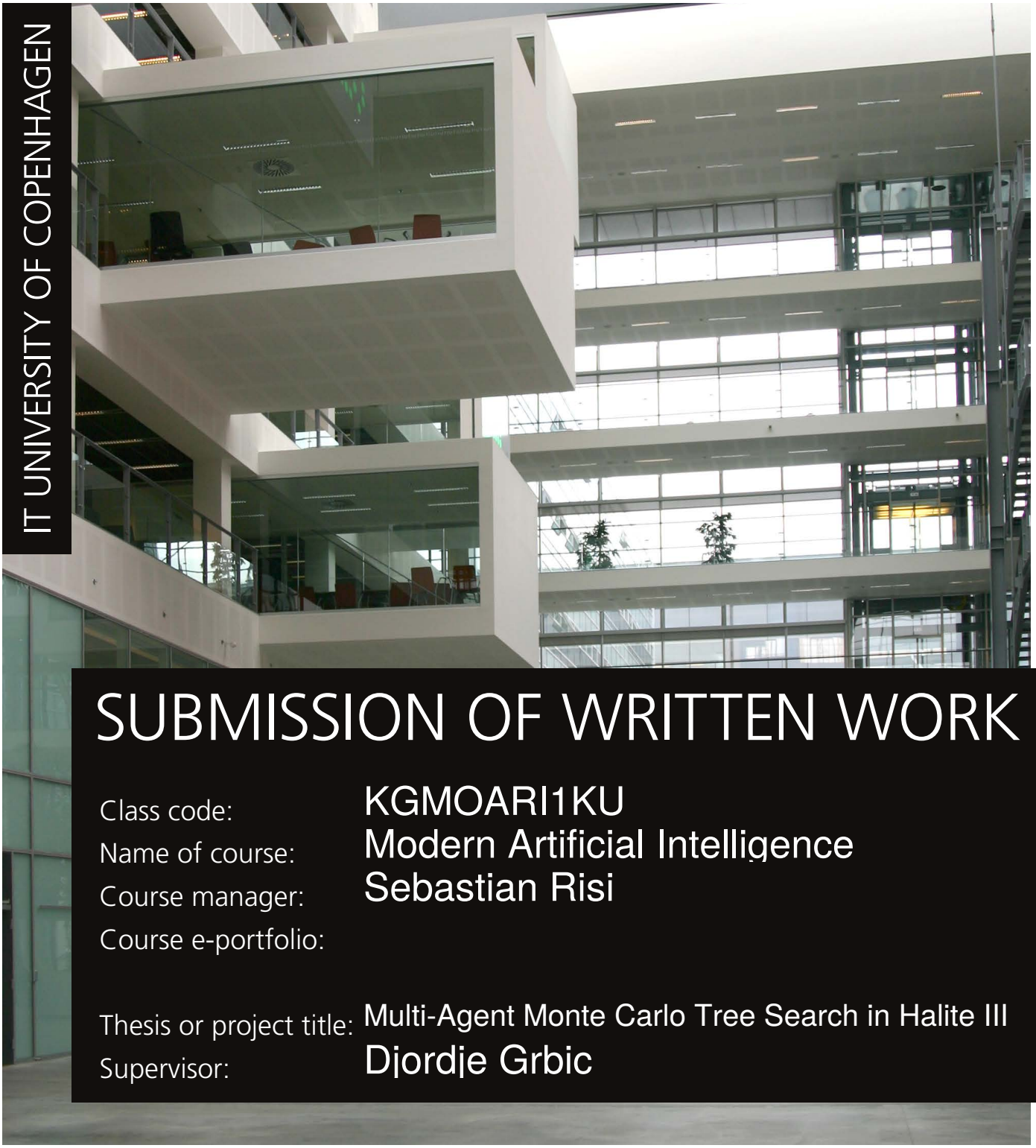
IT UNIVERSITY OF COPENHAGEN



SUBMISSION OF WRITTEN WORK

Class code: KGMOARI1KU
Name of course: Modern Artificial Intelligence
Course manager: Sebastian Risi
Course e-portfolio:

Thesis or project title: Multi-Agent Monte Carlo Tree Search in Halite III
Supervisor: Djordje Grbic

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Michael Vesterli	28/06-1995	miev @itu.dk
2.	Jonas M. Uhrenholdt Tingmose	14/10-1983	joti @itu.dk
3.	Luka Locniskar	14/09-1992	lulo @itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Multi-Agent Monte Carlo Tree Search in Halite III

Luka Locniskar
lulo@itu.dk

Jonas Tingmose
joti@itu.dk

Michael Vesterli
miev@itu.dk

I. Introduction

In this paper, we show how Monte Carlo Tree Search can be used to control many entities, where the number of possible moves would otherwise grow exponentially with the number of entities. One setting where this is important is in the game Halite III, where a large number of ships need to cooperate.

The goal of this paper is then to present the approach used to adapt Monte Carlo Tree Search to be used to create an agent for Halite III. The agent is compared to a hand-crafted agent and other agents submitted online. We also show how our approach could potentially be used in other domains where a large number of entities need to be controlled as long as there are no important helpful interactions between them.

II. Domain Description

Halite III is a game created by Two Sigma for the purpose of an annual AI competition. The competition lasts three months, during which competitors can submit their bots and compete on a global leaderboard.

Halite III is a resource gathering game, which takes place on a two-dimensional grid of cells. The goal of the game is using ships to collect more of the resource *halite* than the opposing player(s). Each player controls a number of ships which can be ordered to either move or mine halite at their current location. Once a ship is full, collected halite needs to be returned to either a *shipyard* or *drop-off*. Each player starts with a shipyard in a fixed location and can spend halite to build drop-offs. Halite can also be spent to build additional ships. An example game state can be seen in figure 1.

The game is turn-based. A turn is executed on a two-second interval, which represents the time players have to provide commands for the entities they own. Before the game starts, thirty seconds is given to all players in a match to do any preliminary calculations. The number of turns is based on the size of the game map (grid), which can be of sizes 32x32, 40x40, 48x48, 56x56, or 64x64. The initial amount of halite on each cell of the grid is based on a unique, randomized symmetric pattern. The map wraps at the edges.

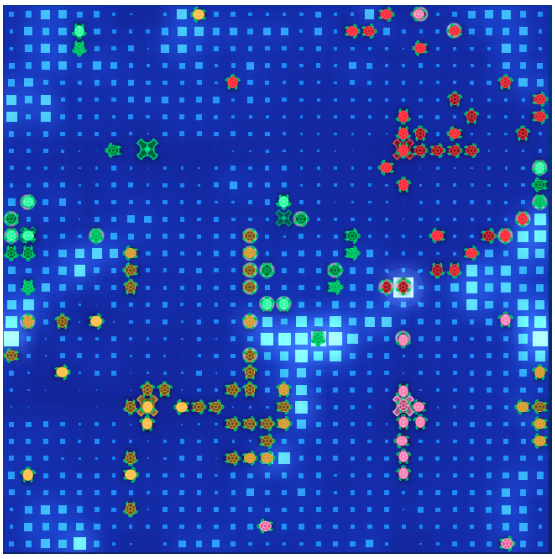


Figure 1: Example game state

Players can give commands to both their ships and the shipyard. The following commands are always available for ships:

- *Move* instructs the ship to move in one of the cardinal directions (north, east, west, south). Moving costs 10% of the halite of the cell that the ship currently occupies. The halite required to move is taken from the ship itself and must be available.
- *Stay still* instructs the ship to stay on the cell it currently occupies. Doing so extracts 25% of the halite on the current cell.
- *Convert into a drop-off* destroys a ship and creates a drop-off point on its position at the cost of 4000 halite. A drop-off point is a static entity to which ships can deploy halite.

Shipyards can receive one command per turn, which is

spawning a ship. This constructs a new ship at the position of the shipyard at the cost of 1000 halite.

Ships can gather and deliver halite. Once the ship gathers halite it is considered to be in its inventory which has a maximum capacity of 1000. Ships can deliver halite by moving to the shipyard or a drop-off and depositing the halite from their inventory to the common halite count. If two or more ships end their turn on the same cell, they are destroyed and their carried halite is dropped on that cell.

There is one additional mechanic related to player interaction called *inspiration*. If there are two or more ships belonging to an opposing player within a certain radius of a ship, that ship receives an inspiration bonus which provides 200% extra halite when the ship is mining.

Players can create bots to play Halite III in various programming languages and submit them to the competition. After being successfully compiled that bot is then pitted against bots of other players 2- and 4-player games. Based on their performance they are ranked by an ELO system. Bots then play against opponents with about the same rank.

III. Related Work

For the implementation, we draw much inspiration from the paper A Survey of Monte Carlo Tree Search Methods (Browne et al., 2012) which contains a detailed description of the background, implementation, and improvements of the Monte Carlo tree search algorithm. What aided the decision for MCTS is the popularity of its use in advanced game AI (Chaslot et al., 2008).

The application of MCTS described in this paper is a novel approach which in part draws inspiration from other approaches. The main problem we faced in our usage of MCTS was the exponentially increasing branching factor. This problem is not uncommon in its application in various problem domains.

A relevant improvement on MCTS for high branching factors is the use of Move Groups (Childs et al., 2008). This approach combines all possible moves into groups which replace individual moves as children in the tree. In our domain, this would not be an effective strategy as it presumes that "many moves are similar"(Browne et al., 2012). This may be true on an abstract level of general ship movement direction but does not take into account that each individual move is very important when it comes to collision avoidance and halite deployment.

As precise individual moves are important in our domain we could have made use of Nested Monte-Carlo Search (Cazenave, 2009) which implements additional roll-outs to aid decision making within the original roll-out. However, due to the time constraint, this was not feasible in our case. Our focus was therefore instead to do as many roll-outs as possible in the time given and use their results as efficiently as possible. This focus is outlined in the papers using the AMAF or all moves as the first approach (Helmbold and Parker-Wood, 2009) which treat all moves in a roll-out as "first" and update the values in the tree for those moves as well which carries a substantial bias. These approaches, such as RAVE (Gelly and Silver, 2011), have shown to "significantly improve the performance of the search" (Gelly and Silver, 2011). However, the increased performance is not enough to combat the extreme branching factor of our domain.

Improving quality through abstraction could be done with an approach such as transposition (Childs et al., 2008). The Halite III contains enables different combinations of moves to achieve similar states which are well suited for the use of transpositions which have proven to be effective in some domains (Kozelek, 2009). However, it does not deal well with our main problem.

Some papers propose using different strategies for node selection and backing up values to reduce the size of the tree and improve decision quality. Good examples of this are Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search (Coulom, 2007) and Single Player Monte Carlo (Schadd et al., 2008). The improvements are not significant enough to merit usage in our domain.

The paper Decentralized Cooperative Planning for Automated Vehicles with Continuous Monte Carlo Tree Search (Kurzer et al., 2018) focuses on a domain that has many similarities to ours. However, they seem to use a standard approach. This is possible in their case as they consider few enough entities that the exponential increase of the branching factor is not prohibiting yet.

IV. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm, which is often used for games involving a series

of moves. Most notably it has been used to make AI for the games Go, Chess and Total War: Rome II. We are using MCTS with UCT as it is presented by Browne et al. (2012) as the basis for our algorithm.

We were primarily inspired by the recent strong performances of Google Deepmind's AlphaGo and its successors (Silver et al., 2017). These use neural networks to suggest moves and evaluate the game state. However, Halite III is likely simple enough that simpler heuristics can be used instead.

A. Algorithm

Given a time frame and the current game state, MCTS searches for the optimal move to make by simulating as many games as there are time for using the rules of the game. It has been shown to converge to minimax, given enough time and memory (Browne et al., 2012). It is thus optimal in theory. In practice there is often not enough time to reach the optimal solution, but it will still find a good solution fairly quickly.

There are four steps involved in the process as seen in figure 2: *selection*, *expansion*, *simulation* and *backpropagation*:

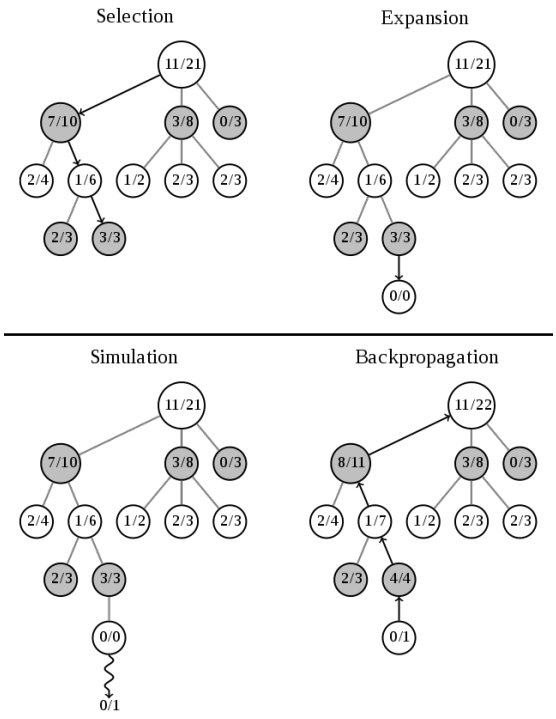


Figure 2: An overview of the MCTS process.
Wikipedia, the free encyclopedia (2017)

In the selection step, a node in the tree is selected for expansion. The path to the expanded node is the starting moves in the next game simulation. The selection process should balance selecting nodes that have yielded a high reward so far (*exploitation*) and nodes that have not been visited many times and therefore might hide a better path (*exploration*).

The nodes to expand are found by starting from the root node and selecting children using a selection function until a leaf node is found. The selected leaf node is then expanded, which means that child nodes are added for each valid move. New games are then simulated where the path up to one of the created child nodes represents the first moves to play.

These selection and expansion steps are referred to as the *tree policy*, as they determine how the tree is traversed.

The next step is running a roll-out, which is a simulation of the game from the current positions until either a terminal state or a cutoff point is reached. Initially, the moves found by the tree policy are played. The remaining moves are taken using a *default policy*, which can be a random move or use some heuristic. Once the simulation ends, the final state is evaluated. If the game reached a terminal state, this will be the result of the game, but it can also be calculated using an evaluation function.

The result of the games are stores in the search tree during the backpropagation step. Each node stores the number of times the corresponding move has been played and the total score of those simulations. These values are updated using the result of the simulated game for each move that was selected by the tree policy. These updated values are then used to select the next node to expand.

Once there is no more time, the best move needs to be determined by selecting one of the children of the root node. As noted by Browne et al. (2012), there are a few ways to choose. We select the node with the highest number of visits.

This is the basic version of MCTS. There are many ways to potentially improve its performance on specific domains as outlined by Browne et al. (2012), but we will not describe these here.

V. Implementation

Plain MCTS is not well-suited for this game as the branching factor grows exponentially with the number of ships. In games like Chess and Go, each player can only make one move

per turn, which makes the branching factor more manageable, even though larger Go-boards still pose a challenge. However, once the branching factor gets so large that a single node cannot be expanded in the allotted time, it will cease to work well. This is due to standard MCTS expanding a node completely before any of its children are expanded. Therefore, it will never search beyond depth 1.

Even if we use a variation where this is allowed, a node will never be expanded again once the tree policy decides that the exploration value of an unexpanded root node exceeds the value of the deeper node. This means that increasing the allotted time will only cause more root nodes to be sampled and searched to low depth. However, nodes with a high reward will not be expanded further within a reasonable time, although the initial depth is slightly greater.

In the case of Halite III, there can be over 100 ships, each of which has 5 possible moves. The branching factor is therefore above 5^{100}. Although there are techniques for significantly reducing this, such as clustering ships, none reduce it to a reasonable level.

A. Monte Carlo Forest Search

To overcome this, we use separate trees for each ship. This way the branching factor is reduced to 5. Additionally, it is possible to reuse simulations to update all trees using one simulation. This avoids having to run as many simulations per iteration as there are ships. Reusing simulations consistently outperforms the alternative. This is likely caused by the improved performance in our time restricted setting and by the improved cooperation mentioned in section V-B.

An issue is how to pass the reward to different trees. If a single reward is passed to every tree as if it was standard MCTS, the trees would just be permutations of each other, such as in figure 3. This is because regardless of the permutation, the same nodes will usually be expanded. After each simulation, both trees would get equal rewards and are updated in the same way. This would never cause the different trees to deviate from each other.

In a forest of MCTS trees that are permutations of each other selecting an action in a node will also imply the action taken by the same node in a different tree. So, even though there are many trees, there are as many choices as if there had been only one. In effect, this is similar to simply randomly sampling 5 moves from the 5^{100} combinations and only considering these.

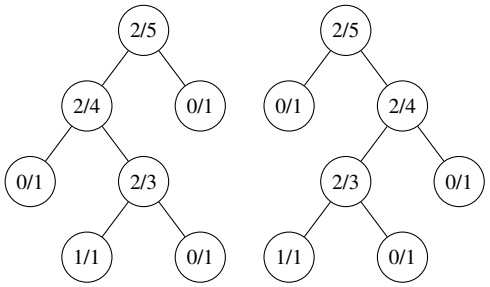


Figure 3: Permutated MCTS trees

One way it would be possible to break this relation between trees is by introducing a random element to the tree policy. However, this would be a difficult parameter to tune. It would also be uncertain if that approach would converge to minimax as efficiently.

Another approach is to give each ship individual rewards, which is what we use. This solves the problem but requires a reward function that ensures that a good individual reward also helps the overall score. In the case of Halite III, one such function is the halite collected per turn. If all ships are collecting halite efficiently then the overall score will also be high.

However, this reward needs to be normalized into a value between 0 and 1. Simply mapping the minimum value to 0 and the maximum value to 1 does not perform well. This is because the score depends on the state that the ship is currently in. If a ship is in a high-halite region, even a poor move will not reduce the score significantly compared to a ship in a low-halite region.

Instead, we compare the halite collected per turn to the average halite collected in the simulations from the last turn. If the collected halite exceeds the average from the last turn, it is counted as a win. Otherwise, it is counted as a loss. In a sense, the ships will each turn strive to be better than they were in the last turn. This has the advantage of being completely independent of other ships. If a ship is currently in a drop-off, there are no suitable simulations to use from the last turn. Instead, a few simulations are run before starting the MCTS.

B. Cooperation

One of the main benefits of MCTS is that ships cooperate by considering each way that they can interact. But since we are

only considering ships independently it is natural to assume that this benefit is lost. Although it is likely true that ships cooperate to a lesser degree than in standard MCTS, they still cooperate through the shared simulations.

When a ship chooses an action that has a high probability of causing a negative interaction with another ship that action accumulates a low reward. The other ship will likewise also avoid actions that cause these interactions. However, when the node representing this action is expanded later in the search, the other ship will likely have selected a sequence of actions which avoid this interaction. Therefore, the interaction becomes less likely to happen and the action is penalized less. This relies on the way simulations are reused across multiple trees.

If an action would otherwise be good if it were not for the negative interaction then it will be rewarded in the later exploration. However, if the other ship elects to do the same, it will still be penalized as the first ship is no longer avoiding the interaction. In this way, the ships should converge to moves where only one ship will approach a cell.

This does not always happen under our time constraints, as a large number of simulations are needed to converge on such behavior. Still, negative interactions are avoided as evidenced by the fact that there are no collisions between ships even though there is no code that specifically avoids that. Ships are also cooperating by not moving towards the same halite deposits.

C. Default Policy

On a typical map in Halite III, there are few areas that worth going to in order to collect halite. Likewise, there is usually only a single place to go to deposit halite. Additionally, if a random action is taken with uniform probability, each ship will lose halite on average as movement has a cost. These are all factors which make using a random default policy unlikely to play well.

We use a default policy with two steps. First, a ship decides whether it should move or mine at its current position. It will only decide on which direction to go if it decides to move.

The probability of mining is based on the amount of halite on the current cell relative to the average halite on the map. It is defined as

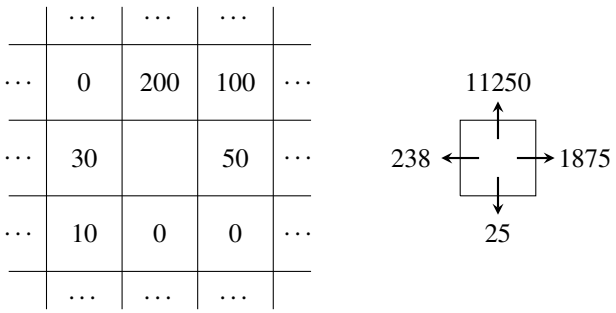
$$\Pr[\text{mining}] = \frac{h_c}{2\sum_{i=0}^{C-1} h_i / C} \tag{1}$$

where h_c is the amount of halite at cell c and C is the number of cells on the board. As an example, a cell with an average amount of halite has a 50% chance of being mined.

To decide which way to move we represent the board as a grid of attractors and base the probability of moving of the pull in each direction. Each cell has an attractor which pulls with a strength equal to the amount of halite on the cell squared. Using the attractors, we calculate the pull in each direction for each cell. The pull p in direction d at cell c is calculated as

$$p_c^d = \sum_{i=0}^{C-1} \frac{h_i^2}{4\,\text{dist}(c,i,d)} \tag{2}$$

where $\text{dist}(c,i,d)$ is the distance from c to i when d is taken as the first move. The path is computed from a path without u-turns, as the differences would otherwise be very small. An example can be seen in figure 4, where only close halite is considered. In this case, the default policy would move up with probability 84%.



(a) Close Halite (b) Resulting pull

Figure 4: Attraction example

This policy will only move toward halite deposits, but when the ship is full it will have to return. Therefore we use another grid which has a strong attractor at the drop-offs. The two attraction grids are then blended depending on the amount of halite the ship is carrying. A ship with no halite will then ignore the drop-offs, whereas a full ship will ignore all halite.

VI. Evaluation

We measure the performance of our bot using MCTS against a strong hand-crafted bot. The hand-crafted bot uses a thorough search to plan a complete path for each ship. The planning ignores other ships but it knows how much halite other ships will mine from each location.

The result is a close to optimal plan, which however takes a long time to compute. As a consequence only a few ships have their plans recalculated each turn. This means that the bot is very slow to react to enemy bots getting in the way, collisions and new drop-offs being constructed.

The comparison is done through the official game runner, but no drop-offs are built. Each bot is given 2 seconds to make their moves. This seems to favor the hand-crafted bots as reducing the number of simulations significantly reduces the performance of the MCTS bot. On the other hand, reducing the time available to the hand-crafted bot does not significantly impact its performance. Due to this, the MCTS bot does not consider inspiration as simulations would take 5 times longer. Unfortunately, it is not possible to increase the amount of time per move.

In two-player games, the hand-crafted bot consistently gathers about 50% more halite than the MCTS bot. However, in four-player, the MCTS bot usually wins. These games have a high variance: some are close with either the hand-crafted or the MCTS bot winning with a low margin, in others, the MCTS bot doubles the score of the hand-crafted one. The MCTS bot does not lose heavily in this mode.

We have also uploaded our bot to the official tournament site, where its performance for two- and four player can be seen in table I and II respectively.

Size	1°	2°	Win %
32x32	14	2	87.5%
40x40	19	1	95%
48x48	26	4	86.67%
56x56	27	4	87.1%
64x64	24	9	72.73%

Table I: Online 2-player performance

Size	1°	2°	3°	4°	Win %
32x32	30	1	0	0	96.77%
40x40	20	4	1	0	80%
48x48	29	6	3	1	74.36%
56x56	26	3	0	3	81.25%
64x64	20	2	8	4	58.82%

Table II: Online 4-player performance

In studied losses, it seems that the bot performs poorly when halite deposits are far from the shipyard. This would explain the lower performance at large map sizes. Note that, due to the bot entering at a low rank, the numbers are inflated as it started playing against easier opponents.

VII. Discussion

The match-up between the hand-crafted bot and the MCTS bot is likely not entirely fair as the MCTS bot does not exploit the weaknesses of the hand-crafted bot. The hand-crafted one performs poorly when opponents get in the way of the planned paths of ships as it never attempts to evade them. This is never exploited by the MCTS bot as it goes out of its way to avoid its opponent.

There are also parts of the game that are not included in the MCTS. The decision of whether to build ships is done by a heuristic instead. Although this could be included in the MCTS, it would lower the search depth which is not feasible at our time constraints. However, there are currently a significant amount of losses that are primarily due to the ship building heuristic being too optimistic.

Since performance decreases significantly when the amount of available time is decreased, it would be interesting to see whether the MCTS bot would perform better when given more time. This would likely help, especially in cases where halite is far from the shipyard, such as on large maps where the bot currently performs poorly.

A. Broader usage

Although the forest of Monte Carlo search trees was used for Halite III, the approach is generally applicable as long as there are no desirable interactions between cooperating entities. This is because it would be difficult for this approach to find them. For them to be found, both entities would need to independently seek the interaction during the same simulations. This is unlikely to happen. On the other hand, undesirable interactions can be avoided by preventing them in the roll-outs. This would still penalize bots approaching such interactions by making them waste time.

In Halite III, there is only an undesirable interaction between cooperating ships, which is a collision. These are prevented in the rollout. On the other hand, collisions with opposing ships are not prevented but instead avoided by modeling them as a loss. Although inspiration is a desirable interaction, it only happens between ships that do not cooperate. It is likely

exploited equally well by our method and standard MCTS when playing against other bots.

In essence, Halite III consists of a single controller with perfect information that controls a swarm of bots which must perform tasks that take various amounts of time. In this specific instance, the tasks consist of collecting halite, where the amount of time spent is dependent on the amount of halite present. Although the ships must return to a central point in periodic intervals, this is not essential to the approach.

In such a setting, finding an optimal plan is difficult as the number of possible actions grows exponentially with the number of bots. Our approach reduces the search space to be independent of the number of bots, which allows MCTS to be used to find good plans regardless of the specifics of the setting.

One use-case where these settings are common is in game AIs. Here, a large number of enemies are controlled centrally and must work together to oppose the player in an intelligent fashion. They also cannot simply choose the same actions as other enemies might be occupying the space. Actions could also be redundant.

Current approaches mostly use heuristics and pathfinding, perhaps with custom behavior per task or level. Often such AIs have a difficult time cooperating without intricate level design. However, using our variant of MCTS it would be possible to simply give a reward to each action and let the algorithm take care of the rest. Some cooperative strategies such as exploiting the positions of other enemies to set up flanks would then occur naturally.

One potential drawback in games is that MCTS can be time-consuming, but with discrete steps that span a long enough time, it should still be possible to get good results in real-time environments. Another option to improve performance in a time-constrained environment that we have not explored is to reuse parts of the search tree. This would lead to trees containing the results of simulations that can no longer occur. However, these invalid simulations can still help guide the search in the right direction. Once a turn is done, the number of valid simulations will still far outweigh the number of invalid simulations. This trade-off between performance and correctness might be worth considering.

VIII. Conclusion

We have implemented a bot for the game Halite III using Monte Carlo tree search. The main challenge was that the branching factor grows exponentially with the number of ships. We solve this by using separate Monte Carlo search trees for each ship, and update them all with a single simulation.

Even though the trees are split, ships still cooperate through the shared simulations. This way, ships avoid colliding with each other and do not attempt to mine the same deposits. The bot performs fairly well against other bots online, where losses are mostly caused by heuristics outside the MCTS and a low search depth due to time constraints.

Although the MCTS variant was used for Halite III, we show that our approach is generally useful. It is applicable in most settings where a single controller issues commands to many entities which do not directly interact, which is the case for AI in many games.

References

C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012. ISSN 1943-068X. doi: 10.1109/TCIAIG.2012.2186810.

T. Cazenave. Nested monte-carlo search. In *IJCAI*, volume 9, pages 456–461, 2009.

G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008.

B. E. Childs, J. H. Brodeur, L. Kocsis, et al. Transpositions and move groups in monte carlo tree search. In *CIG*, pages 389–395, 2008.

R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In H. J. van den Herik, P. Ciancarini, and H. H. L. M. J. Donkers, editors, *Computers and Games*, pages 72–83, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-75538-8.

S. Gelly and D. Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

D. P. Helmbold and A. Parker-Wood. All-moves-as-first heuristics in monte-carlo go. In *IC-AI*, pages 605–610, 2009.

T. Kozelek. Methods of mcts and the game arimaa. 2009.

K. Kurzer, F. Engelhorn, and J. M. Zöllner. Decentralized cooperative planning for automated vehicles with continuous monte carlo tree search, 2018.

M. P. Schadd, M. H. Winands, H. J. Van Den Herik, G. M.-B. Chaslot, and J. W. Uiterwijk. Single-player monte-carlo tree search. In *International Conference on Computers and Games*, pages 1–12. Springer, 2008.

D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, Oct. 2017. URL http://dx.doi.org/10.1038/nature24270.