Synthesizing human-friendly optimal strategies in board games

Thomas Bolander
DTU Compute
Technical University of Denmark
tobo@dtu.dk

Jacob Pjetursson
DTU Compute
Technical University of Denmark
jacob@pjetursson.com

***Abstract*—We present a tool for synthesizing and verifying optimal game playing strategies represented by compact fast-and-frugal trees, i.e., prioritized lists of strategic rules. The purpose of the tool is to create human-friendly optimal strategies for simple board games, e.g. for teaching a human player to play optimally, or to assess the difficulty of a given board game in terms of the length of the generated strategy. The tool supports arbitrary one- or two-player zero-sum games with perfect information specified through the game description language GDL within general game playing. When synthesizing a strategy, the game is initially solved, the solution is turned into a fast-and-frugal tree, and the tree is then minimized. We illustrate the use of the tool to synthesize compact optimal strategies for Tic-tac-toe, Nim, and Sim, which leads us to provide an even shorter optimal strategy for Tic-tac-toe than the well-known Simon & Newell strategy. Additionally, we have developed a visual tool enabling users to build and verify manually crafted fast-and-frugal strategies.**

I. Introduction

Significant amounts of artificial intelligence research is focused on solving games [1]–[4]. However, the game solution itself offers limited value. It may reveal which of the players can force a win from the initial state, or provide a perfectly playing game agent. But it helps little in making humans understand *why* it is a solution, and what it takes to play optimally. In the computer, the perfect strategy is often simply represented as a vast collection of unique state-move pairs. For non-trivial games, human memories are simply inadequate to reliably contain all this information. Instead, humans rely on general strategic rules to understand and memorize strategies. In this paper, we attempt to automatically generate such strategic rules, one of the aims being to teach human players how to play (near-)optimally. We have created a tool that can synthesize optimal game playing strategies represented by *fast-and-frugal trees* containing simple strategic rules. Our tool also supports verifying whether manually crafted strategies are optimal, and minimizing them if they are (by removing or simplifying rules). Fast-and-frugal trees are acknowledged as a decision making tool that is easy to comprehend and that in many scenarios resembles human decision making [5].

We offer two versions of our tool. The first version is a plug-and-play solution built around the *general game playing* (GGP) open source base package.[1] It simply takes a *game description language* (GDL) file as input [6], solves the game, i.e., computes an optimal strategy for the game, and then outputs the strategy as a fast-and-frugal tree. The second version of the tool requires the user to implement various game-specific functions and classes within the code. The benefit of this approach is that the runtime is significantly faster, but typically requires more work to set up for new games. The major issue with the GGP version of our tool is that the prover-based engine used to process and evaluate states is quite slow. Although speed-up techniques exist [7], [8], we won't be considering those in this paper.

The strategic rules we generate are of the form $\phi \Rightarrow a$, where ϕ is a *precondition* (a logical formula) and a is an *action* (a *move*). The rule simply says that if ϕ is true in the current game state, the move a should be made. In Tic-tac-toe, a rule for getting 3-in-a-row by marking the right corner with a cross may look as follows, using simplified GDL syntax (where `(cell` i j k`)` means that cell (i, j) contains k, and k can be either `b` (blank), `x` (cross) or `o` (nought)): `(cell 1 3 x)` $\wedge$ `(cell 2 3 x)` $\wedge$ `(cell 3 3 b)` $\Rightarrow$ `(mark 3 3 x)`.

We use symmetry detection in games to reduce the number of strategic rules required. Whenever checking if a rule applies to a state, we also check if it applies to any symmetric states. We have integrated automatic symmetry detection in our general game playing solution using the graph automorphism detection program made by Schiffel et al. [9].

In addition to symmetry detection, we apply various simplifications to our generated strategies in order to minimize them (make them as simple and short as possible). This includes simplifying the preconditions of strategic rules and removing rules that become obsolete. Our tool handles all games that are finite, deterministic, sequential, 1- or 2-player, zero-sum and has perfect information. Since we generate optimal strategies using Minimax, we can only handle games of rather limited size.

We have tested our tool on the games Tic-tac-toe, Nim and Sim. Although these games are trivially solvable with very small state space sizes, we still expect the tool to be applicable to most strongly solved games like Kalah and Awari [1], [10] when given sufficient computational resources. Apart from storing the solution to the game, using the tool has no significant additional memory overhead.

With our tool, we managed to compress and simplify the

[1] https://github.com/ggp-org/ggp-base

978-1-7281-4533-4/20/$31.00 ©2020 IEEE

famous optimal strategy used in Newell and Simon's 1972 Tic-tac-toe program [11]. Their strategy features a list of 8 prioritized rules expressed in natural language. We formalised these rules in our logical framework and showed that it may be compressed to 6 rules, while still retaining the optimality and simplicity of the strategy. Additionally, our tool independently synthesized an optimal strategy with only 5 strategic rules. We know of no previous succesful attempts to simplify the Newell and Simon strategy.

II. GDL AND FAST-AND-FRUGAL TREES

A *general game playing* (*GGP*) system is one that can understand the rules of arbitrary new games and play them effectively without human intervention [6]. The *game description language* (*GDL*) is the official language for general game playing. GDL is a variant of Datalog and uses prefix notation, along with 9 reserved keywords. GDL requires games to be deterministic with perfect information.

A *fast-and-frugal tree* (*FFT*) is a binary classification tree that can be used as a decision-making tool [12]. The benefits of FFTs is that they are normally very simple and fast to query. FFTs can also be represented in a linear format as a (prioritized) sequence of strategic rules $\phi_1 \Rightarrow a_1 \mid \cdots \mid \phi_n \Rightarrow a_n$. To use such an FFT for decision-making, one first evaluates ϕ_1. If ϕ_1 is true, action a_1 is chosen for execution. If not, one evaluates ϕ_2. If ϕ_2 is true, action a_2 is executed, etc. Gigerenzer [13] presents an example of an FFT to be used at a hospital for deciding whether patients need coronary care. In our linear format it can be written as: *(ST segment changes) ⇒ (to Coronary Care Unit) | (chief complaint of chest pain) ⇒ (to regular nursing bed) | (any other factor (NTG, MI, T, ...)) ⇒ (to Coronary Care Unit) | ⊤ ⇒ to regular nursing bed*.

In this paper, we use the structure of FFTs to represent strategies for board games. In order to be able to automatically generate and simplify FFTs, we represent them in a simple logical language (section IV). Below, we first define games and (optimal) game strategies. To be able to iteratively build up strategies, we introduce two novel notions of partial, optimal strategies.

III. STRATEGIES AND OPTIMALITY

We use state transition systems to describe games. State transition systems describe the underlying state space of the game as well as all possible moves (actions) and their effects. In general, our states are going to be described as sets of atomic formulas, that is, as subsets of a (finite) set of atoms, *Atm*. These atoms can be atomic propositions of propositional logic or ground atoms of a first-order language. The state transition systems induced by GDL descriptions will always have atoms of the second kind.

Definition 1. A *(finite, deterministic) state transition system* over a finite set of atoms *Atm* is $\Sigma = (S, A, \gamma)$, where $S \subseteq 2^{Atm}$ is a finite set of *states*, A is a finite set of *actions* (also called *moves*) and $\gamma : S \times A \to 2^S$ is a *state transition function* satisfying $|\gamma(s,a)| \leq 1$ for all s, a. When $\gamma(s,a) \neq \emptyset$, we say that action a is *applicable* in state s. For $A' \subseteq A$, we define $\gamma(s, A') = \bigcup_{a \in A'} \gamma(s,a)$.

Definition 2. A *(finite, deterministic, perfect information) game* is $G = (\Sigma, s_0, T, P, \rho, u)$ where $\Sigma = (S, A, \gamma)$ is a state transition system, $s_0 \in S$ is an *initial state*, $T \subseteq S$ is a set of *terminal states*, P is a set of *players*, $\rho : S \to P$ is a *player function* determining who has the move, and $u : P \times T \to \mathbb{R}$ is a real-valued *utility function*. A game with $|P| = n$ is called an *n-player game*. A game with $\sum_{p \in P} u(p,t) = 0$ for all $t \in T$ is called *zero-sum*.

This definition of a game is equivalent to the definition of a *multiagent environment* by Schiffel [9], except our definition only allows sequential games (in each state, only one player can make a move). Schiffel uses multiagent environments as a semantics for GDL game descriptions, hence any game described in GDL can be equivalently represented as a multiagent environment, and hence also as a game according to the above definition, as long as it is sequential. Our tool accepts games described in GDL format, but when reasoning about games and game properties, we will most of the time describe these in terms of the induced games according to the definition above.

Definition 3. A *strategy* for player $p \in P$ in game $G = (\Sigma, s_0, T, P, \rho, u)$ with $\Sigma = (S, A, \gamma)$ is a mapping $\sigma : S \to 2^A$ satisfying $\sigma(s) = \emptyset$ for all $s \in S$ with $\rho(s) \neq p$ and satisfying that $a \in \sigma(s)$ implies a is applicable in s. A strategy for p is called *total* if $\sigma(s) \neq \emptyset$ for all s with $\rho(s) = p$, otherwise *partial*. It is *deterministic* if $|\sigma(s)| \leq 1$ for all s. We say that σ is *defined* on a state s if $\sigma(s) \neq \emptyset$, and define the *domain* of σ as $\mathrm{dom}(\sigma) = \{s \in S \mid \sigma(s) \neq \emptyset\}$.

As usual, we can identify mappings $\sigma : S \to 2^A$ with their corresponding relations $\{(s,a) \in S \times A \mid a \in \sigma(s)\}$. In this paper, we will restrict attention to 1- and 2-player zero-sum games. In the following, to keep the exposition simple, we will furthermore restrict our definitions to the two-player case with players $P = \{1, 2\}$. In such games, we can apply the MINIMAX algorithm [14], [15] to assign a *(minimax) value* (game-theoretic value) $v(s)$ to each state s of the game. The value $v(s)$ is the utility achieved by player 1 (the MAX player) when starting the game in s and assuming perfect play by both players.[2] In the following, we will for simplicity only consider strategies for player 1. To consider strategies for player 2, one can always replace any game by a game in which the roles of the two players are swapped.

A total strategy σ for player 1 is *optimal* if for all s with $\rho(s) = 1$ and all $s' \in \gamma(s, \sigma(s))$ we have $v(s') = v(s)$, i.e., if following the strategy always preserves the minimax value of the states to which it is applied. We can define a unique *maximal, optimal strategy opt* (for player 1) by, for all $s \in S$ with $\rho(s) = 1$,

$$opt(s) = \{a \in A \mid \text{for all } s' \in \gamma(s,a) : v(s') = v(s)\}.$$

[2]Note that we have not restricted our state transition systems to be acyclic, so games can potentially loop. We give loop states a minimax value of 0 unless one of the players can enforce a better outcome by breaking the loop. We assume the reader to be familiar with MINIMAX [14], [15].

Any optimal strategy will then be a substrategy of *opt* (where we define σ_1 to be a *substrategy* of σ_2, written $\sigma_1 \subseteq \sigma_2$, if $\sigma_1(s) \subseteq \sigma_2(s)$ for all s). Similarly, we can define the *maximal strategy* (for player 1) by, for all $s \in S$ with $\rho(s) = 1$,

$$max(s) = \{a \in A \mid \gamma(s,a) \neq \emptyset\}.$$

Optimal strategies are *strong* solutions to games in the sense of guaranteeing the game-theoretic (minimax) value for *any* legal state of the game, no matter whether that state is reachable by following the optimal strategy. If a game is always played from the initial state s_0, and player 1 always follows strategy σ, it is of course irrelevant which actions σ specify for states s that are not reachable from s_0 when player 1 plays by σ. Hence we can define a weaker notion of optimal strategy that still guarantees the best possible outcome when following a strategy σ. To define this notion, we first need to define the reachable states when playing according to σ.

Given a state s and total strategy σ for player 1, the set of *successors* of s w.r.t. σ is

$$\Gamma(s,\sigma) = \begin{cases} \gamma(s,\sigma(s)) & \text{if } \rho(s) = 1 \\ \gamma(s,\{a \in A \mid \gamma(s,a) \neq \emptyset\}) & \text{if } \rho(s) = 2 \end{cases}$$

Note that we consider all applicable actions in the moves of player 2, as we are defining all possible successors when player 1 plays by σ and player 2 plays by any strategy. For $S' \subseteq S$, we let $\Gamma(S',\sigma) = \bigcup_{s \in S'} \Gamma(s,\sigma)$. We now let $\Gamma^0(s,\sigma) = \{s\}$, and for all $n \in \mathbb{N}$, we recursively define $\Gamma^{n+1}(s,\sigma) = \Gamma(\Gamma^n(s,\sigma),\sigma)$. Finally, let $\Gamma^*(s,\sigma) = \bigcup_{n \in \mathbb{N}} \Gamma^n(s,\sigma)$: this is the set of *reachable states* from s when player 1 follows strategy σ and player 2 follows any strategy. We define $\Gamma^*(\sigma)$ as an abbreviation of $\Gamma^*(s_0,\sigma)$: the set of reachable states from the initial state when player 1 plays σ.

We can now define a total strategy σ for player 1 to be *weakly optimal* if for all $s \in \Gamma^*(\sigma)$ we have $\sigma(s) \subseteq opt(s)$ (note that when $\rho(s) = 2$, $\sigma(s) = \emptyset$ and hence we automatically have $\sigma(s) \subseteq opt(s)$). This definition is simply the restriction of our earlier definition of optimal strategy to the relevant reachable states. Clearly, if player 1 follows a weakly optimal strategy, player 1 is still guaranteed to receive the game-theoretic value for any play of the game, and against any opponent.

In this paper, strategies are going to be built iteratively, meaning that we start with the empty strategy that is then iteratively extended. Hence, we are mainly going to work with partial strategies. In order to ensure that our partial strategies can be extended to optimal, total strategies, we need a notion of optimality for partial strategies. We are going to define two such notions in the following. First we define the following closure operations on any partial strategy σ (for player 1):

- The *optimal closure* of σ is the strategy σ^{opt} given by, for all $s \in S$ with $\rho(s) = 1$,

$$\sigma^{opt}(s) = \begin{cases} \sigma(s) & \text{if } \sigma(s) \neq \emptyset \\ opt(s) & \text{otherwise} \end{cases}$$

- The *maximal closure* of σ is the strategy σ^{max} given by, for all $s \in S$ with $\rho(s) = 1$,

$$\sigma^{max}(s) = \begin{cases} \sigma(s) & \text{if } \sigma(s) \neq \emptyset \\ max(s) & \text{otherwise} \end{cases}$$

Definition 4. A (possibly partial) strategy σ of player 1 is called *weakly optimal* if for all $s \in \Gamma^*(\sigma^{opt})$ we have $\sigma(s) \subseteq opt(s)$. If $\sigma(s) \subseteq opt(s)$ even holds for all $s \in \Gamma^*(\sigma^{max})$, the strategy is called *strongly optimal*.

Hence for a partial strategy to be weakly optimal, it has to return optimal moves (or no moves at all) on all states that can be reached by following the strategy, where defined, and following arbitrary optimal moves otherwise. This ensures that any partial, weakly optimal strategy can be extended into a total, weakly optimal strategy. Indeed, it follows immediately from Definition 4 that if σ is a partial, weakly optimal strategy, then σ^{opt} is also weakly optimal (and, of course, total). Note that if σ is total then $\sigma^{opt} = \sigma$, so for total strategies, Definition 4 reduces to the previously given definition of weak optimality. A strongly optimal strategy is a partial strategy that returns optimal moves on all reachable states even when extending with arbitrary moves in all the states on which the strategy is undefined. Our goal is to build compact representations of strongly optimal strategies. Such strategies are sufficient to play perfectly (always achieve the game-theoretic value), since the player then just has to follow the strategy when defined and make random moves otherwise.

IV. Strategic rules and logical FFTs

Definition 5. Given partial strategies σ and σ', their *sequential composition* $\sigma \mid \sigma'$ is the strategy defined by

$$\sigma \mid \sigma'(s) = \begin{cases} \sigma(s) & \text{if } \sigma(s) \neq \emptyset \\ \sigma'(s) & \text{otherwise} \end{cases}$$

Definition 6. A *(strategic) rule* r of a game G is an expression of the form $l_1 \wedge \cdots \wedge l_n \Rightarrow a$, where all l_i are *literals* (elements of Atm and their negations) and a is an action (a move in the game G). If $n = 0$, we write $\top \Rightarrow a$. The intended interpretation of a strategic rule is: if $l_1, \ldots, l_n$ are true, then do a. The formula $l_1 \wedge \cdots \wedge l_n$ is called the *precondition* of r, denoted $pre(r)$, and the *action* of r is a, denoted $action(r)$. A *fast-and-frugal tree* (*FFT*) is any *sequential composition* $r_1 \mid r_2 \mid \cdots \mid r_n$ of strategic rules.

Representing strategies by FFTs in a logical form is similar to work by Jiang et al. [16] and Silva et al. [17].

Definition 7. The partial strategy $\sigma(r)$ *induced* by a strategic rule $r = l_1 \wedge \cdots \wedge l_n \Rightarrow a$ is defined by, for all s with $\rho(s) = 1$,

$$\sigma(r)(s) = \begin{cases} \{a\} & \text{if } s \models l_1 \wedge \cdots \wedge l_n \text{ and } \gamma(s,a) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

Here $s \models \phi$ means that ϕ is true in s using standard semantics for propositional logic. A rule r is said to *apply* to state s if

TABLE I
EXAMPLE OF AN FFT f

r_1 : `(cell 1 3 x) ∧ (cell 3 3 x) ⇒ (mark 2 3 x)` |
r_2 : `(cell 1 3 x) ∧ (cell 3 1 x) ⇒ (mark 2 2 x)` |
r_3 : `(cell 3 3 x) ∧ (cell 3 2 x) ⇒ (mark 3 1 x)` |
r_4 : $\emptyset \Rightarrow$ `(mark 2 2 x)`

$\sigma(r)(s) \neq \emptyset$. The partial strategy $\sigma(r_1 \mid \cdots \mid r_n)$ *induced* by an FFT $r_1 \mid \cdots \mid r_n$ is defined by

$$\sigma(r_1 \mid \cdots \mid r_n) = \sigma(r_1) \mid \cdots \mid \sigma(r_n).$$

Whenever there is no risk of ambiguity, we will identify FFTs f with their induced strategies $\sigma(f)$, hence e.g. writing $f(s)$ for $\sigma(f)(s)$. Similarly for strategic rules.

Table I provides an example of an FFT f, again based on Tic-tac-toe, and again using simplified GDL syntax. Consider the state $s = \{$`(cell 1 1 o)`, `(cell 2 1 b)`, `(cell 3 1 b)`, `(cell 1 2 b)`, `(cell 2 2 o)`, `(cell 3 2 x)`, `(cell 1 3 x)`, `(cell 2 3 o)`, `(cell 3 3 x)`$\}$. We can now compute $f(s)$: Starting from r_1, we check if it applies to s, which is the case, but since `(mark 2 3 x)` is not applicable in s, we continue to check r_2. However, $s \not\models pre(r_2)$, so we move on to r_3 that successfully applies, hence $f(s) =$ `(mark 3 1 x)`.

V. SYNTHESIZING FFTS

Let $f = (r_1 \mid \cdots \mid r_n)$ be an FFT. Note that for any state $s \in \mathrm{dom}(f)$, $f(s) = r_i(s)$, where r_i is the earliest rule in f with $s \models pre(r_i)$ and $\gamma(s, action(r_i)) \neq \emptyset$. Suppose that for some j we have, for all $s \in S$, if $s \models pre(r_j)$ and $\gamma(s, action(r_j)) \neq \emptyset$ then there exists $i < j$ with $s \models pre(r_i)$ and $\gamma(s, action(r_i)) \neq \emptyset$. Then $\sigma(f) = \sigma(r_1 \mid \cdots \mid r_{j-1} \mid r_{j+1} \mid \cdots \mid r_n)$, i.e., we can remove r_j without changing the induced strategy. In this case we say that the rule r_j is *dead* in f.

Definition 8. Given a state s and an action a, the *rule induced* by the pair s, a is $r_{(s,a)} := (\bigwedge_{p \in s} p \wedge \bigwedge_{q \in Atm - s} \neg q) \Rightarrow a$.

A naive way to synthesize a strongly optimal FFT for a game G is to: 1) compute a deterministic, total, optimal strategy $\sigma = \{(s_1, a_1), \ldots, (s_n, a_n)\}$ for G using the MINIMAX algorithm; 2) Construct the FFT $f = r_{(s_1,a_1)} \mid \cdots \mid r_{(s_n,a_n)}$. This FFT is necessarily total and optimal, since $\sigma(f) = \sigma$ (each rule $r_{(s_i,a_i)}$ only applies at exactly one state s_i). However, the entire purpose of constructing FFTs is to have compact strategies that are easy to follow, and for this purpose it is clearly not helpful to represent strategies simply as a list of optimal moves. We now introduce ways to simplify FFTs.

Definition 9. Let $f = r_1 \mid \cdots \mid r_n$ be an FFT. A *single-step simplification* of f is achieved by performing the following modifications:

1) Removing a single literal from the precondition of a single rule of f. That is, replace f by the FFT $f' = r_1 \mid \cdots \mid r_{i-1} \mid r'_i \mid r_{i+1} \mid \cdots \mid r_n$ for some i, where $r_i = l_1 \wedge \cdots \wedge l_m \Rightarrow a$ and $r'_i = l_1 \wedge \cdots \wedge l_{j-1} \wedge l_{j+1} \wedge \cdots \wedge l_m \Rightarrow a$ for some j.

Algorithm 1 Naive synthesis of FFT for game G

1: **procedure** NAIVE-SYNTHESIZE-FFT(G)
2: Compute strategy *opt* for game G using MINIMAX[3]
3: Pick a deterministic, total strategy $\sigma \subseteq opt$
4: From $\sigma = \{(s_1, a_1), \ldots, (s_n, a_n)\}$ construct the FFT $f = r_{(s_1,a_1)} \mid \cdots \mid r_{(s_n,a_n)}$
5: Iteratively apply valid single-step simplifications to f until it becomes simplification minimal
6: **return** f

2) Removing all rules that have become dead by the previous modification.

A *multi-step simplification* is a sequence of single-step simplifications. A (single- or multi-step) simplification f' of a weakly optimal FFT f is called *valid* if f' is also weakly optimal. A weakly optimal FFT is called *simplification minimal* if it allows no further valid simplifications.

Letting $f = (p \wedge q \Rightarrow a \mid q \wedge r \Rightarrow b)$, both $f' = (p \Rightarrow a \mid q \wedge r \Rightarrow b)$ and $f'' = (q \Rightarrow a)$ are single-step simplifications of f (assuming a is applicable in any state satisfying q). It is possible to define a game in which both f' and f'' are valid simplifications of f, and both are simplification minimal, showing that the the length of a simplification minimal FFT can depend on the order in which we apply the simplification steps. We can now devise a first non-trivial, but still naive, algorithm for synthesizing FFTs, Algorithm 1. Note that line 5 iteratively applies simplification steps until the FFT is minimal. The reason for doing this is that a simplification step most often changes the reachable states of the induced strategy, hence potentially making other simplification steps valid that weren't valid previously.

Consider a single-step simplification g of some weakly optimal FFT f. How costly is it to verify that the simplification step is valid? We need to verify that g is also weakly optimal, i.e., assigns optimal moves to all states reachable by g^{opt}. For the subset of states that are also reachable by f^{opt} and for which f and g specify the same action, there is of course nothing to prove (since f is assumed weakly optimal). But for all $s \in \Gamma^*(g^{opt}) - \Gamma^*(f^{opt})$ we need to verify that $g(s) \subseteq opt(s)$. In the worst case $\Gamma^*(g^{opt}) - \Gamma^*(f^{opt})$ can have size $\Omega(|S|)$, the size of the state space, e.g. if $g(s_0) \neq f(s_0)$ in a game where the underlying state transition system is a (binary) tree. We hence clearly want to both keep the number of such validations low, and choose simplifications for which the number of states to be checked is as low as possible.

It is not obvious how to achieve this. Is it for instance best to first build an entire FFT and then try to simplify its rules afterwards, or should we try to simplify each added rule as much as possible before adding a new? The answer is that unfortunately it depends on the game, so no approach is universally best. The issue is that there is a non-trivial trade-off:

[3]We compute the maximal, optimal strategy instead of just a single deterministic, total, optimal strategy since we anyway need σ^{opt} for validating simplifications in line 5.

Algorithm 2 Synthesis of FFT for game G

```
1: procedure SYNTHESIZE-FFT(G)
2:     Let f be the empty FFT
3:     Compute strategy opt for game G using MINIMAX
4:     σ ← opt
5:     while σ ≠ ∅ do
6:         σ ← σ − {(s, a)} for some choice of (s, a) ∈ σ
7:         if max(s) = opt(s) or f(s) ≠ ∅ then skip
8:         f ← f | r where r = r_(s,a)   ▷ extend f with r = r_(s,a)
9:         for all literals l in pre(r_(s,a)) do
10:            if removing l from r is a valid simplification then
11:                f ← (f with the simplification applied) [5]
12:    Apply valid single-step simplifications to f until minimal
13:    for all rules r in f do
14:        if f with r removed is (still) strongly optimal then
15:            f ← (f with r removed)
16:    return f
```

1) Extending an FFT f with new rules to get g, g^{opt} will generally have fewer reachable states than f^{opt}.[4] This speaks in favour of extending an FFT before simplifying its rules, as the extended FFT will have fewer states in which to check for optimal moves when verifying weak optimality.
2) When verifying the validity of a simplification, we don't need to consider the reachable states that were also reachable before the simplification and for which the simplified FFT specifies the same action as before the simplification. Hence, the fewer modifications we make to an FFT before we attempt to simplify one of its rules, the bigger is the proportion of such states where there is nothing to verify. This speaks in favour of simplifying rules before extending with new rules.

Which approach is best, simplify first or simplify last, depends on the particular game in question, but our experimental results suggest that most often it is better to simplify before adding new rules. This leads us to Algorithm 2, simplifying each rule before adding new rules, and applying a few other tricks to optimise computation time compared to Algorithm 1 (see section IX for a comparison of the performance of the two algorithms).

Theorem 1. For any game G, Algorithm 2 returns a strongly optimal FFT f.

Proof. First we prove that throughout the algorithm, f is weakly optimal. Initially f is empty and hence trivially weakly optimal. Consider the extension of f with $r_{(s,a)}$ in line 8. We need to prove that if f is weakly optimal, so is the extension $f \mid r_{(s,a)}$. So assume f is weakly optimal and let $s' \in \Gamma^*((f \mid r_{(s,a)})^{opt})$ be chosen arbitrarily. Then we need to prove $(f \mid r_{(s,a)})(s') \subseteq opt(s')$. We must have $f(s) = \emptyset$, since otherwise we would have skipped the current iteration of the loop in line 7 and never have reached line 8. This implies $f^{opt}(s) = opt(s)$. Since $a \in opt(s)$ (by choice of (s,a)), we get $(f \mid r_{(s,a)})(s) = \{a\} \subseteq opt(s) = f^{opt}(s)$ and hence $(f \mid r_{(s,a)})^{opt}(s) \subseteq f^{opt}(s)$. For any $s'' \neq s$, we clearly have $(f \mid r_{(s,a)})^{opt}(s'') = f^{opt}(s'')$. Hence, in total, for all $s'' \in S$, $(f \mid r_{(s,a)})^{opt}(s'') \subseteq f^{opt}(s'')$. This implies $\Gamma^*(f \mid r_{(s,a)})^{opt}) \subseteq \Gamma^*(f^{opt})$. Hence $s' \in \Gamma^*(f^{opt})$. By weak optimality of f we can then conclude $f(s') \subseteq opt(s')$. If $s' \neq s$, we now get $(f \mid r_{(s,a)})(s') = f(s') \subseteq opt(s')$, as required. If $s' = s$, we get $(f \mid r_{(s,a)})(s') = \{a\} \subseteq opt(s) = opt(s')$, again as required. This proves the extension $f \mid r_{(s,a)}$ is weakly optimal. For the other modifications of f, weak optimality is guaranteed by construction. This proves that throughout the algorithm, f is weakly optimal.

We now prove that the FFT returned in line 16 is strongly optimal. First we prove that the FFT is already strongly optimal when the while loop terminates. Call the FFT when the while loop terminates g. Since g is weakly optimal, it suffices to prove that $g^{max} = g^{opt}$. Letting $s \in S$, we need to prove $g^{max}(s) = g^{opt}(s)$. First assume $g(s) \neq \emptyset$. Then we immediately get $g^{max}(s) = g(s) = g^{opt}(s)$, as required. So assume instead $g(s) = \emptyset$. Since σ in line 4 is initialised to be the maximal, optimal strategy, it must initially contain a pair (s,a) with $a \in opt(s)$. This pair will in some iteration of the while loop be chosen in line 6. In line 8, the current FFT f will be extended with $r_{(s,a)}$ unless the iteration is skipped in line 7. Assume first that the extension in line 8 is executed. Then f is replaced by $f \mid r_{(s,a)}$ and clearly $(f \mid r_{(s,a)})(s) \neq \emptyset$. Now note that no simplification step on an FFT can make its domain smaller (simplifications only make rules apply to more states and delete dead rules). Hence, if the extension in line 8 is executed, s will also be in the domain of f when the while loop terminates, i.e., in the domain of g. This implies $g(s) \neq \emptyset$, contradicting our assumption. Assume instead that the extension in line 8 is not executed. This means that in line 7 we will have $max(s) = opt(s)$ or $f(s) \neq \emptyset$. If $f(s) \neq \emptyset$, we will also have $g(s) \neq \emptyset$, which again contradicts our assumption. If $max(s) = opt(s)$ we get, using that $g(s) = \emptyset$, $g^{max}(s) = max(s) = opt(s) = g^{opt}(s)$, as required. This proves that f is strongly optimal when the while loop terminates. Strong optimality is clearly also preserved under valid simplification steps, and in lines 13-15 we explicitly only perform operations preserving strong optimality. Hence, the returned FFT f will be strongly optimal. □

Let's show an example of applying Algorithm 2, where G is a simplified version of Tic-tac-toe with the objective to get 2 in a row instead of 3. The algorithm extracts a pair (s,a) from opt and turns it into a rule $r_{(s,a)}$: `(cell 1 1 b) ∧ (cell 2 1 b) ∧ (cell 3 1 b) ∧ (cell 1 2 o) ∧ (cell 2 2 b) ∧ (cell 3 2 b) ∧ (cell 1 3 x) ∧ (cell 2 3 b) ∧ (cell 3 3 x) ⇒ (mark 2 2 x)`. The empty FFT f is extended with $r = r_{(s,a)}$, and we now for each literal in $pre(r)$ check whether the literal can be removed from r while still

[4] The optimal closure f^{opt} of f returns all optimal moves in the states not in $\mathrm{dom}(f)$. If g is an extension of f, $\mathrm{dom}(g)$ will be larger than $\mathrm{dom}(f)$ and hence g^{opt} will have fewer states than f^{opt} in which all optimal moves are returned. In other words, going from f^{opt} to g^{opt} reduces the non-determinism of the strategy, and hence the set of reachable states.

[5] We greedily simplify the added rule r by attempting to remove one precondition literal at a time.

ensuring weak optimality. For this game, each precondition literal can actually be removed (it is always an optimal move to put a cross in the center), so we end with the simplified rule $\top \Rightarrow$ `(mark 2 2 x)`, which then becomes the current FFT f. All of the remaining states $s \in \mathrm{dom}(opt)$ satisfy either $max(s) = opt(s)$ or $f(s) \neq \emptyset$, meaning the remaining iterations of the while loop will skip in line 7. Hence when the while loop terminates, we still only have the one rule in our FFT. The algorithm will now try to first simplify the rule (line 12) and afterwards try to remove it (lines 13-15), and both will fail. Hence the algorithm will eventually return the single-rule FFT $\top \Rightarrow$ `(mark 2 2 x)`, which indeed induces a strongly optimal policy in this simple game (it is a winning strategy for the first player to place a cross in the center whenever possible and otherwise just make random applicable moves).

VI. Implementation

In line 3 of Algorithm 2, we use MiniMax to compute the maximal, optimal strategy. Our version of MiniMax employs transposition tables in combination with iterative deepening, to save memory and avoid re-expanding existing states (loop-checking) [18]. When checking if a strategy σ is weakly or strongly optimal, we compute the set of reachable states $\Gamma^*(\sigma^{opt})$ or $\Gamma^*(\sigma^{max})$, respectively, by performing a Breadth-First Search (BFS) on the game tree. We maintain a set of hashed states to avoid expanding unique states more than once. We make use of parallelization by assigning worker threads to compute the results of sub-branches of the tree.

VII. A visual tool for building, generating, verifying, playing and understanding strategies

The purpose of generating strongly optimal strategies is to teach human players to become better at a game, to learn to play the game optimally, or just to understand what the optimal strategy is (e.g. why a certain player has a winning strategy in a given game). We have built a visual tool that can help making strategies easier to understand and follow for the user. Additionally, the tool can help users build and verify their own strategies, as well as turn user-generated partial, weakly optimal strategies into strongly optimal strategies.

a) Playing with a strategy: Figure 1 shows how a strongly optimal strategy generated by our tool can be followed step by step in Tic-tac-toe when playing against a player making arbitrary optimal moves. On the left we see the states, where every legal move is represented by a color and a number. The number represents the maximum number of turns to a terminal node from the corresponding next state with optimal play. The color represents the outcome of a game, where green, yellow and red means a win, draw and loss respectively. The blue color represents the move chosen by the computed optimal strategy. On the right we see the strategy represented as an FFT, where the chosen move is highlighted in blue. This representation allows the user to quickly identify the move chosen by the FFT.

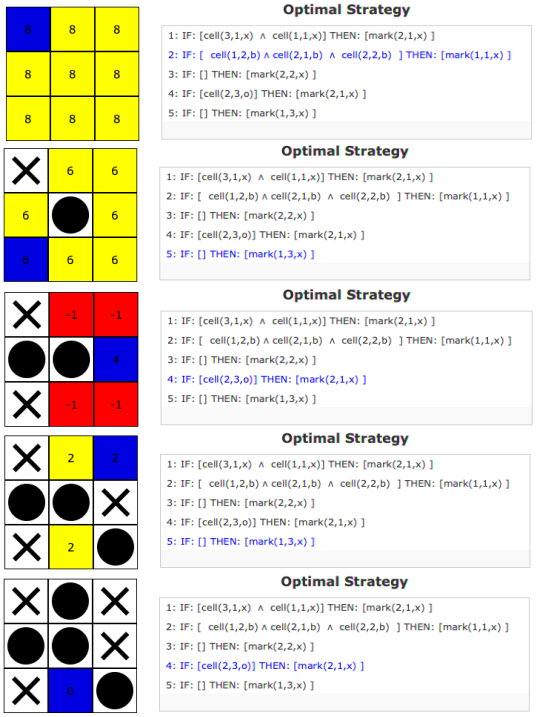

Fig. 1. Tic-tac-toe states during game play with tool

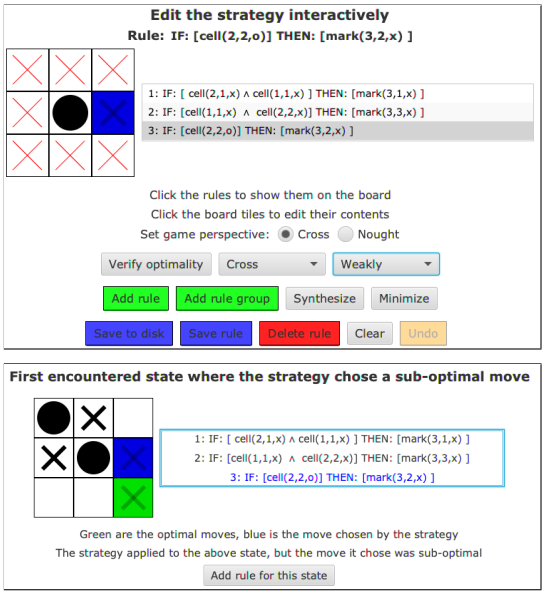

Fig. 2. Building and verifying a strategy via an interactive interface

b) Building and verifying manually crafted strategies: We provide an interactive interface where a strategy can be built from scratch by adding, removing and moving rules. It is possible to verify whether the strategy is (weakly/strongly) optimal or not. If the verification fails, the user is shown an example of a reachable state where the strategy is sub-optimal. An example of this from Tic-tac-toe is shown in Figure 2. In the first image we see the interactive interface with the Tic-

TABLE II
RESULTS OF SYNTHESIZING OPTIMAL STRATEGIES IN VARIOUS GAMES

	$\|S\|$	no. rules	no. precond. lit.	comp. time
Tic-tac-toe	4520	5	6	0.828s
Nim	342	18	24	0.1974s
Sim	1350022	32	166	14651.58s

tac-toe board on the left and the current strategy on the right. The last rule of the strategy is highlighted and showed on the board. The red crosses indicate that these positions are not part of the current rule. In the second image the user has asked the tool to check whether the strategy is weakly optimal, which it is not. The user is then presented with a state where the strategy chooses a sub-optimal move, and is given the option to create a new rule with an optimal move in that state.

Additionally, it is possible for the user to add what we call *meta-rules* to the strategy. A *meta-rule* is a sequential composition of strategic rules $r_1 \mid \cdots \mid r_n$ that can not be modified (only moved). This will allow the user to e.g. define a single meta-rule for getting 3-in-a-row in Tic-tac-toe, and then afterwards let the tool synthesize the remaining rules.

The tool is available at https://github.com/JacobPjetursson/Board-game-strategy-synthesis

VIII. EXPERIMENTAL RESULTS

We have run benchmarks on Tic-tac-toe, Nim[6] and Sim, using the version of the tool requiring game-specific functions and classes. In all cases, the tool successfully generated a strongly optimal strategy. The results are shown in Table II, where we report: 1) the size $|S|$ of the state space of the game; 2) the number of rules in the FFT generated by the tool; 3) the total number of precondition literals in the generated FFT; 4) the total computation time of generating the FFT (running time of Algorithm 2). Note that symmetry detection has been used to shorten our FFTs such that a rule that applies to a state s also applies to states symmetric to s.

For Tic-tac-toe and Nim, we chose to generate an optimal strategy for player 1, whereas in Sim we did it for player 2 (who has a winning strategy in the game). Due to the non-determinism of the algorithm (lines 6, 9, 12 and 13 in Algorithm 2), the algorithm generates FFTs of varying length. The results in Table II are the best results measured in number of rules of the produced FFTs by running the algorithm 10 times. The benchmarks were made using an Intel Xeon Gold 3.7GHz 6126 Processor with 24 threads and 384GB RAM. Note that for Nim, we generate a much larger FFT than for Tic-tac-toe, even though the state space of Tic-tac-toe is significantly larger. For Sim we generate a very small FFT compared to the size of the state space. The size of the generated FFTs are clearly more indicative of how difficult the games are to play (optimally) than the state space sizes.

The tool synthesized an optimal strategy for player 1 in Tic-tac-toe consisting of 5 rules and 6 precondition literals in total, as shown in Table III, and an optimal strategy for a variant of Nim as shown in Table IV. For the original Nim, the tool generated a strategy consisting of 71 rules.

[6]Nim rules: https://en.wikipedia.org/wiki/Nim. Note that Nim is played as a misre game and is a variant of the original game where the four heaps (noted a,b,c,d) have sizes 1,2,4,5 respectively, instead of the standard 1,3,5,7.

TABLE III
OPTIMAL TIC-TAC-TOE STRATEGY FOR PLAYER 1

```
r1 : (cell 1 1 x) ∧ (cell 3 1 x) ⇒ (mark 2 1 x)
r2 : (cell 2 2 b) ∧ (cell 1 2 b) ∧ (cell 2 1 b)
     ⇒ (mark 1 1 x)
r3 : ⊤ ⇒ (mark 2 2 x)
r4 : (cell 2 3 o) ⇒ (mark 2 1 x)
r5 : ⊤ ⇒ (mark 1 3 x)
```

TABLE IV
OPTIMAL NIM STRATEGY FOR PLAYER 1

```
r1 :  (heap a 0) ∧ (heap d 0) ⇒ (reduce c 0)
r2 :  (heap c 0) ∧ (heap a 0) ⇒ (reduce d 0)
r3 :  ⊤ ⇒ (reduce b 0)
r4 :  (heap c 1) ∧ (heap a 0) ⇒ (reduce d 1)
r5 :  (heap d 1) ∧ (heap a 0) ⇒ (reduce c 1)
r6 :  (heap d 1) ⇒ (reduce c 0)
r7 :  (heap c 0) ⇒ (reduce d 1)
r8 :  (heap c 1) ⇒ (reduce d 0)
r9 :  (heap d 0) ⇒ (reduce c 1)
r10 : (heap c 2) ∧ (heap a 0) ⇒ (reduce d 2)
r11 : (heap a 0) ∧ (heap d 2) ⇒ (reduce c 2)
r12 : (heap a 1) ∧ (heap d 3) ⇒ (reduce c 2)
r13 : (heap c 3) ∧ (heap a 0) ⇒ (reduce d 3)
r14 : (heap c 3) ⇒ (reduce d 2)
r15 : (heap d 3) ⇒ (reduce c 3)
r16 : (heap d 2) ⇒ (reduce c 3)
r17 : (heap c 2) ⇒ (reduce d 3)
r18 : ⊤ ⇒ (reduce d 4)
r19 : ⊤ ⇒ (reduce a 0)
```

Simon & Newell created an optimal strategy for Tic-tac-toe in 1972 featuring 8 rules expressed in natural language [11]. We converted these rules into our logical FFT format and used the tool to: 1) verify the strong optimality of the Simon & Newell strategy, and 2) simplify their strategy to only 6 rules. The simplified list of rules generated by the tool is, translated back into natural language:

1) **Win:** If the player has two in a row, they can place a third to get three in a row.
2) **Block:** If the opponent has two in a row, the player must play the third themselves to block the opponent.
3) **Fork:** Create an opportunity where the player has two ways to win (two non-blocked lines of 2).
4) **Blocking an opponent's fork:** Prevent the opponent from creating a fork.
5) **Center:** The player marks the center.
6) **Empty corner:** The player plays in an empty corner square.

To follow the strategy, one has to always choose the first rule that applies and make random moves if none of them do.

IX. BENCHMARKS FOR ALTERNATIVE APPROACHES

As earlier mentioned, there are four sources of non-determinism in Algorithm 2 (lines 6, 9, 12 and 13). We now consider each of these in turn, where we have benchmarked different implementations of the non-determinism.

First we consider the non-determinism in line 6 of Algorithm 2. We made experiments to test whether the order in which pairs (s, a) are extracted from the maximal, total

strategy matters for the size of the FFT generated by the algorithm. More specifically, we compared 1) random order; 2) extracting pairs (s, a) s.t. s has minimal distance to a terminal state with perfect play; 3) extracting pairs (s, a) s.t. s has minimal distance to the initial state. The differences were negligible on Nim, but on Tic-tac-toe, using 2) worked significantly better than both 1) and 3), generating only 8.4 rules on average compared to 13 for 1).

Now consider the non-determinism in line 9 of Algorithm 2: the choice of precondition literals to remove. The algorithm greedily tries to remove one literal at a time, i.e., a literal is only considered for deletion once. So if e.g. the precondition literals of the rule $l_1 \wedge l_2 \wedge l_3 \Rightarrow a$ are considered in order from left to right, we might remove l_1, whereafter it might not be possible to remove l_2 or l_3. However, had we removed l_2 first, we might still have been able to remove l_3 as well, thus ending with a rule containing fewer precondition literals. We made a non-greedy version of the algorithm that considers all subsets of precondition literals in order to find a subset of minimal size to which the rule could be validly simplified. Synthesizing 10 strategies for Tic-tac-toe, the greedy approach of Algorithm 2 on average generated 8.4 rules with 14.66 precondition literals in 0.83s. The non-greedy approach generated on average 8.14 rules with 10.57 precondition literals in 556.27s. Hence, the improvement in terms of quality of the produced FFTs using the non-greedy approach appears relatively small, and the penalty in computation time quite significant.

Finally, consider the non-determinism in line 12 of Algorithm 2: the order of simplifications. Here we also tried various orders, e.g. random, left-to-right (simplifying the left-most rules first) and right-to-left. The difference in both computation time and size of generated FFTs were negligible. Similarly for the non-determinism in line 13.

We have also compared the performance of Algorithm 2 against the naive Algorithm 1. The comparison didn't lead to huge differences in the size of the produced FFTs, but the impact on running time was significant. On Tic-tac-toe, Algorithm 2 produced an FFT 132 times faster than Algorithm 1 (0.879s vs 116.3s).

X. Related and future work

The closest related research we are aware of are by Silva et al. [17] and Antos and Pfeffer [19]. Silva et al. presents an algorithm that generates a strategy represented as an FFT, but it differs from our approach by not generating optimal strategies—and by not being a general solution that applies to any game. Antos and Pfeffer construct non-optimal reasoning patterns which models the human decision-making process, in an attempt to cover the various ways a player may try to win a game. These patterns offers advice to human game-players. They proved that human players who received advice prior to playing had better performance than those who did not.

A natural extension to our tool would be to make the definition of our rules more flexible, e.g. by allowing rules of the form $l_1, \ldots, l_n \Rightarrow \{a_1, \ldots, a_n\}$ meaning "choose any applicable action a_i", as well as allowing lifted rules using first-order logic. This would make it easier to describe existing game strategies as logical FFTs, such as the optimal strategy in Sim by Mead [20]. Additionally we could generalize the game class by allowing n-player, simultaneous games by improving our MiniMax implementation [21], [22]. Furthermore, a deeper algorithmic analysis might allow additional optimizations of the algorithm, e.g. by not having to recompute all reachable states when simplifying a strategy.

References

[1] G. Irving, J. Donkers, and J. Uiterwijk, "Solving kalah," *Icga Journal*, vol. 23, no. 3, pp. 139–147, 2000.

[2] P. Henderson, B. Arneson, and R. B. Hayward, "Solving 8x8 hex," in *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[3] R. Gasser, "Solving nine men's morris," *Computational Intelligence*, vol. 12, no. 1, pp. 24–41, 1996.

[4] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen, "Checkers is solved," *science*, vol. 317, no. 5844, pp. 1518–1522, 2007.

[5] J. N. Marewski and G. Gigerenzer, "Heuristic decision making in medicine," *Dialogues in clinical neuroscience*, vol. 14, no. 1, p. 77, 2012.

[6] M. Genesereth, N. Love, and B. Pell, "General game playing: Overview of the aaai competition," *AI magazine*, vol. 26, no. 2, pp. 62–62, 2005.

[7] Y. A. Liu and S. D. Stoller, "From datalog rules to efficient programs with time and space guarantees," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 31, no. 6, pp. 1–38, 2009.

[8] K. Waugh, "Faster state manipulation in general games using generated code," *Proceedings of the 1st general intelligence in game-playing agents (GIGA)*, 2009.

[9] S. Schiffel, "Symmetry detection in general game playing," in *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[10] J. W. Romein and H. E. Bal, "Solving awari with parallel retrograde analysis," *Computer*, vol. 36, no. 10, pp. 26–33, 2003.

[11] A. Newell, H. A. Simon *et al.*, *Human problem solving*. Prentice-Hall Englewood Cliffs, NJ, 1972, vol. 104.

[12] L. Martignon, O. Vitouch, M. Takezawa, and M. R. Forster, "Naive and yet enlightened: From natural frequencies to fast and frugal decision trees," *Thinking: Psychological perspective on reasoning, judgment, and decision making*, pp. 189–211, 2003.

[13] G. Gigerenzer, "Fast and frugal heuristics: The tools of bounded rationality," *Blackwell handbook of judgment and decision making*, vol. 62, p. 88, 2004.

[14] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[15] Y. Shoham, K. Leyton-Brown *et al.*, "Multiagent systems," *Algorithmic, Game-Theoretic, and Logical Foundations*, 2009.

[16] G. Jiang, D. Zhang, and L. Perrussel, "Gdl meets atl: A logic for game description and strategic reasoning," in *Pacific Rim International Conference on Artificial Intelligence*. Springer, 2014, pp. 733–746.

[17] F. de Mesentier Silva, A. Isaksen, J. Togelius, and A. Nealen, "Generating heuristics for novice players," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2016, pp. 1–8.

[18] A. Plaat, J. Schaeffer, W. Pijls, and A. De Bruin, "Best-first fixed-depth minimax algorithms," *Artificial Intelligence*, vol. 87, no. 1-2, pp. 255–293, 1996.

[19] D. Antos and A. Pfeffer, "Using reasoning patterns to helps humans solve complex games," in *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[20] E. Mead, A. Rosa, and C. Huang, "The game of sim: A winning strategy for the second player," *Mathematics Magazine*, vol. 47, no. 5, pp. 243–247, 1974.

[21] A. Saffidine, H. Finnsson, and M. Buro, "Alpha-beta pruning for games with simultaneous moves," in *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[22] C. Luckhart and K. B. Irani, "An algorithmic solution of n-person games." in *AAAI*, vol. 86, 1986, pp. 158–162.