Comparing tools for Uniform Strategy Synthesis for Multi-Agent Systems with Imperfect Information and Imperfect Recall

AUGUST HEDDINI
KONRAD WIHL
Comparing tools for Uniform Strategy Synthesis for Multi-Agent Systems with Imperfect Information and Imperfect Recall

AUGUST HEDDINI
KONRAD WIHL
Abstract

This report aims to study and compare currently available tools for uniform strategy synthesis in multi-player versus environment games of imperfect information and imperfect recall. These games are represented as a graph of game states, and goal sequences of states are typically represented using temporal logic formulae. Strategies for achieving those goals can then be found by evaluating the formulae. Specifically the time complexity and addressed game objectives of the tools were of interest. Not too much is known about this particular field of game theory, and strategy synthesis is complicated and computationally heavy. Today there is only one completed tool specialized for the particular game structure of interest, the Strategic Model Checker (SMC).

Three tools were found which could be used for the task, out of which two seemed worthy of further investigation. Firstly, the aforementioned Strategic Model Checker (SMC) provided a good reference as a completed program. Secondly, an experimental version of the ATL\textsubscript{ir}, Model Checker, a newer tool still under development specialized for the same task, could provide an example at the cutting edge of the field. The evaluation of the literature as well as the tools confirmed and corroborated earlier findings in that there are few implementations available, that the time complexity of their performance is steep, albeit improving, and that they deal with a potentially very wide range of game objectives, which can be described using Alternating Time Temporal Logic, or ATL. The performance analysis concluded that the newer tool shows promising results for smaller games, but that SMC still produces more reliable results for larger instances as well as more complicated test cases.
Sammanfattning

Att undersöka konsten att fatta välavvägda beslut är av största intresse inom flertalet akademiska områden. Vare sig det gäller att utveckla artificiell intelligens med pålitlig prestanda, eller att undersöka det mänskliga psyket, så är förmågan att basera kritiska avgöranden på en begränsad mängd datapunkter definitivt mycket viktig. Detta är ett ämne som flitigt tas upp inom spelteori.

Spelteori är ett matematiskt område som studerar spel och deras egenskaper. Spelen kan representera nästan vilket scenario som helst där en mängd spelare försöker nå ett mål. Denna rapport behandlar ett särskilt område inom spelteorin, som beskriver s.k. flerspelar-spel med ofullständig information och ofullständigt minne. I sådana spel kan flera spelare delta, men de är inte alltid medvetna om spelets nuvarande tillstånd (ofullständig information), och de kan inte minnas något av sina tidigare drag (ofullständigt minne).

För att ytterligare förtydliga: Ofullständig information syftar på fall där vissa (eller samtliga) spelare inte kan skilja på vissa tillstånd i spelet. En sådan grupp av sammanblandade tillstånd kallas för en observation; spelaren i fråga observerar att den befinner sig i något av gruppens tillstånd, men inte vilket. Observationerna är också individuella - olika spelare har i regel olika observationer.

Ofullständigt minne innebär att spelaren eller spelarna inte minns vilka tidigare tillstånd de befunnit sig i; de kan således bara fatta beslut med avseende på vad de de observerar i nuläget. Man kan föreställa sig att spelarna är robotar utan minneslagring - de läser av spelets läge via sina sensorer, men de kan inte minnas vad som tidigare skett.


Vid syntesen av dylika strategier är det vanligt att beskriva spelarnas mål med hjälp av temporallogik-formler, och utvärdera dessa (Se stycke 2.2). Dessvärre är utvärderingsprocessen mycket prestandakravande, och nya tekniker krävs för att göra den genomförbar. Denna rapport syftar till att utforska och jämföra några av de nuvarande verktyg som finns tillgängliga för att finna och skapa enhetliga strategier i spel med flera spelare, ofullständig information och ofullständigt minne.

Denna typ av modeller kan användas för att undersöka många typer av sce-
narion, både inom beteendevetenskaper som ekonomi såväl som datalogiska ämnen, exempelvis AI och robotik. De senare kategorierna tjänar väl som konkreta exempel, eftersom AI-system kan hamna i situationer där de har tillgång till begränsad information om det rådande läget, och behöver fatta välgrundade, säkra beslut.
Contents

1 Introduction ........................................ 1
  1.1 Problem Statement ............................... 2
  1.2 Purpose ......................................... 2
  1.3 Approach ....................................... 3
  1.4 Restrictions .................................... 3

2 Background ........................................... 4
  2.1 Game Theory ..................................... 4
    2.1.1 Single-Player Games ....................... 4
    2.1.2 Multi-Player Games ....................... 6
    2.1.3 Objectives and Strategies ................ 8
    2.1.4 Knowledge Based Subset Construction ... 9
  2.2 Alternating-time Temporal Logic ............... 9
  2.3 Model Checking .................................. 10
  2.4 Strategy Synthesis .............................. 11
    2.4.1 The Model Checker for Multi-Agent Systems (MCMAS) 11
    2.4.2 Strategic Model Checker (SMC) ............. 12
    2.4.3 The ATL_{ir} model checker ............... 12
  2.5 The Castles game ................................ 13

3 Method ................................................. 14
  3.1 SMC .............................................. 14
  3.2 ATL_{ir} Model Checker ........................ 15
  3.3 Adaptations for the model checkers .......... 15

4 Results ................................................ 17
  4.1 Model Checker Comparison ..................... 17
  4.2 Viable game objectives ......................... 18

5 Discussion ........................................... 20
6 Conclusions 22

Bibliography 23

A Castles.ispl 25

B SMC Output 32

C ALT prep Output 34
Chapter 1

Introduction

Studying the art of watertight decision making is of prime interest in several academic pursuits. Whether it’s to develop artificial intelligence with consistent performance, or to gain a deeper understanding of human thought patterns, being able to base vital decisions on a limited set of data points is certainly of importance. This is a topic frequently brought up in game theory.

Game Theory is a mathematical discipline that studies the properties of some game which can, loosely, represent almost any scenario where some player(s) attempt to reach a goal. This report covers a certain field of Game Theory, namely that of multi-player games with imperfect information and imperfect recall. In such games, multiple players may participate but they might not always be fully aware of the current game state (imperfect information), and also cannot recall any previous moves (imperfect recall).

To be more specific: Imperfect information refers to the case when some (or all) of the players cannot differentiate between certain game states. Such a group of indistinguishable states is called an observation; the player in question observes that they are at one of the game states of the group, but cannot tell which one. Observations are player specific - one player may observe what another cannot.

Imperfect recall means that the player(s) have no memory of previous states; they can only make decisions based on what is currently being observed. A way to visualize this might be to imagine the player(s) as robots without memory units. They can still read the current input from their sensors to determine the game state, but they cannot remember any past inputs or game states.

In such a game one may then wish to find some method which will always lead the player(s) to victory. This can be achieved by analyzing the game and
coming up with (i.e. synthesizing) a set of guidelines for the player(s) to follow - a strategy, based on the current game state. The player(s) will then have to find actions which work in their favour regardless of what they currently observe. If this can be done, the resulting strategy will be called an observation based or uniform strategy.

When attempting to synthesize such strategies it is common to describe the objective (sometimes called goal) of the player(s) using temporal logic formulae which can then be evaluated (for examples see 2.2). However, this evaluation is very computationally demanding, and novel techniques must be employed to make it practical and affordable. This report aims to explore and compare some of the currently existing techniques for finding and creating uniform strategies in games with multiple players, imperfect information and imperfect recall.

These ideas can be used to study many different scenarios, ranging topic-wise from behavioural sciences like economics, to AI and robotics. The latter categories can be used as quite concrete examples, as AI-systems might be put in situations where they receive limited information about the current state of affairs, but will still have to perform water-tight decision making.

1.1 Problem Statement

The problem this project seeks to address is that of synthesizing observation based strategies for multi-player games of imperfect information and imperfect recall. In particular, the following questions are of interest:

- What algorithms and tools are available for solving it?
- What is the practical time complexity of this problem?
- What types of game objectives do existing solutions deal with?

1.2 Purpose

This particular branch of game theory may be an interesting topic to expand upon as there could be many real life scenarios where the agents involved do not know all the relevant variables, and where a strategy synthesis tool might prove useful.

Given that this field of research is relatively recent, its nature is still somewhat abstract and it can thus be difficult to describe and quantify wider societal impact. However, it is not unreasonable to suggest that any potential insights
gained in this project might contribute to the general body of game theory and its ability to describe scenarios in AI, robotics and economics etc.

1.3 Approach

The study will be conducted by reviewing the relevant literature and searching for tools which have good support for performing strategy synthesis on game graphs of the type described in this report. Finally performance tests will be run on any tools found in order to get a clearer picture of the complexity of the problem at hand. The goal is then to compare and discuss the results garnered from the above process.

1.4 Restrictions

This project is primarily restricted to only consider uniform strategy synthesis in multi-player versus nature games of imperfect information and imperfect recall. This restriction is to keep the scope minimal and realistic.

It is worth noting that all games considered in this report are theoretically infinite, but as temporal logic formulae are evaluated, there is no need to consider game rounds after the formulae have been validated or dismissed. As such, the game effectively ends after a strategy has been found, or found not to exist.

For the comparison of model checkers (See Chapters 3 and 4) the Castles game was chosen, as it has several easily modifiable variables which were deemed useful for testing the model checker performance both when the size of the graph was changed and when the number of players was changed.
Chapter 2

Background

2.1 Game Theory

"Game theory can be defined as the study of mathematical models of conflict and cooperation between intelligent rational decision-makers." [1]

This study deals with a specific field of game theory known as multi-player games of imperfect information and imperfect recall, as well as the associated logic formulae of alternating-time temporal logic, abbreviated ATL (see section 2.2) [2]. In essence, one attempts to model games (systems in which players seek to reach some type of objective) as discrete graphs consisting of nodes representing game states, and edges representing valid transitions between those states. A game is then played on this graph by some number of players for infinitely many rounds. In each round, every player must choose an action to take, and depending on their choices the game state is changed. The following subsections describe such games in more detail, using notation from earlier KTH bachelor’s students Nylén and Jacobsson [3] which in turn is based upon older notes from the more senior team of Goranko, Gurov and Lundberg [4].

2.1.1 Single-Player Games

Mathematically the game graph mentioned above is denoted:

\[ G = (L, l_0, \Sigma, \Delta) \]  [4]

Here, \( G \) is the game, which consists of a tuple of information about the structure of the game graph. \( L \) is the set of all possible states in the game \( G \), meaning
they make up the vertices of the graph, and \( l_0 \) is the initial state where the game begins. \( \Sigma \) is the set of all possible actions, for example it could be the set of every possible move any piece can make on a chess board. \( \Delta \) is then the set which describes which actions \( \sigma \) are legal in which game states \( l \), meaning \( \Delta \) describes the edges of the game graph. Define \( \Delta \) such that \( \Delta \subseteq L \times \Sigma \times L \), meaning that performing an action \( \sigma_i \) in game state \( l \) resulting in the game reaching state \( l' \) is a legal move iff \( (l, \sigma_i, l') \in \Delta \). In other words this notation describes that when in game state \( l \) performing action \( \sigma_i \) would move the game to state \( l' \), but only in games where \( (l, \sigma_i, l') \in \Delta \). In the chess example, it would be \( \Delta \) which describes that you cannot move a rook along a diagonal, simply by not containing such a transition [5].

The model above describes only a single-player game with perfect information. This report studies games with imperfect information which means that individual players do not necessarily have full knowledge of the current game state at any given time. In the model this is represented by introducing a set of equivalence classes of states (also known as the set of observations, \( O \)), meaning that the player \( p \) cannot differentiate between states in a particular observation \( o \in O \). Each observation \( o \) then contains some number of game states \( l \in o \), which are indistinguishable to that player. In other words, in games of imperfect information, a player does not know about the currently held state but rather they know the current observation. Whenever an observation contains only one game state, it is known as a singleton and is regarded as equivalent to that state. Adding this to the mathematical definition above one must add the set of observations \( O \) to \( G \) [4].

\[
G = \langle L, l_0, \Sigma, \Delta, O \rangle
\]
Figure 2.1: An example single-player game of imperfect information.

In Figure 2.1 we see an example of a single-player game of imperfect information. The player will start at the state $l_0$, and can then perform actions $\sigma_1 - \sigma_3$, depending on which state they are currently in. The states $l_2$ and $l_3$ are bundled together in the observation $O_1$, which makes them indistinguishable.

2.1.2 Multi-Player Games

Upon expanding this definition of games to include multi-player games several of the constituent parts of the game must be changed to contain information about each player and their actions and observations. The elements $\sigma \in \Sigma$ are extended to instead be tuples consisting of one action per player $p \in \Pi$, where $\Pi$ is the set of all players. Then $\sigma = \langle a_0, a_1, ... a_n \rangle$, where $n$ is the number of players, and each $a_i$ is an action of player $p_i$. $\Sigma$ then contains all permutations of such tuples, and $\Delta$ still limits this set by describing the legal actions $(l, \sigma, l')$ in each state $l$. Additionally, $O$ must now contain subsets, one per player, composed of the observation(s) held by that player. As such, denote $O = O_1 \times O_2 \times ... \times O_n$, where $O_i$ is the set of observations for player $p_i$ [4].
Figure 2.2: An example multi-player game of imperfect information. The action tuples \([a_i, b_j]\) now represent the actions of players \(a\) and \(b\) respectively.

Figure 2.3: The same multi-player game as above (figure 2.2) but with some adjustments.

In figure 2.2 the previous game has been expanded to account for two play-
ers $a$ and $b$. Note that each unique action tuple $\sigma_i$ now shows individual components for each player. All action tuples which are not present at some state $l_i$ can be regarded as a loop back to the same state, or if one wishes, as simply being illegal for that state, depending on the game being modelled. Moreover, an observation $O_{b[1]}$ has been added to specify that player $b$ cannot distinguish between states $l_0$ and $l_3$.

### 2.1.3 Objectives and Strategies

A strategy can be defined by deciding on an action tuple $\sigma$ to take (edge to traverse) for every game state $l$. This strategy is defined as a function $\alpha : L \rightarrow \Sigma$. In this study only uniform strategies will be examined, meaning that if $l_i, l_j \in o_w, o_w \in O_k$ for some player $p_k$ then $\alpha_{p_k}(l_i) = \alpha_{p_k}(l_j)$. In other words the action decided by the strategy must be the same for every location in an observation. In effect this ensures that the strategy is not impossible for the players to follow. It is worth noting that the players also do not communicate and have no memory of prior moves and are thus oblivious to any information held by another player, and to any information gained from studying the game history [4].

One particularly interesting subset of strategies are those that, when followed, cause the game to always reach some game state(s) which are defined as the goal/winning game states. Such a goal can be known as for instance a reachability objective $\text{Reach}(\tau)$, or a safety objective $\text{Safe}(\tau)$. The reachability objective $\text{Reach}(\tau)$ requires that the game at some point reaches the goal state $\tau$, while a safety objective $\text{Safe}(\tau)$ also requires that the game never leaves the goal state $\tau$ again, i.e. there must exist some action $\sigma \in \Sigma$ such that $(\tau, \sigma, \tau) \in \Delta$. A strategy which ensures that the goal is reached is called a winning strategy. In this report, however, the objective is more commonly described using temporal logic formulae, explained further in section 2.2 [6].

Consider a scenario in the game depicted in Figure 2.1, where the player wishes to complete the objective $\text{Reach}(l_4)$, and cannot recall previous moves. An observation based strategy for this would be as follows:

$$\alpha : L \rightarrow \Sigma = \{ \alpha(l_0) = \sigma_2, \alpha(l_1) = \sigma_2, \alpha(O_1) = \sigma_1, \alpha(l_4) = \sigma_2, \alpha(l_5) = \sigma_1 \}$$

Following the strategy, we see that performing $\sigma_1$ in $O_1$ will take the player closer to the goal regardless of which state in the observation they are at. This is what separates the particular objective from one which has no solution. For instance, if the goal were to reach $l_5$, we see that there is no observation based
strategy to complete the objective, as the player cannot know whether to perform \( \sigma_1 \) or \( \sigma_3 \) while in \( O_1 \). Note that all states except those in \( O_1 \) are technically individual observations as well, albeit singletons.

To illustrate the matter of expanding this strategy synthesis problem to a multi-player context, consider figure 2.2. Here, the objective of reaching \( l_4 \) has no uniform strategy, because of the uncertainty caused by the overlapping observations. Once the game reaches \( l_3 \), the players cannot proceed to \( l_4 \) by using the same actions as they did in the previous states in their respective observations. However, in figure 2.3 the action tuples have been modified to remedy this. Now, player \( a \) can simply perform \( a1 \) whenever observing \( O_{a[1]} \), and \( b \) can similarly perform \( b1 \) whenever observing \( O_{b[1]} \).

### 2.1.4 Knowledge Based Subset Construction

One way of dealing with imperfect information is to use a method known as knowledge based subset construction, or KBSC. This approach involves transforming the game graph \( G \) into a new graph \( G^+ \), which is larger, but describes a game with perfect information. The method can be applied with relative practical computational ease, and strategies can then be generated for the resulting graph utilizing methods like the attractor method (see Section 2.4). There have also been recent attempts at generalizing the KBSC to multi-player games [3]. However, it turns out that finding a memoryless strategy in \( G^+ \) will not necessarily yield one for the original game. Rather, it can be translated to generate a strategy of finite memory (meaning with some memory of past game states) in single-player games, and for multiple players there is no way to ensure that all imperfect information is removed [4]. Seeing as this report seeks to synthesize memoryless strategies for the latter case, these approaches cannot be used.

### 2.2 Alternating-time Temporal Logic

Alternating-time Temporal Logic (ATL) is a more general alternative to Computational Tree Logic (CTL). ATL is very well suited to describing and evaluating statements in multi-player versus environment games, and is used heavily in the type of game system this report concerns [2].

For example, if one considers some game \( G = \langle L, l_0, \Sigma, \Delta, O \rangle \) with players \( p \) in \( \Pi \) as described in section 2.1 above, one could then use ATL to create logic statements like \( \langle\langle C \rangle\rangle \models \phi \). This statement would describe the ability of some coalition of players \( C \subseteq \Pi \) to achieve game state \( \phi \) at some
chapter 2. background

point in the game. Note that since \( C \) is defined as a subset of \( \Pi \), it may contain any number of the players in \( \Pi \). If \( C = \Pi \), then it is called a "grand coalition".

Since ATL is a temporal logic, one can formulate the goal using temporal operators, \( X \), \( F \), \( G \) and \( U \). These represent "on the next turn", "finally", "globally" and "until", respectively. For instance the temporal logic statement \( \langle\langle C \rangle\rangle |:= G \phi \) would describe the coalition \( C \)'s ability to cause the game to reach state \( \phi \) in the next turn of the game and remain there forever, causing the game state \( \phi \) to be globally true from then on [2]. Notice that this models the behaviour of a Safe(\( \phi \)) objective as described in section 2.1.3. The most complicated of the four to understand is arguably the "until" operator \( U \). \( \langle\langle C \rangle\rangle |:= \phi U \gamma \) describes the coalition \( C \)'s ability to cause the game to reach state \( \phi \) and remain there until game state \( \gamma \) is reached. For instance this might model the ability of some team in a video game to capture a control point and hold it until they gain a point. In a more serious scenario it might describe the ability of a search-and-rescue crew to find a person in need of rescue and then stay with them until their wounds have been treated. It is also intuitive that statements like \( \langle\langle C \rangle\rangle |:= \langle\langle D \rangle\rangle F \phi \) (describing the coalition \( C \)'s ability to enable the coalition \( D \) to achieve something else) are possible under ATL syntax, however such formulae will not be considered in this report since they are much harder still than the regular case to synthesize strategies for and often not accepted as input for existing tools [6][2].

It is known that model checking for ATL (in essence, the construction of a winning strategy) is a PSPACE-complete problem [2][7][8]. Thus some more clever and effective algorithms for model checking are required.

2.3 Model Checking

Model checking refers to the task of validating whether or not some properties hold within a given model. This can be done by representing the model as a graph/automata (or in this case, a game), formulating the relevant properties using temporal logic and then computationally traversing the graph to determine the validity of said properties [9]. A naive approach to the problem at hand would be to simply perform an exhaustive test of all possible strategies, which obviously quickly becomes very computationally heavy. Therefore the main problem in building a model checker for use in strategy synthesis is to create algorithms for reducing this search space. For instance, Pilecki et al. [6] have found ways to 'remove' or ignore parts of the game graph that they consider unfavorable to reach. If one does not plan to ever reach some game state, then no strategy needs to be defined in that state to find a practical solu-
The remaining set is then reduced even further by weeding out effectively equivalent strategies, and finally path-finding heuristics are employed to find a successful strategy, if one exists.

2.4 Strategy Synthesis

Strategy synthesis is the process of algorithmically generating winning strategies $\alpha$ for a given game $G$. In a game with perfect information, where all players perfectly know the game state, there are known algorithms which consistently synthesize winning strategies if such strategies exist. A common and reliable method for this is to begin at the goal state and then iterate backwards to the starting game state. This is achieved by letting all neighbour states of the goal state have the strategy of moving to the goal state, and all neighbours of those states have the strategy to move to the state they neighbour; and so on until the starting game state has been reached. This is often referred to as the attractor method [10], and can be performed in linear time with respect to the graph size. In a game of imperfect information this process of synthesis is not guaranteed to work, as each game state cannot always be assigned a unique action in a uniform strategy. The problem of strategy synthesis for games with imperfect information has in fact been proven to be $\Delta_2^P$ to $\text{PSPACE}$-complete [6]. To create useful algorithms it then becomes necessary to find and utilize some heuristic to reduce the problem size. The model checkers (see Section 2.3) covered in this report make effective use of different heuristics to perform strategy synthesis for a subset of games.

Since model checkers work by evaluating different possible strategies, model checking becomes a way to synthesize winning strategies. If some model checker can efficiently evaluate ATL-formulae within a model with imperfect information, then the task of synthesizing a strategy becomes possible. By testing every possible strategy, or even simply some 'strong candidate' strategies, synthesis is trivially performed by returning any successful strategy found.

2.4.1 The Model Checker for Multi-Agent Systems (MC-MAS)

The Model Checker for Multi-Agent Systems (MC-MAS) is an open source model checker specifically tailored for models with multiple agents. The tool accepts input in a language called ISPL (interpreted systems programming language) and comes with a variety of features, including checking of CTL and
ATL formulae. MCMAS also includes a feature for providing counterexamples to formulae it finds untrue. It is possible to make use of this feature for strategy synthesis, by asking the tool to test the negation of a formula. For example, if a strategy is to be synthesized for $\langle \langle C \rangle \rangle \models F\phi$, then MCMAS could be used to verify $\langle \langle C \rangle \rangle \models !F\phi$. A counterexample to the latter would intuitively be a successful example to the former. This counterexample then becomes the synthesized strategy. However, the unmodified tool cannot check models with imperfect information [7], and as such MCMAS will not be studied in-depth in this report.

2.4.2 Strategic Model Checker (SMC)

The Strategic Model Checker (SMC)[6], is a tool which can check ATL statements in multi-player games of imperfect information and imperfect recall by weeding out redundant and uninteresting cases from the set of all possible strategies. It does this by algorithmically finding 'practical equivalences' of strategies with regards to the goal. For instance it chooses to ignore any and all game states which cannot be reached or which create 'dead ends' with regards to the goal state. It is then only necessary to check one strategy of each 'practical equivalence', reducing the problem size significantly [6].

Following this the SMC tool applies a simple heuristic to the remaining solution set in order to attempt to produce a suitable strategy. Much like MCMAS, SMC uses ISPL as the input format in order to describe the game graph and the formulae to be checked. This language makes SMC input files highly modifiable, and allows the tool to be used for any game graph which can be described using ISPL syntax.

2.4.3 The ATL$_{ir}$ model checker

The ATL$_{ir}$ model checker is a tool currently under development by researcher Damian Kurpiewski of the Polish Institute of Sciences [11]. It is designed to work similarly to SMC, but using a variety of techniques for the actual strategy synthesis task. So far, test code exists for a range of specific problem instances, using both a depth first search approach, and a so called fixpoint approximation technique. The former can currently be used to find strategies for simple test cases, and output them. It is somewhat dubious to make broad statements about the details of the ATL$_{ir}$ model checker, as it is still in development, and its inner workings may change before its release [11]. However, it is designed to take on the exact same types of games as the SMC tool, and therefore sits
very well within the scope of this report.

2.5 The Castles game

One example of multi-player games of imperfect information and imperfect recall is the castles game. It was originally designed by Pilecki et al [6] (see section 5.1) for performance testing reasons. It will be used in this report as well, as it offers several ways to scale the game graph and both strategy synthesizing tools examined already have implementations of it for testing purposes.

In this game three castles are at war. Each castle has health points, represented as an integer as well as a number of workers, which are the players of the game. Each player is assigned as a worker to some castle and may take any of three actions each round, namely attacking another castle, defending their own, or doing nothing. No worker may defend their own castle for two consecutive turns. In any round where the number of workers attacking a castle is greater than the number of defending ones for a particular castle, it loses health points equal to the difference between attackers and defenders. All workers know the health points of their own castle, but not the other ones. For those, they may only see whether or not the opposing castles have been defeated (reduced to 0 health points or less). Once a castle is defeated, its workers only available action is to do nothing, effectively taking them out of the game.

This game fits the description of games concerned in this report by incorporating both multiple players, imperfect information and imperfect recall. How multi-player and imperfect recall fit the Castles game scenario is evident, but it may be worth explaining where imperfect information arises. The players of the game can not tell the difference between the different health point levels of the opposing castles, other than 0. This creates several observations for each player. However, it is important to note that the rule stating that players cannot defend a castle on two consecutive turns, combined with imperfect recall, also gives rise to several observations. A player can not tell whether the opposing players are eligible to defend their castle on any given turn. As such the castles game is a good fit for this report.
Chapter 3

Method

After a number of model checkers had been found, these were individually evaluated to find some model checkers which were most suited to the problem discussed in this report. The model checkers were evaluated based on which types of game graph they were specialized for. Favoured were those that were designed to work for specifically uniform strategy synthesis in multi-player games with imperfect information and imperfect recall.

The chosen model checkers were used to synthesize strategies for some variants of the Castles problem, to compare their efficiency under a number of variables. The variables chosen were: number of players per castle and number of health points per castle. The reasoning behind the choice of these parameters was quite simple; they were the only available variables which could be adjusted to change the game graph using the input formats of the tools. Furthermore it was reasoned that the health points parameter would primarily increase the graph size, by adding more possible permutations of health states for the castles. On the other hand, altering the number of workers per castle should increase the set of available moves/transition between states, resulting in a greater number of edges in the game graph.

All experiments were run using an HP Pavilion Notebook running Ubuntu 18.04, with a timeout limit of 40 minutes (not including the time for model generation and set up, only the strategy synthesis).

3.1 SMC

The Strategic Model Checker was used to synthesize strategies in some example games of Castles. Using the tool "castlesGenerator", provided by the authors of SMC it was possible to generate ISPL files describing games with
three castles versus the environment. It was also possible to specify the number of players working for each castle, and this was used to create different test cases. To alter the maximum health points (HP) of each castle it was necessary to edit the ISPL file directly. This edit was quite simple. In the generated ISPL file the HP of each castle is represented as a vector 0..n, where n is the max HP. This value is stored as a variable castle\textsubscript{x}HP for each castle \( X \) in the Environment (under Vars). Then it was also necessary to change the value of Environment.castle\textsubscript{x}HP under InitStates, as the castle hit points would otherwise be initialized to a lower value than their maximum. Both of these edits were easily performed using a text editor (See Appendix A).

### 3.2 ATL\textsubscript{\textsc{ir}} Model Checker

The ATL\textsubscript{\textsc{ir}} Model Checker was also used to synthesize strategies for some configurations of the Castles example. Modelling the game graph was done as an executable Python3.6.5 file, most of which had been done in advance by the tool’s developer Damian Kurpiewski. The test model used for the ATL\textsubscript{\textsc{ir}} model checker uses a depth first search (DFS) approach to attempt to reach the given objective. This currently appears to be the only implementation of the castles game for the ATL\textsubscript{\textsc{ir}} model checker which is able to find and output a strategy, and could thus be adapted to compare with SMC. It has been shown by Jamroga et al. that the other implemented approach (fixpoint approximation) is not applicable to the castles game, because of the structure of the game [11].

### 3.3 Adaptations for the model checkers

The ISPL syntax for describing game graphs used in the SMC and MCMAS, as well as the syntactic style used in the preliminary ATL\textsubscript{\textsc{ir}} Model Checker slightly differs in its definitions of the game constituents from the ones used in this report. More specifically, the games described in ISPL format (and the current python files used in the ATL\textsubscript{\textsc{ir}} Model Checker) do not consider observations as sets of game states directly. Rather the language represents the observations as variables belonging to the game states. These variables may then be observed by the players, and any game states for which the variables a given player \( p \in \Pi \) can observe are the same belongs to the same observation \( o \in O_p \). For example each game state may have a temperature variable, representing the temperature of the local environment. A player who can observe the temperature would then see all game states with the same temperature as
part of the same observation, but if the temperature differed the player would be able to tell them apart. Different players observe different variables, and consequently end up with different equivalence classes of game states. To exemplify, consider a player \( p \in \Pi \) who observes an observation \( o \in O_p \). Previously, this scenario has been described by simply stating that \( l_i \sim l_j \forall l_i, l_j \in o \). However, in ISPL the same scenario is represented by introducing a set of logical variables \( Q_i = \{q_{i,1}, ..., q_{i,n}\} \) which are present (but may differ in value) at every game state \( l_i \in L \). A player \( p \in \Pi \) will then observe some set of these variables \( O_p = \{q_{\_i}, q_{\_j}, ...\} \). Thus observations are formed for the player as all game states \( \{l_i, l_j \in L \mid q_{i,k} = q_{j,k} \forall q_{\_k} \in O_p\} \). Functionally both these ways of representing the observations are the same, i.e. the game graph described is equivalent.

Another key difference between the games described by Lundberg et al. as well as Nylén and Jacobsson, and the ones used to evaluate the performance of SMC and the ATL\(_{ir}\) Model Checker is that of coalitions. Both of the tools are capable of storing and processing several coalitions of players at once, while the aforementioned sources mainly discuss games of the type grand coalition versus nature\([4][3]\). This is reflected in the Castles game 2.5 where the workers of the three castles may be grouped into arbitrary coalitions who then compete against each other. However, in practice, with ATL-formulae only involving one coalition, the game can effectively be viewed as grand coalition versus nature, where the opposing coalition(s) become non-deterministic properties of the nature agent. In essence, the coalitions are a convenient way of toggling between different partitions of \( \Pi \) - each partition containing two subsets: the grand coalition of players, and the remaining agents, who represent nature. This becomes possible considering that the formulae being evaluated are of the type 'can \( (\langle C \rangle) \) achieve \( \phi \) regardless of what the other agent(s) (nature) do?'
Chapter 4

Results

After the literature study it stood clear that SMC was the most relevant tool. It is designed to synthesize uniform strategies for multi-player games of imperfect information and imperfect recall, which is exactly what this thesis concerns. Upon further investigation, additional information arose regarding a newer tool for the same class of problems. The tool is still under development and was recommended by Wojtek Jamroga, one of the researchers behind SMC [6]. Acquiring a preliminary version of this tool allowed for a more rigorous complexity analysis, as the tools could then be compared and contrasted. After testing both programs on several variations of the Castles test case the following results were gathered:

4.1 Model Checker Comparison

Presented in this section is the data gathered for analysis and comparison of the two chosen Model Checkers, SMC and the unreleased ATL$_{ir}$ MC tool. For hardware details, see chapter 3.

For the following tables the evaluated formula was the reachability objective:

\[
\langle\langle\text{Castle1}, \text{Castle2}\rangle\rangle \models F \text{Castle3Defeated}
\]

When analyzing the times recorded in Table 4.1 it becomes clear that the ATL$_{ir}$ model checker performs significantly better for small game graphs, but also significantly worse as the problem size grows. Specifically the ATL$_{ir}$ model checker seems to struggle when the total player count exceeds 6 agents, and when some single castle has significantly more players assigned to it than the others. It is also clear that even for small problem sizes the execution time
Table 4.1: The real time used to solve the castles problem by the two model checkers. Timeout was set to 40 minutes. Any larger problem size caused a timeout for both tools.

<table>
<thead>
<tr>
<th># players/Castle</th>
<th>SMC</th>
<th>ATL_{\alpha}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
<td>1.138s</td>
<td>0.097s</td>
</tr>
<tr>
<td>2 1 1</td>
<td>4.995s</td>
<td>0.775s</td>
</tr>
<tr>
<td>3 1 1</td>
<td>41.420s</td>
<td>25.318s</td>
</tr>
<tr>
<td>4 1 1</td>
<td>691.593</td>
<td>2218.579</td>
</tr>
<tr>
<td>2 2 1</td>
<td>29.652s</td>
<td>22.178s</td>
</tr>
<tr>
<td>2 2 2</td>
<td>325.902s</td>
<td>121.802s</td>
</tr>
<tr>
<td>3 2 1</td>
<td>196.473s</td>
<td>TIMEOUT</td>
</tr>
</tbody>
</table>

Table 4.2: The real time used to solve the castles problem where the health of Castle 3 is varied by the two model checkers. In each case the Castles were assigned one player each. No strategy found means that the execution of the tool finished before the timeout limit, but that no possible strategy was found.

<table>
<thead>
<tr>
<th># HP/Castle</th>
<th>SMC</th>
<th>ATL_{\alpha}</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 3 4</td>
<td>3.007s</td>
<td>NO STRATEGY FOUND</td>
</tr>
<tr>
<td>3 3 5</td>
<td>3.928s</td>
<td>NO STRATEGY FOUND</td>
</tr>
<tr>
<td>3 3 6</td>
<td>4.653s</td>
<td>NO STRATEGY FOUND</td>
</tr>
<tr>
<td>3 3 7</td>
<td>NO STRATEGY FOUND</td>
<td>NO STRATEGY FOUND</td>
</tr>
</tbody>
</table>

4.2 Viable game objectives

The main restriction found on the viability of strategy synthesis for game graphs was clearly the problem size. Even for very small games the computation time increased drastically with the addition of just one or two more players, and to model any more complex problems is practically impossible as a result. Other than this any objective which can be described using ATL formulae are viable in tools using the ISPL input language, as long as only one
coalition of players is involved. This is due to the ISPL syntax and how the language models games.
Chapter 5

Discussion

Clearly, progress is being made in tackling the strategy synthesis problem, not only in the theoretical domain, but also in that of implementation. Both the tools which we were able to evaluate have been implemented in the last decade. Still, the time complexity of the problem appears daunting, as exemplified by the performance analysis in the results chapter (See Section 4.1 & Table 4.1). Both tools show a steep increase in execution time as more players are added to the winning coalition, which expands the number of possible game states and consequently, the game graph. SMC shows this same steep increase in execution time as the health of the third castle is increased, as shown in table 4.2. It is possible that with further advancements in computer science and technology this time complexity becomes more surmountable in practice and that strategies for more practically useful games might be synthesized, but today it only seems possible with a large amount of computing resources or for small, highly specialized games.

It is also promising that new tools are being developed for the strategy synthesis problem, as this shows that the academic world still believes there are improvements to be made. From the results of our literature study it is clear that there are not many effective tools for the specific case considered in this report. The most prominent tools for strategy synthesis seem to have been SMC and a modified version of MCMAS, both of which are quite slow.

Reviewing the results from the performance tests, we can make a few interesting observations. Firstly, the ATL model checkers outperform SMC for smaller problem instances, but it seems that whenever 6 or more players are unevenly distributed among the castles the algorithm runs into problems and becomes significantly slower than its counterpart. This might be due to the nature of the DFS-based approach that was used, as this may work very well
for smaller problem sizes but lack in efficiency as the complexity of the game graph grows. The results also bode well for the future of the ATL$_{ir}$ tool. If it can find ways to solve larger problem sizes with the same efficiency as the smaller, it will be the most effective tool by quite a margin. We also believe these results speak volumes to the great complexity of the strategy synthesis problem, as it highlights the exponential nature of the computations required to solve it, with regards to problem size.

Secondly, SMC was able to produce strategies for more test cases than the ATL$_{ir}$ model checker, as exemplified when modifying the health points of the third castle (which was the coalition to be defeated, see Table 4.2). SMC seemed able to generate winning strategies for at least some instances of this problem type, whereas the ATL$_{ir}$ model checker was unable to do so even when the health points of the enemy castle was only one step higher than that of its individual opponents, instead giving a false negative. This is not surprising, however, as the tool is still in development and it might be the case that none of the strategy synthesis algorithms are fully realized.
Chapter 6

Conclusions

In conclusion we can see that there are not many tools available to perform uniform strategy synthesis for multi-player games of imperfect information, but more refined tools are under development and progress is being made towards reducing the practical complexity of the problem. Tools like the Strategic Model Checker (SMC) or the yet-unreleased ATLr MC exist and are able to reliably synthesize strategies for simpler game graphs, but it is likely that many more complex games would require unrealistic computing power. This is due to the fact that the time complexity of the strategy synthesis problem for this particular flavour of game is at least $\Delta_2^P$-complete [6], and may be even greater. However, using heuristics to reduce the search space of possible strategies the tools mentioned above have achieved success in synthesizing strategies with somewhat better performance.

It seems from our testing that any Alternating-time Temporal Logic (ATL) formulae involving only a single coalition of players can have strategies synthesized for them using these tools,
Bibliography


Appendix A

Castles.ispl

The following is the contents of the castles.ispl file as generated by the castles-Generator.jar file provided with the SMC tool. The following ispl code describes the castles game for 3 players, one assigned to each castle.

--- System notice: Assuming default scalability factors.
--- Castles v 0.0.2
--- N[castle1]==1
--- N[castle2]==1
--- N[castle3]==1
Semantics=SingleAssignment;
--- SemanticsOfProtocol=SingleAssignment;

Agent Environment
Obsvars:
castle1Defeated:boolean;
castle2Defeated:boolean;
castle3Defeated:boolean;
end Obsvars
Vars:
castle1HP:0..3;
castle2HP:0..3;
castle3HP:0..3;
end Vars
Actions = {doNothing};
Protocol:
Other: {doNothing};
end Protocol
Evolution:
castle1Defeated = true if !(Worker1.Action = defend) and !(Worker2.Action = attack1) and Worker3.Action = attack1 and castle1HP < 2;
castle1Defeated = true if !(Worker1.Action = defend) and Worker2.Action = attack1 and !(Worker3.Action = attack1) and castle1HP < 2;
castle1Defeated = true if !(Worker1.Action = defend) and Worker2.Action = attack1 and Worker3.Action = attack1 and castle1HP < 3;
castle1Defeated = true if Worker1.Action = defend and Worker2.Action = attack1 and Worker3.Action = attack1 and castle1HP < 2;
castle1HP = 0 if !(Worker1.Action = defend) and !(Worker2.Action = attack1) and Worker3.Action = attack1 and castle1HP < 2;
castle1HP = 0 if !(Worker1.Action = defend) and Worker2.Action = attack1 and !(Worker3.Action = attack1) and castle1HP < 2;
castle1HP = 0 if Worker1.Action = defend and Worker2.Action = attack1 and Worker3.Action = attack1 and castle1HP < 2;
castle1HP = castle1HP−1 if !(Worker1.Action = defend) and !(Worker2.Action = attack1) and Worker3.Action = attack1 and castle1HP > 1;
castle1HP = castle1HP−1 if !(Worker1.Action = defend) and Worker2.Action = attack1 and !(Worker3.Action = attack1) and castle1HP > 1;
castle1HP = castle1HP−1 if Worker1.Action = defend and Worker2.Action = attack1 and Worker3.Action = attack1 and castle1HP > 1;
castle1HP = castle1HP−2 if !(Worker1.Action = defend) and Worker2.Action = attack1 and Worker3
.Action = attack1 and castle1HP > 2;
castle2Defeated = true if !(Worker1.Action = attack2) and !(Worker2.Action = defend) and Worker3.Action = attack2 and castle2HP < 2;
castle2Defeated = true if Worker1.Action = attack2 and !(Worker2.Action = defend) and !(Worker3.Action = attack2) and castle2HP < 2;
castle2Defeated = true if Worker3.Action = attack2 and castle2HP < 2;
castle2Defeated = true if Worker2.Action = defend and Worker3.Action = attack2 and castle2HP < 2;
castle2HP = 0 if !(Worker1.Action = attack2) and !((Worker2.Action = defend) and Worker3.Action = attack2 and castle2HP < 2);
castle2HP = 0 if Worker1.Action = attack2 and !(Worker2.Action = defend) and !(Worker3.Action = attack2) and castle2HP < 2;
castle2HP = 0 if Worker1.Action = attack2 and Worker2.Action = defend and castle2HP < 2;
castle2HP = 0 if Worker1.Action = attack2 and !((Worker2.Action = defend) and Worker3.Action = attack2) and castle2HP < 3;
castle2HP = 0 if Worker1.Action = attack2 and Worker2.Action = defend and Worker3.Action = attack2 and castle2HP < 2;
castle2HP = 0 if Worker1.Action = attack2 and Worker2.Action = defend and Worker3.Action = attack2 and castle2HP < 3;
castle2HP = castle2HP−1 if !(Worker1.Action = attack2) and !(Worker2.Action = defend) and Worker3.Action = attack2 and castle2HP > 1;
castle2HP = castle2HP−1 if Worker1.Action = attack2 and !(Worker2.Action = defend) and !((Worker3.Action = attack2) and castle2HP > 1);
castle2HP = castle2HP−1 if Worker1.Action = attack2 and Worker2.Action = defend and !((Worker3.Action = attack2) and castle2HP > 1);
castle2HP = castle2HP−2 if Worker1.Action = attack2 and !(Worker2.Action = defend) and Worker3.Action = attack2 and castle2HP > 2;
castle3Defeated = true if !(Worker1.Action = attack3) and Worker2.Action = attack3 and !(
Worker3.Action = defend) and castle3HP < 2;
castle3Defeated = true if Worker1.Action = attack3
and !(Worker2.Action = attack3) and !(Worker3.
Action = defend) and castle3HP < 2;
castle3Defeated = true if Worker1.Action = attack3
and Worker2.Action = attack3 and !(Worker3.
Action = defend) and castle3HP < 3;
castle3Defeated = true if Worker1.Action = attack3
and Worker2.Action = attack3 and Worker3.Action
= defend and castle3HP < 2;
castle3HP = 0 if !(Worker1.Action = attack3) and
Worker2.Action = attack3 and !(Worker3.Action =
defend) and castle3HP < 2;
castle3HP = 0 if Worker1.Action = attack3 and !(Worker2.
Action = attack3) and !(Worker3.Action =
defend) and castle3HP < 2;
castle3HP = 0 if Worker1.Action = attack3 and
Worker2.Action = attack3 and Worker3.Action =
defend and castle3HP < 2;
castle3HP = castle3HP+1 if !(Worker1.Action =
attack3) and Worker2.Action = attack3 and !(Worker3.
Action = defend) and castle3HP > 1;
castle3HP = castle3HP+1 if Worker1.Action =
attack3 and !(Worker2.Action = attack3) and !(Worker3.
Action = defend) and castle3HP > 1;
castle3HP = castle3HP+1 if Worker1.Action =
attack3 and Worker2.Action = attack3 and Worker3.
.Action = defend and castle3HP > 1;
castle3HP = castle3HP+2 if Worker1.Action =
attack3 and Worker2.Action = attack3 and !(Worker3.
.Action = defend) and castle3HP > 2;
end Evolution
end Agent

Agent Worker1

Vars:
APPENDIX A. CASTLES.ISPL

canDefend : boolean;
end Vars
Actions = {defend, attack2, attack3, doNothing};
Protocol:
Environment.castle1Defeated = true : {doNothing};
Environment.castle1Defeated = false and canDefend = false : {attack2, attack3, doNothing};
Environment.castle1Defeated = false and canDefend = true : {defend, attack2, attack3, doNothing};
end Protocol
Evolution:
canDefend = false if Action = defend;
canDefend = true if canDefend = false;
end Evolution
end Agent

Agent Worker2
Vars:
  canDefend: boolean;
end Vars
Actions = {defend, attack1, attack3, doNothing};
Protocol:
Environment.castle2Defeated = true : {doNothing};
Environment.castle2Defeated = false and canDefend = false : {attack1, attack3, doNothing};
Environment.castle2Defeated = false and canDefend = true : {defend, attack1, attack3, doNothing};
end Protocol
Evolution:
canDefend = false if Action = defend;
canDefend = true if canDefend = false;
end Evolution
end Agent

Agent Worker3
Vars:
  canDefend: boolean;
end Vars
Actions = {defend, attack1, attack2, doNothing};
Protocol:
Environment. castle3Defeated = true : { doNothing };  
Environment. castle3Defeated = false and canDefend = false : { attack1, attack2, doNothing };  
Environment. castle3Defeated = false and canDefend = true : { defend, attack1, attack2, doNothing };  
end Protocol
Evolution:
canDefend = false if Action = defend;  
canDefend = true if canDefend = false;  
end Evolution
end Agent

Evaluation
castle1Wins if Environment. castle1Defeated = false and ( Environment. castle2Defeated = true or Environment. castle3Defeated = true );  
castle1Defeated if Environment. castle1Defeated = true;  
castle2Defeated if Environment. castle2Defeated = true;  
castle3Defeated if Environment. castle3Defeated = true;  
allDefeated if Environment. castle1Defeated = true and Environment. castle2Defeated = true and Environment. castle3Defeated = true;  
castle3Damaged if Environment. castle3HP < 3;  
end Evaluation

InitStates
Environment. castle1HP = 3 and Environment. castle1Defeated = false and Environment. castle2HP = 3 and Environment. castle2Defeated = false and Environment. castle3HP = 3 and Environment. castle3Defeated = false and Worker1. canDefend = true and Worker2. canDefend = true and Worker3. canDefend = true;  
end InitStates
Groups
all = \{ Worker1, Worker2, Worker3, Environment \};
w12 = \{ Worker1, Worker2 \};
c1 = \{ Worker1 \};
c2 = \{ Worker2 \};
c3 = \{ Worker3 \};
c12 = \{ Worker1, Worker2 \};
end Groups

Formulae
<\texttt{c12}>F(\texttt{castle3Defeated});
end Formulae
Appendix B

SMC Output

The following images show the terminal output of the SMC tool when running some cases of the castles game. The strategy found (if it exists) is generated for each player, and is based on the environment variables which each player can observe. As such it is not necessary to print the strategy for every player in each game state, and the output may be presented in this more readable format.

Figure B.1: The output result from the SMC tool for the Castles game with one player/castle.
Figure B.2: The output result from the SMC tool for the Castles game with two players/castle.
Appendix C

ALT* Output

The following images show the terminal output of the ATL* tool when running some cases of the castles game. The strategy (if found) is generated for each game state, and represented as a vector of actions (one per player). As such it is quite difficult to read, and is likely intended for further parsing prior to human inspection. Execution time, however, is more legible.
Figure C.1: The output result from the ATL\textsubscript{IR} tool for the Castles game with one players/castle.
Figure C.2: The output result from the ATL ir tool for the Castles game with two players/castle.