Exploring Monte Carlo Negotiation Search with Nontrivial Agreements

Elijah Malaby, John Licato
Advancing Machine and Human Reasoning (AMHR) Lab
Department of Computer Science and Engineering
University of South Florida

Abstract

The application of automated negotiations to general game playing is a research area with far-reaching implications. Non-zero sum games can be used to model a wide variety of real-world scenarios and automated negotiation provides a framework for more realistically modeling the behavior of agents in these scenarios. A particular recent development in this space is the Monte Carlo Negotiation Search (MCNS) algorithm, which can negotiate to find valuable cooperative strategies for a wide array of games (such as those of the Game Description Language). However, MCNS only proposes agreements corresponding to individual sequences of moves without any higher-level notions of conditional or stateful strategy. Our work attempts to lift this restriction. We present two contributions: extensions to the MCNS algorithm to support more complex agreements and an agreement language for GDL games suitable for use with our algorithm. We also present the results of a preliminary experiment in which we use our algorithm to search for an optimal agreement for the iterated prisoners dilemma. We demonstrate significant improvement of our algorithm over random agreement sampling, although further work is required to more consistently produce optimal agreements.

1 Introduction

Game playing provides a valuable framework for developing and experimenting in artificial intelligence. Games provide controlled environments and clear goals. Furthermore, a broad class of multi-agent systems can be analyzed in terms of games. Turn-based zero-sum games like Chess and Go have already seen extensive research (recently, (Silver et al. 2018)), and more recent efforts have been made to develop AI for more complex environments like the real-time strategy game StarCraft 2 (Churchill et al. 2016). Cooperative games are ones where agents are allowed to collaborate in their strategies (and, broadly speaking, are incentivized to do so). Automated negotiation is a similarly important field in AI, tasked with solving the problem of agents coming to mutually-beneficial agreements in complex environments (Fatima, Kraus, and Wooldridge 2014). At the intersection of automated negotiation and game playing is the potential to build robust AIs that can work cooperatively based only upon knowledge of their shared environment and understanding of each other's goals.

Copyright © 2021by the authors. All rights reserved.

A particularly compelling example of a game where negotiation and careful cooperation are valuable is Diplomacy. Diplomacy is a perfect information game with a large state space, no chance, simultaneous moves, and up to seven concurrent players. Players are encouraged to come to agreements during the game. The large state space makes analyzing Diplomacy games challenging, let alone discovering effective cooperative strategies and negotiating agreements. Recent work has explored the problem of automated negotiation in Diplomacy (Fabregues and Sierra 2011; de Jonge et al. 2018; Theodoridis and Chalkiadakis 2020; Marinheiro and Cardoso 2018).

A recent development in the game-negotiation space is the work on Monte Carlo Negotiation Search (de Jonge and Zhang 2017), an algorithm for negotiating cooperative agreements that do not assume any foreknowledge about the game in question. Instead, it is provided the rules to the game in the form of the Game Description Language (GDL) (Genesereth, Love, and Pell 2005) and has to be prepared to negotiate over any such games. However, MCNS as described works over an agreement space where agreements correspond to sequences of moves. These move sequences are discovered by the random sampling of its internal Monte Carlo Tree Search. For simple games, MCNS can reliably find an optimal strategy but this approach does not scale as effectively to more complex games (the likelihood that a given random game will be optimal decreases rapidly) or to games where a subset of players are party to the agreement (a move sequence may be insufficient to describe the optimal strategy under all opposing responses). Our work explores an approach to decoupling agreement search from the rest of the MCNS algorithm, allowing it to be applied to more expressive agreement spaces capable of describing more complex strategies. We make two main contributions. First, in §3 and §4 we document our proposed extensions to MCTS and MCNS to support these agreements. Second, in section §5 we document our approach to a general-purpose GDL-based agreement language suitable for our algorithm. Finally, §6 documents a preliminary experiment we performed applying our agreement search process to generating the optimal agreement for the iterated prisoners dilemma.

2 Foundation and related work

Games

We focus on simultaneous-move two-player games (games with two players where each player's moves are selected concurrently and applied simultaneously). For the rest of the paper, definitions are based on an abstract game with two players, p_i for $i \in \{0, 1\}$ and a finite set of game states $s \in \mathcal{S}$. There is an initial state $s_0 \in \mathcal{S}$ and a nonempty subset of terminal states $\mathcal{T} \subset \mathcal{S}$. There are two sets of actions $\mathcal{A}_0$ and $\mathcal{A}_1$ corresponding to the full set of actions the respective player could take. There are two legality functions, $\mathcal{L}_0$ and $\mathcal{L}_1$, defined on $(\mathcal{S} \backslash \mathcal{T}) \to 2^{\mathcal{A}_i}$ and identifying the nonempty subset of actions legal for a player to take in a given nonterminal state. For each nonterminal state $s \in \mathcal{S} \backslash \mathcal{T}$ there is a game transition function δg_s defined on $\mathcal{L}_0(s) \times \mathcal{L}_1(s) \to \mathcal{S}$ giving the transitions out of that state. Finally, there are two utility functions $\mathcal{U}_0$ and $\mathcal{U}_1$ defined on $\mathcal{T} \to \mathbb{R}^+$ giving the utility values of each terminal state to each player.

Monte Carlo Tree Search

Monte Carlo Tree Search is a family of algorithms for estimating properties of games (in particular, Nash Equilibria). A more detailed discussion of MCTS can be found in (Kocsis and Szepesvári 2006) and particular policies for simultaneous move games are discussed in (Lisy et al. 2013). MCTS has recently used for automated negotiations in the work of (Buron, Guessoum, and Ductor 2019), in which they directly apply MCTS to a negotiation game formalism of their design.

MCTS algorithms work by iteratively exploring the move tree of the game, with each step incrementally improving the overall estimates. Each node in the game's move tree corresponds directly to a game state. The children of a state correspond to valid state transitions (moves). Each iteration of MCTS proceeds in four steps:

1. Selection: Select a promising unvisited node
2. Rollout: Simulate a random game starting at this node
3. Update: Use the reward from the simulation to update metrics about this node and its parents
4. Expansion: Add the children of this node as unvisited nodes in the tree

Selection proceeds recursively from the root by selecting visited nodes according to a policy. A simple greedy policy selects the node with the highest average result so far, but this policy is highly chaotic in its results. Many heuristics have been studied, but a commonly used one is UCT (Kocsis and Szepesvári 2006). In UCT, the average utility score of each state (node) is combined with an "exploration" term encouraging visits to new subtrees. Selection stops when recursion encounters a node with unvisited children, one of which is selected arbitrarily.

The Rollout step then performs a simulation of the game starting at the selected node. For our purposes, actions are randomly selected in this simulation step. Each player's actions are selected with uniform probability from their legal actions in each state. Finally, rollout records the utility values of the terminal state for each player.

The Update step walks back up the tree to update each node based on the rollout result. The update is dependent on what information is being recorded and what policy is being used to select nodes. UCT, for example, records the cumulative record and number of times each node is updated.

The Expansion step performs maintenance needed for selection to pick children from the explored node. It allows the tree to be generated lazily in memory, which is important given even finite games have trees with explosive size.

General game playing and GDL

General Game Playing explores the problem of agents playing games without any foreknowledge of the game rules. The Game Description Language (GDL) was popularized by the AAAI General Game Playing competition (Genesereth, Love, and Pell 2005) (which provides a complete reference for it) as the format all submitted agents must consume to know what game they are playing. Since then, GDL has been used to develop game agents based upon minimax, alpha-beta pruning, and varieties of MCTS. de Jonge and Zhang (2016) explore the general premise of applying GDL as a foundation for developing general negotiating agents.

GDL is a language based on Datalog (Ceri, Gottlob, and Tanca 1989), a logic programming language. In particular, GDL describes a game in terms of a set of logical rules which must hold in every state of the game. Each of the rules defines logical inferences to be made about the game, framed as implications. A special predicate $true$ is used to refer to facts about particular game states, so a rule like $true(x) \to legal(p, y)$ says "in states where x holds, it is legal for player p to take action y. Like Datalog (and Prolog before it), the rules are written with the consequent first. So in practice that rule would be written `legal(p, y) :- true(x)` or in the more common s-expression notation `(<= (legal p y) (true x))`. The design of GDL allows agents to follow these logical inferences and derive all of the relevant information about any given game state, including how the actions performed in a given state influence the set of $true$ facts in the next state.

Automated Negotiation

Automated negotiation is a broad subfield of AI, with many recent works centering on the Automated Negotiating Agents Competition (Fujita et al. 2016; Aydoğan et al. 2020). However, much of the work in this space either makes too many simplifying assumptions about the negotiation space to be applied directly to game negotiations or is highly domain specific (like the Diplomacy agents referenced in the introduction). One of the more general purpose works recently has been that of Marinheiro and Cardoso (2018), but this only presents a framework which must be tailored for particular games.

Monte Carlo Negotiation Search

Monte Carlo Negotiation Search (de Jonge and Zhang 2017; 2020) is a recent algorithm which builds upon MCTS to produce a generic algorithm for negotiating cooperative agreements between agents in non-zero-sum games. MCNS accomplishes this by using a modified MCTS to compute the

Negotiation Value and *Reservation Value* of states. The reservation value refers to the utility each agent expects to get despite failing to negotiate, while the negotiation value refers to the utility each agent can expect to receive from successful negotiations. Under MCNS agents get an opportunity to negotiate a new agreement at the start of each state (regardless of if an agreement had already been established). So, the reservation value is taken as the negotiation value of the expected next state if negotiation fails (or as the average utility of the state itself, as a lower bound). The negotiation value is itself also computed based on the reservation value of the same state (although in a less direct fashion).

The mutually-referential nature of the negotiation and reservation values makes them difficult to compute directly. To compute these values, MCNS replaces the normal loop of MCTS with a nested loop. The outer loop of MCNS performs a modified MCTS but replaces the rollout step with a nested instance of MCTS. In the outer loop, the negotiation values are used for selection instead of the average rollout scores (to select a state which is promising under negotiation). Then MCTS is used to compute the average rollout score of the selected node (the expected utility despite failing to negotiate). This nested MCTS loop is also augmented to record each of its rollout simulations, the best of which are used as agreements. Using the average rollout score as a lower bound for the reservation value, the update step can estimate the negotiation value of that state. This new negotiation value can be used to estimate the reservation value of the parent state and update its negotiation value. This process continues back up the tree, updating the negotiation values of each parent. Depending on the exact implementation, the expansion step of the outer loop can be skipped as the nested MCTS will have had to explore that subtree already.

The full MCNS algorithm implements a negotiation strategy based upon these negotiation and reservation values. However, given the only agreements it generates are move sequences found by guided MCTS, the algorithm has challenges scaling optimally to games with larger move spaces. As mentioned above, the focus of this work is expanding MCNS to work with more complex agreements.

3 Monte Carlo Agreement Tree Search

Our first contribution is the development of Monte Carlo Agreement Tree Search (MCATS) and Monte Carlo Agreement Forest Search (MCAFS), described in this and the following sections respectively. Monte Carlo Agreement Tree Search builds on MCTS by augmenting the underlying game state and transition system with agreements. MCATS is instantiated with an agreement space on the game in question, based on a set of agreements $\alpha \in Ag$. In addition, an agreement space has three functions. The first two are the *execution functions* ex_i defined on $\mathcal{S} \times Ag \to 2^{\mathcal{A}_i}$, giving the actions legal for each player in the given state according to the agreement. The second is the *agreement transition function* δa defined on $\mathcal{S} \times \mathcal{A} \to \mathcal{A}$ which defines how the agreement itself changes as the game progresses. The agreement transition function allows us to capture stateful agreements without needing to introduce an explicit notion of state, which would otherwise complicate our model, by forcing any state to be encoded into the agreement itself.

Based on our agreement model, we define an agreement-restricted game. Intuitively, an agreement-restricted game is a game whose legal transitions are restricted by an initial agreement between the players. The states of this game are $\mathcal{S} \times \mathcal{A}$ state-agreement pairs, and the terminal states are similarly $\mathcal{T} \times \mathcal{A}$ terminal state-agreement pairs. The initial state is the pair (s_0, α). The full sets of actions are the same. The legality functions are defined as $\mathcal{L}'_i(s) := \mathcal{L}_i(s) \cap ex_i(s, \alpha)$ if said intersection is nonempty, otherwise just $\mathcal{L}_i(s)$. Without handling this edge case it would be possible for an agreement to prevent a player from taking any move. The transition functions are defined as $\delta_{(s,\alpha)}(a_0, a_1) := (\delta g_s(a_0, a_1), \delta a_s(\alpha))$.

Finally, Monte Carlo Agreement Tree Search is MCTS over an agreement-restricted game. MCATS allows us to approximate the Nash Equilibrium and expected utility of the game under the assumption that the given agreement is binding, which allows us to compare agreements by direct simulation.

4 Monte Carlo Agreement Forest Search

Monte Carlo Agreement Forest Search applies MCNS to the game as a whole and MCATS to a pool of agreements. To remain entirely agnostic to the agreement space, MCAFS takes an evolution function that operates on the agreement pool and each agreement's MCATS tree. The purpose of the evolution function is to use the MCATS data to improve the pool of agreements as the corresponding trees are explored: removing agreements that do not appear promising and replacing them with new ones. The idea of MCAFS is to use MCNS to estimate the negotiation and reservation values of the current state but not to generate the agreements themselves. We need the negotiation values from MCNS to calculate the reservation value of the current state, with which we can calculate the negotiation value of the current state based on our pool of agreements. Then alternation between exploring the MCATS trees and calling the evolution function to try new agreements is used to search the agreement space.

5 An Agreement Language

Our second contribution in this work is the design of an agreement language for simultaneous move games (in particular, GDL games) compatible with MCAFS. While existing agreement languages exist (in, for example, the blockchain smart contract space), and constructing one for a specific game is straightforward, to our knowledge a suitable language for concisely expressing agreements over arbitrary GDL games does not exist. This section gives a high-level description of our agreement language. The language is tree-structured (using s-expressions) and has five main constructs broken into three categories: the temporal operators, the conditional operator, and the requirement operators. All of these form clauses and an agreement is simply a set of such clauses. We chose these operators to strike a balance between keeping the overall language simple and capturing

$Rcl(state, \texttt{(next clauses...)}) ::= \{\}$

$Rcl(state, \texttt{(until pred clauses...)}) ::=$

$\texttt{if } pred(state) \texttt{ then } \{\} \texttt{ else } \bigcup_{c \in clauses} Rcl(state, c)$

$Rcl(state, \texttt{(when pred clauses...)}) ::=$

$\texttt{if } pred(state) \texttt{ then } \bigcup_{c \in clauses} Rcl(c, state) \texttt{ else } \{\}$

$Rcl(state, \texttt{(force agent moves...)}) ::=$
$\{\texttt{(force agent moves...)}\}$

$Rcl(state, \texttt{(block agent moves...)}) ::=$
$\{\texttt{(block agent moves...)}\}$

$Ragent(\texttt{(force agent moves...)}) ::= agent$

$Ragent(\texttt{(block agent moves...)}) ::= agent$

$Rmoves(state, \texttt{(force agent moves...)}) ::=$
$\{moves...\}$

$Rmoves(state, \texttt{(block agent moves...)}) ::=$
$\mathcal{L}_{agent}(state) \backslash \{moves...\}$

$R_{p_i}(state, agreement) ::=$
$\{r | c \in agreement \wedge r \in Rcl(c) \wedge Ragent(r) = p_i\}$

$ex_{p_i}(state, agreement) ::=$

$$\bigcap_{s \in R_{p_i}(state, agreement)} Rmoves(s)$$

$\delta cl(state, \texttt{(next clauses...)}) ::=$
$\{clauses...\}$

$\delta cl(state, \texttt{(until condition clauses...)}) ::=$
$\texttt{if } condition(state)$
$\texttt{ then } \{\}$
$\texttt{ else } \{\texttt{(until condition clauses...)}\} \cup$
$\bigcup_{c \in clauses} \delta cl(state, c)$

$\delta cl(state, \texttt{(when condition clauses...)}) ::=$
$\texttt{if } condition(state)$
$\texttt{ then } \bigcup_{c \in clauses} \delta cl(state, c)$
$\texttt{ else } \{\}$

$\delta cl(state, \texttt{(force agent moves...)}) ::= \{\}$

$\delta cl(state, \texttt{(block agent moves...)}) ::= \{\}$

$\delta a(state, agreement) ::= \bigcup_{c \in agreement} \delta cl(state, c)$

Figure 1: Execution and Transition functions of our agreement language. For the execution functions: Rcl collects the requirements of a given clause, $Ragent$ extracts the agent of a requirement, $Rmoves$ gives the set of legal moves under a requirement, R_{p_i} is the set of requirements for a given agent in a state. For the transition function: δcl is the transition function for a given clause

$clause ::= \texttt{(next } S+ \texttt{)}$
$\quad | \texttt{(until } condition\ agreement \texttt{)}$
$\quad | \texttt{(when } condition\ agreement \texttt{)}$
$\quad | \texttt{(force } agent\ \mathcal{A}+ \texttt{)}$
$\quad | \texttt{(block } agent\ \mathcal{A}+ \texttt{)}$
$agreement ::= clause | clause\ agreement$

Figure 2: ENBF of our agreement language

a large range of possible agreements. Figure 1 gives the semantics of these operators in terms of the execution and transition functions, while Figure 2 gives the syntax of the language in EBNF.

We incorporate GDL terms in three ways: predicates, moves, and identifying agents. By predicates, we mean GDL queries: logical statements whose truth value can be determined by exhaustively searching the game's GDL inference rules concerning the current game state. Incorporating GDL queries allows our agreements to capture any properties of the game state or rules used to define the game itself. Using GDL terms for moves reflects the GDL-centric nature of the language. We use the GDL agent identities for the same reason, but we have to keep in mind that GDL does not assign agents to integers (or even limit the number of agents to two). As such, to use MCATS as defined above, a mapping must be defined to the two agents of our formal game definition in section §2.

The two temporal operators are `next` and `until`. These operators define how the contract can evolve as the game changes. The `next` operator takes an agreement and defers its enforcement (execution/transition) to the next state. The `until` operator takes a GDL predicate along with an agreement. If the predicate holds it enforces the nested agreement and is included in the next agreement (the output of the transition function). The conditional operator `when` also takes a GDL predicate alongside an agreement and enforces the nested agreement if the predicate holds. Finally, the requirement operators are `force` and `block`, both of which take the name of an agent and a set of moves for that agent. The requirement operators are the leaves of any agreement, the building blocks which the agreement uses to enforce higher-level goals. The `force` operator requires that the given agent's move come from among its set, while the `block` operator prevents the agent from taking any of the given moves. These operators are complementary. However, for games with large move spaces, it would be impractical to simulate the effect of one with using the other: enumerating all moves less one in a `force` to simulate a `block` does not scale well.

Example agreements

```
(force white lay_claim)
(next
  (force black end_game))
```

The above agreement is based on the GDL rules for the Dollar Auction (DA) game and represents the optimal (cooperative) solution to it. In DA, players take turns laying claims to some prize. Each claim costs 5 points and being the last player to lay a claim wins you back 25 points. The first player to pass ends the game, winning it for the other player. Let us say this agreement goes into force on the first turn and white goes first. The agreement forces `white` to lay a claim and on the next turn forces `black` to pass. Thus, both players have minimized their losses. For this agreement, just having `(next (force black end_game))` would be equivalent in effect: safe in the knowledge that `black` will skip next turn, white is already incentivized to lay a claim.

```
(until false
  (when (and (true (cell 1 1 x))
             (true (cell 3 1 x)))
    (block x (mark 2 1))))
```

This agreement is based on a tic-tac-toe game, where `x` and `o` are the agents and the `cell` predicate is used to identify what markings are in what (x,y) coordinates of the grid. Informally, the contract blocks the `x` player from ever completing a column by marking the (2,1) location. `until false` is used to enforce the clauses of the contract on the state it goes into force and all subsequent states. `false` here is a stand-in for any GDL query which would never succeed: GDL makes no mention of a guaranteed false statement, but any condition requiring mutually exclusive facts about the game board would suffice. The `when` condition checks if `x` has already marked cells (1,1) and (3,1), and finally the `block` requirement prevents marking (2,1). Importantly, the contract as stated only prevents marking (2,1) *last*: if the column were filled in from the top to bottom (for example) the contract would never come into play.

6 Experiment

We performed a preliminary experiment to test the effectiveness of MCAFS-guided agreement evolution versus a random sampling of our agreement space. Our evolution mechanism was randomly mutating (re-generating) subtrees of randomly generated agreements. The agents and respective valid moves needed to generate agreements were queried from the GDL rules and sampled randomly. To generate the queries we queried for all of the possible valid state propositions and randomly generated conjunctions of these as needed. We focused our experiment on the GDL definition of the iterated prisoners dilemma (IPD). We tested if preferentially mutating agreements that scored well under MCATS would more consistently generate an optimal IPD agreement than random mutations would.

Although our algorithm is game agnostic, we chose to focus on IPD for a couple of reasons. Firstly, it is not only non-zero-sum but the optimal cooperative strategy is significantly better than the nash equilibrium. Equally importantly howver, despite being a fairly simple game, IPD already shows the limitation of MCNS (de Jonge and Zhang 2017). Although MCNS performed well against IPD, with the full 20 rounds it did not perform optimally. This sub-optimal play was attributed to the large state space and MCNS only sampling move sequences for its agreements. These properties of IPD make it an important benchmark for our work, where an optimal agreement as described by our agreement language should be significantly easier to find.

Such an optimal agreement is given by `white` and `black` both being forced to cooperate every round, as in:

```
(until false
  (force white cooperate)
  (force black cooperate))
```

We performed 40 executions of MCAFS, 20 where the evolution step accounted for the MCATS results and the other 20 where random agreements were selected. In both cases, we maintained a pool of 8 agreements. MCAFS was configured to run 100 samples for each agreement in the pool on each iteration. On the evolution step we selected two agreements and replaced four others with mutations of those two. In the MCATS-aware pool, agreements were sorted primarily by the sum of their rewards to all players and secondarily by their reward for one of the players (white, in our tests, to simulate the preference a particular player would give to agreements that reward them). In the end, the MCATS-aware runs generated an optimal agreement 10 times out of 20 while random generation only found it once. Despite the simplicity of our evolution step, using MCAFS to guide the search through the agreement space shows a clear advantage over random sampling.

7 Conclusion and Future Work

Autonomous systems of the future will increasingly find themselves in environments where they are bound by a hierarchy of rules, which may include laws, ethical guidelines, norms of proper behavior, and temporary agreements. Negotiating and reasoning about possible agreements still is out of reach of even the most advanced AI systems currently available, especially when those agreements are themselves subject to constraints set by additional factors such as an agent's goals, preferences of other parties involved, and so on. The work we describe in this paper represents small, but important, steps towards negotiation-capable AI.

How an environment's rules should be represented by a negotiation-capable AI is a question which leads to many unanswered questions, which we hope to address with future work. For example: How can such an AI reason over agreements containing open-textured terms? For example: consider the rule "Compensation shall be granted as soon as is reasonably possible." How should the open-textured term "reasonably possible" be defined? Though it is somewhat ambiguous and can lead to differing interpretations, it is difficult to avoid the use of such terms because it is implausible (and ill-advised) to explicitly state every possible scenario which does and does not count as "reasonably possible." Such open-textured terms are a common, but necessary, part of both legal and ethical systems (Hart 1961; Franklin 2012), and improving automated reasoning over them should be considered an imperative for AI research (Licato and Marji 2018; Licato, Marji, and Abraham 2019; Quandt and Licato 2019).

The clear next step we see is to develop our implementation of MCNS and MCAFS to the point where we can directly test negotiating agents against the benchmark of (de Jonge and Zhang 2017). We also see improvements to be made in our agreement evolution: for this experiment, we did not take advantage of any of the MCTS-tree data to guide mutating agreements (beyond just comparing the agreements by their average utility). A more detailed analysis of the game trees could be used to prefer certain mutations over others. Our agreement generation algorithm is also only able to generate ground GDL queries— ones that do not involve logic variables. Allowing for logic variables in these conditions introduces complicated questions of scope and state, but it may prove vital for capturing agreements in spaces as complex as (for example) Diplomacy.

In conclusion, our work presents an incremental but important step forward in automated game negotiations. We have described an extension to MCNS which allows it to operate in more complex agreement spaces as well as an agreement language to test our extensions upon. We are hopeful that our work will serve as a foundation for further exploration into the game negotiation space.

8 Acknowledgement

This material is based upon work supported by the Air Force Office of Scientific Research under award numbers FA9550-17-1-0191 and FA9550-18-1-0052. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Air Force.

References

Aydoğan, R.; Baarslag, T.; Fujita, K.; Mell, J.; Gratch, J.; de Jonge, D.; Mohammad, Y.; Nakadai, S.; Morinaga, S.; Osawa, H.; Aranha, C.; and Jonker, C. M. 2020. Challenges and main results of the automated negotiating agents competition (anac) 2019. 366–381.

Buron, C. L. R.; Guessoum, Z.; and Ductor, S. 2019. Mcts-based automated negotiation agent. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, volume 11873, 1850–1852.

Ceri, S.; Gottlob, G.; and Tanca, L. 1989. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering* 1(1):146–166.

Churchill, D.; Preuss, M.; Richoux, F.; Synnaeve, G.; Uriarte, A.; Ontañnón, S.; and Čertický, M. 2016. Starcraft bots and competitions. *Encyclopedia of Computer Graphics and Games* 1–18.

de Jonge, D., and Zhang, D. 2016. Using gdl to represent domain knowledge for automated negotiations. In *International Conference on Autonomous Agents and Multiagent Systems*, 134–153.

de Jonge, D., and Zhang, D. 2017. Automated negotiations for general game playing. In *AAMAS '17 Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, 371–379.

de Jonge, D., and Zhang, D. 2020. Strategic negotiations for extensive-form games. *Autonomous Agents and Multi-Agent Systems* 34(1):1–41.

de Jonge, D.; Baarslag, T.; Aydogan, R.; Jonker, C. M.; Fujita, K.; and Ito, T. 2018. The challenge of negotiation in the game of diplomacy. In *Agreement Technologies 2018, Revised Selected Papers*, 100–114.

Fabregues, A., and Sierra, C. 2011. Dipgame: A challenging negotiation testbed. *Engineering Applications of Artificial Intelligence* 24(7):1137–1146.

Fatima, S.; Kraus, S.; and Wooldridge, M. 2014. *Principles of Automated Negotiation*.

Franklin, J. 2012. Discussion paper: How Much of Commonsense and Legal Reasoning is Formalizable? A Review of Conceptual Obstacles. *Law, Probability and Risk* 11(2-3):225–245.

Fujita, K.; Aydoğan, R.; Baarslag, T.; Hindriks, K.; Ito, T.; and Jonker, C. 2016. The first automated negotiating agents competition (anac 2010). volume 674. 139–151.

Genesereth, M. R.; Love, N.; and Pell, B. 2005. General game playing: Overview of the aaai competition. *Ai Magazine* 26(2):62–72.

Hart, H. 1961. *The Concept of Law*. Clarendon Press.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *ECML'06 Proceedings of the 17th European conference on Machine Learning*, 282–293.

Licato, J., and Marji, Z. 2018. Probing formal/informal misalignment with the loophole task. In *Proceedings of the 2018 International Conference on Robot Ethics and Standards (ICRES 2018)*.

Licato, J.; Marji, Z.; and Abraham, S. 2019. Scenarios and recommendations for ethical interpretive ai. In *Proceedings of the AAAI 2019 Fall Symposium on Human-Centered AI*.

Lisy, V.; Kovarik, V.; Lanctot, M.; and Bosansky, B. 2013. Convergence of monte carlo tree search in simultaneous move games. In *Advances in Neural Information Processing Systems 26*, volume 26, 2112–2120.

Marinheiro, J., and Cardoso, H. L. 2018. Towards general cooperative game playing. *Trans. Comput. Collect. Intell.* 28:164–192.

Quandt, R., and Licato, J. 2019. Problems of autonomous agents following informal, open-textured rules. In *Proceedings of the AAAI 2019 Spring Symposium on Shared Context*.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science* 362(6419):1140–1144.

Theodoridis, A., and Chalkiadakis, G. 2020. Monte carlo tree search for the game of diplomacy. In *11th Hellenic Conference on Artificial Intelligence*, 16–25.