which dimensions are expanded, which has a significant impact on the algorithm's performance. This report researches four heuristic strategies that order the available dimensions during the Expansion phase of MCTS, as well as one strategy that attempts to learn an ordering based on the captured Side Information.

HE was already shown to perform well in the game of Berlin. For this research, a Turn-Based Multi-Agent strategy game was created, named HunterKiller. An AI-Competition [8] has been created for HunterKiller, mainly as a learning project for students at the Department of Data Science and Knowledge Engineering at Maastricht University, but also as a way to discover strong heuristic strategies that can act as a performance baseline to compare the researched strategies to.

This report is structured as follows. Section 2 discusses the CMAB problem. Section 3 presents current techniques for approaching the CMAB problem. Section 4 presents the new techniques developed during this research. HunterKiller is discussed in detail in Section 5. Section 6 outlines the experiments conducted during this research and presents their results. This report is concluded in Section 7.

1.1. Internship

This research has been conducted during an internship at CodePoKE, directly supervised by Gijs-Jan Roelofs. The supervisor from Maastricht University was Mark Winands. All work took place between 01-12-2016 and December 19, 2017. The goal of this internship was to gain experience in programming AI for games and research various state-of-the-art techniques. We set out to achieve this goal by creating our own game, and implementing different strategies for it. The results of this research are intended to be used in the AI for an upcoming commercial game, Xenonauts2. We also implemented and set up an AI competition for HunterKiller, which could aid us in finding adequate strategies to use in the experiments.

2. Combinatorial Multi-Armed Bandit

The Multi-Armed Bandit (MAB) problem can be formulated as a slot-machine presenting the player with m arms, where the reward for choosing an arm is unknown prior to selecting it. The player is tasked with repeatedly choosing an arm to play each round and the objective is to get as close to the optimal reward as possible.

The Combinatorial Multi-Armed Bandit (CMAB) problem presents the player with a choice between super-arms as their play for a turn [7]. These super-arms represent a set of arms of the MAB. A player's combined-action becomes a set of partial-actions that represent a choice for an individual arm. A combined-action can consist of an arbitrary number of partial-actions. This model allows the reward function to be nonlinear.

When approaching the game HunterKiller as a CMAB problem, the Units and Structures a player controls are the dimensions and an action performed by either of these is a partial-action.

3. Related Research

This section describes the various strategies that were used in our experiments. Specific settings for some of these strategies are detailed in Section 6.

3.1. Monte Carlo Tree Search

Several of the strategies discussed in this report are based on, or are extensions of MCTS. MCTS is a best-first search that builds a search-tree in four phases [9]:

1) Selection: from the perspective of a node, select a child node according to a *selection-strategy*.
2) Expansion: from the perspective of a leaf node, create its children according to an *expansion-strategy*.
3) Playout: from the perspective of a leaf node, play out the game according to a *playout-strategy*.
4) Backpropagation: from the perspective of a leaf node, evaluate the simulated game state and propagate the evaluation back to the root node according to a *backpropagation-strategy*.

After a number of sequential iterations of these phases, the best child of the root node is selected as the final move, which is returned as the solution to the search.

A *selection-strategy* selects one of the child nodes of the node that it is invoked upon. MCTS starts by selecting the root node which represents the current state of the game. Upper Confidence Bound applied to Trees (UCT) [10] is the selection-strategy that is used by the strategies discussed in sections 3.4 and 4.1. UCT selects the next node by the following formula:

$$I_t = \max_{i \in K}\{\overline{R}_i + 2 \times C \times \sqrt{\frac{\ln V_p}{V_i}}\} \tag{1}$$

In Equation 1, I_t is the node selected at round t from the set of available nodes K, $\overline{R}_i$ is the average evaluation for node i, V_i is the number of times node i has been visited and V_p is the number of times i's parent node p has been visited. The constant C has to be tuned experimentally.

An *expansion-strategy* dictates when and how the child nodes of a node should be generated. The children of a node represent the set of game states that can be reached from the current node. An example of an expansion-strategy is to only expand a node after it has been visited T number of times. The strategies discussed in Subsections 3.4 and 4.1 extend upon this expansion-strategy.

The *playout-strategy* starts with a copy of the game state as stored in the leaf node and continuously produces moves and executes them, moving the game forward until it reaches an end state or stop condition. A trivial strategy is to create random moves, but this can weaken the resulting play strength [9]. The experiment discussed in Subsection 6.5 compares the performance of a random and a heuristic playout strategy.

A *backpropagation-strategy* takes the evaluation score of a simulated game state from a leaf node and moves back up the game tree to the root node. Each node visited during this process has its value updated with the evaluation score. A straightforward way to calculate the value of a node is to take the average score over the number of visits.

3.2. Naïve Monte Carlo

Naïve Monte Carlo (NMC) is an algorithm designed to solve the CMAB problem, using the Naïve Sampling process [11].

This process works under the naïve assumption that partial-actions do not influence each other, i.e. the reward function for a combined-action can be rewritten as summation of rewards for individual partial-actions. Each iteration of the process a choice is made whether to *Explore* or *Exploit*, according to a policy π_0. If Explore is chosen, a partial-action is selected for each dimension independently according to a policy π_l, forming a combined-action which is sampled and stored along with its value in a set of combined-actions, G. If Exploit is chosen, a combined-action is selected from G according to a policy π_g and sampled. At the end of the search the combined-action in G with the best value is returned as the solution.

In our experiments, the policies π_0, π_l and π_g were all ϵ-greedy policies.

3.3. Linear Side Information

Linear Side Information (LSI) divides its search process into two phases: Explore and Exploit, each having a separate iteration budget [12].

In the Explore phase, the set of all possible combined-actions is reduced to a relatively small sub-set. To generate this sub-set a table of Side Information [13] is created for each dimension and it's available partial-actions. This table contains the average reward after sampling each dimension's available partial-actions an equal number of times, based on the available budget. To sample a partial-action a playout is performed either within an otherwise empty combined-action, or one that consists of randomly generated partial-actions for the other dimensions. When this Side Information per dimension is normalised it can be used as a probability distribution for selecting partial-actions during the generation of the sub-set of combined-actions.

In the Exploit phase, the generated sub-set of combined-actions is reduced by using Sequential Halving. This algorithm samples each combined-action in the sub-set a number of times according to the available budget and then removes half of the combined-actions; those that performed worse on average. This continues until only one combined-action remains, which is returned as the solution to the search.

The Sequential Halving algorithm as presented in [14] is shown below.

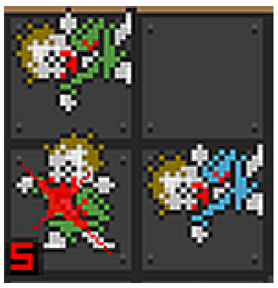
```
  input : total budget $T$
1 initialise $S_0 \leftarrow [n]$
2 for $r = 0$ to $\lceil \log_2 n \rceil - 1$ do
3     sample each arm $i \in S_r$ for $t_r = \lfloor \frac{T}{|S_r| \lceil \log_2 n \rceil} \rfloor$
        times, and let $\hat{p}_i^r$ be the average reward
4     $S_{r+1} \leftarrow$ the set of $\lceil |S_r|/2 \rceil$ arms in $S_r$ with the
        largest empirical average
5 end
  output: arm in $S_{\lceil \log_2 n \rceil}$
```

Algorithm 1: Sequential Halving

3.4. Hierarchical Expansion

Hierarchical Expansion (HE) is an extension of MCTS and designed to cope with the action space of the Combinatorial Multi-Armed Bandit problem [6].

In an effort to deal with a high branching factor as a result of the combinatorial action space, HE modifies the Expansion phase of MCTS. It expands into the partial-actions for one dimension per level, which means one ply can encompass multiple levels. How this expansion changes the structure of the search tree can be seen in Figure 1 and 2 where a default MCTS-expansion and HE are compared. These figures are based on those presented in [6]. The combined-action that is constructed in a single level in


(a) Blue player attacks the green unit from 10 health to 5.

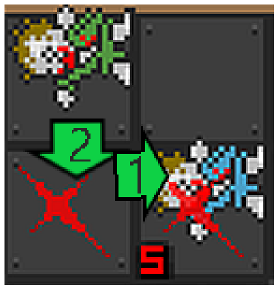

(b) Green player kills their own unit to deny the opponent a kill on the next turn.

Figure 3: Action sequences in HunterKiller.

MCTS is in HE instead incrementally created and stored in the tree's nodes. HE assumes that some ordering for selecting which dimension to expand into is known or that such an ordering can be learned during the search.

As a solution to the search, HE will select the best child nodes of the root until a combined-action is constructed that contains a partial-action for all dimensions. This approach allows exploration on the level of partial-actions. When a game has partial-actions that are significantly more valuable than others, HE will allow the search to be focussed on branches of the tree that contain these valuable partial-actions. Because HE assumes an order in the dimensions is known or can be learned, it can exploit high-value action-sequences that are represented by a non-linear component in the reward function, such as the one shown in Figure 3. Because of these sequences, HE is able to let go of the naïve assumption of linear separability in the reward function.

4. Proposed Techniques

This section describes the new techniques developed during this research. We start by proposing a new Move Completion technique for HE, after which we discuss several strategies for Dimensional Ordering in HE.

4.1. Side Informed Move Completion

Because HE expands over partial-actions it can no longer guarantee that MCTS will return a complete combined-action, i.e. containing a partial-action for each dimension [6]. We propose a solution to this problem called Side Informed Move Completion (SIMC).

SIMC creates a table containing the average evaluation score for each partial-action, for each available dimension in the root state. The data in this table is referred to as Side Information [13]. In the playout phase, if the current combined-action is incomplete, a randomly selected partial-action is chosen for each undecided dimension. These dimension-*partial-action* pairs are stored if the current simulation is run from a state which is still in the same ply as the root state. In the Backpropagation phase, the Side Information for each stored dimension-action pair is updated

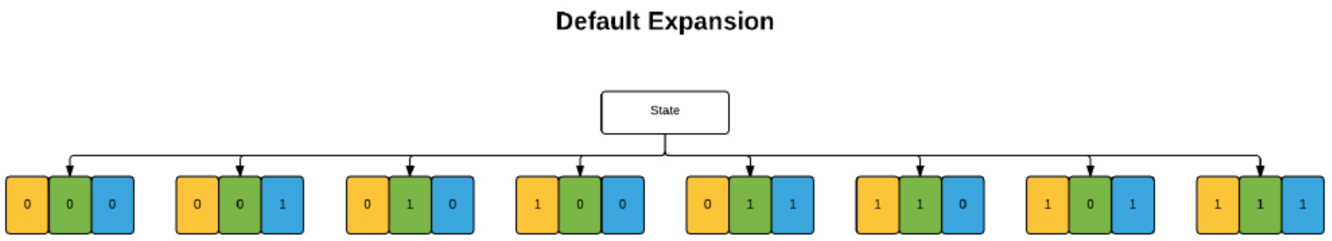


Figure 1: Example of default expansion of a combinatorial action space.

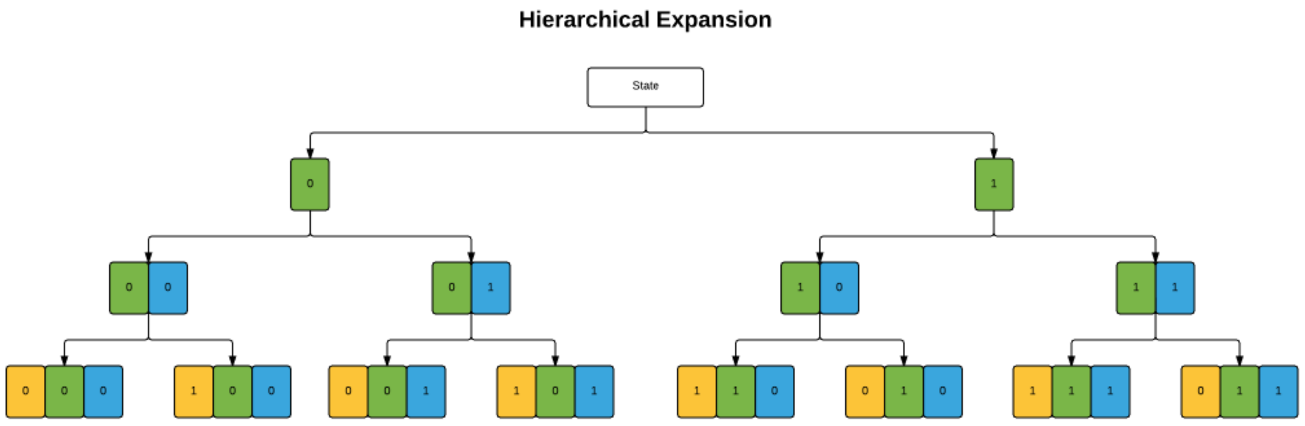


Figure 2: Example of hierarchical expansion of a combinatorial action space.

with the evaluation score. When the solution of the search is created and the combined-action in the selected leaf-node is not complete, SIMC will fill the combined-action with the best scoring partial-actions for each undecided dimension. It should be noted that the Side Information is not retained between searches.

```
1 During Backpropagation:
  input : Filled Actions $fA$, Side Information $sI$,
          Evaluation $E$
  if ply == 1 then
      forall $a \in fA$ do
          $sI(D_a, a) = sI(D_a, a) + E$
          increment visit count for $sI(D_a, a)$
      end
  end

  During Solution:
  input : Combined Action $cA$, Side Information $sI$
  if $D \notin cA$ then
      $cA(D) \leftarrow \underset{a}{\text{argmax}} \frac{sI(D_a,a)}{visits}$
  end
```

Algorithm 2: Side Informed Move Completion

4.2. Dimensional Ordering Strategies

The ordering of dimensions is fundamental to HE [6]. The following subsections discuss several ordering strategies developed in an effort to find an optimal ordering.

4.2.1. Random-per-Expansion. As a baseline for performance, we include a strategy that at each expansion expands a randomly selected dimension that has not been expanded yet in the current ply. When an optimal ordering is not known, this strategy tries to maximize its chances to find the optimal dimension by randomizing this choice each time. A disadvantage of this strategy is that if a suboptimal dimension is selected, some of MCTS's iteration budget will effectively be lost to exploring it before a new and perhaps better dimension can be reached. Roelofs already found such a strategy to perform well [6], which lead us to use it as a base of comparison in our experiments.

4.2.2. Random-per-Turn. This strategy creates a random ordering over the dimensions at the root node and retains this throughout the search. When the search is started, an ordering for the currently available dimensions is randomly created and fixed. At each expansion the next dimension in the fixed order is expanded, if it is available (i.e. the unit is still alive). At the start of a new ply any dimensions that are not present in the fixed ordering are randomly ordered and appended to a copy of it, which is used in the subtree. This strategy tries to alleviate the disadvantage of the Random-per-Expansion strategy by fixing the order of dimensions it expands into at the start of plies the same, so that a larger part of the iteration budget can be spent on the root dimensions.

4.2.3. Least Distance. This heuristic strategy orders the dimensions by their distance from any known enemy position. Units that are close to the enemy have a higher chance to get into combat, which in presumed to be crucial to the outcome of the game. Prioritising these dimensions puts an emphasis on finding the best actions in an engagement.

4.2.4. Attack. Another heuristic strategy, this orders the dimensions by whether or not they are able to attack any enemy. While quite similar to the Least Distance ordering, this strategy focusses even more on immediate combat actions. It is conceivable that an engagement would occur close to the friendly base while a scouting force is exploring an area away from the base. In such a game state the Least Distance heuristic would prioritise the scouting force, while Attack prioritises the defending units.

4.2.5. Informed. This strategy takes advantage of the Side Information created by the SIMC algorithm. It orders the dimensions by an entropy measure of their available partial-actions, in our case the Shannon Entropy [15], which is defined as:

$$H = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{2}$$

When a dimension has a high entropy for the average scores of its available partial-actions, it indicates that there is a significant choice to be made, i.e. there is a significant disparity between the average evaluations. Adhering to such an ordering ensures that dimensions that have a significant difference to the outcome of their actions are expanded before dimensions that do not. Note that by using the Side Information, which is constructed alongside the search tree, this strategy attempts to learn the optimal ordering during the execution of the search.

5. Research Domain

This section explains the research domains in detail. It starts with the game Bokkenschieten, which was used as an introductory domain to gain acquaintance with CodePoKE's AI framework. The game HunterKiller is the domain for which the strategies discussed in Section 4 have been implemented.

5.1. Bokkenschieten

Bokkenschieten is a card game, played with 3 to 6 players. The goal of the game is to collect the highest number of 'goats', while not going over a specific number. The game consists of a deck of cards, uniquely numbered from 1 to 50. There is also a deck of 12 island-cards and a small wooden goat. At the start of a game, each player is randomly dealt 8 cards from the deck. The island-cards are not dealt out to anyone. A game-round consists of each player, in a specific order, playing one card from their hand. Therefore a single game of Bokkenschieten takes 8 rounds. The player that played the highest ranking card in a round, wins that round. Each card represents an amount of goats, as displayed on the card. The winning player collects the cards that were played during that round and adds the combined amount of goats to his total. During the first four rounds of a game, one of the island-cards is drawn from the deck by the player that played the lowest ranking card. An island-card has two quarters of an island on its face side. It is up to the player that drew the island card to decide which side will be used to make up an island. This island is constructed in the first four game-rounds by the four chosen quarters. Each quarter also shows a number: this represents the maximum amount of goats that can be held on that quarter of the island. So after round 4, the island will be complete and a maximum number (or capacity) of goats that the island can hold will be known. At the end of the 8th round, each player adds up the total number of goats they have collected by winning rounds. A player's score at the end of the game will be that number if it is lower or equal to the island maximum. If a player collected too many goats, they scores 0. A special case is when a player managed to collect no goats at all (i.e. won no round), then they will receive a score equal to the chosen number of the first island-quarter. The game is a multi-player game with imperfect information. The game tree complexity is approximately 3.5×10^{53}. The state-space complexity would be including the island-cards, but this has not been calculated yet. This research domain was used to get acquainted with CodePoKE's AI framework.

5.2. HunterKiller

This section starts by outlining the rules for HunterKiller. We then discuss the LoS system that has been implemented. Next we present the types of units and structures, and the types of actions they can be ordered. Then we discuss some optimizations that have been implemented to aid with research, before reviewing the game's state evaluation method. We end this subsection by briefly exploring HunterKiller's game complexity.

5.2.1. Rules. While HunterKiller can be played by between 2 and 4 players, this report focusses on the 2 player variant. A game of HunterKiller ends when one player has defeated all of their opponents, or the maximum number of 200 rounds is reached. A player is considered defeated when the structure that is defined by the map setup as their Command Center is destroyed. When a player is defeated, any units they were controlling are removed from the map and their points total is halved. The player with the highest number of points at the end of the game wins, or in the case of an equal amount a draw is declared. Points can be gained by destroying enemy units, or by controlling Objectives on the map. Units can be spawned at Bases or Outposts,

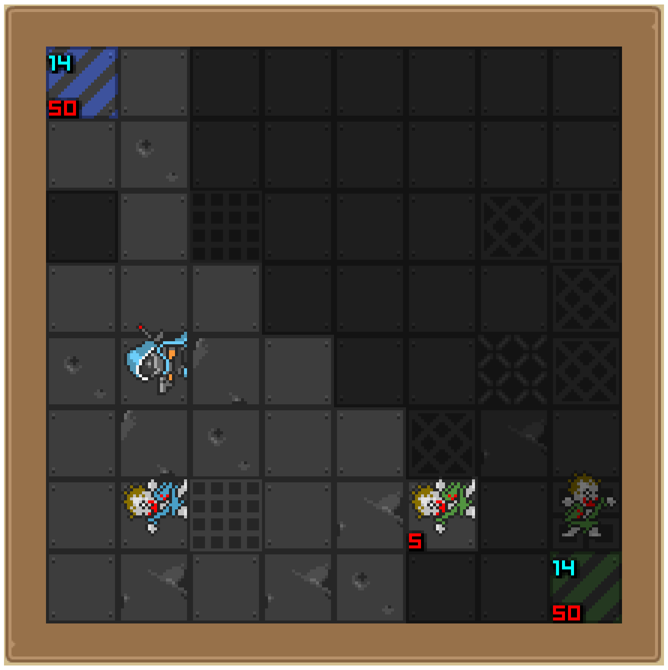


Figure 4: Example of an empty 8-by-8 HunterKiller board. Shaded squares indicate locations that are outside of the active player's Field-of-View

situated on the map, for a resource cost. Each player starts the game with an equal amount of resources, but during the game extra resources will be generated by Bases, Outposts and Strongholds, for the player that is controlling them. The game is played sequentially, which means a single round consists of each player being asked for input in a predetermined order. A player's input consists of at most one order for each object they control (i.e. Units and Structures). If an object is not issued an order, it will do nothing.

5.2.2. Line-of-Sight. HunterKiller's LoS system is based on an algorithm designed by Milazzo [16]. Each type of unit has a Field-of-View (FoV) angle and range characteristic that, together with the LoS system, determines which locations are visible to the unit. Structures have a FoV of 1 square around its location. A player's FoV consists of all its controlled objects' FoV combined, introducing imperfect information to the game. See Figure 4 for an example of a HunterKiller board where the LoS system is active.

5.2.3. Objects and Actions. A player can control two main type of objects: Units and Structures. There are three types of units: Soldier, Medic and Infected. A unit can be ordered to execute one of the following:

- Move 1 square in a cardinal direction.
- Rotate its orientation clockwise or counter-clockwise.
- Attack a location within its range.
- Use its special ability.

Note that units cannot move to a square that is being occupied by another unit. Attacking a location deals damage to any unit or structure on that location. Special abilities adhere to a cooldown period that is different for each type of unit. A Soldier's special ability is an attack that affects an area 3 squares wide and 3 squares long, dealing damage to anything within the area. A Medic's special ability is a beneficial action that heals any unit on the targeted location. Unlike the other two types, an Infected's special ability cannot be ordered; it will automatically be executed when the Infected destroys a non-Infected unit (ability cooldowns are still taken into account). When this happens, an Infected will be spawned on the location of the destroyed unit, controlled by the same player that controls the Infected which destroyed the unit. A structure can be ordered to spawn a unit of a specific type if the player has enough resources to pay for that type of unit. There are two types of structures that can spawn units: Base and Outpost. These types are identical, except that the Base is a player's Command Center by default (and therefore cannot be captured by other players) and spawns units in an adjacent square, where the Outpost spawns units on its own location. There are two more types of structures: Stronghold and Objective. These types have no function other than generating an advantage for the player that controls them; a Stronghold generates resources and an Objective generates score points. Structures generate their respective bonus once every 5 rounds.

5.2.4. Research optimizations. Some optimizations to the code framework of HunterKiller have been implemented to increase speed of a single game when run in simulation mode. One optimization is the caching of FoV per location, based on unit type and orientation. The LoS calculations are frequently called during the simulation of a single game and are CPU-intensive. Another optimization is to provide the playout strategy with a subset of all legal attack orders in order to speed up the playout games. Removed orders include, but are not limited to:

- Attacking an empty location.
- Attacking an allied structure.
- Healing an enemy.
- Healing a unit that is not damaged.

The bot PlayoutBot randomly chooses an order from this subset for each dimension.

These optimizations allow a standard MCTS agent, using PlayoutBot as a playout strategy, to run approximately 2000 iterations per second (as measured on the author's machine). See Section 6 for a more detailed description of the experimental set up.

5.2.5. State evaluation. Several of the strategies discussed in Section 3 use a state evaluation in their process. For HunterKiller we have defined the following state evaluation function:

$$E(s) = F(s) + 4 \times (\Delta S/Uw)^3 + 10 \times Ua^3 + Pr + FoV \quad (3)$$

Note that each term is evaluated from the root player's perspective. $F(s)$ is function that returns a value representing a game win if state s is a win, a value representing a game loss if state s is a loss, or 0 otherwise. ΔS is the difference in scores. Uw is the average score points a unit is worth when destroyed by the enemy team. Ua is the number of allied units present in the state. Pr is the lowest distance of any allied unit to any enemy structure. FoV is the number of locations that are currently in the player's Field-of-View.

The evaluation scores are normalised by a sigmoid function.

5.2.6. Complexity. While calculating exact numbers for HunterKiller's state-space and game-tree complexity was outside the scope of this research, we would still like to estimate the game-tree complexity within the context of our experiments. During the optimization of the code framework, some statistics were recorded. The average number of turns for a single game is 150 and the average amount of units under the control of one player is 8. Since each unit has 8 possible actions each turn, we can estimate the game-tree complexity to be:

$$(8^8)^{150} = 5.1 \times 10^{1083} \tag{4}$$

As a comparison: Shannon calculated the game-tree complexity of Chess to be 1×10^{120} [17].

6. Experiments and Results

This section presents the experiment for the game of Bokkenschieten, as well as the experimental setup and various experiments conducted to research the proposed improvements to HE.

6.1. Bokkenschieten: Playout Improvements

In this experiment, we compare the performance of several strategies for the game of Bokkenschieten:

- Random play
- MCTS using a pure random playout
- MCTS using an epsilon greedy mixed playout strategy

The mixed strategy uses a random playout 80% of the time and a heuristic playout 20% of the time. The heuristic playouts are performed by a heuristic bot. The settings for MCTS are similar to those detailed in Subsection 6.2.

Table 1 shows the results of 500 4-player games of Bokkenschieten, where two players use the Random strategy and the MCTS-Random and MCTS-Mixed strategies are played by one player each. Note that the number of game wins does not add up to 500 because of the possibility of two players attaining the same number of points in a single game, making it a draw. The amount of points scored by the mixed-playout strategy is significantly higher than that of the random-playout at the 99% confidence interval, even though the number of game wins is not. This would suggest that adding higher quality playouts increases the performance of MCTS.

6.2. Experimental Setup

All experiments are run on an empty 8-by-8 HunterKiller map, with each player's base on opposite corners of the map.

For the algorithms that use MCTS as a base strategy, its iteration budget is set to 1000, with playouts being limited to run 20 turns ahead and are performed by PlayoutBot. It should be noted that this iteration budget is low compared to research focussed on MCTS [9], but similar to what was used in [11] and [12].

After some experimentation, the C-constant used in UCT was set to 0.1 for the HE algorithm. Initial tests showed that a value between 0 and 0.2 achieved the best results, after which this range was more closely tested. See Figure 5 for the results of these tests. A greedy approach was to be expected to work well due to HE being able to exploit significant dimensions more, therefore reaching deeper into the tree.

The ϵ-greedy strategies used by NMC use an ϵ of 0.25.

The iteration budgets T_g and T_e used by LSI are set to $0.25N$ and $0.75N$ respectively. Like MCTS, LSI's playouts are also limited to run 20 turns ahead.

The playout agent used by all algorithms is the bot that was optimized for run speed, as discussed in Subsection 5.2.4.

The heuristic bot used in these experiments is the bot that the authors submitted to the first AI-competition for HunterKiller.

6.3. Side Informed Move Completion in HE

To test the impact of SIMC in the HE algorithm we set up an experiment where 200 games are run pitting a bot using SIMC against one that completes its incomplete combined-actions randomly. Both bots use the same dimensional ordering strategy and PlayoutBot as their playout strategy.

The results of this experiment are presented in Table 2. Because the iteration budget for MCTS is relatively low for a domain of HunterKiller's complexity, in the mid and late stages of a game not all dimensions will be able to be fully explored. SIMC allows HE to learn which partial-actions have higher expected value than others during the Simulation phase, leading to better quality partial-actions being selected during Move Completion. Reviewing the games revealed that the bot using SIMC won by elimination (i.e. destroying its opponent's base) in 191 of them. In only one game did its own base come under attack, suggesting that the bot using SIMC attained strong control over the board.

6.4. Dimensional Ordering in HE

Subsection 4.2 discussed various strategies for ordering the dimensions during a HE search. To review the effectiveness of these different strategies, a round-robin tournament of 200 games was run with bots where the only difference between them was which dimensional ordering strategy they

	Total Score	Avg. Score	Game Wins
Random	6635	6.635	139
MCTS-Random	4137	8.274	154
MCTS-Mixed	4379	8.758	156

TABLE 1: Results for Experiment 6.1.

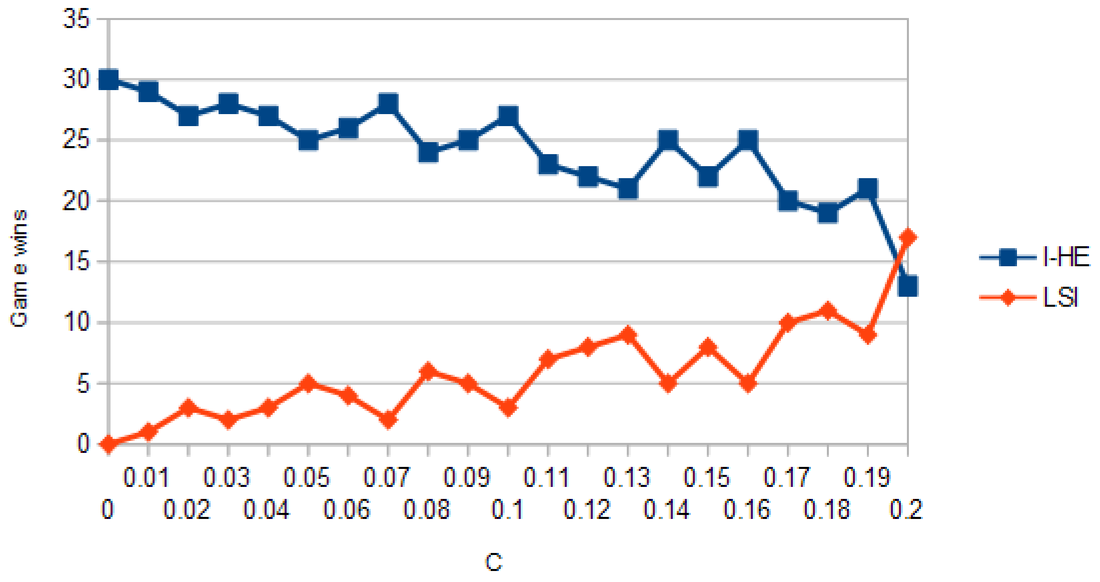


Figure 5: Game wins in 30 games of I-HE versus LSI. The C-constant used in UCT varies between 0 and 0.2.

	Game Wins
I-HE	200
HE	0

TABLE 2: The amount of game wins of a bot using SIMC versus a bot using random Move Completion (Experiment 6.3).

used in HE. All bots use PlayoutBot as their playout strategy. Because the Informed strategy uses the Side Information data collected by SIMC, all bots use the I-HE algorithm. Table 3 shows the results of this experiment.

We see that the Random-per-Turn ordering performs best, significantly better than Informed at the 99% confidence interval. The results also show that the LeastDistance ordering performs significantly better than Attack at the 95% confidence interval, but not significantly better than the Random-per-Expansion ordering even at a 90% confidence interval.

The performance of the heuristic strategies could be explained by the often occurring cluttered board states in HunterKiller. When around 15 to 20 units are available to both players, as can happen in the later stages of a game, many units will meet the requirements of these heuristic strategies. This abundance of viable dimensions to expand into can regress them back to being close to random.

Ordering the dimensions by high entropy between their available partial-actions achieves a significant performance increase over a Random-per-Expansion ordering. Because MCTS spends more time exploring these valuable dimensions, it can more often exploit good partial-actions or avoid bad ones.

Another 200-game tournament was created, this time using HE without SIMC. The Informed strategy was not included in this tournament, due to the fact that no Side Information is gathered. Table 4 shows the results of this tournament. Once again the LeastDistance ordering performs significantly better than the Attack ordering at the 99% confidence interval. However, the Random-per-Turn ordering now performs not even significantly better than the Random-per-Expansion ordering at the 90% confidence interval. This change in performance shows the influence of SIMC: it provides a baseline of performance for even sub-optimal orderings, but when a good ordering is selected by the Random-per-Turn strategy it can take full advantage of exploiting this good ordering ánd have its move completed with high value partial-actions.

6.5. Effect of Playout Strategies

A key part of the MCTS algorithm is the playout phase. While playing out a game in a random fashion can result in an accurate assessment of the position given enough iterations, when that resource is limited, it stands to reason that a higher quality playout strategy would lead to better performance. To test this, a round-robin tournament was

	Random-per-Expansion	Random-per-Turn	LeastDistance	Attack	Informed	Win	Loss	Win-%
Random-per-Expansion	-	36	103	98	49	286	514	35.8%
Random-per-Turn	164	-	163	157	122	606	194	75.8%
LeastDistance	97	37	-	118	60	312	488	39.0%
Attack	102	43	82	-	43	270	530	33.8%
Informed	151	78	140	157	-	526	274	65.8%

TABLE 3: Results for the Dimensional Ordering experiment (6.4) using SIMC. Rows represent game wins versus Column algorithm.

	Random-per-Expansion	Random-per-Turn	LeastDistance	Attack	Win	Loss	Win-%
Random-per-Expansion	-	106	1	3	110	490	18.3%
Random-per-Turn	94	-	1	1	96	504	16.0%
LeastDistance	199	199	-	146	544	56	90.7%
Attack	197	199	54	-	450	150	75.0%

TABLE 4: Results for the Dimensional Ordering experiment (6.4) without SIMC. Rows represent game wins versus Column algorithm.

set up between the three general approaches discussed in this paper; I-HE, NMC and LSI. Each strategy participated with a variant using the PlayoutBot and a variant using the HeuristicBot, as their playout strategy. The I-HE algorithm used the LeastDistance dimensional ordering strategy. A match between strategies consisted of 100 games. Table 5 shows the results of this experiment.

These results show that increasing the quality of moves made during the Simulation phase (or playouts) increases playing strength. All three approaches perform significantly better when using heuristic playouts at the 99% confidence interval.

It should be noted that using the HeuristicBot as a playout strategy significantly increases computation time. The exact increase was not recorded, but observations during the experiments indicate this increase to be at least tenfold.

6.6. Algorithm Performance

In this experiment the performance of the SIMC enhancement to the HE algorithm is tested relative to the other approaches discussed in this paper. The following algorithms are tested in a round-robin tournament where each pair played 200 games:

- PlayoutBot, the bot used during the playouts in various strategies, see Subsection 5.2.4.
- HeuristicBot, a bot using a set of heuristics to select its actions.
- HE$_{re}$, Hierarchical Expansion using the Random-per-Expansion dimensional ordering.
- HE$_{rt}$, Hierarchical Expansion using the Random-per-Turn dimensional ordering.
- I-HE$_{re}$, Hierarchical Expansion using SIMC and the Random-per-Expansion dimensional ordering.
- I-HE$_{rt}$, Hierarchical Expansion using SIMC and the Random-per-Turn dimensional ordering.
- I-HE$_{i}$, Hierarchical Expansion using SIMC and the Informed dimensional ordering.
- NMC, Naïve Monte Carlo Search.
- LSI, Linear Side Information.

All bots are using PlayoutBot as their playout strategy. Table 6 shows the results of this experiment.

From these results we can see that the HeuristicBot and I-HE$_{rt}$ algorithm perform best. Although the difference between their overall performance is not significant at the 99% confidence interval, it should be noted that the HeuristicBot did beat the I-HE$_{rt}$ algorithm by a significant margin in their match. The algorithms using SIMC perform better than LSI, NMC, HE without SIMC and the PlayoutBot, but cannot consistently beat the HeuristicBot. Only LSI reaches an even match up against the HeuristicBot but loses to the HE algorithms using SIMC.

The worst performing strategies in this experiment are the HE algorithms that do not use SIMC. This could be caused by the low iteration budget restricting the depth that these strategies can reach in the game tree, causing the Partial Move Completion problem, which SIMC is designed to handle. The difference in performance between the two dimensional orderings is not significant at the 99% confidence interval.

6.7. Effect of Line-of-Sight on Performance

As a final experiment we tested the effect of removing the LoS-system. This essentially transforms HunterKiller into a game with perfect information. We decided to take two of the best performing strategies in Experiment 6.6 and review if the outcome of their match changes significantly with perfect information. These bot use the PlayoutBot as their playout strategy. Table 7 shows the results of this experiment.

The difference between these results and those in Experiment 6.6 are not significant at the 99% confidence interval. This can be explained by the observation that the best performing strategies favour the Infected unit. It is strong in close range combat but also has a large 360 degree Field-of-View. Spreading out enough Infected units across the board can essentially emulate perfect information, since a large portion of the board will be visible to the player.

	I-HE$_{random}$	I-HE$_{heuristic}$	LSI$_{random}$	LSI$_{heuristic}$	NMC$_{random}$	NMC$_{heuristic}$	Win	Loss	Win-%
I-HE$_{random}$	-	13	78	1	99	74	265	235	53.0%
I-HE$_{heuristic}$	87	-	96	27	97	81	388	112	77.6%
LSI$_{random}$	22	4	-	4	98	96	224	276	44.8%
LSI$_{heuristic}$	99	73	96	-	99	90	457	43	91.4%
NMC$_{random}$	1	3	2	1	-	35	42	458	8.4%
NMC$_{heuristic}$	26		4	10	65	-	124	376	24.8%

TABLE 5: Game wins for varying playout strategies (Experiment 6.5). Rows represent game wins versus Column algorithm.

	PlayoutBot	HeuristicBot	HE$_{re}$	HE$_{rt}$	I-HE$_{re}$	I-HE$_{rt}$	I-HE$_{i}$	NMC	LSI	Win	Loss	Win-%
PlayoutBot	-	0	189	193	0	0	0	130	34	546	1054	34.1%
HeuristicBot	200	-	199	200	162	145	144	188	95	1333	267	83.3%
HE$_{re}$	11	1	-	106	0	0	0	5	0	123	1477	7.7%
HE$_{rt}$	7	0	94	-	0	0	0	9	0	110	1490	6.9%
I-HE$_{re}$	200	38	200	200	-	36	49	192	164	1079	521	67.4%
I-HE$_{rt}$	200	55	200	200	164	-	122	194	183	1318	282	82.4%
I-HE$_{i}$	200	56	200	200	151	78	-	193	180	1258	342	78.6%
NMC	70	12	195	191	8	6	7	-	3	492	1108	30.8%
LSI	166	105	200	200	36	17	20	197	-	941	659	58.8%

TABLE 6: Results of 200 games between each algorithm pair (Experiment 6.6). Rows represent game wins versus Column algorithms.

	Game Wins
I-HE$_{rt}$	191
LSI	9

TABLE 7: Results of 200 games with HunterKiller's Line-of-Sight system deactivated (Experiment 6.7).

7. Conclusion

This report presented new approaches to the open problems of the Hierarchical Expansion algorithm, a Search Technique designed to cope with the action space of the Combinatorial Multi-Armed Bandit problem. Side Informed Move Completion was proposed as a solution to the problem of Partial Move Completion and shown to outperform a HE algorithm that did not use SIMC. To solve the Dimensional Ordering problem four heuristic orderings were discussed, as well as one ordering strategy that uses the Side Information gathered by SIMC for ordering. When the HE algorithm does not use SIMC, the heuristic orderings perform significantly better than the two Random orderings. While using the Side Information to complete partial moves, the Informed ordering outperforms both the LeastDistance and Attack orderings. However, the Random-per-Turn ordering improves a significant amount from the addition of Side Information, even performing better than the Informed ordering. Future work could research the effects of increasing the iteration budget, which should reduce the influence of Move Completion strategies due to the fact that the search will be able to reach deeper into the tree.

Experiment 6.6 showed that the I-HE$_{rt}$ algorithm provided the best play strength in the research domain, but it was not able to beat the best heuristic strategy consistently. Preliminary results show that by using heuristic playouts the I-HE is able to defeat the heuristic bot. The HE strategies performed poorly, this could be related to the low number of iterations preventing them from reaching deep into the game tree and therefore having to randomly complete its moves, a problem that SIMC is shown to fix.

Heuristic playouts improve performance over random playouts, but at a cost of increased calculation time.

Disabling Line-of-Sight does not significantly change the difference in performance between strategies.

Acknowledgements

The author would like to thank Gijs-Jan Roelofs for his unending patience and support, and Mark Winands for his timely advice and supervision.

References

[1] H. J. van den Herik, J. W. H. M. Uiterwijk, and J. van Rijswijck, "Games solved: Now and in the future," *Artificial Intelligence*, vol. 134, no. 1, pp. 277–311, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370201001527

[2] M. Moravcík, M. Schmid, N. Burch, V. Lisý, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, and M. H. Bowling. (2017, Jan.) Deepstack: Expert-level artificial intelligence in no-limit poker. Computing Research Repository. [Online]. Available: http://arxiv.org/abs/1701.01724

[3] N. Justesen, T. Mahlmann, and J. Togelius, "Online evolution for multi-action adversarial games," in *Proc. Applications of Evolutionary Computation (EvoApplications 2016)*, Porto, Portugal, 2016, pp. 590–603. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-31204-0_38

[4] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial Intelligence*, vol. 6, no. 4, pp. 293–326, 1975. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0004370275900193

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel,

and D. Hassabis, “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, pp. 484–489, 2016. [Online]. Available: http://dx.doi.org/10.1038/nature16961

[6] G.-J. Roelofs, “Action space representation in combinatorial multi-armed bandits,” Master’s thesis, Maastricht University, Department of Knowledge Engineering, Maastricht, The Netherlands, Jul. 2015. [Online]. Available: https://project.dke.maastrichtuniversity.nl/games/files/msc/Roelofs_thesis.pdf

[7] W. Chen, Y. Wang, and Y. Yuan, “Combinatorial multi-armed bandit: General framework and applications,” in *Proceedings of the 30th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, S. Dasgupta and D. McAllester, Eds., vol. 28, no. 1. Atlanta, Georgia, USA: PMLR, 17-19 Jun 2013, pp. 151–159. [Online]. Available: http://proceedings.mlr.press/v28/chen13a.html

[8] CodePoKE. Hunterkiller ai-competition. [Online]. Available: ai.codepoke.net

[9] G. M. J.-B. Chaslot, M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and B. Bouzy, “Progressive strategies for monte-carlo tree search,” *New Mathematics and Natural Computation*, vol. 4, no. 3, pp. 343–357, 2008. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S1793005708001094

[10] L. Kocsis and C. Szepesvári, “Bandit based monte-carlo planning,” in *ECML-06*. Springer, 2006, pp. 282–293. [Online]. Available: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.102.1296

[11] S. Ontañón, “The combinatorial multi-armed bandit problem and its application to real-time strategy games,” in *Proc. Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2013)*, Boston, USA, 2013, pp. 58–64. [Online]. Available: https://www.aaai.org/ocs/index.php/AIIDE/AIIDE13/paper/download/7377/7589

[12] A. Shleyfman, A. Komenda, and C. Domshlak, “On combinatorial actions and CMABs with linear side information,” in *Proc. 21st European Conference on Artificial Intelligence (ECAI 2014)*, 2014, pp. 825–830. [Online]. Available: https://doi.org/10.3233/978-1-61499-419-0-825

[13] J. Chen, X. Liu, and S. Lyu, *Boosting with Side Information*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 563–577. [Online]. Available: https://doi.org/10.1007/978-3-642-37331-2_43

[14] Z. Karnin, T. Koren, and O. Somekh, “Almost optimal exploration in multi-armed bandits,” in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, S. Dasgupta and D. Mcallester, Eds., vol. 28, no. 3. JMLR Workshop and Conference Proceedings, May 2013, pp. 1238–1246. [Online]. Available: http://jmlr.org/proceedings/papers/v28/karnin13.pdf

[15] C. E. Shannon, “A mathematical theory of communication,” *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948. [Online]. Available: http://dx.doi.org/10.1002/j.1538-7305.1948.tb01338.x

[16] A. Milazzo. Roguelike vision algorithms. [Online]. Available: http://www.adammil.net/blog/v125_Roguelike_Vision_Algorithms.html

[17] C. E. Shannon, *Programming a Computer for Playing Chess*. New York, NY: Springer New York, 1988, pp. 2–13. [Online]. Available: http://dx.doi.org/10.1007/978-1-4757-1968-0_1