Bachelor's Thesis Computing Science

An MCTS-based AI playing The Crew

Iwan Litjens
s1022950

July 7, 2022

First supervisor/assessor:
dr. Sebastian Junges

Second assessor:
dr. Nils Jansen

Radboud University

Abstract

In this thesis we conclude that an MCTS-based algorithm can solve fully co-operative games that have information fog. For this purpose we take a look at The Crew, a cooperative card game where players have limited information. We analyse where these algorithms struggle, and the effectiveness of improving parts of the algorithm. Determining which parts are more crucial that others. We demonstrate that MCTS-based AI is effective in solving this game. We conclude that its simulation-based decisions make it versatile, which contributes to this effectiveness in this environment. Finally we conclude that it has the drawback of not being able to take into account how other players gaining new information leads to them playing better.

Contents

1 Introduction **2**

2 Preliminaries **4**
2.1 The Crew 4
2.1.1 Concepts 4
2.1.2 The Rules 4
2.2 The game tree 7
2.3 MCTS 9

3 Solving The Crew **10**
3.1 Adapting MCTS 10
3.2 Implementation 11
3.2.1 General purpose 11
3.2.2 Detailed implementation 12
3.2.3 The Player class 12
3.2.4 The GameController class 15
3.2.5 Strategy 18
3.3 Results 20

4 Related Work **25**

5 Conclusions **27**

Chapter 1

Introduction

Game trees are used when algorithmically solving games. These trees consists of nodes representing the current game state. Each node either offers one or more state transitions leading to another node or they are leaves in which case there is a value assigned to it. In a classic game tree there are two players one has the goal to reach the highest possible value the other to reach the lowest value possible. Both players take alternating turns, in their turns they choose which of the possible state transitions is the best for them and then the current game state transitions accordingly. Then the other player does the same until an endpoint is reached, at which point a winner is determined.

We are interested in how you can optimally traverse this tree when the game played is cooperative and has information fog.

A cooperative game, for the purposes of this thesis, is defined as a game in which all players have to work together to achieve a common goal in which they either win as a team or lose as a team. A game with information fog is defined as a game in which the players have limited information such that only a partial game state can be determined by each player. When games are cooperative and have information fog this changes how the trees are traversed and/or how those trees will look like.

The information fog has the most impact as no full game state can be determined by a player, therefore they can only construct a part of the game tree. Switching to cooperative games does not change the structure of the game trees directly, however it does change how they are searched as all parties will be trying to reach an endpoint with a value as high as possible. For example, a commonly used method to make tree searches more efficient is Alpha-Beta pruning [4], however this technique relies on your opponent trying to reach the opposite goal than you, which of course will not happen in a cooperative game.

This makes cooperative games with information fog very different when compared to games like chess for example, and in order to algorithmically

solve them we will adapt an existing solution to this new environment. For this solution we have chosen MCTS[4].

First we explain the rules of The Crew. Then we will explain how MCTS works and how we will adapt it for our environment. This is followed by an explanation of the implementation we made to test how well our algorithm works. Then we will discuss how our algorithm performed and its strengths and weaknesses. We will talk about what existing research there is with regard to MCTS and finally we will discuss what can be done to improve the algorithm, its strengths and weaknesses with regards to our environment.

Chapter 2

Preliminaries

2.1 The Crew

In order to put the algorithms to the test, we need an environment to test them on, for this purpose we have chosen The Crew[1] the dutch version is published by 999 games[2]. The Crew is a cooperative game where information is limited by information fog and we have created an environment which closely resembles the rules of the game, with some exceptions which will be explained below.

2.1.1 Concepts

The deck contains all cards which fulfill the following definition. $deck = \{(color, number) | number \in \mathbb{N}, 0 < number < 10, color \in \{"pink", "yellow"$ $, "blue", "green", "black"\}, color = "black" \rightarrow number < 5\}$
The color number tuple is a card, we will abbreviate cards to the first and last letter of the color followed by the number, e.g. a yellow 9 becomes Yw9 and a black 4 becomes Bk4.

Starting hands Starting hands are a partition of the deck, where the number of subsets is equal to the number of players and the size of those subsets is as equal to each other. Some starting hands contain one card more than the others, if and only if the deck size is not divisible by the number of players.

2.1.2 The Rules

Playing the game The crew is played in rounds, each round consists of a opportunity to communicate and a trick phase. A player can communicate

[1] https://cdn.1j1ju.com/medias/8f/13/09-the-crew-the-quest-for-planet-nine-rulebook.pdf
[2] https://www.999games.nl/de-crew-kaartspel.html

once each game, in order to communicate a player shows one of their cards and indicates whether that card is the highest, lowest or only card from that color. The communicated card has to fulfill one of those conditions and cannot be black. In the trick phase the lead player, the player who won the previous trick, plays a card from their hand. Whenever a card is played it is removed from that players hand. Then, in clockwise order, the other players play a card in the same color as the card played by the lead player, if possible. Otherwise they play any card in their hand. The exception to this rule are black cards, which can be played even if the player has the leading color. Then a winner is determined based on the cards that were played, this player wins all the cards played this round. The highest black card wins, if there is no black card the highest card of the leading color wins. Rounds continue until the game ends. The game ends if there is a player with no more cards in their hand, or if the game is won/lost, see below.

Preparing the game Before we start the rounds we do some preparation. First we create starting hands for each player. The player with the Bk4 announces this and becomes the captain, further communication regarding the cards in players hands is prohibited. Then a number of task cards are shown. Then the captain chooses one task, this continues clockwise until all the shown tasks have been divided.

Task cards $taskCard \in \{c \in deck | color \neq "black"\}$ A task card is completed successfully if and only if the player that has the task card wins the card from the deck with corresponding number and color. Before a game begin a mission is chosen, which determines how many task cards need to be divided.

Winning the game The players collectively win the game if and only if they completed all the task cards successfully. A task card is completed if the player who chose it wins the corresponding card, if another player wins it the task card cannot be won and thus the game can no longer be won resulting in an early end.

Example Now we will provide an example of how this would look like, a number of rounds have already been played.

- Alice has a Be4, Yw9, Gn9, Bk1

- Bob has a Pk3, Be6, Yw3

- Charlie has a Be1, Be8, Gn5

Furthermore Bob has the task card Be4.
A new round starts.
First everyone has the opportunity to communicate. Bob decides to do so and communicates its Be6 with the token in the middle, indicating that this is the last blue card he has. The other players decide not to communicate.
Charlie won the previous round so is now the starting player and can play any of his cards, as there is no leading color yet. Charlie decides to play Gn5.
Alice has two options now, playing Gn9 or Bk1, and decides to play Gn9.
Bob does not have a green card and is therefore free to play any of his cards. In order to save the blue card for the task, Bob decides to play Pk3.
No one played a black card, so the highest green card wins. Alice played a 9 which is higher than the 5 so Alice wins this round.
There is again a possibility for communication, but no one decides to do so.
Currently these are the hands of the players.

- Alice has a Be4, Yw9, Bk1

- Bob has a Be6, Yw3

- Charlie has a Be1, Be8

A new round starts.
Alice won the previous round so can play any card and goes first. Alice decides to play the Be4.
Bob now has no other option to play the Be6.
Charlie now has two options, playing the Be8 would win him the round, but as the Be4 is already played Bob should win this round, or that task fails. So Charlie decides to play the Be1 instead.
Again no black card was played so the highest blue card wins. The 6 played by Bob is the highest. So Bob wins this round and completes the last task successfully. This ends the game and the players win.

The Rules we ignore (and why)

There are a number of rules we decided to ignore for the purposes of this thesis. This is mostly done because they would change the game, but have little impact on how the MCTS based algorithms would function. For some rules there were more reasons and below is a list of the rules we ignore and what additional reasons were for ignoring them, if applicable.

The game has a lot of challenges, most missions other than deciding how many task cards are used also decide which of these challenges are used for this game. Some examples of these challenges are: communication can be forbidden from occurring until the third round, some tasks might have to be completed before others, the division of tasks could be different. The missions we chose are without such challenges. Additional motivation for not implementing this is that the difficulty increase is more intuitive when it is just the number of tasks that increases.
In the rules there exists a help mechanic called distress signals. This mechanic can be used after the preparation phase, if it is used every player gives one card to the player to the left or right, everyone to the same side. This comes at the cost of adding one additional attempt to that mission. An additional reason not to implement this is because it affects things beyond the mission itself.
As we run our simulations with three players all rules in regard to 2 and 5 player games are also ignored.
There is a rule that says black cards can only be played if the player does not have a card of the leading color (or the leading color is black). In our implementation black cards can be played no matter what the leading color is.[3]

2.2 The game tree

The crew does not have a very different game tree when compared to a classical game tree. There are only two differences when compared to classical game trees. One is that there are tree players instead of two and the second one is that the turn order is dependent on the moves made. The problem is that no player has the information necessary to construct this game tree, which we will refer to as the true game tree. Instead they can construct a tree that contains all possible game trees, see Figure 2.1. The node labeled R is a random node which randomly chooses one of the possible paths to traverse. This node represents the different possible hands players can have, one of those hand combinations is the combination that is the case and this will lead to the true game tree although players will not now which

[3] This is due to a misunderstanding of the rules

tree that is. The number of different hands the other two players can have is $\binom{26}{13} \cdot \binom{13}{13} = 10.400.600$. For all possible hand combinations there is a game tree tailored to that division of cards. The node R leads to one of the possible game trees. Based on the example in section 2.1.2 we created an tree of how one of those trees will look like, see Figure 2.2. In this tree nodes colored green are leaves where the game is won, whereas the nodes colored red are leaves where the game is lost. Note that this is only part of a full tree, containing just one round, therefore the uncolored nodes lead to the next round. Additionally because communication only changes the information players have and not which moves can be made or which effect they have the communication phase is omitted as it has no influence on how the game tree looks, only how the players traverse it.

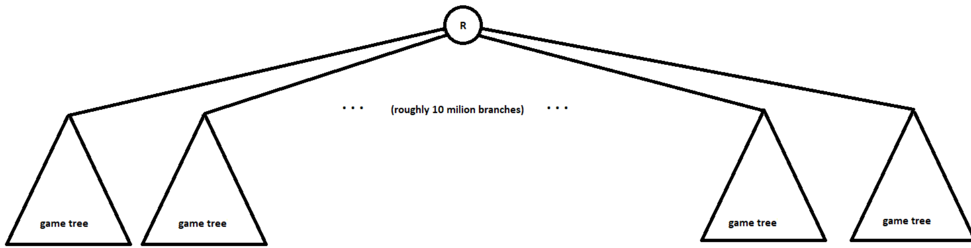

Figure 2.1: Game tree of all possible situations

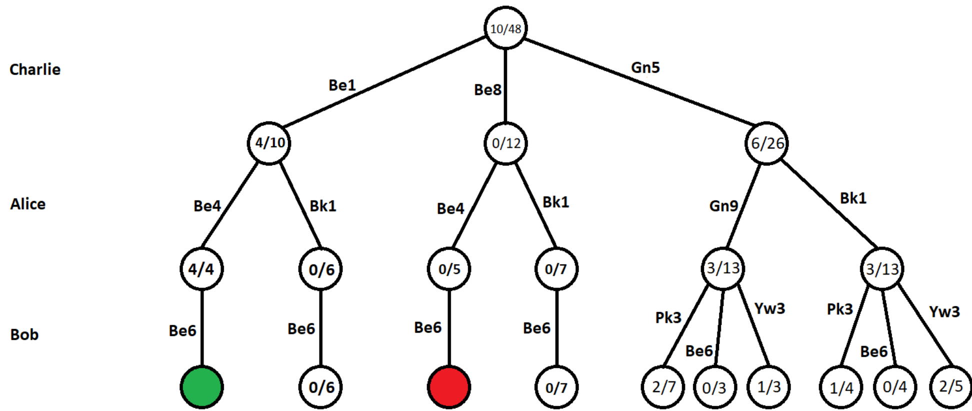

Figure 2.2: True game tree from section 2.1.2

2.3 MCTS

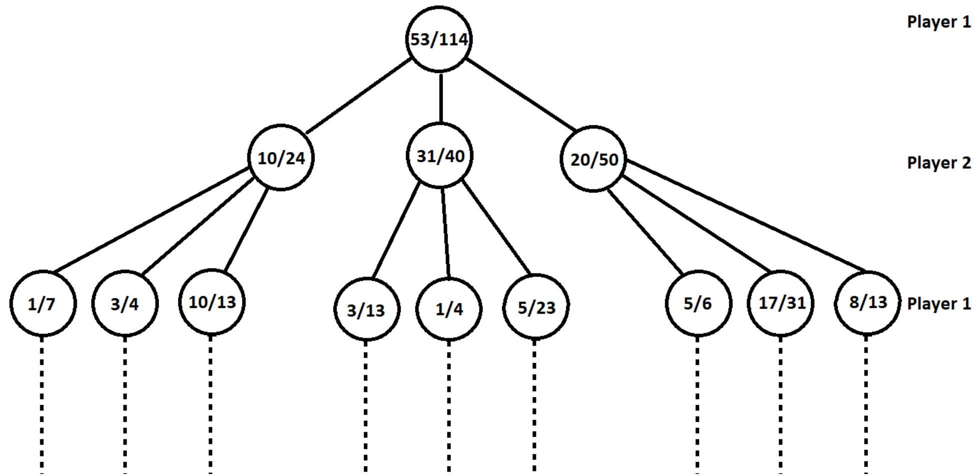

Figure 2.3: Example tree with MCTS

In order the cope with the challenges provided by the cooperative nature and information fog, we use the Monte Carlo Tree search[4]. This is because the use of simulations lends itself well to situations where game trees cannot be fully constructed. The three core concepts of this method are: using simulations to determine the optimal move, the playout policy and the selection policy. In a simulation a path of the tree is traversed, when this path ends all previous nodes are updated depending on the result of the simulation. While simulations are running a tree is constructed, consisting of the paths taken by the simulations, see Figure 2.3. In that figure there is (a part of) an example tree, in an competitive environment. The lines at the bottom lead to more of the tree. The numbers within the nodes are the won games divided by the total attempts that passed through that node. The selection policy chooses which part of the tree needs further exploration, balancing places where very little simulations have been made vs further exploring places that yielded the best results. If for each move within this simulation the path is determined using this same method, the number of simulations would grow exponentially. This is where the playout policy is used. The playout policy decides which moves are played within simulations. Preferably this policy has a bias towards good moves, but this is not strictly required, although performance will likely suffer. In the next chapter we will explain what changes we make to MCTS to make it work for The Crew.

Chapter 3

Solving The Crew

3.1 Adapting MCTS

As we employ MCTS in a cooperative environment with information fog, we need to change certain aspects to ensure it still works. The core idea of using simulations to determine the best move is central to MCTS and it was the concept that makes MCTS suitable to deal with incomplete game trees. The concept of the playout policy requires little adaption, however we do need to create a playout policy tailored to this game. We have created several playout policies and we will cover them in detail in section 3.2.5. Which policy we use changes based on the algorithm. The selection policy requires the most adapting to the new environment. This is because the players cannot build a true game tree due to the information fog, see section 2.2. The tree they can build has random node R at the beginning, see Figure 2.1. Given the large amount of branches in that node, exploring all possible game trees is infeasible given the amount of computational power available in one laptop. This means that in nearly every simulation the game tree it traverses will be unexplored. Therefore our selection policy will focus on determining which hand configurations are more likely to be the actual hand configuration, or which are closer to it. We then use that to traverse random node R such that we are more likely to end up in a game tree closer to the true game tree. To accomplish this we do two things. First we want to be able to avoid going to trees for which we know that they cannot be the true game tree. We do this by keeping track of the cards played and which player played them. This means that we can reduce the number of hand configurations with certainty. The more rounds are played the more configurations can be eliminated, as the only uncertainty is which cards there are in the hands of other players and those hands become smaller after each round. We can reduce the amount of possibilities further by using the cards the players communicate and deduce from that what they can/cannot have left in their hands. For the remaining configurations we

try to focus on those that, based on how the other players play, are more likely to be the correct configuration. This is achieved by simulating the part of the game that has already happened with a strategy that tries to play the same cards as in the original. For this purpose we pick a starting hand combination that could still be the case, although we cannot do this for every combination. Using these starting hands we attempt to play the same moves made in the original. However we also use the playout policy to determine whether or not the move in the original would have been made in this situation. If any of the moves cannot be done be playing legal moves, or a player would have played a different card following the selection policy, we know that this is unlikely to be the correct hand configuration. By doing more of these simulations some cards will be present more/less often in the hands of certain players. We use this ratio to perform a weighted random division of the hands at the other simulations. By doing this we recreate the purpose of the selection policy, which is directing your exploration of the tree to the places where they are the most effective.

3.2 Implementation

3.2.1 General purpose

In order to test our adapted version of MCTS we need to implement the game logic so that we can use it to run games using the strategy. For this purpose we will write a java program, the source code can be found in this[1] GitHub repository. In this the most important classes are the `Player` class and the `GameController` class, the `Player` class is very comparable to a real player in what it can do. The `GameController` class has mostly a managing role, tells the players when to perform which actions, it determines who won the trick, etc. When simulating games both of these classes make clones of themselves, generated such that all lists and other variables passed by reference are remade. This ensures that the original classes remain unchanged whenever games are simulated.

In Figure 3.1, a diagram is made representing a simplified version of the main flow of the program. There are five numbered boxes, which will get some elaboration. The box numbered 1 contains the three functions needing to be called to run a game. The arrows pointing outside the box represent the inner workings of these functions. Boxes 2, 3 and 4 each contain one of the three situations where a player needs to make a decision, communicating, playing a card and picking a tasks respectively. Each box has the following pattern. The first function is the environment in which this choice needs to be made. It then calls the second function from the desired player, which will call upon its controller to make a decision for it. The first function

[1]https://github.com/iwan382/crew

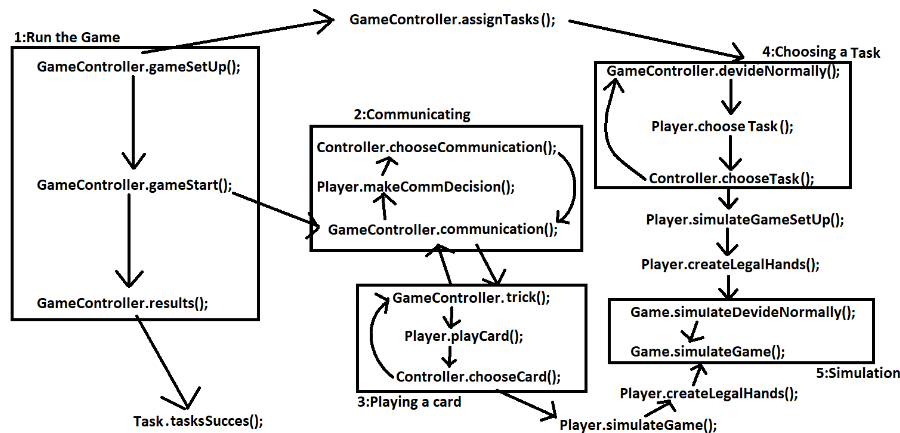

Figure 3.1: Flow overview

repeatedly does this until that choice no longer needs to be made in this phase. The fifth box consists of two functions which simulate an entire game. This looks basically like a copy of Figure 3.1, except no simulations are done within simulations, so that part would not be present, instead a playout policy decides which move is played.

3.2.2 Detailed implementation

3.2.3 The Player class

The `Player` class contains many functions and attributes. In Table 3.1 there is an overview of most of the functions their return values and the arguments they are given. The other functions are explained below.[2]

simulateGameSetUp First this function starts a lot of simulations, starting when players are still choosing tasks. The simulation returns the history object associated with it, which is stored in a list. As there are still history objects in that list from earlier simulations, we check if they are still useful. We consider simulations useful if and only if the same moves were made as in the original. The function will remove all history objects that are not considered useful. It then calculated which chosen task had the highest success rate and returns it. It is given several arguments.

- A GameController object, so that it can be copied and used in the simulations.

[2] Additionally there are several instances where the code is built to support features we ended up ignoring, but as these were built into the code before those decisions were made it remains in place. As an example the code is build to work with a variable amount of players even though we only use it in games with 3.

Function name	Return type	Arguments
maxHandSizeExeeded	boolean	int, LinkedList<Card>
legalComm	LinkedList<Communication>	none
isCaptain	boolean	none
handEmpty	boolean	none
forcePlayCard	void	Card
forceChooseTask	void	Task
legalMoves	LinkedList<Card>	int, Player[]
getCardPlayed	Card	none
getComm	Communication	none
getPlayNum	int	none
getDecision	int	none
getWinPile	Card[][]	none
getHand	LinkedList<Card>	none
getPersonalTasks	LinkedList<Task>	none
getGameInfo	double[][][]	none
removeCardPlayed	void	none
setCardPlayed	void	Card
setGame	void	GameController
setDecision	void	int
setStrategy	void	Strategy
setWinPile	void	int, Card[]
setHand	void	LinkedList<Card>
clonePlayer	Player	GameController

Table 3.1: Player functions

- A list of the tasks that can be chosen.
- A number indicating how many simulations are run per task.
- An array containing the strategies to be used when simulating games.
- Finally a `Boolean` value, if true the hands of the players are replaced based on the players knowledge, otherwise the actual hands are used.

cardInfo This is not a function, but an attribute which warrants some further explanation. This three dimensional array represents the knowledge the player has about where the cards can and cannot be. Every player and card combination can be found in this array. A value of 1.00 means that this player is absolutely certain that that player has that card. A value of 0.00 means that this player is absolutely certain that that player cannot have the card. The values 1.00 and 0.00 are only ever assigned as a consequence of a played/communicated card. Information based on simulations always has a false data-point for both possibilities such that these certainties cannot be made incorrectly.

createLegalHands This function makes and returns a list of hands, based on the knowledge it has in `cardInfo`. It starts with assigning all the cards of which the player knows the true owner of. It then fills the hands with the remaining cards, this is done using random numbers, these random numbers are weighted by the numbers in `cardInfo`.

simulateGame This function act similar to the `simulateGameSetUp` function. There are however two key differences. The first it that returns the card that when played has the highest success rate instead of the task. Additionally, it also preforms a second set of simulations. These simulations simulate the past to see whether or not moves/tasks in the original game could/would have been selected with the hands it generates in `createLegalHands`. The odds of a player having a certain card are then changed based on these simulations.

updateInfoCommunication Whenever a player communicated the `gameController` calls this function on all the players. This function gets as arguments the communication that was made and the player who made it. It then proceeds to update the `cardInfo` array, with the information it gains from this. The information is, that the communicated player has the card it communicated; that it does not have cards that would contradict its token and that those cards might now have only one possible owner; which will then be noted as a certainty. In the example in 2.1.2, in the first round Bob communicated that he has a Be6 card and that he has no other card of that

color. The player object associated with bob would then call this function by Alice and Charlie, they would then both run this function, where they would learn that Bob has no Blue cards except for the Be6.

updateInfoCard Whenever a player plays a card this function will be called. It is given two arguments, the number of the player who played a card and the card that was played. This function updates the array `cardInfo` accordingly.

playCard Determine the moves that this player can make with `legalMoves`. Then call the controller to determine which card should be played based on the strategy we use. Remove that card from our hand. In the example in 2.1.2, the first round Alice needs to play a card the `legalMoves` function returned a list with a Gn9 and a Bk1, that list is put into a function deciding which of the cards to play which returned a Gn9 so that card was played by Alice.

chooseTask Functions very much like the `playCard` function, but with tasks instead. As there are no restrictions for choosing tasks the determining of which tasks can be chosen is skipped.

makeCommDecision Functions very much like the `playCard` function, but with communications and the appropriate functions instead.

bestTask This function iterates through the list of tasks it has been given. Each task is passed as argument to the `calculateSuccesValue` function, which returns a value. The `calculateSuccesValue` function calculates a value for a task, the higher this value the more favourable circumstances exist when choosing it. The task with the highest value is returned.

bestCommunication This function iterates through the list of communications it has been given. Each communication is passed as argument to the `calculateGain` function, which calculates a value based on how much information others would gain from this communication. A bonus is given to that value when there are tasks related to the color of that card. The communication with the highest value is returned.

3.2.4 The GameController class

As with the `Player` class this class contains many functions/attributes and we listed the functions where no explanation is given for in table 2.

Function name	Return type	Arguments
chooseTask	void	int, Task
makeMove	void	int, Card
results	boolean	none
cardsPlayed	boolean	none
quickFail	boolean	Card, int
couldWin	boolean	Card, Color
winningCard	boolean	Color, Card
isInATask	boolean	Card
playerHasTask	boolean	Card, Player
assignTasks	void	int, LinkedList<Card>
createTasks	LinkedList<Task>	LinkedList<Card>, int, int[]
cloneGameController	GameController	none
getNextTask	Task	none
getNextMove	Card	none
getTaskOverview	LinkedList<Task>	none
getPlayer	int	none
getHistory	History	none
setGame	void	GameController
setLeadPlayer	void	none
setPreselectedTasks	void	LinkedList<Task>
setPreselectedMoves	void	LinkedList<Card>
setHands	void	LinkedList<LinkedList<Card>>
reassignStrategy	void	Strategy[]

Table 3.2: GameController functions

runGame This function calls the three functions in box 1 from Figure 3.1. In other words it runs a game. After the game is finished the function returns a 1.00 if the game was successful and a 0.00 otherwise.

historyCheck It calls `simulateGameStart` to simulate a game, then it returns false if either the chosen tasks or the played cards contain a black 9 which is a non existing card indicating that the forced card would not have been played.

simulateGameStart This function first assigns the strategies used for the simulations, then it simulates a game starting in the phase where players choose tasks. After the game is finished it returns the History object from that game.

gameSetUp This function performs the first part of the game where tasks are generated and then divided between the players. It also sets the player variable, which is used to determine whose turn it is, to the right number.

divideNormally Starting with the captain then continuing in ascending order each player picks a task until all tasks are divided.

simulateDivideNormally The same as the `divideNormally` function, but it copies the list of tasks beforehand so that the original list remains unchanged. Additionally it does not necessarily start with the captain, but instead with the player whose turn it currently is.

simulateGame Check if we should be in a trick phase, if so it calls the trick function to perform a trick phase. Then it calls `gameStart` which plays the rest of the game. This function then calculates the results before returning the history object.

gameStart Performs a communications phase, as follows. As long as the `Boolean` returned by the communication phase returns false and the progress in the tasks does not cause the game to end. This function will continue to run a trick phase followed by a communication phase.

communication Gives every player a chance to communicate, continues to do this until every player has rejected this in a row. Then checks if the players have enough cards to play another trick, if they do return false. If any player does not have enough cards return true.

trick Because due to simulations this function can be called in the middle of a trick phase, first check which players have already played a card. Then continue having players play cards until everyone has played a card. Then determine who won the trick. Assign all the played cards to their `winPile`, which is an attribute keeping tract of which cards this player won. Set the `leadPlayer` to that player and then remove the `cardsPlayed` from all the players.

quickFail This function is made in order to make the players play better inside simulations by warning them if their move would cause the task to fail. This warning consists of returning true if such a move is detected.
For this purpose it checks if there is a card needed to complete a task. If there is and the task must be won by this player and then warns the player if the current move is insufficient to win the trick. If another player must win that task, this function will give a warning if the current move would win the trick.
Additionally it has to consider the situation where the current move is playing a card needed for a task. If the player who needs to win this has played a card, we check if that player would win the trick in this situation, if not we give a warning. If that player has not played a card yet we only give a warning if we know that the player cannot have the cards required to win the card. In the example in 2.1.2, in the second round Charlie can play a Be1 or a Be8. If he plays the Be8 he wins this round, however Bob played the Be6 which he needs to win. Therefore If Charlie plays the Be8 the players lose the game. Charlies other move, playing the Be1, would let Bob win the Be6 instead. In this situation the move Be8 would be detected by this function as a move that would lose the game, and since there is another move available this function warns the player not to play it.

3.2.5 Strategy

This enum type contains the various strategies which can be used.

PLAYER Print the choice in the console and wait until input tells you what to do. This strategy is used for debugging, but for nothing else.

Playout policies

These three strategies are the different playout policies we use with the different strategies.

RANDOM Self explanatory, whenever presented with a choice pick a random option.

NOTCOMPLETELYINCOMPETENT Acts the same as `RANDOM` with regards to tasks and communications, but when playing a card it will try to detect if playing the card results in a failed task, if it does it will select another move. If all moves fail a random move is selected.

SMART Acts the the same as `NOTCOMPLETELYINCOMPETENT` with regards to the playing of cards. For tasks and communications it calls a function that evaluates a choice and returns a score, for tasks and communications the one with the highest score is selected.

MCTS based strategies

These four strategies use our adapted version of MCTS, each with differences in playout policies or number of simulations.

SIMPLESIMHONEST With this strategy 100 simulations are performed for each option per choice, except for communications in which it acts randomly. The option with the highest success rate is chosen. All simulations assume the use of the `RANDOM` strategy and hands are reassigned, based on its own knowledge.

COMPETENTSIMHONEST This is an extension of `SIMPLESIMHONEST`, where for its simulations it assumes that everyone uses the `NOTCOMPLETELYINCOMPETENT` strategy instead. It also tries to gain information about other players' hands by replaying history using the `SETLISTNOTDUMB` strategy.

BULKSIMHONEST The same as `COMPETENTSIMHONEST`, but with the number of simulations increased from 100 per move to 1000 per move.

SMARTSIMHONEST This is an extension of `COMPETENTSIMHONEST`, where in the simulations it assumes players use the SMART strategy. Additionally when communicating it uses the same evaluation function as used in SMART.

Forced strategies

These strategies are exclusively used during the simulations done to check whether or not moves could/would have been made with certain starting hands. These strategies all attempt to follow the moves that have been played in the original game. Returning impossible values based on where this attempt went wrong.

SETLIST It will attempt to follow the list of moves present in the `GameController` class. If the move contains a black 7 somewhere, it returns a random card. If the move cannot be made for another reason it will return an appropriate object containing a black 9 to signify that.

SETLISTNOTDUMB Acts the same as `SETLIST` with regards to tasks and communications. However when a card needs to be played it checks whether or not the card it should play according to the list fulfills two conditions. The first is that this move is detected to result in a failed task. The second is that there is another move that does not do that. If both conditions are met a black 9 is returned instead.

SETLISTSMART Acts the same as `SETLISTNOTDUMB` with regards to the playing of cards. For tasks and communications it checks if the proposed task/communication aligns with the scores assigned to them. If this does not align an appropriate object containing a black 9 is returned.

3.3 Results

In order to obtain our results we ran 6 different simulation with 3 different task card amounts, see Table 3.3. For each strategy involving simulations we attempt to run them as many times as possible during the night, roughly 7 to 9 hours. An excel file containing the raw data can also be found at the GitHub repository[3].

	1 Task card	4 Task cards	9 Task cards
`RANDOM`	100.000	100.000	1.000.000
`SMART`	100.000	100.000	1.000.000
`SIMPLE_SIM`	10.000	3000	2000
`COMP_SIM`	2481	1000	1000
`SMART_SIM`	2500	1000	1500
`BULK_SIM`	1000	500	750

Table 3.3: Number of runs

From each run we save the `History` object and print its content to an excel file. The data this object contains is, the starting hands of each player, the tasks they chose (in order), the cards that were played and by who, in which round which player communicated what, the time the game took (in ms) and whether or not the game was won. Exceptions of this are the `RANDOM` strategy and the `SMART` strategy, for these we only recorded the amount of

[3]https://github.com/iwan382/crew

wins. There are two reasons for this, the first is that they are mainly there for comparison sake and their inner workings are pretty directly defined, so not much information is gained from this. The second reason is that since both take very little time to run a single game, exporting the data would be the large majority of the time it would be busy exporting data instead of running games. By removing the export function a million games were run in three and a half minute.

In Figure 3.2, 3.3 and 3.4, we have mapped the accuracy scores of all six algorithms, for 1, 4 and 9 task cards respectively. As the percentages where obtained by random trails we cannot be entirely certain that they are accurate. Because of this the graphs depict a 90% confidence bound. This means that every percentage below the box, has a 5% chance (or less) to achieve an equal (or better) accuracy score with the same amount of runs. The same goes for the percentages above it, except that they achieve an equal or worse result. This is calculated using the `Binom.Inv`[1] function in excel. This function calculates the minimal amount of successes needed to get at least the given chance as result from the `Binom.Dist` function, which is a Binomial Distribution functions. From the minimal amount of successes we can calculate the corresponding percentage for which the 5% boundary holds, but because the only return values can be whole numbers, the actual boundary has a small deviation.

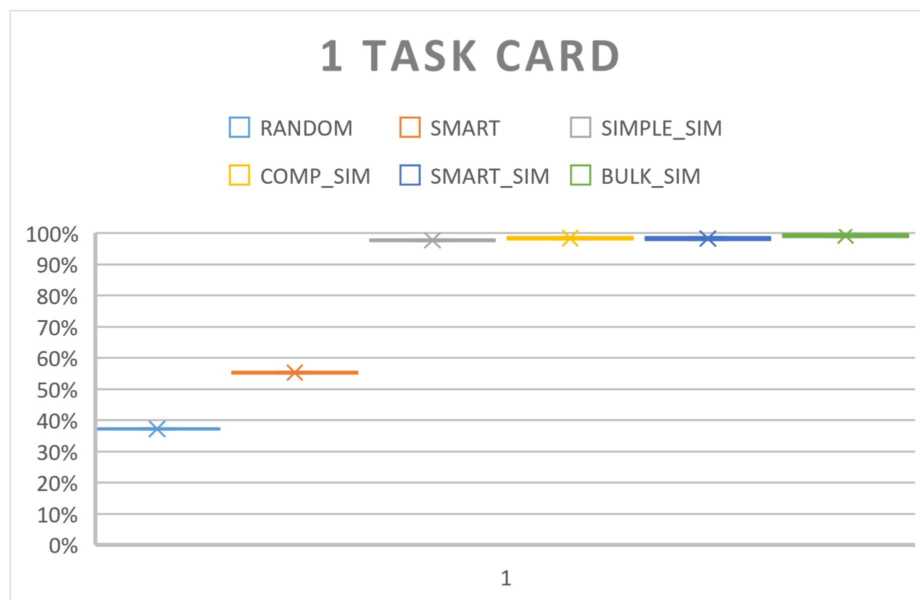

Figure 3.2: Accuracy with 1 task card

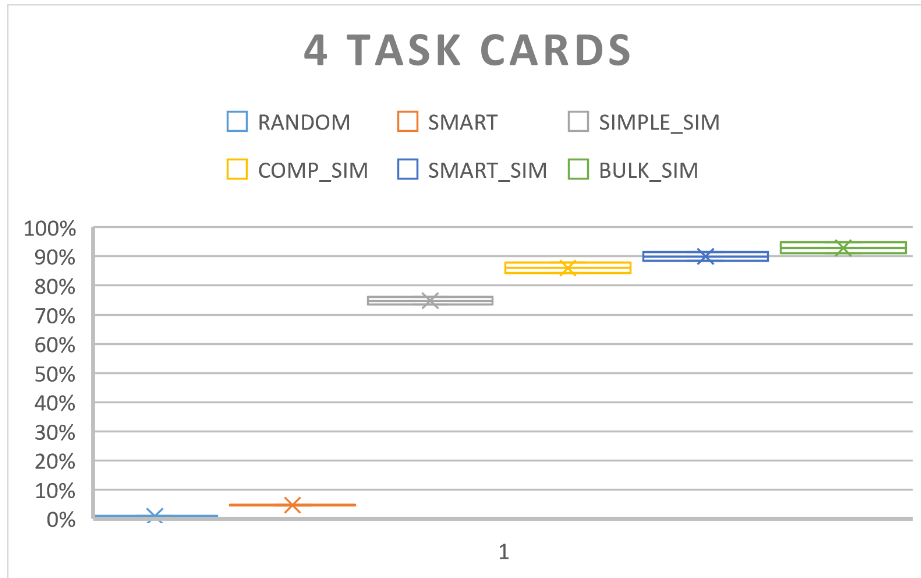

Figure 3.3: Accuracy with 4 task cards

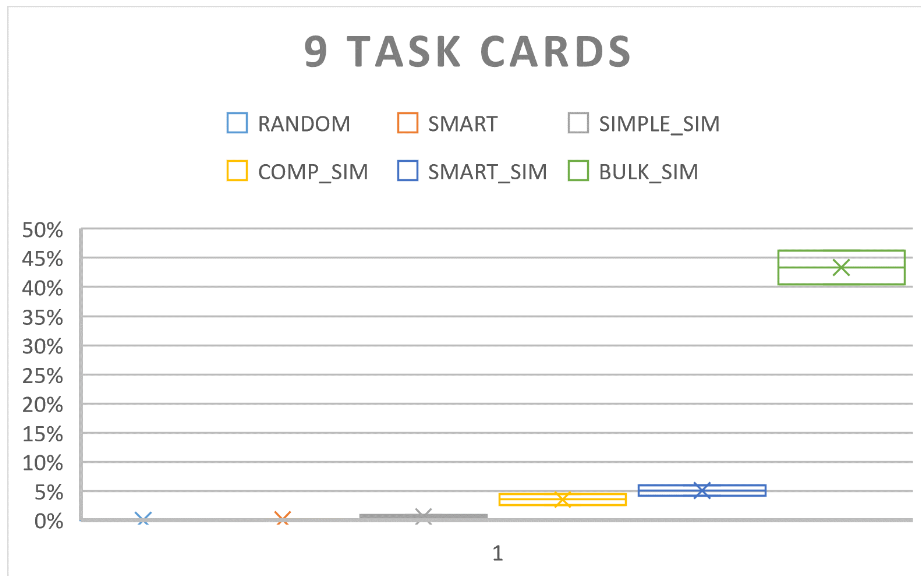

Figure 3.4: Accuracy with 9 task cards

A game with 1 task card is usually very easy, however due to the way the task cards are divided the captain always gets that task. This sometimes leads to a much more difficult game, in this scenario the task card has a low number e.g. a green 1 and the captain has that card in their hand. This means that the captain will at some point have to play a green 1, if any other player has a green card in their hand they will be forced to play it and win the card, thereby failing the task. This situation is made easier if the captain has a lot of that color in their hand, enabling them to root out that color before playing. This the case where the strategies that use simulations struggle the most with as very careful play is required.

When there are 9 task cards, every strategy has fallen dramatically in accuracy, with the exception of BULK_SIM, which now towers over the others. The data collected from the history objects does not show the cause of this. However when running separate games some output that is not stored in the history object provides the answer for this. When running the game like this players will print the moves they have available to them and how high the success rate of them was in the simulations however often every success rate will be 0.0%, see figure 3.5. This means that it will play effectively randomly. The BULK_SIM strategy has more often at least some of those percentages higher, ensuring it will not play randomly.

```
The currect round is 2
A communication phase starts.
Player 2 has decided not to communicate
Player 3 has decided not to communicate
Player 1 has decided not to communicate
There have been: 6 moves made
Playing a (GREEN, 5) won an estimated 0.0% of the times
Playing a (YELLOW, 2) won an estimated 0.0% of the times
Playing a (GREEN, 3) won an estimated 0.0% of the times
Playing a (GREEN, 2) won an estimated 0.0% of the times
Playing a (GREEN, 1) won an estimated 0.0% of the times
Playing a (YELLOW, 1) won an estimated 0.0% of the times
Playing a (GREEN, 8) won an estimated 0.0% of the times
Playing a (BLUE, 3) won an estimated 0.0% of the times
Playing a (BLUE, 2) won an estimated 0.0% of the times
Playing a (BLUE, 9) won an estimated 0.0% of the times
Playing a (BLACK, 1) won an estimated 0.0% of the times
Player 2, played a (GREEN, 5)
There have been: 7 moves made
Playing a (GREEN, 9) won an estimated 0.0% of the times
Playing a (GREEN, 4) won an estimated 0.0% of the times
Playing a (BLACK, 4) won an estimated 0.0% of the times
Player 3, played a (GREEN, 9)
There have been: 8 moves made
Playing a (GREEN, 7) won an estimated 0.0% of the times
Playing a (BLACK, 3) won an estimated 0.0% of the times
Playing a (GREEN, 6) won an estimated 0.0% of the times
Player 1, played a (GREEN, 7)
The currect round is 3
A communication phase starts.
Player 3 has decided not to communicate
Player 1 has decided not to communicate
```

Figure 3.5: Part of the simulation success rates of a single game

Chapter 4

Related Work

MCTS has become a very popular method to use in various games and other problems that are solved by algorithms. The use of simulations leads it to be usable in a wide variety of scenarios. Its versatility led to a lot of research being done with it. In our thesis we will attempt to make MCTS work with the new environment "The Crew" and take a look at its strengths and weaknesses. In this chapter we will give a brief overview of some of the research done with MCTS.

An overview reviewing recent modifications and applications with MCTS is made by Maciej, Konrad, Bartosz and Jacek[8]. This study reviews many techniques used to improve MCTS in a large amount of different environments. A way to examine how MCTS performs with limited information is by using MCTS based algorithms to play games and then examine how their performance is affected by limiting information. This is done by Piers R. Williams[7]. A study that focuses on cooperative games and how various degrees of partial observability affect performance, e.g. pac man, where they start limiting the amount of visible tiles to see its impact. They also mix computer agents with human players. War type games have a constraint of imperfect information. Additionally they have players take multiple actions per turn. This adds to the complexity as the branching factor will be much higher because there is one branch for each combination of actions. These studies[2][5] have a focus on this type of games. MCTS has to manage many different tasks, balancing building with fighting and exploration and more. With the goal to improve MCTS when there are more than 2 players, a study[3] was done by J. A. M. Nijssen and Mark H. M. Winands. It proposes and tests improvements to MCTS which help when there are more than 2 players in the game, although our environment does not allow these improvements to be used. How MCTS-based agents act in a cooperative environment is examined in a study[6] by Piers, Joseph, Diego and Simon. Here the agents try to complete a puzzle where they need to cooperate to push buttons and open doors. The less moves they need to complete it the

better their performance.

Chapter 5

Conclusions

Considering the 40 to 50% win rate on the 9 task card variant, we can conclude that MCTS can be adapted to function in an cooperative environment with information fog. We can also conclude that a situation were all simulations return a fail is likely to occur when the goal becomes more difficult. This is due to the goal remaining the same while, in the simulation, all players will work sub-optimally. When adapting MCTS to this environment one must take this into account. In this thesis two methods were used to prevent that situation from occurring, increasing the competence of the playout policy and increasing the number of simulations that were run. The latter seemed to be the most effective in these circumstances. Although since a random playout policy is also significantly worse when goal becomes more difficult, we can conclude that a competent playout policy is useful for boosting performance.

We can also see some of the strengths and weaknesses of MCTS. An important strength being that it is able to be adapted in a wide variety of situations. Another strength is that very little needs to be changes to the algorithm if we would change the rules of the game, or apply it to a different game that fits in this environment. The only thing that would need real change is the playout policy as it is tailored to The Crew. A weakness is that, due to not being able to run simulations within simulations recursively, moves that improve or worsen how well the other players play remain undetected. This is notable here with the communication opportunities. These would allow other players to reduce the number of possible hand distribution, however in the relatively simple playout policies there is no usage of this as no simulations are performed. This means that simulating different communications has no real use, hence why we do not do that.

In this thesis we restricted ourselves to only let all players use the same strategy. An interesting avenue to pursue would be how these algorithms perform when the players use different strategies. Additionally one could replace one or more players with a human player to see how this would impact

their performance. Another interesting avenue would be to try an approach based on a different method, one could try using a Heuristic search[4] or neural networks instead.

Bibliography

[1] `https://www.excelfunctions.net/excel-binom-inv-function.html`.

[2] Connor M. Pipan. Application of the monte-carlo tree search to multi-action turn-based games with hidden information. 03 2021.

[3] J. A. M. Nijssen and Mark H. M. Winands. *Enhancements for Multi-Player Monte-Carlo Tree Search*, pages 238–249. Lecture Notes in Computer Science. Springer, United States, 2011.

[4] Stuart Russel and Peter Norvig. *Artificial Intelligence A Modern Approach*, volume 4, chapter 6. Pearson, 2020.

[5] Dennis Soemers. Tactical planning using mcts in the game of starcraft. 06 2014.

[6] Piers Williams, Joseph Walton-Rivers, Diego Perez-Liebana, and Simon Lucas. Monte carlo tree search applied to co-operative problems. 09 2015.

[7] Piers R. Williams. *Artificial intelligence in co-operative games with partial observability*. PhD thesis, University of Essex, 02 2019.

[8] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte carlo tree search: A review on recent modifications and applications. 03 2021.