# Adapting IC3 for Combinatorial Two-Player Games


| | |
|---|---|
| Name: | Jos Zandvliet |
| Date: | July 2020 |
| 1st supervisor: | Dr. Alfons Laarman |
| 2nd supervisor: | Dr. Jan van Rijn |


MASTER'S THESIS


Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

**Abstract**

Solving a combinatorial two-player game is a difficult problem. This thesis presents an approach to solving two-player games by modeling them as a symbolic transition system. A model checking algorithm, IC3, which is meant for such models and has proven to be a powerful verification method, was adapted to solve them.

This novel algorithm was validated on a number of "toy" games, and the performance of the algorithm was analyzed. The primary performance bottleneck that was identified stems from a deficiency of the generalization step, which is an important aspect of the efficiency of both the original as well as the adapted two-player version of IC3.

# Contents

# Chapter 1

# Introduction

In computer science, two-player games are used as a means to model the behavior of competing agents. Such games are often used as a "stand-in" for real problems. An example of such a problem is the "job scheduling game", in which multiple computer users compete for CPU time for their jobs. Algorithms for analyzing two-player games therefore have applications beyond trying to win board games: the goal is not necessarily to just solve a game, but to develop strategies and methods that can be used to solve other problems.

An example of the usefulness of games research is DeepMind Technologies' AlphaGo project. Winning the board game Go posed a challenging problem to prevailing methods. For AlphaGo, novel methods were developed that not only proved successful at solving Go, but were applicable in other fields, such as the synthesis of organic molecules for the development of medicine [19].

This thesis will focus on the following problem: for a given two-player game, determine if there is a strategy for the starting player such that, regardless of any possible counter-moves by the opponent, the player is guaranteed a path to victory. This is called "weakly solving" a game [8], and is a computationally harder problem than finding finding moves that are good enough to beat a human player, like AlphaGo does.

The algorithm IC3 by Aaron Bradley [12] has proven successful in the area of

model checking. Despite its unusual method of operation, which mirrors the method of a human verifier and relies heavily on the use of SAT solvers, it ranked third overall in the Hardware Model Checking Competition 2010 [14].

The main question of this thesis is: Can the algorithm IC3 be adapted to solve games? To answer this question, an implementation of IC3 was written in Python and modified to use two-player game models. This modified algorithm was used to perform an experimental evaluation.

The new algorithm was evaluated using a number of small games, which already have been weakly solved, to test if the algorithm could give the correct result. Experiments on these models show that the algorithm correctly decides whether or not a game is winnable, but for models with a large state space it is very slow.

Chapter 2 will give an overview of the relevant background topics used in this thesis, including a formal description of two-player games. Chapter 3 describes how IC3 was modified for use in such games. Chapter 4 compares this method with some related work. Chapter 5 documents the experiments that were performed to evaluate the algorithm. The final chapter is a summary of results and a conclusion.

# Chapter 2

# Background

This chapter will cover some necessary background topics, including boolean logic and SAT, as well as give a formal description of the concept of a two-player game. Finally, model checking is discussed and a brief outline of the operation of the algorithm IC3 is provided.

## 2.1 Boolean logic and the satisfiability problem

Boolean logic is a fundamental algebra used in computer science. This algebra works exclusively with boolean variables: variables that have a value in the boolean domain ($\mathbb{B}$): either true (written $\top$ or 1) or false (written $\bot$ or 0). A boolean formula is a formula over a set of boolean variables, constructed from the following elements:

- $p$ or $\neg p$: a single literal, which represents the boolean variable $p$ having the value true ($p$ means $p = \top$) or false ($\neg p$ means $p = \bot$).
- $\varphi \wedge \psi$, the conjunction (logical AND) of two formulas $\varphi, \psi$.
- $\varphi \vee \psi$, the disjunction (logical OR) of two formulas $\varphi, \psi$.
- $\neg\varphi$, the negation of another formula $\varphi$.

Boolean formulas are often expressed as the conjunctive normal form or CNF. In this form, a formula may only consist of a conjunction of clauses, where a clause is a disjunction of literals; so the general form of a CNF formula is $(a_1 \vee \cdots \vee a_n) \wedge (b_1 \vee \cdots \vee b_m) \wedge \cdots$. The negation of a clause gives a conjunction of literals, which is called a cube.

Any boolean formula can be encoded in CNF. A naïve transformation by the repeated application of distributivity and De Morgan's laws often causes an exponential increase in the amount of literals. For example, the formula $(a \wedge b) \vee (c \wedge d)$ becomes $(a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d)$.

However, this exponential increase in literals can be avoided by introducing new variables. The Tseytin transformation [5] of a formula uses this method, and gives a CNF formula with only a linear size increase compared to the original.

When a variable has a finite and discrete domain, it is possible to convert it into multiple boolean variables. One method is by encoding the value in binary, referred to as "bit-blasting". Another possible method is to give each possible value a unique bit, and add additional constraints to ensure only one bit of the group is asserted; this encoding is called "one-hot". In other words, for a variable with a domain of size $x$, a one-hot encoding needs $x$ bits (one per value) to encode the variable, whereas bit-blasting only needs $\log_2(x)$ bits.

Determining whether a boolean formula can be satisfied, that is, finding an assignment to all variables such that the entire formula is true, is called the boolean satisfiability problem (or SAT) [3]. A SAT solver is a program that, given a formula (in CNF form), either returns a cube representing a satisfying assignment, or declares the formula unsatisfiable if no such assignment exists.

SAT is an NP-complete problem, which ranks it amongst the most difficult problems in computer science. Nevertheless, much progress has been made in the development of SAT solvers [3]. Because of this other such hard problems are often expressed as SAT in order to utilize SAT solvers; model checking is an example of such a problem.

Using the notion of satisfiability, a formula can be considered to represent a set: namely, it represents the set of all variable assignments that satisfy the formula. Consequently, the solutions of a SAT solver on a formula give elements of the set it represents.

In this interpretation, the formula is called the "indicator function" for that set. For example, the set given by a formula $F$ is $[\![F]\!] = \{v \in \mathbb{B}^n \mid F(v) = \top\}$. In this thesis, the two interpretations will be used interchangeably, simply writing $F$ to represent the set of satisfying assignments of a formula $F$.

With this representation of sets as formulas, a binary relation can also be represented by a formula. A binary relation $R$ over two sets $A$ and $B$ is a subset of the set of tuples $A \times B$. If a tuple $\langle a, b\rangle \in R$, then $a \in A$ is related by $R$ to $b \in B$ (denoted $aRb$). The tuple is represented as an assignment to a set of variables. This set of variables is divided into variables used to represent $a$, and variables used to represent $b$.

If the sets $A$ and $B$ intersect, the variables used to represent $b$ are counterparts to those used to represent $a$. These counterparts, known as primed variables (and denoted with a prime symbol), represent the value of the variable in the image of the relation. After the relation is applied, the primed variables are renamed back into the original domain.

A function is a kind of binary relation. For a function $F$, $F(X)$ will be used to represent the image of $F$ over a set $X$, with this renaming applied: $F(X) \triangleq \{w \in F \mid v \in X, \langle v, w\rangle \in F\}$.

For example, let $\mathrm{NOT} : \mathbb{B} \to \mathbb{B}$ be a function mapping elements from the boolean domain $\mathbb{B} \triangleq \{\top, \bot\}$ to $\mathbb{B}$. Represented as a set, this function is $\{\langle \top, \bot\rangle, \langle \bot, \top\rangle\}$. In a boolean formula, $\mathbb{B}$ is simply a single variable, say $b$. In its formula representation, NOT can then be expressed with the constraint $(b \wedge \neg b') \vee (\neg b \wedge b')$.

The game models used in this thesis will be encoded in this way. For a game $G$ with the domain of all its variables limited to booleans, a state can simply be represented as a cube. The transition function, which is a binary relation from a state to one or more states, is expressed in the manner described above.

## 2.2 Games

For the purposes of this thesis, a game is defined as follows: A game consists of a set of positions or states [10, 11]. There are two players, which will be called A for Alice and B for Bob, of which only one can move at a time. The rules of the game specify which player has the turn, and which moves each player may perform from a given state. Player A (Alice) always moves first.

A game is assumed to be completely deterministic and have perfect information. In other words: there is no aspect of chance (such as dice rolls or card deals), and both players have complete knowledge of the current game state. Games with these restrictions are known as "combinatorial games" [11].

If the set of valid moves is identical for both players, the game is called "impartial". The game Notakto (introduced in Section 2.2.2) is an example of an impartial game. If the set of valid moves differs depending on whose turn it is, the game is called "partisan". Tic-tac-toe is an example of a partisan game, since players may only put down their own symbol.

A game is won if it reaches a state in which the win condition has been met. This condition is specified by the rules of the game; furthermore, the rules specify which player is the victor when such a state is reached. It is assumed that a game can not continue indefinitely.

### 2.2.1 Formal definition of games

This section will give a formal definition of a game. A game is a graph, where each vertex represents a game state. Each state is associated with either Alice or Bob; that player has the turn.

The definition used here is adapted from that of the problem GAME described by Greenlaw, Hoover, and Ruzzo [9: pg. 208]. To accommodate the approach used in this thesis, GAME was extended by adding the variable domains over which the game states and moves are defined. The transition function is defined over

these variables.

Consider a two-player game G, played between Alice A and Bob B. A game is represented in symbolic form as $G \triangleq \langle V, S, \mathit{turn}, s_0, T, \mathbf{won}_A, \mathbf{won}_B \rangle$, where:

- $V$ is the set of variables, used to represent the game state. It is assumed, without loss of generality, that all variables have a boolean domain.
- $S \triangleq 2^V$ is the finite set of all states. A state $s \in S$ is an assignment to all variables in the set $V$. The notation $s[x]$ is used to denote the value of the variable $x$ in a state $s$.
- $\mathit{turn} \in V$ is a variable that designates which player has the turn to move: $s[\mathit{turn}] \in \{A, B\}$.

  The notation $S_p \triangleq \{s \in S \mid s[\mathit{turn}] = p\}$ for $p \in \{A, B\}$ is used to denote the states where it's player $p$'s turn to move.
- $s_0 \in S_A$ is the initial state.
- $T : S \to 2^S$ is the transition function. The notation $T_p$ for $p \in \{A, B\}$ is used to denote the transition function for each player, where the domain has the restriction that $\mathit{turn} = p$.
- $\mathbf{won}_p \subseteq S$ for $p \in \{A, B\}$ are all states where the win condition for the game has been met (the game has been won) with player $p$ as the victor. Note that this does not necessarily mean that there are no more moves possible from such a state.

  Since there can be only one winning player, the sets of won states are disjoint. A tie or draw condition is not considered a win for either player; such states are not explicitly captured in this model.

As described in Section 2.1, the sets of states and transition function used in G can be expressed in the form of boolean formulas. For clarity, the variable *turn* is expressed in $\{A, B\}$ rather than $\{\top, \bot\}$; the two forms are equivalent.

### 2.2.2 An example game: Notakto

As an example, the game Notakto [18] is a variant of the well-known game tic-tac-toe. This game was chosen as an example because of its simplicity; its states and moves can easily be encoded as boolean variables.

A game of Notakto is played on a board of 3 by 3 squares. Two players take turns placing a cross in an unoccupied square. The game ends when the board has three crosses in the same row, column, or diagonal. When the game ends, the last player to move loses (such a game is sometimes called a "misère game"), so in order to win, a player must force the opponent to make the ending move.

A single-board game of Notakto is formalized as follows: each square on the board has a boolean variable that represents whether that square is occupied or not: $\{\boxtimes_{xy} \mid x, y \in \{1,2,3\}\} \subset V$. In the initial state, all squares are empty: $s_0 \mathrel{\hat{=}} \left(\bigwedge_{x,y\in\{1,2,3\}} \neg\boxtimes_{xy}\right)$.

The win condition, forcing the *other* player to create a three-in-a-row, is defined as:

$$\mathbf{won}_A \mathrel{\hat{=}} W \wedge \mathit{turn} = B$$
$$\mathbf{won}_B \mathrel{\hat{=}} W \wedge \mathit{turn} = A$$

where W is defined as:

$$W \mathrel{\hat{=}} \left( \bigvee_{x\in\{1,2,3\}} (\boxtimes_{x1} \wedge \boxtimes_{x2} \wedge \boxtimes_{x3}) \vee (\boxtimes_{1x} \wedge \boxtimes_{2x} \wedge \boxtimes_{3x}) \right)$$
$$\vee (\boxtimes_{11} \wedge \boxtimes_{22} \wedge \boxtimes_{33}) \vee (\boxtimes_{13} \wedge \boxtimes_{22} \wedge \boxtimes_{31})$$

The transition function consists of several parts. First of all, all squares except for the one being filled remain unchanged. Secondly, the chosen square can not already be occupied by a cross, and after the move it *is* occupied by a cross. Finally, whenever a move is made, the turn goes to the other player.

An additional constraint is used here that a move can only be made if the game has not already ended, but generally this need not be the case. Combining these constraints, the transition function T is then as follows:

$$T \triangleq \bigvee_{x,y\in\{1,2,3\}} \left( \bigwedge_{\langle a,b\rangle\in(\{1,2,3\}^2\setminus\{\langle x,y\rangle\})} (\boxtimes_{ab} = \boxtimes'_{ab}) \wedge (\neg\boxtimes_{xy} \wedge \boxtimes'_{xy}) \right) \wedge \neg W \wedge (turn' = \neg turn)$$

As Notakto is an impartial game, the transition function $T_p$ for either player $p$ is identical (excepting, of course, the fact that *turn* must have the appropriate value): $T_p = T$ for $p \in \{A, B\}$.

## 2.3 Model checking

Model checking is the problem of deciding whether or not a given hardware design or software program conforms to a specification. The subject is modeled by a symbolic transition system $\langle V, S, s_0, T\rangle$, where:

- $V$ is a set of (boolean) variables.
- $S \triangleq 2^V$ is the finite set of all states. A state $s \in S$ is an assignment to all variables in the set $V$.
- $s_0 \in S$ is a the initial state.
- $T : S \to 2^S$ is the transition function. As described in Section 2.1, such a function can be symbolically represented as a boolean formula.

In its most basic form, a specification for a model describes safety properties. A safety property $P$ is a contraint over the variables $V$, which represents the set of "good" states. In order for a model to conform to $P$, all reachable states must be in this set, or in other words, the property must always hold true in every reachable state. A model that conforms to a safety property is called "safe".

Most model checking algorithms work by exploration of the reachable state space. The size of the state space is, in the worst case, exponential in terms of the number of variables in the model. However, only those states reachable from the initial state have to be considered.

### 2.3.1 Reachability

The set of reachable states is the smallest fixpoint that includes the initial state. A fixpoint is a point of a function where the input and output of the function are identical. So, the set of reachable states is the smallest set $X$ where $s_0 \in X$ and $T(X) = X$. The reachable states can be computed by iteratively applying the transition function $T$ to a set of states, starting with only the initial state. Pseudocode for computing the reachable states is shown in Algorithm 1.

```
1 set reach (V, S, s_0, T):
      // R is the working set:  start with s_0
2     R := {s_0};
3     R' := ∅;
4     while R ≠ R':
          // add all states reachable in one step from current
             working set
5         R' := R;
6         R := T(R');
      // no new states are added, all reachable states are in R
7     return R;
```

Algorithm 1: An algorithm to return the set of reachable states.

Backwards reachability is the same principle, but applied backwards. From the set of states that violate $P$, the set of backwards reachable states can be computed by iteratively applying the predecessor function $\mathbf{pred}(T, X)$.

This function, defined as $\mathbf{pred}(T, X) \triangleq \{v \in X \mid \exists w \in X : \langle v, w \rangle \in T\}$, gives all states that are predecessors of states in $X$. The model is safe if the set of states backward reachable from $\neg P$ does not include the initial state $s_0$.

### 2.3.2 Floyd's theorem

Another approach to proving a safety property $P$ holds true is by making use of Floyd's theorem [2]. This theorem states that instead of finding the smallest fixpoint, *any* pre-fixpoint will be sufficient:

**Theorem 1 (Floyd)** *A symbolic transition system* $\langle V, S, s_0, T\rangle$ *satisfies a property* $P$ *if there is an inductive invariant* $F$ *such that* $s_0 \subseteq F$ *and* $F \subseteq P$. *A constraint* $F$ *is an inductive invariant iff* $T(F) \subseteq F$ *(i.e.* $F$ *is a pre-fixpoint).*

The major advantage of this method is that, unlike with reachability methods, there is no need to traverse the entire state space.

## 2.4 IC3

IC3 (Incremental Construction of Inductive Clauses for Indubitable Correctness) [12] is a model checking algorithm that decides the safety problem (see Section 2.3). The approach of IC3 is based on Floyd's Theorem (Theorem 1, described in Section 2.3.2).

To construct the inductive invariant required by Floyd's theorem, IC3 maintains a sequence of CNF formulas called "frames", $F_0 = s_0, F_1, \cdots F_k$. In all frames, the following invariants are maintained:

1. $s_0 \subseteq F_0$
2. $F_i \subseteq F_{i+1}$ for all $0 \leqslant i < k$
3. $F_i \subseteq P$ for all $0 \leqslant i \leqslant k$
4. $T(F_i) \subseteq F_{i+1}$ for all $0 \leqslant i < k$.

Due to these invariants a frame $F_i$ can be considered to be an over-approximation of the states of $S$ reachable from the initial state in $i$ steps, because $F_0 = s_0$, each frame $F_i$ is a strict superset of the previous frame $F_{i-1}$, and contains at least $T(F_{i-1})$.

The inductive invariant is constructed by maintaining relative inductiveness of these frames: a constraint $F$ is inductive relative to another constraint $G$ if $s_0 \subseteq F$, and $T(G \wedge F) \subseteq F$. The invariants ensure that each frame is inductive relative to the previous; so when two consecutive frames become equal, then $T(F_i) \subseteq (F_{i+1} = F_i)$: an inductive invariant is found and the algorithm finishes.

Pseudocode for IC3 is shown in Algorithm 2. It begins (line 4) by ensuring that the invariants 2, 3, and 4 apply for the initial state $s_0$ (and by extension the zeroth frame $F_0$), as a base case for the proof.

For each frame, the algorithm consists of essentially two phases: a refinement phase of finding and removing counterexamples to induction in the frame (lines 7–21), and a phase of propagating newly found constraints to later frames (lines 22–26). In the following subsections, the operation will be described in detail.

### 2.4.1 Searching and refining $F_k$

The frame $F_k$ is the frontier of the search. Initially, $k = 1$ and $F_k = P$. Until one of the frames becomes equal to its successor (meaning that it has become an inductive invariant), the goal is to prove that $T(F_k) \subseteq P$, so that with $F_{k+1} = P$, invariant 2 and 4 apply for $i \leqslant k$ and as a result $k$ may be incremented and the search expanded by another step.

#### Searching a counterexample

During the search, a set of "proof obligations" is maintained: a proof obligation is a tuple $\langle n, p \rangle$ that represents the knowledge that a formula $\neg p$ is inductive relative to $F_{n-1}$, and that $F_n \subseteq \neg p$: in other words, $p$ can not be reached from $F_0$ in $n$ steps. It is known by invariants 3 and 4 that the property $P$ is inductive relative to $F_{k-1}$, and that $F_k \subseteq P$, so the initial set of obligations is $\{\langle k, \neg P \rangle\}$ (line 8).

This initial proof obligation indicates that frame $F_k$ (but not any earlier frame) may still contain a state that leads to $\neg P$; such a state is known as a counterexam-

ple to induction (CTI) for $P$. The presence of such a state means that $T(F_k) \not\subseteq P$: invariant 4 is not met for $i = k$ yet, which is needed in order to increment $k$. In order to prove this invariant, the over-approximation $F_k$ must be strengthened by removing these CTIs for $P$.

Until all obligations are handled, the obligation with the lowest $n$ is taken (line 10). This tuple $\langle n, p \rangle$ indicates that $p$, which is a known CTI, may be reachable from $F_n$ (but not any earlier frame). To handle it, the frame $F_n$ is searched for any counterexample state $s$ that causes $T(F_n) \not\subseteq \neg p$ (that is, $s$ is a predecessor of $\neg p$; line 12).

By handling obligations sorted by $n$, this CTI can never already be present in the set of obligations (because it is a predecessor); this ensures that any cycles in the reachable state space of the model do not cause an infinite loop, ensuring that the algorithm terminates on finite systems.

#### Removing the counterexample

The found counterexample must be removed from all frames. To remove a CTI $s$ from a frame, it is negated into a clause $\neg s$; this clause is generalized to capture a larger set of removable states (explained later). The (generalized) clause is added to the frame as an additional constraint. However, $\neg s$ may not be inductive relative up to $F_k$ immediately. The highest frame $F_m$ is found where $\neg s$ is inductive relative to it (lines 14–16).

If $m = 0 : s_0 \not\subseteq \neg s$ (line 11), then the property $P$ has been violated: there is a path from the initial state to a state in $\neg P$, known as a "trace". If not, the CTI can be removed from all frames $F_0, \cdots, F_{m+1}$ from which it is known not to be reachable.

However, it may be the case that $m < k - 1$, in which case $s$ has not been excluded from $F_k$. This happens when there are one or more states $t$ in $F_m$ that lead to $s$ (the situation shown in Figure 2.1). Such states need to be eliminated before the removal of $s$ is complete, so $\langle m, s \rangle$ is added as a new obligation to handle these states.

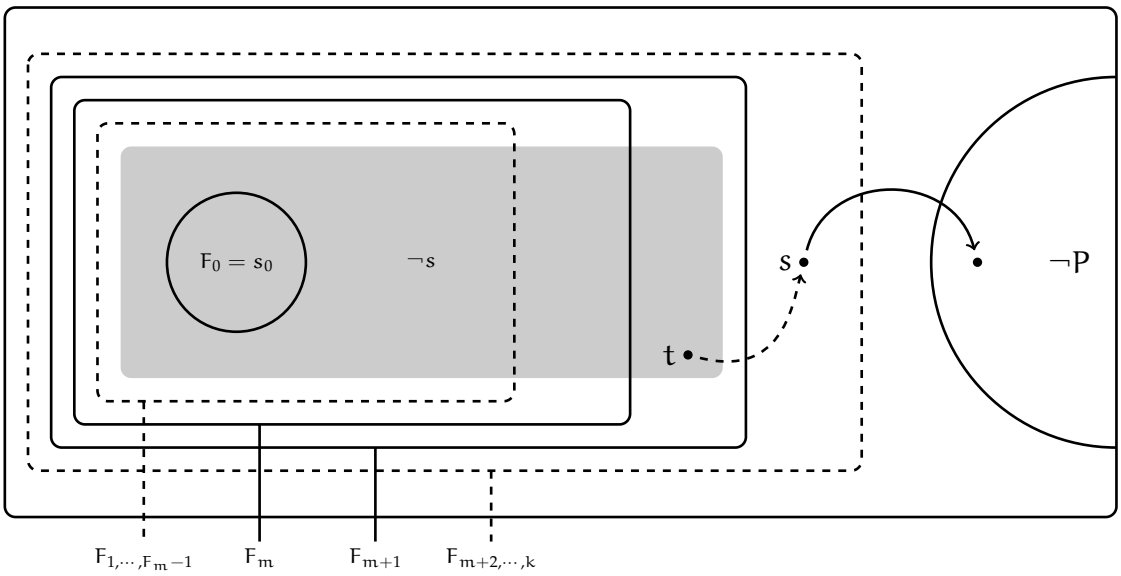



Figure 2.1: Diagram showing an example of $\neg s$ inductive relative up to an $m < k - 1$.

---

If there are no CTIs for an obligation $\langle n, p \rangle$, then $p$ itself can be excluded (as above with $s$, lines 18–21) from $F_{n+1}$ and later frames, because it is now known there are no transitions leading to it from $F_n$. The old obligation is removed; if $p$ is still not inductive relative to $F_k$, a new obligation $\langle m, p \rangle$ is added to handle $p$ in the new highest inductive frame $F_m$ (essentially, the level of the old obligation is increased).

As can be seen, when an obligation is handled, it either adds a new state not already present in the set of obligations, or it increases the level of an old obligation, removing it when that level exceeds $k$. This means that the number of obligations per frame is bounded at $(k + 1) \times$ the size of the state space.

### 2.4.2 Finishing a frame

When all outstanding obligations for the current frontier have been handled, invariant 4 has been proven for $F_k$, so $k$ can be incremented. The new frame $F_k$ is initialized to $P$.

At this point, a clause which was not inductive at an earlier point may have become inductive due to the addition of other clauses (these are called mutually inductive clauses). Such clauses are now found and "pushed forward" as far as possible, by taking each clause $c \in F_i$ where $0 \leqslant i \leqslant k$, and adding it to $F_{i+1}$ if $T(F_i) \subseteq c'$ (there are no predecessors to $\neg c$ in $F_i$). This is known as propagation.

The process of trying to increment $k$ continues until any two consecutive frames have become identical, at which point the inductivity has been proven; or until a trace is found, at which point the property is found be be violated.

### 2.4.3 Generalization

While not needed for correctness, an important aspect of IC3 is the fact that found counterexamples are generalized (in lines 15 and 19) before they are removed from the frames.

A generalization for a state is a subclause of the original formula that retains the same relative inductiveness. The advantage of this generalization process is the fact that the new clause covers more states than just one, significantly reducing the amount of work that needs to be done. The particulars of how such a subclause can be computed are not discussed here.

```
1  cube hasCTI(F_i, p):
       // Returns a counterexample to induction for a property p
          in a frame F_i.
2      return SAT(F_i ∧ T ∧ ¬p′) projected to V;
3  bool prove():
4      if (s_0 ⊈ P) ∨ (T(s_0) ⊈ P):
5          return Reachable;
6      F := [s_0, P]; k := 1;
7      while True:
           // Ob is a set of tuples ⟨n, p⟩ such that ¬p is inductive
              relative up to F_{n−1} and F_n excludes p.
8          Ob := {⟨k, ¬P⟩};
9          while Ob is not empty:
10             ⟨n, p⟩ := min(Ob);
11             if n = 0: return Reachable;
12             s := hasCTI(F_n, p);
13             if s exists:
14                 find smallest m for which SAT(F_m ∧ ¬s ∧ T ∧ ¬s′);
15                 remove generalized s from ∀0 ⩽ i ⩽ m : F_i;
16                 if m ⩽ k: Ob := Ob ∪ {⟨m, s⟩};
17             else:
18                 find smallest m for which SAT(F_m ∧ ¬p ∧ T ∧ ¬p′);
19                 remove generalized p from ∀0 ⩽ i ⩽ m : F_i;
20                 Ob := Ob \ {⟨n, p⟩};
21                 if m ⩽ k: Ob := Ob ∪ {⟨m, p⟩};
22         k := k + 1;
23         F_k := P;
24         propagate clauses F_0 ··· F_k;
25         if ∃1 ⩽ i ⩽ k : F_{i+1} ⊆ F_i:
26             return Unreachable;
```

Algorithm 2: The general outline of the IC3 algorithm.

# Chapter 3

# Solving Two-Player Games

With a game expressed in symbolic form $G \hat{=} \langle V, S, \mathit{turn}, s_0, T, \mathbf{won}_A, \mathbf{won}_B \rangle$, as described in Section 2.2.1, the problem of weakly solving a two-player game can be approached similarly to a model checking problem: namely utilizing retrograde analysis, which is a form of backward reachability (defined in 2.3.1) applied to games.

In this case, the analysis starts with a set of states that are a guaranteed win ($\mathbf{won}_p$), traversing the state space backwards to determine all states that are "winning", defined as follows.

A game state $s \in S$ is considered winning for Alice ($s \in \mathbf{winning}_A$) when there is at least one sequence of moves for Alice from $s$ that results in the game being in a state $s' \in \mathbf{won}_A$, and Bob can not make a move that results in there being no such sequence; Alice can always reach $\mathbf{won}_A$. The goal of the algorithm can then be stated as: is $s_0 \in \mathbf{winning}_A$?

Expressed more formally, $\mathbf{winning}_p$ is the smallest fixpoint under the transition function $T$, where:

- $\mathbf{won}_p \subseteq \mathbf{winning}_p$.
- $(\mathbf{pred}(T, \mathbf{winning}_p) \cap S_p) \subseteq \mathbf{winning}_p$,
  includes all states that player p can steer into $\mathbf{winning}_p$.

- $\forall\exists\mathbf{pred}(T, \mathbf{winning}_p) \subseteq \mathbf{winning}_p$,
  includes all states that can only move into $\mathbf{winning}_p$.

In the description above, the operator $\mathbf{pred}(T, X)$ is the predecessor operator, as defined in Section 2.3; and the operator $\forall\exists\mathbf{pred}(T, X) \mathrel{\hat{=}} \mathbf{pred}(T, X) \setminus \mathbf{pred}(T, \neg X)$ represents all states that have at least one successor in X, and no successors outside X.

Note that it is not needed to restrict the result of $\forall\exists\mathbf{pred}$ to one player (e.g. $\forall\exists\mathbf{pred}(T, \mathbf{winning}_p) \cap S_{\neg p}$), because it doesn't matter which player is forced to move into $\mathbf{winning}_p$; such a state is part of $\mathbf{winning}_p$ regardless.

Further, by requiring all states to have at least one successor, the result of $\forall\exists\mathbf{pred}$ excludes any deadlocked states (states that have no successors). Any deadlocked states must either be won for either player, or a tie condition (which, as noted in Section 2.2.1, are not a win for either player). As a result, deadlocked states outside $\mathbf{won}_p$ can never be part of $\mathbf{winning}_p$.

## 3.1 Adapting IC3 for Games

The approach to solving games used in this thesis is to use a method like that of IC3, using retrograde analysis with Floyd's theorem. With this method, there is no need for $\mathbf{winning}_A$ to be the smallest fixpoint; any fixpoint will suffice. The benefit is that by not requiring the smallest fixpoint, less constraints are needed to describe the set of states in each frame.

The adaptation of IC3 for games uses the same invariants as described in Section 2.4, except for the invariant 4. This invariant, (which guarantees that transitions from a frame $F_i$ are all to states contained in $F_{i+1}$) is split into two separate invariants, as follows:

4a. $T_A(F_i) \subseteq F_{i+1}$ for $0 \leqslant i < k$,
  Alice's states must have all successors in $F_{i+1}$.

4b. $T_B(F_i) \cap F_{i+1} \neq \varnothing$ for $0 \leqslant i < k$,

Bob's states must have at least one successor in $F_{i+1}$.

In order to maintain these new invariants, the following modifications are made to Algorithm 2. In regular IC3, when a counterexample to induction is sought (line 2), such as when handling the obligation of a state $s$, a predecessor to $s$ can be found with the following test: $\exists t \in F_i : T(t) \not\subseteq \neg s$. If a state has any outgoing edge to $s$, it is a counterexample.

However, in the case of invariant 4b, a state with an outgoing edge to a bad state does not necessarily need to be a counterexample. To determine this, the other successors need to be considered as well. In other words, to invalidate invariant 4b, a state must have *all* successors outside $F_{i+1}$ — not just $s$.

Take for example the transition system shown in Figure 3.1. When solving this (artificial) miniature game, at a certain point, the state `0011` will be removed from the frame $F_1$, which is shown as an outline. Before its removal, the state `0001` does not violate any invariants, because it is a Bob state with at least one

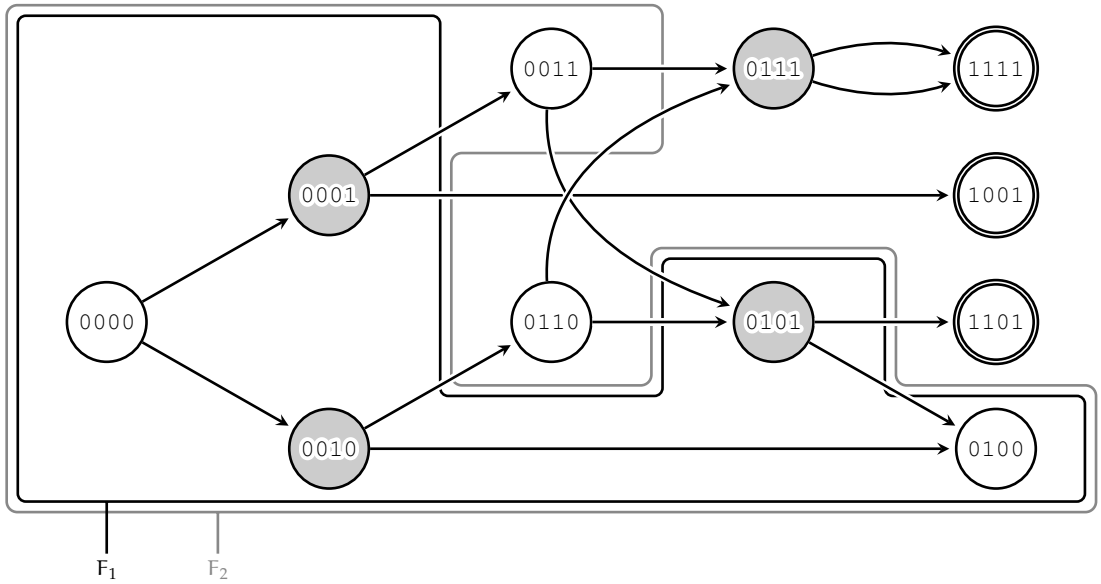



Figure 3.1: Transition system for a game "minigame". Shaded nodes represent states where Bob has the turn. States in $\mathbf{won}_A$ are indicated with a double outline.

successor in $F_2$, namely `0011`. But when this state is removed, `0001` has no other successors in $F_2$: invariant 4b is now violated, and `0001` is a CTI.

To find counterexamples that invalidate the new invariants, the test must explicitly constrain $\neg s$ to the following frame: $\exists t \in F_i : T(t) \cap (\neg s \cap F_{i+1}) = \varnothing$, or equivalently, $\exists t \in \forall\exists\textbf{pred}(\neg s \cap F_{i+1})$. (This essentially tests if there is a CTI in $F_i$ if $s$ were to be removed from $F_{i+1}$.)

This constraint was not necessary for the original IC3, nor for the test of invariant 4a, as invariant 2 already guarantees that this constraint applies in these cases.

Furthermore, in order to find counterexamples for invariant 4b, the algorithm needs some way to apply the operator $\forall\exists$**pred**. $\forall\exists$**pred** is an instance of the quantified boolean formula problem, specifically 2QBF. A solver for this operator can be implemented using two SAT solvers [4], shown in Algorithm 3.

1 **cube** $\forall\exists$***pred***$(T, X)$**:**
2 $C := S$ `// the entire state space`
3 **while** *True***:**
`// q := ∃q ∈ (C ∩ pred(T, X))`
4 $q :=$ SAT$(C \wedge T \wedge X')$ projected to $V$;
5 **if** *no such* $q$ *exists***: return** *None*;
`// r := ∃r ∈ (T(q) ∩¬X')`
6 $r :=$ SAT$(q \wedge T \wedge \neg X')$ projected to $V$;
7 **if** *no such* $r$ *exists***: break**;
8 compute generalized $\hat{q}$, such that
1. $q \subseteq \hat{q}$
2. $\forall s \in \hat{q} : \langle s, r\rangle \in T$
9 $C := C \setminus \hat{q}$;
10 **return** $q$;

Algorithm 3: Implementation of $\forall\exists$**pred** using SAT solvers.

First, a candidate state is $q$ found (line 4) that has a transition into $X$. The candidate $q$ may have no transitions to any state $r$ outside of $X$; this is verified in

line 6. If there are no such transitions, then q is included in $\forall\exists\mathbf{pred}(X)$, so it is returned.

If there is a transition to such a state r, then q is not part of $\forall\exists\mathbf{pred}(X)$. It must be removed from the set of candidates; preferably, all states $\hat{q}$ that lead to r (that is, the set $\mathbf{pred}(q)$) are removed as well. This process repeats until a valid forall-exists predecessor is found, or there are no more candidates.

Summarized, the modified `hasCTI` is as shown in Algorithm 4. The $\forall\exists\mathbf{pred}$ operator is implemented using the method described in Algorithm 3. The rest of the IC3 algorithm remains the same.

1 **cube** `hasCTI(`$F_i, p$`):`
  `// Returns a counterexample to induction for a property` $p$ `in a frame` $F_i$`.`
  `// exists-predecessor`
2  $s := \mathrm{SAT}(F_i \wedge (s[\mathit{turn}] = A) \wedge T \wedge \neg p)$ projected to $V$;
3  **if** *such an* s *exists*:
4  **return** $s$;
  `// forall-predecessor`
5  $C := F_i$;
6  $p := p \wedge F_{i+1}$;
7  **while** *True*:
  `//` $s$ `:=` $\exists s \in C$`: (T(s)` $\cap$ `(`$F_{i+1} \cap p) \neq \varnothing$
8  $s := \mathrm{SAT}(C \wedge T \wedge \neg p)$ projected to $V$;
9  **if** *no such* s *exists*: **return** *None*;
  `//` $t$ `:=` $\exists t \in$ `(T(s)` $\cap \neg p$`)`
10  $t := \mathrm{SAT}(s \wedge T \wedge p')$ projected to $V$;
11  **if** *no such* t *exists*: **break**;
12  $C := C \setminus \mathbf{pred}(T, t)$;
13  **return** $s$;

Algorithm 4: `hasCTI` modified for two-player games.

In line 12, all predecessors of t are removed. The set of predecessors of t is essentially the generalized form of s, as mentioned above.

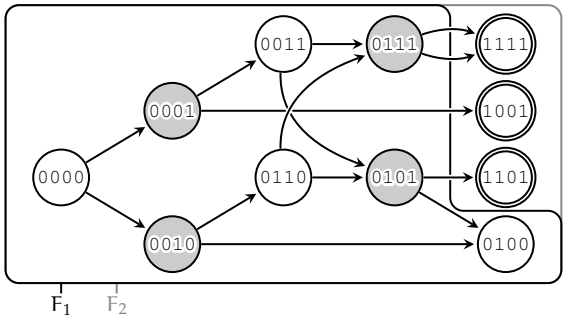
As mentioned in Section 2.4, generalization of found counterexamples to induction (not to be confused with the generalization done within $\forall\exists$**pred**) is an important point for the efficiency of the algorithm. In IC3 for games, a different strategy for generalization is needed for counterexamples found as $\exists$-predecessors and those found as $\forall$-predecessors.

For the $\exists$-step, the "down" algorithm described by Bradley [12] can be used; however, care must be taken that the variable *turn* isn't removed. Regrettably, there was not enough time to develop a generalization method for the $\forall$-step.

## 3.2 Step-by-step example: solving Minigame

This section gives a step-by-step worked example to illustrate how the algorithm solves a game. The model used here is "minigame", the toy game introduced in Figure 3.1.

$k = 1$; $Ob = \{\langle F_1, won_A \rangle\}$



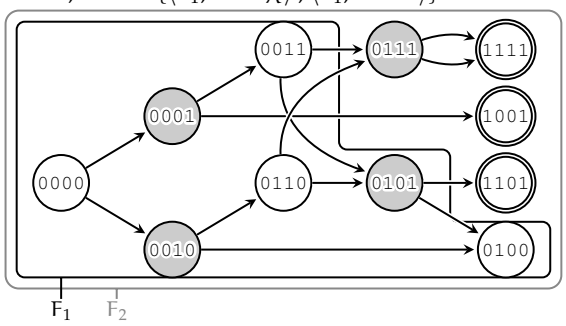

The algorithm begins with the frames as shown here. The frame $F_0$, which is not explicitly drawn, contains only the initial state `0000`. The states in $\mathbf{won}_A$ are drawn with a double outline, and as shown $F_1$ is equal to $\neg\mathbf{won}_A$. The frame $F_2$ is currently empty (contains everything).

At this point, the only obligation is $\langle F_1, \mathbf{won}_A \rangle$. To handle this obligation, the algorithm searches for a CTI for $\mathbf{won}_a$ in $F_1$, meaning: either an Alice-state (drawn in white) that has a successor outside of $F_2 \cap \neg\mathbf{won}_A$, or a Bob-state (drawn in gray) that has no successors in $F_2 \cap \neg\mathbf{won}_A$.

Is there such a state? Yes, the state `0111` is in $F_1$, and has no successors outside of $F_2 \cap \neg\mathbf{won}_A$.

To remove `0111`, the first frame $F_m$ is found where $\neg$`0111` is no longer inductive relative to it, by searching for a CTI for $F_{i+1} \cap \neg$`0111` in every frame $F_i \cap \neg$`0111`.

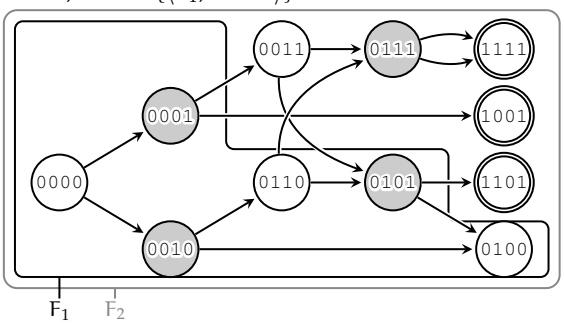
$k = 1$; $Ob = \{\langle F_1, \mathbf{won}_A \rangle, \langle F_1, 0111 \rangle\}$




In this case $m = 1$, because $F_1 \cap \neg$`0111` still contains a predecessor to `0111`, namely `0011`. $\neg$`0111` is now added to all frames up to and including $F_m$: $F_0, F_1$.

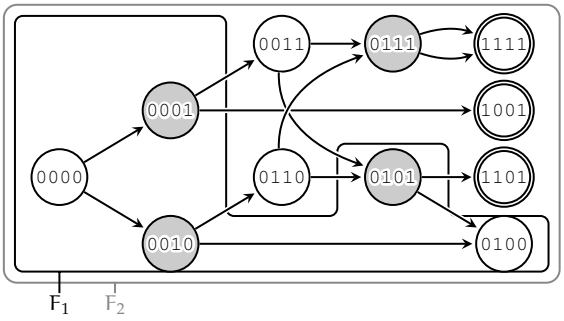
Because $\neg$`0111` is not yet relatively inductive to $F_k$, a new obligation $\langle F_1, 0111 \rangle$ is added. Another iteration for $\langle F_1, \mathbf{won}_A \rangle$ shows there are no other counterexamples to P, so that obligation is removed.

$k = 1$; $Ob = \{\langle F_1, \texttt{0111}\rangle\}$




Now $\langle F_1, \texttt{0111}\rangle$ is handled: is there a CTI? Yes, the state `0011` is in $F_1$ and has `0111` as a successor. There are no CTIs blocking its removal, so it is removed from all frames, and no new obligation is added.

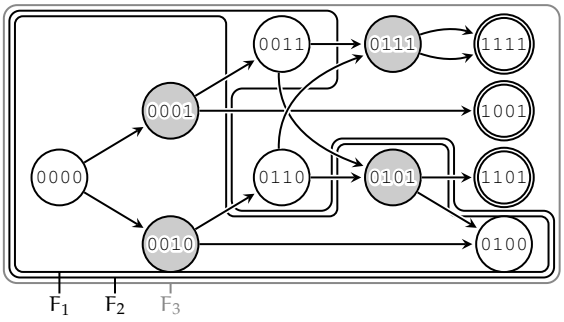
$k = 1$; $Ob = \{\langle F_1, \texttt{0111}\rangle\}$




The obligation for `0111` is still not done. This time the state `0110` is found, and it too has no blockers, so it is removed from all frames without adding a new obligation.

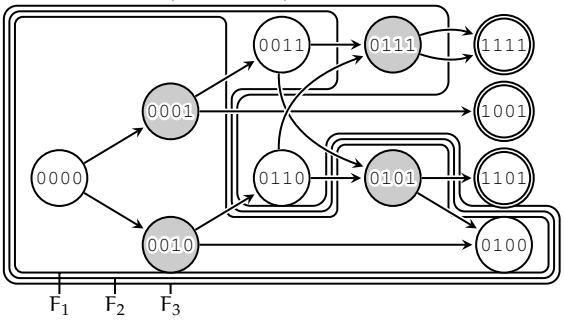
A final check for `0111` reveals there are no other counterexamples. It is now known to be inductive relative to $F_k$, so the obligation is removed. Since the set of obligations is now empty, $k$ is incremented to 2.

$k = 2$; $Ob = \{\langle F_2, won_A\rangle\}$




After propagation of clauses, $F_2$ is as shown here. The first obligation for this frame is $\langle F_2, \mathbf{won}_A\rangle$; however, there are no counterexamples. The obligation is removed, and $k$ is incremented to 3.

$k = 3$; $Ob = \{\langle F_3, won_A\rangle\}$

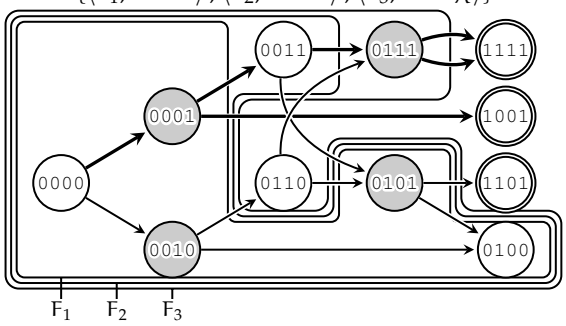



After propagation of clauses, $F_3$ is as shown here. Handling the obligation $\langle F_3, \mathbf{won}_A\rangle$ reveals that `0111` is a counterexample state.

The highest frame $F_m$ is found where $\neg$`0111` is no longer inductive relative to it, which is $F_2$. $\neg$`0111` is added to all frames up to and including $F_2$ (already the case), and a new obligation $\langle F_2, \texttt{0111}\rangle$ is added.

The smallest obligation is $\langle F_2, \texttt{0111} \rangle$; and it has a counterexample state, namely `0011`. This state is inductive relative up to $F_1$, and like the last obligation it is already excluded from those frames. A new obligation $\langle F_1, \texttt{0011} \rangle$ is added.

$\text{Ob} = \{\langle F_1, \texttt{0011} \rangle, \langle F_2, \texttt{0111} \rangle, \langle F_3, \mathbf{won}_A \rangle\}$




The smallest obligation is $\langle F_1, \texttt{0011} \rangle$; and searching for a counterexample state reveals `0001`. But this time, the state `0001` is not even inductive relative to $F_0$ (because it is reachable from $s_0$). Because this counterexample state is reachable from $F_0$, the algorithm has found a trace to $\mathbf{won}_A$ and it ends: Minigame is indeed winnable for the first player.

# Chapter 4

# Related Work

The algorithm in this thesis is based on IC3, an algorithm developed by Aaron Bradley [12]. Bradley has authored multiple papers which describe IC3 in detail [12, 13]. An alternative description of IC3 and related algorithms, written in a non-deterministic fashion, is given by Gurfinkel [15].

An earlier approach to solving games with IC3, though in a different context, is that of Morgenstern, Gesell, and Schneider [16]. In this paper, a modified form of IC3 is applied to the field of "controller synthesis": rather than weakly solving a two-player game, the goal is to automatically synthesize a system that conforms to a specification.

A limited form of synthesis is explored: namely: given a partial system, determine whether there exists a controller that can complete the system according to the specification. The synthesis process is modeled as a two-player game: one player is the controller, and the opponent is the partial system. Together, the two players decide on each transition. The controller "loses" if the specification is violated.

Rather than maintaining a single set of frames, Morgenstern et al. separate the frames (which they call rank traces) by "player" (since they can assume that the turn always changes after each move), as well as keeping an explicit set of winning states.

# Chapter 5

# Experiments

This section describes experiments done in order to evaluate the working and performance of the new algorithm. Implementations of IC3 and the new IC3 for games algorithm as described in this thesis were written in Python. This implementation makes use of the SAT solver MiniSat 2.2 [6] (available in the package `pysat` [7]). The input format used for models is the AIGER format [17], a format that describes model checking problems in the form of "and-inverter graphs".

Several models were written in the SMV modeling language. They were converted into AIGER format using the `smvtoaig` utility, which is part of the standard AIGER utility distribution [17]. The models developed for this thesis are: "minigame", the small model shown in Figure 3.1; Notakto, described in Section 2.2.2, as well as cut-down variants of Notakto played on smaller boards; and finally, regular tic-tac-toe, also including smaller board variants.

The generalization step in Algorithm 4 (line 12) proved to be difficult to implement. In the current implementation, the predecessors are found by enumerating all satisfying assignments to $T \wedge t'$ and removing them from C individually (by adding the negated form to C); there seems to be no simple way to remove the entire set from C directly: a naïve attempt to substitute $t'$ in the transition function would interfere with the SAT call in line 8.

For all Notakto variants, the first to three-in-a-row loses; for all tic-tac-toe variants except $2 \times 2$, the first to three-in-a-row wins. The $2 \times 2$ tic-tac-toe variant has two-in-a-row as its win condition, which makes it trivial to see that the first player can always win.

To analyze the performance of the algorithm, the run time of the execution of the new algorithm on these models was measured. Furthermore, to validate the correct working of the algorithm, the final result (whether the game in question is winnable for the first player or not) was compared to the (known) result for each game; the result given by the algorithm was correct on all models.

Table 5.1 summarizes the performance of the algorithm on these models. The column "|V|" gives the size of the (syntactic) state space for that model; note that the amount of reachable states is often smaller. The column "obligations" gives the total amount of obligations handled, and the column "SATs" gives the total amount of calls to the SAT solver.

Run times are given in seconds. These times were measured with Python's `time.process_time()`, which gives the sum of CPU time in both kernel and user space. The run time measured is the difference of these times sampled before and after the execution of the main loop. In order to only measure the behavior of the algorithm itself, the initialization and loading of the model is not included in the run time.

To gauge the impact of generalization (described in Section 2.4.3), statistics are given for each model with and without generalization. As mentioned in Section 3.1, generalization can currently only be applied to exists-states. The rows where generalization was enabled are marked with $\star$. As can be seen, generalization causes an increase in the amount of SAT calls and run time for models up to a certain size. It appears that for small models, the overhead of generalization is greater than the performance gain it gives.

The standard Python profiler `cProfile` was used to analyze the run time behavior of the program. This indicated that a large amount of time was being spent in the forall-part of the new `hasCTI` function: for the $3 \times 3$ Notakto model

with generalization enabled, a little over 78% of the run time is spent searching for forall-predecessors.

The reason for can be attributed to the algorithm's lack of a generalization function for the forall-states. Since each state is now removed individually, each frame contains more constraints, and as a result each SAT call takes more time.

| model | | $\lvert V\rvert$ | obligations | SATs | run time | winnable? |
|---|---|---|---|---|---|---|
| minigame | | 16 | 9 | 80 | 0.036 s | yes |
| | $\star$ | | 9 | 157 | 0.056 s | |
| notakto | $(2 \times 3)$ | 128 | 103 | 4902 | 8.124 s | no |
| | $\star$ | | 102 | 7288 | 21.840 s | |
| | $(2 \times 3, \boxtimes AA)$ | 64 | 68 | 2172 | 2.327 s | yes |
| | $\star$ | | 52 | 2945 | 3.518 s | |
| | $(3 \times 3)$ | 1024 | 279 | 65512 | 879.098 s | yes |
| | $\star$ | | 207 | 23620 | 231.895 s | |
| tic-tac-toe | $(2 \times 2)$ | 162 | 33 | 422 | 0.325 s | yes |
| | $\star$ | | 33 | 609 | 0.407 s | |
| | $(2 \times 3)$ | 1458 | 458 | 110187 | 6042.698 s | no |
| | $\star$ | | 225 | 48190 | 1269.657 s | |
| | $(3 \times 3)$ | 39366 | — | — | —.000 s | no |
| | $\star$ | | — | — | —.000 s | |

Table 5.1: Results of experiments.

The Notakto variant marked as "$2 \times 3, \boxtimes AA$" is the same as the $2 \times 3$ variant, except that the initial state has the first square $\boxtimes AA$ already occupied with a cross. This, in effect, reverses the roles of the two players, turning Alice into the "second" player. As seen in the table, the first player can not force a win in a $2 \times 3$ game. Because Notakto can not end in a tie, this means that the $\boxtimes AA$ variant *can* be forced.

# Chapter 6

# Conclusion

In this thesis the algorithm IC3 was adapted to weakly solve two-player combinatorial games. The new algorithm was applied to a number of games that were already weakly solved, in order to verify that the correct result is returned and to measure the performance of the algorithm.

The experiments show the adapted algorithm gives the correct result for all games tried. However, as the size of the model (measured in the amount of states) increases, the run time soon becomes too large. While generalization can improve this, the largest model (tic-tac-toe) proved still too large to be solved.

As mentioned in Chapter 5 and Section 3.1, the generalization method used in the current implementation is deficient: it is only able to perform generalization on CTIs from the exists-predecessor, and not those of the forall-predecessor.

Given the large importance of generalization for the performance of regular IC3, it seems likely that the ability to generalize the forall-states could yield a significant performance gain by keeping frames small and SAT calls fast. Development of a generalization method for forall-states remains future work.

# Bibliography


### Model checking

[1] Tarski, A. *A lattice-theoretical fixpoint theorem and its application*. Pacific Journal of Mathematics, vol. 5 (1955): 285–309.

[2] Floyd, R. W. *Assigning Meanings to Programs*. Proceedings of a Symposium in Applied Mathematics, vol. 19 (1967): 19–32.

### SAT

[3] Biere, A.; Heule, M.; Van Maaren, H.; Walsh, T. *Handbook of Satisfiability*. IOS Press (2009).

[4] Dutertre, B. *Solving Exists/Forall Problems With Yices*. 13th International Workshop on Satisfiability Modulo Theories (2015).

[5] Tseytin, G. S. *On The Complexity of Derivation in Propositional Calculus*. Leningrad Seminar on Mathematical Logic (1966).

[6] Eén, N.; Sörensson, N. *An Extensible SAT-solver*. Lecture Notes in Computer Science, vol. 2919 (2003): 502–518.

[7] Ignatiev, A.; Morgado, A.; Marques-Silva, J. *PySAT: A Python Toolkit for Prototyping with SAT Oracles*. Lecture Notes in Computer Science, vol. 10929 (2018): 428–437.

### Games

[8] Allis, L. V. *Searching for Solution in Games and Artificial Intelligence*. Rijksuniversiteit Limburg (1994).

[9] Greenlaw, R.; Hoover, H. J.; Ruzzo, W. L. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press (1995).

[10] Berlekamp, E R.; Conway, J. H.; Guy, R. K. *Winning Ways for Your Mathematical Plays*. A K Peters (2001).

[11] Ferguson, T. S. *Game Theory* (2014). Retrieved from `https://www.math.ucla.edu/~tom/Game_Theory/Contents.html`

### IC3

[12] Bradley, A. R. *SAT-Based Model Checking Without Unrolling*. In "Verification, Model Checking, and Abstract Interpretation". Springer (2011): 70–87.

[13] Somenzi, F.; Bradley, A. R. *IC3: Where Monolithic and Incremental Meet*. Formal Methods in Computer-Aided Design (2011).

[14] Biere, A.; Claessen, K. *Hardware Model Checking Competition 2010*. First Hardware Verification Workshop (2010).

[15] Gurfinkel, A. *IC3, PDR, and Friends*. Summer School on Formal Techniques (2015). Retrieved from `https://arieg.bitbucket.io/pdf/gurfinkel_ssft15.pdf`.

[16] Morgenstern, A.; Gesell, M.; Schneider, K. *Solving Games Using Incremental Induction*. Lecture Notes in Computer Science, vol. 7940 (2013): 177–191.

### Miscellaneous

[17] Biere, A. *The AIGER And-Inverter Graph (AIG) Format Version 20071012*. Institute for Formal Models and Verification (2007).

[18] Plambeck, T. E.; Whitehead, G. *The Secrets of Notakto: Winning at X-Only Tic-Tac-Toe* (2013). Retrieved from `https://arxiv.org/abs/1301.1672`

[19] Segler, M. H. S.; Preuss, M.; Waller, W. P. *Planning chemical syntheses with deep neural networks and symbolic AI*. Nature, vol. 555 (2018): 604–610.