

Games, Strategies, and Boolean Formula Manipulation

by

Ben Andrews

B.A., Mathematics, Hendrix College, 2001

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December 2005

©2005, Ben Andrews

Dedication

To my wife.

Acknowledgments

This material is based upon work supported by the National Science Foundation (grants CCR-0085792, CCR-0219587, EIA-0218262, EIA-0238027, and EIA-0324845), the Defense Advanced Research Projects Agency (grant F30602-02-1-0146), Microsoft Research, and Hewlett-Packard (gift 88425.1). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the sponsors.

Games, Strategies, and Boolean Formula Manipulation

by

Ben Andrews

ABSTRACT OF THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December 2005

Games, Strategies, and Boolean Formula Manipulation

by

Ben Andrews

B.A., Mathematics, Hendrix College, 2001

M.S., Computer Science, University of New Mexico, 2005

Abstract

Biomolecular automata based on chemical logic gates can play two-player games of perfect information provided that a legal strategy for the game can be expressed in terms of Boolean formulas. To facilitate the construction of such automata, we developed a formal procedure for the analysis of game strategies, their translation into Boolean formulas, and the manipulation of the resultant formulas into canonical forms determined by the available logic primitives. As a case study in the application of this procedure, we generated strategies for the first player in the game of tic-tac-toe that, by construction, are favorable (do not admit a loss), and we found that many can be translated into Boolean formulas. Moreover, the resultant formulas can be expressed in logic with a fan-in of five, but, apparently, not in simpler logic. Additionally, we studied the combinatorics of strategies and, for the first player in the game of tic-tac-toe, we computed the number of all strategies, about $1.423 \cdot 10^{124}$, and the number of favorable strategies, about $2.648 \cdot 10^{103}$.

Contents

List of Figures	xiii
1 Introduction	1
1.1 Research Progression	2
1.2 Representing Boolean Formulas in Chemistry	3
1.3 Representing Game Strategies with Boolean Formulas	4
1.4 A Final Introductory Word	4
1.5 Organization of the Thesis	5
2 Background	6
2.1 Fundamental Chemistry of Molecular Logic Gates	6
2.2 The Niche of This Work	9
3 Games and Strategies	10
3.1 Strategy Trees	10
3.2 The Process of Converting Strategies into Boolean Formulas	11

Contents

3.2.1	Resolvable Conflict	14
3.2.2	Unresolvable Conflict	16
3.2.3	Feasibility	18
3.3	Favorability	18
4	Boolean Formula Manipulation	20
4.1	Introduction	20
4.2	Motivation for new tools	21
4.3	Minimo	22
5	Tic-Tac-Toe: A Case Study	24
5.1	Game Description	24
5.2	Strategy Tree Description	24
5.3	The Number of Strategies	25
5.3.1	Upper and Lower Bounds: A Simple Overcount and A Simple Undercount	26
5.3.2	The Exact Number of Strategies	28
5.3.3	The Exact Number of Favorable Strategies	30
5.4	The Process of Strategy Generation	31
5.4.1	Generating Favorable Strategies	32
5.4.2	Generating Feasible Strategies	33

Contents

5.4.3	Generating Feasible, Favorable Strategies	36
5.5	Generating Boolean Formulas from Strategies	36
5.5.1	A Defense of Redundancy	37
5.5.2	Checking the Formulas	38
5.6	Our Tools, Their Locations, and Their Uses	38
5.6.1	File Formats	39
6	Conclusions	40
6.1	Strategy Conversions	40
6.2	Boolean Manipulation	41
6.3	Tic-Tac-Toe Results	41
7	Future Work	43
7.1	Tic-Tac-Toe	43
7.2	Other Games	44
7.2.1	Connect Four	44
7.2.2	Othello	44
	Appendices	44
A	A Strategy	45
B	Our Tools, Their Locations, and Their Uses	63

Contents

B.1	Stratgen	63
B.1.1	Usage and Output	64
B.2	Stratcounter	64
B.2.1	Usage and Output	64
B.3	Checker	65
B.3.1	Usage and Output	65
B.4	CandBuild	65
B.4.1	Usage and Output	66
B.5	Minimo	66
B.5.1	Usage	66
B.5.2	Output	66
B.6	Minimo Strategy Input File	67
C	Computing Values For f	68
C.1	Computing the Value of $f([], [0])$	69
C.2	Computing the Value of $f([], [1])$	69
C.3	Computing the Value of $f([], [4])$	69
D	Computing Values For g	70
D.1	Computing the Value of $g([], [0])$	70
D.2	Computing the Value of $g([], [1])$	70

Contents

D.3 Computing the Value of $g([], [4])$	70
References	71

List of Figures

2.1	Reactions catalyzed by phosphodiesterase and ligase currently under construction are exact reverses of each other.	7
2.2	YES gate, in which an oligonucleotide I_A activates the deoxyribozyme by opening an inhibitory stem.	7
2.3	NOT gate, in which an oligonucleotide I_B deactivates the deoxyribozyme by opening a stem and destroying a catalytic core.	8
2.4	More unimolecular gates: A. AND NOT gate (also known as NOT IF or ONLY gate), active only when I_A is present and I_C is absent; B. AND gate, active when both I_A and I_B are present; C. A complex, three-input “INHIBIT” gate combining an AND gate and a NOT gate; gate is active only when both I_A and I_B are present, but not I_C	8
3.1	An example of a strategy tree. Each edge is labelled with the opponent’s move. Each node is labelled with the strategy’s move. The strategy moves first.	11
3.2	An example of a strategy tree. Each edge is labelled with the opponent’s move. Each node is labelled with the strategy’s move. The strategy moves first.	13

List of Figures

3.3	An example of a strategy tree with an unresolvable conflict.	17
3.4	A strategy tree that demonstrates the idea of move hiding.	17
3.5	An example of a (slightly larger) feasible strategy tree.	19
5.1	A labelled board for the game of tic-tac-toe.	25
5.2	A strategy tree for the game of tic-tac-toe.	26

Chapter 1

Introduction

Computer science has always been about revolutionary ideas. Alan Turing modeled his first ideas of computers around his understanding of the process behind the work of mathematicians (infinite time and tape) [15]. More recently Adleman demonstrated a technique of computation that solves any instance of an NP-hard problem in linear time (if you set enough chemicals up first) [1]. Some might say that this is a form of “brute-forcing” the solution.

This complaint of “brute force” could certainly not be made if instead of encoding the edges of the graph into strands of DNA someone had encoded a machine of simple Boolean logic gates to solve the same problem. And that is the jumping-off point of this work. A chemical computation paradigm has been introduced that simulates Boolean logic at the individual gate level (meaning the Boolean logic gates are what is actually synthesized) [11, 12, 14]. We will explore what we can do with this new technology (what limits must be placed on the Boolean functions to be simulated) and describe research into a field of possible applications: automata that play games.

1.1 Research Progression

Some time ago we set out to expand on the results printed in the MAYA paper [14]. We wanted to demonstrate a complete strategy for the game of tic-tac-toe. In order to do this we needed to understand how the chemical computation paradigm represented Boolean functions. We knew that the representation did not perfectly simulate Boolean functions. Specifically, there is no way to remove inputs from solution, and there is no way to stop a vat from fluorescing.¹ In essence, we are dealing with “monotonic” Boolean functions. Once they evaluate to true, they will always “report” true. This monotonicity actually served to help us as we defined a process for converting strategies into correctly-functioning Boolean formulas.

Generating strategies for tic-tac-toe that can subsequently be converted into correctly-functioning Boolean formulas is not too straightforward a task. A strategy is correct if it simply makes proper moves. At the beginning of this work it was unknown what constraints needed to be applied to strategies to allow them to be represented correctly by Boolean formulas. This thesis describes the refinement process we went through in order to correctly define the criteria for a strategy that was a candidate for this conversion process. This strategy-theoretic knowledge is discussed before its application to the game of tic-tac-toe.

Once a strategy has been generated and it has been converted into disjunctive normal form (DNF) Boolean formulas, each of its disjuncts must be mapped to gates that can be represented in the chemistry. Because of the scope of our problems, it became clear that we should develop a special-purpose Boolean formula manipulation program.

Thus, our approach was the following: We wrote programs to generate strategies for the game of tic-tac-toe. We went on to write programs to generate Boolean formulas from

¹These performance restrictions are a byproduct of the closed systems the reactions currently operate in.

the strategies.² Next, we wrote programs to check those strategies. Finally, we needed to tackle the problem of attempting to fit formulas to the current technological constraints of the chemical computation paradigm.

1.2 Representing Boolean Formulas in Chemistry

As previously stated, our chemical computation paradigm is capable of simulating logic gates up to a certain point. In its current (closed-system) form, the paradigm has some limitations. The “computation” is performed in a vat or test tube. The computation begins when the molecules representing the logic gates are added to the solution in the vat. After the gates are added, the chemicals representing the inputs to the gates may be added. The molecules representing the gates fluoresce when they are activated. This corresponds to the gate reporting that it has evaluated to true.

Let us say that we have a system in which we are representing the Boolean formula $y = a \vee (b \wedge c)$. We represent this formula in a vat with two kinds of gate molecules. The first kind of gate molecule will fluoresce in the presence of the molecule representing a while the second will fluoresce in the presence of the molecules representing b and c . An input to the formula is considered to have a value of true if its corresponding input molecule has been added to the vat. The chemical computation paradigm will correctly represent the formula so long as the inputs to the formula (a , b , and c) make state changes only from false to true. Once the chemical representing a has been added to the vat, it can not be removed.

A vat will represent a Boolean formula in disjunctive normal form. Each gate molecule will “work” independently. As a consequence, only one kind of gate molecule in a certain vat need fluoresce for the entire vat to fluoresce. Each gate molecule, then, will be repre-

²In order to do this properly, we had to develop a method for doing this for strategies not just for the game of tic-tac-toe but for any game.

senting a conjunction of inputs. The presence of more than one type of gate molecule in a vat will result in the implicit “OR”-ing of the values of the gates.

1.3 Representing Game Strategies with Boolean Formulas

This work began with a predefined idea of how we wanted to represent a strategy using Boolean formulas. This work did not begin, however, knowing whether or not this representation (and the implicit conversion) would be able to provide enough freedom to represent a game strategy with more than two or three moves in sequence.

The chemical logic gates will represent a strategy’s play. The inputs to those logic gates will represent the strategy’s opponent’s play. From here on, we will refer to these two players as “the strategy” (or “automaton”) and “the opponent” (or “the human”). A formula will represent a field of the game board. If the formula evaluates to true then the strategy wishes to play in that field. When the human wishes to play in a field X , the human sets the variable representing that field to true. Again, this corresponds to adding a chemical representing the input field X to all³ of the vats.

1.4 A Final Introductory Word

This work has a practical application in the laboratory. While most of the work is idealized, we would like to be able to perform experiments demonstrating our ideas. There are

³Adding the chemical input to all of the vats changes the make-up of all of the vats. After any move by the opponent, all of the vats have a chance to respond to a change in their chemical make-up. We could add the chemical representing input field x to only the vat that we are expecting to be the automaton’s next play, but this would presuppose that we know what the automaton’s next play is going to be.

current technological limitations to the sizes and shapes of the gates that can be simulated in the chemistry. This affects the disjunct sizes (number of positive and negative literals) in the (DNF) Boolean formulas to be simulated. After we have converted a strategy into a set of Boolean formulas, we will attempt to find an equivalent representation of those formulas using only such formulas as can be simulated under the current technological limitations.

1.5 Organization of the Thesis

Chapter 2 describes the setting for the application of the end results of this work. Chapter 3 formalizes the process of converting strategy trees into Boolean formulas. Chapter 4 describes the development and implementation of a special-purpose Boolean formula manipulator (Minimo) inspired by the algorithm used in the BOOM Boolean formula minimizer [6]. Chapter 5 describes our application of this theory to the game of tic-tac-toe.

Chapter 2

Background

This work was done in support of a chemical computation paradigm used in ongoing research [11, 12, 14]. In Section 2.1 I will describe the basic ideas behind this research. In Section 2.2 I will describe how this work fits into the structure provided by earlier work.

2.1 Fundamental Chemistry of Molecular Logic Gates

Deoxyribozyme logic gates are constructed via a modular design that combines molecular beacon stem-loops [11] with hammerhead-type deoxyribozymes. Deoxyribozymes are nucleic acid enzymes; of particular interest for our purposes are those that can catalyze cleavage [3, 4] or ligation [5] reactions (see Figure 2.1).

The gates (Figures 2.2–2.4) use oligonucleotides as both inputs and outputs. Correct functioning of individual gates was experimentally verified through fluorescent readouts **F** [12] and PAGE (polyacrylamide gel electrophoresis). Once an individual gate has begun to fluoresce, it will continue to fluoresce no matter what inputs are subsequently added to the system.

Chapter 2. Background

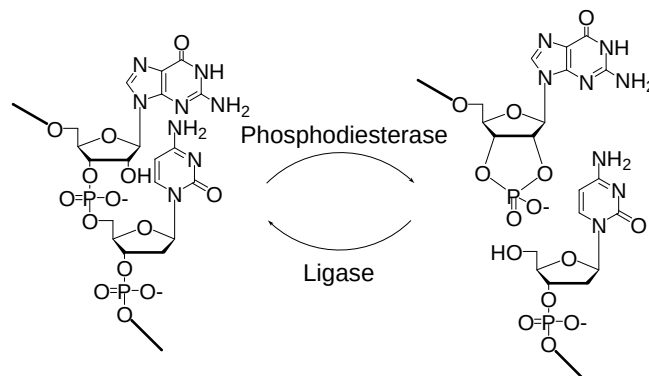


Figure 2.1: Reactions catalyzed by phosphodiesterase and ligase currently under construction are exact reverses of each other.

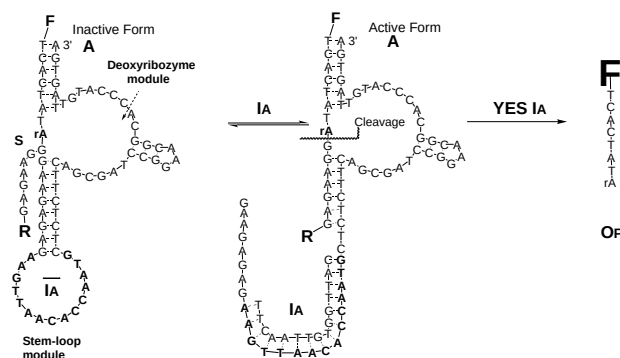


Figure 2.2: YES gate, in which an oligonucleotide I_A activates the deoxyribozyme by opening an inhibitory stem.

In recent years there has been a number of demonstrations of molecular systems that can carry out logic operations required in computer circuitry, including synthetic molecules that act as switches. In most of these prototype molecular logic gates the output was a change in spectroscopic properties. Until our demonstration, it was not possible to couple solution-phase molecular elements into circuits without using macroscopic (non-molecular) interfaces that would fully dominate the system [2].

We have constructed a full set of molecular scale logic gates with oligonucleotides as inputs and outputs. The gates use a modular design to combine deoxyribozymes (nucleic

Chapter 2. Background

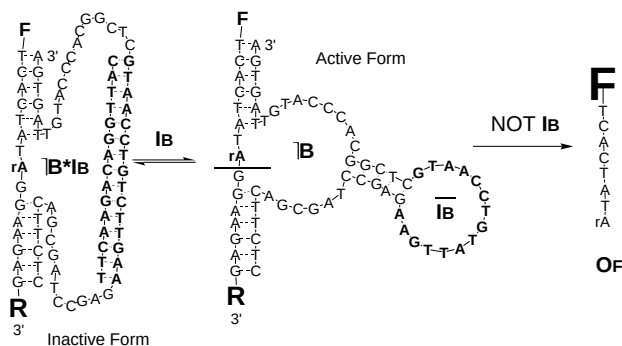


Figure 2.3: NOT gate, in which an oligonucleotide I_B deactivates the deoxyribozyme by opening a stem and destroying a catalytic core.

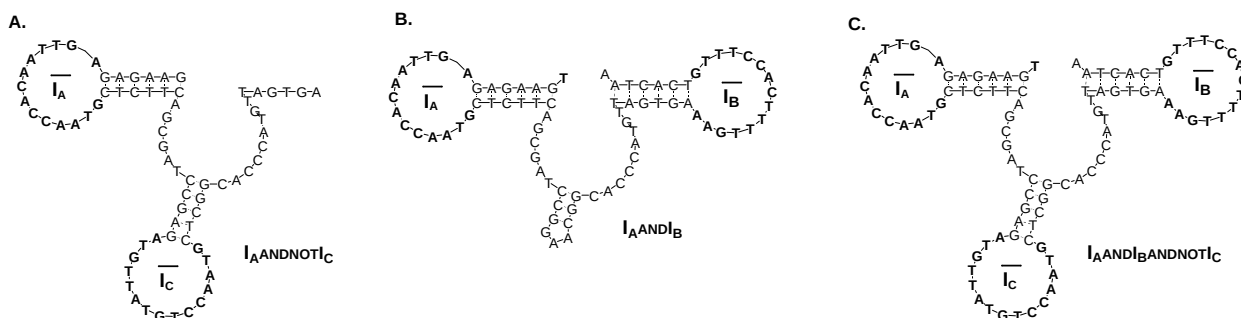


Figure 2.4: More unimolecular gates: A. AND NOT gate (also known as NOT IF or ONLY gate), active only when I_A is present and I_C is absent; B. AND gate, active when both I_A and I_B are present; C. A complex, three-input "INHIBIT" gate combining an AND gate and a NOT gate; gate is active only when both I_A and I_B are present, but not I_C .

acid enzymes) with phosphodiesterase activity and stem-loop elements. We have built one-input YES,¹ NOT, two-input OR,² NOR, XOR, AND NOT, and AND gates (Figures 2.2–2.4), and more recently three-input gates (INHIBIT or AND AND NOT), as in Figure 2.4C, and a two-input two-output system, the half adder [13]. An implicit XOR system with an increase in green fluorescence (fluorescein) as readout was combined with an AND gate with an increase in red fluorescence as readout, yielding a half-adder with two inputs and

¹The term "YES gate" for a sensor or repeater is established in chemistry literature.

²Note that an implicit OR is easy to achieve by having two upstream gates yield the same product.

two outputs.

2.2 The Niche of This Work

Stojanovic and Stefanovic [14] built a molecular automaton using the technology described in the previous section. The automaton plays a restricted form of the game of tic-tac-toe against a human opponent. Specifically, the automaton plays first (always in the center square) and the human opponent is limited (in his first move) to a specific corner or a specific side move.

It can be argued that this indeed captures the breadth of the tic-tac-toe experience. Many people argue that the second player really only has two distinct choices after the first player plays in the center field (namely, to play in a corner or on one of the sides). While this argument does have some merit as far as rotations of the game board are concerned, it hides the fact that at the time of the inception and implementation of MAYA, the implementation was limited by the fact that the technology did not support a more complex system of Boolean equations (as would presumably be required by a more complex strategy).

This thesis describes efforts to improve our understanding of the process of converting strategies for games into Boolean logic that is amenable to simulation by technology in the vein of that used for the construction of MAYA. At the time of the inception of MAYA, the strategy and the formulas implemented were compiled ad hoc. While this implementation works, it is clearly not a good starting point for a new branch of development. Therefore, this thesis concerns itself with games and strategies that could be used in an extension of MAYA.

Chapter 3

Games and Strategies

This chapter introduces the game theory concepts that will be used throughout this document.

3.1 Strategy Trees

A *sequential game* is a game in which players take turns making decisions known as *moves*. A *game of perfect information* is a sequential game in which all the players are informed before every move of the complete state of the game. A *strategy* for a player in a game of perfect information is a plan that dictates what moves that player will make in every possible game state¹[10]. We consider two-player, sequential games of perfect information exclusively.

A *strategy tree* is a (directed, acyclic) graph representation of a strategy. The nodes of the graph represent reachable game states. The edges of the graph represent the opponent's moves. The target node of the edge contains the strategy's response to the move encoded

¹For our purposes, a game state encodes the order of the players' moves.

on the edge. A leaf represents a final game state, and can, usually, be labelled either win, lose, or draw. Thus, a path from the root of a strategy tree to one of its leaves represents a game.

3.2 The Process of Converting Strategies into Boolean Formulas

Let us begin by describing the process of converting a very simple strategy into Boolean formulas.² Consider the strategy tree given in Figure 3.1.

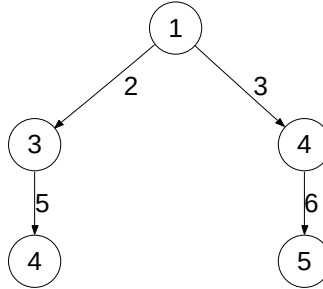


Figure 3.1: An example of a strategy tree. Each edge is labelled with the opponent's move. Each node is labelled with the strategy's move. The strategy moves first.

As in [14], we want to define Boolean functions that will represent an automaton's move selection. We want the function representing a field x to be true if and only if we intend the automaton implementing the formulas to claim field x . The inputs to the Boolean formulas will be the opponent's moves.

Once a gate has been activated, it cannot return to a state of inactivity. This property makes the paradigm we are using ideal for simulating games in which play selection is

²This section is partly an elaboration on a process described in the supplementary material to [14].

Chapter 3. Games and Strategies

permanent, i.e., once a move has been made, it cannot be undone.³

In a tree, there is only one path from the root of the tree to each node. This path defines sets of moves made by the players in the game. We will call these sets *move sets* for the players. A player's *move set* at any node is the set of moves made by that player up to that point in the game. Every path in the strategy tree will be represented in exactly one formula (the formula representing the strategy's play at the end of that path) by a conjunction of the elements of the opponent's move set at that node. From this description, we can construct the following formulas for the strategy in Figure 3.1:

$$o_1 = 1$$

$$o_3 = i_2$$

$$o_4 = i_3 \vee (i_2 \wedge i_5)$$

$$o_5 = i_3 \wedge i_6$$

When the game begins, all of the input variables are set to false (indicating that the opponent has not claimed any fields as his own), and the formula o_1 is true. So the strategy plays in field 1. At this point, the opponent may play in field 2 or in field 3. Let us say that the opponent wishes to play in field 2. This corresponds to setting the input variable i_2 to true. We then reevaluate our functions and find that o_1 and o_3 are now true. Recall that the formula o_1 was true before the opponent made his play. So after the opponent plays, there is exactly one output formula that newly evaluates to true. This corresponds to the strategy choosing a single field (which is a requirement of the desired functioning of the system). That field is o_3 . From this point, the opponent chooses to play in field 5 by setting i_5 to

³There are games in which this paradigm may not be applicable. For example, if the players control pieces that move from one field to another, a player may move into the same field more than once. Perhaps such a game could use a different encoding of moves rather than "move equals field."

Chapter 3. Games and Strategies

true. At that time, the formulas o_1 , o_3 , and o_4 evaluate to true. This is one more formula than before the opponent played. So the strategy plays in field 4. We would have gone through a similar scenario if the opponent had played in field 3 at his first opportunity.

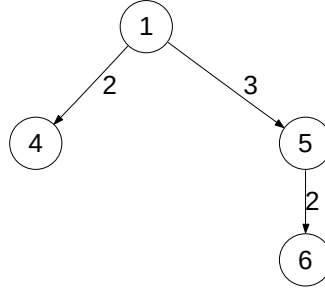


Figure 3.2: An example of a strategy tree. Each edge is labelled with the opponent's move. Each node is labelled with the strategy's move. The strategy moves first.

Up to this point, our conversion process holds up. If the opponent claims one field at a time, the strategy claims one field at a time. This is not always the case. Take the strategy in Figure 3.2 for instance. If we convert this strategy using the method described above, we will get the following formulas:

$$o_1 = 1$$

$$o_4 = i_2$$

$$o_5 = i_3$$

$$o_6 = i_2 \wedge i_3$$

When the game begins, all of the input variables are false. The formula o_1 evaluates to true. Let us say that the opponent chooses to play in field 2. This sets i_2 to true and thus, o_4 evaluates to true. On the other hand, let us say that the opponent plays in field 3. This makes o_5 true. When the opponent plays in field 2, things go awry. When i_3 is set to true,

o_1 and o_5 evaluate to true. When i_2 is set to true, o_1 , o_4 , o_5 , and o_6 evaluate to true. With the addition of one new input, two new output formulas become true. This means that after the opponent claimed a field, the strategy claimed two fields. Moreover, the strategy does not indicate that the strategy should claim field 4 after the opponent plays in field 3 and then field 2. Our formulas have not correctly represented the strategy.

The problem is that the disjunct i_2 in the formula for o_4 can evaluate to true not only after the first move of the opponent (the move from which the disjunct was derived) but also after the opponent's second move. This problem of more than one output function at a time becoming true (hereafter called a *conflict*) is not limited to this path through this strategy tree. In general, there will be such conflicts in other trees. For reasons to be explained shortly, this type of conflict is called a *resolvable conflict*.

3.2.1 Resolvable Conflict

From the preceding description it should be clear that every path in a strategy tree (originating at the root) corresponds to a disjunct in an output formula.⁴ A *resolvable conflict* occurs when the introduction of an input triggers two or more output function disjuncts that arose from strategy tree edges from different levels of the tree. A resolvable conflict can only be brought about by two conjunctions $x_1 \wedge x_2 \wedge \dots \wedge x_j$ and $y_1 \wedge y_2 \wedge \dots \wedge y_k$ (representing edges $e_{x_i} \dots e_{y_l}$ with $1 \leq i \leq j$ and $1 \leq l \leq k$) of the decision tree if certain conditions are met (without loss of generality, let us assume that $j \leq k$):

- $j \neq k$,
- $\{x_1, x_2, \dots, x_j\} \subset \{y_1, y_2, \dots, y_k\}$,
- the target of edge e_{x_j} encodes a different response from the target of edge e_{y_k} , and

⁴Note that this correspondence can be one-to-one or many-to-one.

Chapter 3. Games and Strategies

- the target of edge e_{x_j} encodes a response that is not encoded in any of the targets of edges e_{y_k} for any i with $1 \leq i \leq k - 1$.

The first condition ensures that the offending disjuncts arise from edges on two *different* levels of the decision tree. The second condition ensures that the gameplay path defined by the x_i 's will indeed activate the output associated with the y_i 's. The third condition ensures that the targets of the two edges in question are indeed different. The fourth condition ensures that the spurious output function has not already evaluated to true in the current gameplay path (this is an artifact of the chemical computation paradigm).

Resolvable conflicts arise only between disjuncts with different numbers of terms. The terms of the disjuncts are necessarily in a strict subset-superset relation. We can correct for a resolvable conflict by adding negative literals to the shorter disjunct. With this in mind, we can evolve our set of Boolean formulas to the following:

$$o_1 = 1$$

$$o_4 = i_2 \wedge \neg i_3$$

$$o_5 = i_3$$

$$o_6 = i_2 \wedge i_3$$

According to the original formulas, if the opponent plays in field 3 and then in field 2, the automaton will then respond to the second move by playing in both field 4 and field 6. After this change in the formulas, we note that after the opponent plays in field 3 and then in field 2, the automaton will then play only in field 4. The question arises as to what to do about conflicts between elements that lie on the *same* level of the tree.

3.2.2 Unresolvable Conflict

In the case of resolvable conflicts, we know to introduce negative literals to differentiate between disjuncts arising from different levels of the strategy tree. If the conflict is between two disjuncts on the same level of the decision tree, there is no strict subset-superset relationship to be exploited (indeed, the sets' sizes are necessarily equal). We call such a conflict an *unresolvable conflict* because there is no way to “fix” it using our method of strategy conversion. The fault is not corrigible in the Boolean formula representation of the strategy. It is an intrinsic fault of the strategy tree.

An unresolvable conflict is defined to occur between two disjuncts $x_1 \wedge x_2 \wedge \dots \wedge x_j$ and $y_1 \wedge y_2 \wedge \dots \wedge y_j$ (representing edges $e_{x_i} \dots e_{y_l}$ of the decision tree) under the following conditions:

- $\{x_1, x_2, \dots, x_j\} = \{y_1, y_2, \dots, y_j\}$,
- the target of edge e_{x_j} encodes a different response from the target of edge e_{y_j} and
- the target of edge e_{x_j} encodes a response that is not encoded in any of the targets of edges e_{y_i} for any i with $1 \leq i \leq j - 1$.

The first condition ensures that the offending disjuncts are set-wise equal (the sets of the opponent's moves on both paths are equal). The second condition ensures that the targets of the two edges in question are different. The third condition ensures that the spurious output function has not already evaluated to true in the current gameplay path. This third condition is more important than it may seem at first.

The strategy tree given in Figure 3.3 contains an unresolvable conflict. As before, we can define Boolean functions that correspond to the decision tree. The unresolvable conflict arises when the opponent plays in field 1 then in field 3. Upon the play in field 1,

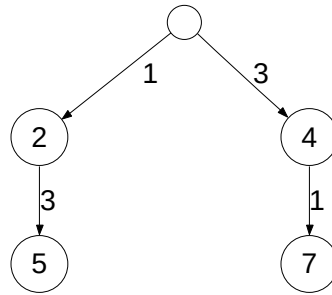


Figure 3.3: An example of a strategy tree with an unresolvable conflict.

the automaton using the formulas plays in field 2. After the following play by the opponent in field 3, the output functions o_5 and o_7 both (newly) evaluate to true.

Move Hiding

It may, after the previous discussion, be tempting to say that for any set of inputs corresponding to a set of the opponent's possible moves the strategy must encode a single response for all possible permutations of these moves. This is certainly a "safe" and correct way to design and verify strategies. But there are strategies that can be verified as correct that do not adhere to this convention. Consider the strategy tree given in Figure 3.4. The

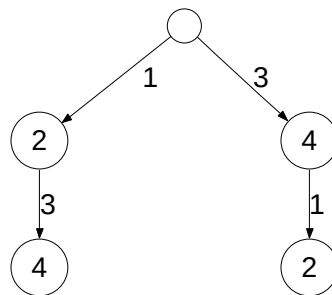


Figure 3.4: A strategy tree that demonstrates the idea of move hiding.

path given by the opponent's plays in field 1 then field 3 would conflict with the path given

by the opponent's plays in field 3 then field 1 if it were not for the third condition in the definition of unresolvable conflict. This leads us to our definition of *feasibility*.

3.2.3 Feasibility

A strategy is said to be *feasible* if, for every pair of nodes in the strategy tree for which the opponent's move sets are equal, one of the following two conditions holds:

- the nodes encode the same decision (i.e., they dictate the same move), or
- the strategy's move sets are equal.

The strategy tree in Figure 3.5 is a feasible strategy tree.⁵ In this strategy tree, the response associated with the opponent playing in field 2 then in field 3 is the same as the response associated with the opponent playing in field 3 then in field 2. This pair of responses conforms to our definition of feasibility because the response to the ordered set of moves (2, 3) is the same as the response to the ordered set of moves (3, 2). There exist two other pairs of nodes with equal move sets. In each case, the responses to both orderings of the moves is the same. The response to the ordered set of moves (2, 4) is the same as to (4, 2). Likewise, the response to (3, 4) is the same as the response to (4, 3).

3.3 Favorability

Upon completion, every game has an outcome that can be judged to be favorable or unfavorable for the player following the strategy. For instance, we might choose to consider favorable only final game states labelled “win”; or perhaps also those labelled “draw”. A strategy is said to be *favorable* if it admits only favorable outcomes for the player using it.

⁵It is also worth noting that the strategy trees in Figures 3.1 and 3.2 are also feasible.

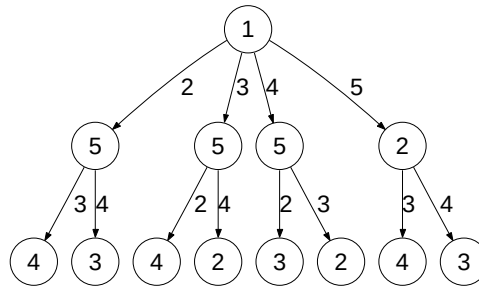


Figure 3.5: An example of a (slightly larger) feasible strategy tree.

Favorability is a concept that is uniquely defined by the rules of the game itself and by our demands on the automaton's play quality.

Thus, favorability is a largely arbitrary concept. This makes it difficult, usually, to prove properties of strategies that involve favorability, especially in conjunction with other, more structural concepts, such as feasibility.

Chapter 4

Boolean Formula Manipulation

4.1 Introduction

This work is directed at converting game strategies into Boolean functions and transforming those functions so that they conform to a certain (predefined) set of standards. In some ways, the goal of transforming the functions to conform to a set of standards can be seen as similar to the desired outcome of the process of Boolean minimization [8].¹ In both cases, we are dealing with large search spaces.

The similarities end there, though. Boolean minimization is a question of optimization. This work, on the other hand, is concerned with the question of whether there exists an equivalent representation of a function under a set of representational constraints.

¹In saying that we have minimized a Boolean function f , we claim that we have defined a function g that is equivalent to f and that every function h which has fewer literals than g can not be equivalent to f . More formally, we can restate this using quantifiers as follows:

$$\forall \vec{x}: f(\vec{x}) = g(\vec{x}) \wedge \forall h: (|h| \geq |g| \vee \exists \vec{x}: f(\vec{x}) \neq h(\vec{x}))$$

This problem is in the complexity class Π_2P . This question led to the definition of the polynomial hierarchy [9].

We say that the Boolean function g is equivalent to the Boolean function f if, for all input vectors $\vec{x}$ in the domain of f , we have $f(\vec{x}) = g(\vec{x})$. A Boolean function of n variables is defined by its truth table. For each of the 2^n input vectors, the function can be defined to be true (these vectors form the ON-set of the function) or false (these vectors form the OFF-set of the function). From this we can deduce that there are 2^{2^n} Boolean functions defined on n variables. Together, the ON-set and OFF-set comprise the CARE-set of a function. If, however, a function does not have a defined value for an input vector, then we say that we “do not care” about the output of the function when given that vector of inputs. These inputs form the DON’T CARE-set of the function.

4.2 Motivation for new tools

While the concept of Boolean minimization may be seen as akin to what we are attempting to do, there are some striking differences. First and foremost, we are interested in a simple decision problem. Namely, given a function f , we want to know if there exists a function g with the following properties:

1. the ON-set of g is a superset of the ON-set of f ,
2. the OFF-set of g does not intersect with the ON-set of f , and
3. g can be expressed as a disjunction of conjunctions that each fit a certain shape.

The first two properties are in line with the goals of the Boolean minimization process (i.e., to produce a function that correctly simulates the original function²). However, the third property is foreign. In fact, the third property could be described as the crux of the problem.

²The ON- and OFF-sets of f and g need not be equal since we do not really care about correctly representing the OFF-set of f . We are treating the OFF-set of f as part of the DON’T CARE-set.

The shapes to which the clauses must conform are quite limiting. We chose to develop a generalized technique for Boolean formula manipulation that takes the available clause shapes as a parameter. Also to the point, we are interested in functions with small numbers of variables. This leads us to the conclusion that this problem can be solved by a brute force algorithm.

4.3 Minimo

Minimo is our Boolean formula manipulator. It works by constructing a set of conjunctions and “fitting” them to the function we are looking to simulate. Let us begin by constructing the set of conjunctions S that can be simulated using current laboratory techniques. With S in hand, the methodology of **Minimo** can be described in pseudocode as follows:

```
for each  $s \in S$ 
    if  $s$  encompasses any truth table elements of  $f$  that are set to false
        then  $S = S - \{s\}$ 
for each vector of inputs in the ON-set of  $f$ 
    if none of the elements  $s \in S$  encompasses that vector
        end in failure
end in success.
```

At this point, if **Minimo** has ended in failure, we can be assured that there is no way to map the function f to a DNF function consisting only of conjunctions from S . If, however, **Minimo** ends in success, there exists a DNF function g , consisting only of conjunctions

Chapter 4. Boolean Formula Manipulation

from S , which simulates f correctly. Constructing g requires re-running the **Minimo** algorithm with a slight modification as follows:

```
for each  $s \in S$ 
    if  $s$  encompasses any truth table elements of  $f$  that are set to false
        then  $S = S - \{s\}$ 
for each vector of inputs  $v$  in the ON-set of  $f$ 
    if none of the elements  $s \in S$  encompasses  $v$ 
        end in failure
    otherwise, add (Boolean OR) the “shortest”  $s \in S$  that encompasses  $v$  to  $g$ 
end in success.
```

Minimo is implemented in such a way as to make it easy to change the types of clauses it will use to simulate the functions given as input. This is done using another (simple) program that builds “candidate disjunctions” to be used as building blocks for complete functions.

Finding the “shortest” $s \in S$ that encompasses a vector v is handled in the construction of the list of candidate disjunctions. So long as the candidates list is ordered so that candidates with fewer literals are listed before candidates with more literals, **Minimo** will always select the smallest candidate.

Chapter 5

Tic-Tac-Toe: A Case Study

5.1 Game Description

Tic-tac-toe is a two-player game played on a 3×3 grid of fields. The two players take turns claiming fields¹ and marking them with their mark. No player may claim a field that has already been claimed. A player wins the game (and the game ends) if that player has claimed three fields “in a row” (whether vertically, horizontally, or diagonally). The game ends in a draw if there is no winner after the players have claimed all nine fields. We define wins and draws to be favorable outcomes.

5.2 Strategy Tree Description

Tic-tac-toe is a game that ends after a maximum of five moves by the first player and a maximum of four moves by the second player. A strategy tree for the first player will have a maximum height of five (one edge for each of the opponent’s moves and one root node).

¹Numbered 0 through 8 left-to-right and top-to-bottom with 0 in the top-left field, 4 in the center field and 8 in the bottom-right field. See Figure 5.1.

0	1	2
3	4	5
6	7	8

Figure 5.1: A labelled board for the game of tic-tac-toe.

The strategy tree given in Figure 5.2 is a favorable strategy for the first player because all of the possible outcomes are either wins or draws. There is no way for the second player to win a game as long as the first player sticks to this strategy. This strategy also happens to be feasible. It is (admittedly) very hard to visually inspect Figure 5.2. Appendix A reproduces the `dot` file used to generate Figure 5.2.

5.3 The Number of Strategies

While there are a very limited number of games of tic-tac-toe (there are 255168 games [7]), the number of strategies is a bit larger. We will begin by describing a simple method to give an upper bound on the number of strategies, then we will describe a rather involved method for computing the exact number.

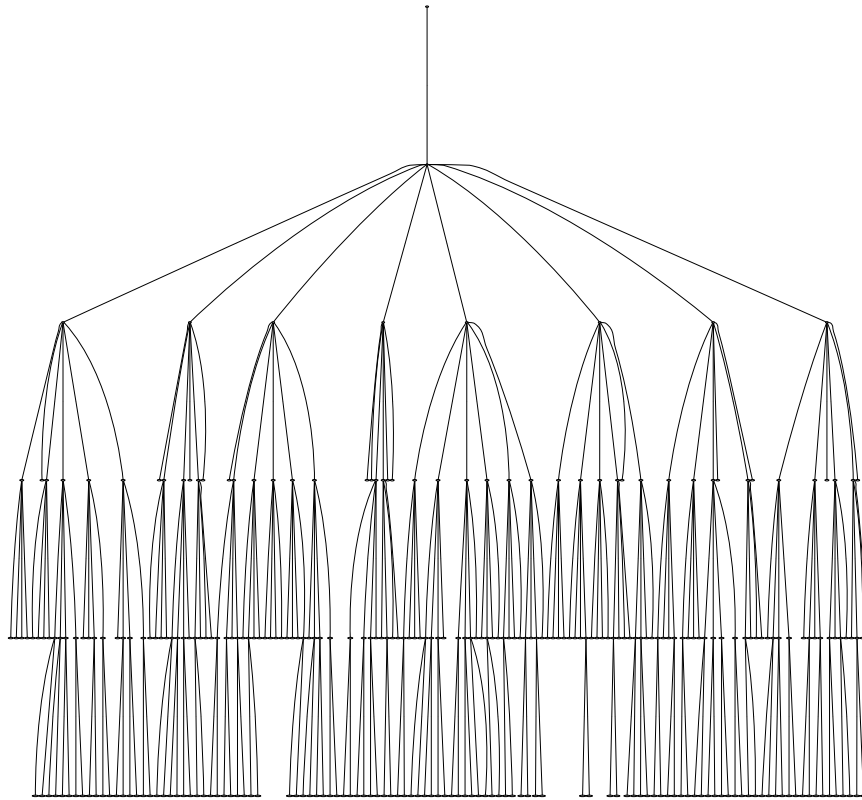


Figure 5.2: A strategy tree for the game of tic-tac-toe.

5.3.1 Upper and Lower Bounds: A Simple Overcount and A Simple Undercount

Let us consider a simplified game of tic-tac-toe in which play ends only when all the fields have been claimed.

The strategy may be required to make the first move in a game. Let us assume that we are designing strategies for the first player. The first player in a game of tic-tac-toe has nine choices for his first move.

Chapter 5. Tic-Tac-Toe: A Case Study

How many one-move ordered lists of the opponent's moves are there? There are eight. After the strategy dictates the first field claimed, the opponent has eight remaining choices for his first play. These eight choices correspond to edges in a strategy tree. Each of these edges requires a label on its target node. Each of these labels may take the form of any of the remaining seven fields.

Every path in the strategy tree corresponds to a game of tic-tac-toe played. With this in mind, it should be clear that the eight labels are independent. This concept will be repeated throughout this derivation. Up to this point in the growth of the tree, there are $9 \cdot 7^8$ possible distinct trees.

After the strategy's second move, there are six fields remaining. These six fields correspond to six edges in a decision tree. As before, the target nodes of these edges require labels. Each of these labels may take the form of any of the five remaining fields. Up to this point in the growth of the tree, there are $9 \cdot 7^8 \cdot 5^{6 \cdot 8}$ possible distinct trees.

After the strategy's third move, there are four fields remaining. These four fields correspond to the edges in a decision tree. The target nodes of these edges require labels. Each label may take the form of any of the remaining three fields. Up to this point in the growth of the tree, there are $9 \cdot 7^8 \cdot 5^{6 \cdot 8} \cdot 3^{4 \cdot 6 \cdot 8}$ possible distinct trees.

At the next level of the tree, the opponent has only two choices. The responses to both of these choices are determined by earlier moves (as there is only one field remaining).

Until we label the nodes of the decision tree, it is impossible to know the labels on the edges from said nodes. Even though this is the case, we know that the "shape" of a strategy is the same regardless of the labels assigned to the nodes.

- Every strategy will have 8 nodes one edge away from the root of the tree that require a label from a pool of 7 possible labels.
- Every strategy will have $8 \cdot 6 = 48$ nodes two edges away from the root of the tree

that require a label from a pool of 5 possible labels.

- Every strategy will have $8 \cdot 6 \cdot 4 = 192$ nodes three edges away from the root of the tree that require a label from a pool of 3 possible labels.
- Every strategy will have $8 \cdot 6 \cdot 4 \cdot 2 = 384$ nodes four edges away from the root of the tree that require a label from a pool of 1 possible label.

Each of the labels is independent of one another (insofar as the number of required and available labels is independent of the actual label placed anywhere in the tree). With this in mind, we can say that there are

$$9 \cdot 7^8 \cdot 5^{8 \cdot 6} \cdot 3^{8 \cdot 6 \cdot 4} \cdot 1^{8 \cdot 6 \cdot 4 \cdot 2} \approx 7.4622 \cdot 10^{132}$$

strategies for this simplified game of tic-tac-toe.² This is also an upper bound on the number of strategies for the actual game of tic-tac-toe.

Every game of tic-tac-toe must include at least three plays by the first player. So there are certainly at least

$$9 \cdot 7^8 \cdot 5^{6 \cdot 8} \approx 1.8433 \cdot 10^{41}$$

strategies for the game of tic-tac-toe. Thus, we have a lower and an upper bound, but they are very far apart.

5.3.2 The Exact Number of Strategies

In order to properly count the number of strategies, we will employ some extra definitions. A game state may be specified by two lists of players' moves up to that point in the game. Thus, a game state $([a_1, \dots, a_i], [h_1, \dots, h_j])$ is the state reached after the automaton has played into field a_1 , the human has played into h_1 , etc.; note that either $i = j$ or $i = j + 1$.

²This result is an exercise found in [10].

Chapter 5. Tic-Tac-Toe: A Case Study

Let us define a very special function f . The function f will map some game states to the natural numbers. Given a game state (x, y) , $f(x, y)$ will be the number of strategies for tic-tac-toe that are the same except for game states that are “extensions” of (x, y) . A game state $([a_1, \dots, a_j], [b_1, \dots, b_k])$ is an extension of another game state $([x_1, \dots, x_m], [y_1, \dots, y_n])$ if there exists a number p for which the following is true: for all i between 1 and p (inclusive), we have $a_i = x_i$ and $b_i = y_i$.

Using this new terminology, we are interested in computing $f([\], [\])$ (note that every conceivable correct game state is an extension of $([\], [\])$). There are a certain number of strategies that begin every game with a play in field one. This corresponds to $f([\], [1])$. Similarly, for every field x between 0 and 8, we can compute $f([\], [x])$. Every strategy must begin with an opening move. We can then draw the following conclusion:

$$f([\], [\]) = \sum_{i=0}^8 f([\], [i]).$$

Let us consider, for example, the computation of $f([\], [4])$. This is the number of strategies that begin by playing in field four (the center field). After the automaton plays in field four, the opponent has eight choices for his play. Each of these choices requires a response from the automaton (and the strategy). Since these responses are independent (as discussed before), we are interested in the product expressed by the following:

$$f([\], [4]) = \prod_{\substack{i=0 \\ i \notin \{4\}}}^8 f([i], [4]).$$

In computing $f([0], [4])$, we are again interested in a sum of choices for the automaton:

$$f([0], [4]) = \sum_{\substack{i=0 \\ i \notin \{0,4\}}}^9 f([0], [4, i]).$$

Using the same reasoning as before, we can deduce the following equation:

$$f([0], [4, 5]) = \prod_{\substack{i=0 \\ i \notin \{0,4,5\}}}^9 f([0, i], [4, 5]).$$

Chapter 5. Tic-Tac-Toe: A Case Study

We can continue with this process until we reach elements that are obvious such as the following:

$$\begin{aligned}f([0, 1], [4, 3, 5]) &= 1, \\f([0, 2], [4, 3, 5]) &= 1, \\f([0, 5, 7], [4, 3, 1, 2]) &= 1, \\f([0, 5, 7], [4, 3, 1, 6]) &= 1, \\f([0, 5, 7], [4, 3, 1, 8]) &= 1, \\f([0, 5, 7], [4, 3, 1]) &= 3.\end{aligned}$$

The process for computing $f([\], [\])$ seems very daunting and tedious at first blush. This process is a nice candidate for automation. We wrote a program that generated the requisite proof. The result is that

$$\begin{aligned}f([\], [\]) &= 4 \cdot f([\], [0]) + 4 \cdot f([\], [1]) + f([\], [4]) \\&\approx (1.90478 \cdot 10^{123} \cdot 4) + (7.45027 \cdot 10^{122} \cdot 4) + (3.6333 \cdot 10^{123}) \\&\approx 1.4233 \cdot 10^{124}.\end{aligned}$$

That is, there are $1.4233 \cdot 10^{124}$ strategies for the game of tic-tac-toe. Note that this is within eight orders of magnitude of the simple overcount proposed in section 5.3.1. Appendix C contains a full derivation of this value.

5.3.3 The Exact Number of Favorable Strategies

Consider a function g that counts only favorable strategies. The computation of $g([\], [\])$ is very similar to the computation of $f([\], [\])$ in the early stages. We can quite simply

Chapter 5. Tic-Tac-Toe: A Case Study

compute the following values of $g([], [])$:

$$\begin{aligned}g([1, 2], [4, 5, 6]) &= 1, \\g([1, 3], [4, 5, 6]) &= 1, \\g([1, 6, 8], [5, 4, 2]) &= 3, \\g([1, 2], [5, 4, 7]) &= 0, \\g([1, 3], [5, 6, 7]) &= 0.\end{aligned}$$

We wrote another program to produce a proof, i.e., derive a value of $g([], [])$. The result is:

$$\begin{aligned}g([], []) &= 4 \cdot g([], [0]) + 4 \cdot g([], [1]) + g([], [4]) \\&\approx (4.3689 \cdot 10^{95} \cdot 4) + (2.28863 \cdot 10^{86} \cdot 4) + (2.64833 \cdot 10^{103}) \\&\approx 2.64833 \cdot 10^{103}.\end{aligned}$$

Appendix D contains a full derivation of this value.

5.4 The Process of Strategy Generation

A strategy for the game of tic-tac-toe is not (in and of itself) very hard to construct. In order for a strategy to be valid it must have a valid response to the opponent's moves. These responses need not make common sense, though, in order for the strategy to be deemed "correct" or "valid".

The process of generating strategies for the game of tic-tac-toe that have certain properties differs depending on the desired properties. In the following sections, we will outline how to generate strategies that are favorable, feasible, and both feasible and favorable.

5.4.1 Generating Favorable Strategies

From Section 5.3 we gather that there are many strategies that are not favorable. Indeed, there are about 10^{21} unfavorable strategies for every favorable strategy. Randomly sampling the space of all strategies in order to find a favorable strategy would be very uneconomical. Instead, strategies should be generated so that they are favorable by construction.

Favorable strategies are simply strategies which do not admit a loss. Favorability does not really come into play in selecting the strategy's first and second moves. If the strategy's first three moves are not in a row, then it is possible that the opponent's moves may require a block after the opponent's second move. Clearly, we need a way to intelligently choose strategy responses to the opponent's second move. We need to devise a set of possible moves for the strategy's response to the opponent playing in field i and then in field j . This response is referred to as s_{ij} .

Let $N = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ be the set of fields. Let W be the set of sets of fields that form winning combinations.

$$W = \{\{0, 1, 2\}, \{3, 4, 5\}, \{6, 7, 8\}, \{0, 3, 6\}, \{1, 4, 7\}, \{2, 5, 8\}, \{0, 4, 8\}, \{2, 4, 6\}\}.$$

We will be using N and W throughout the remainder of this section. Let us create a set P of possible fields for s_{ij} that maintain the favorability of the strategy. The following algorithm computes P .

$$P = \{\}$$

if $\exists k$ such that $\{i, j, k\} \in W$ and $k \notin \{s_0, s_i\}$

then

$$P = P \cup \{k\}$$

if $\exists w$ such that $w \notin \{i, j\}$ and $\{w, s_0, s_i\} \in W$

Chapter 5. Tic-Tac-Toe: A Case Study

then $P = P \cup \{w\}$
else $P = N - \{s_0, i, s_i, j\}$.

After the strategy's fourth move, the strategy has no say in the outcome of the game. If the strategy is to be favorable, then the strategy's fourth move must prevent its opponent winning on his fourth move. The following algorithm must be followed when constructing a set of possible responses P to the opponent playing in fields i then j then k (assume that the strategy's first three plays in this game were in fields s_0 , s_i , and s_{ij}):

$P = \{\}$
if $\exists m \in N - \{s_0, i, s_i, j, s_{ij}, k\}$ such that $\exists x \in W$ with $x \subset \{i, j, k, m\}$
then
 $P = P \cup \{m\}$
if $\exists w$ such that $w \notin \{i, j, k\}$ and $\exists x \subset \{w, s_0, s_i, s_{ij}\}$ with $x \in W$
then
 $P = P \cup \{w\}$
else $P = N - \{s_0, i, s_i, j, s_{ij}, k\}$.

5.4.2 Generating Feasible Strategies

Consider the situation in which we are making an assignment of the strategy's response to the opponent playing in field i then in field j . The plays up to this point have been in fields s_0, i, s_i, j . Making an assignment in response to i then j forces the response to j then i (hereafter, s_{ij} and s_{ji} , respectively). If we do not need a value for s_{ji} , then s_{ij} can be taken to be any of the available fields without affecting the feasibility of the strategy.

Consider the case that we need an assignment to both s_{ij} and s_{ji} . Let us create two sets P_1 and P_2 of possible fields for s_{ij} . Selecting a member of the two P sets has different

Chapter 5. Tic-Tac-Toe: A Case Study

implications for the feasible assignments for s_{ji} . The fields in P_1 are those fields which could be assigned both to s_{ij} and s_{ji} : $P_1 = N - \{s_0, i, s_i, j, s_j\}$. In order for a strategy to be feasible, if s_{ij} is assigned a value $x \in P_1$, then s_{ji} must be assigned the value x (this explains why $s_j \notin P_1$). The fields in P_2 are those fields which demonstrate the concept of move hiding (see Section 3.2.2): $P_2 = \{s_j\} - \{s_0, i, j, s_i\}$. If P_2 is not empty, then we can assign s_j to s_{ij} so long as we also assign s_i to s_{ji} . We can see from this discussion that when assigning values to s_{ij} and s_{ji} we are either using move hiding or making the assignments equal. This idea will extend nicely to the next level of the strategy.

As before, let us construct P_1 and P_2 for a strategy response s_{ijk} to opponent's play in field i then j then k . There could potentially be five other strategy plays that are dependent upon this assignment. In constructing P_1 we need to make sure that all of the fields in P_1 are valid plays for each of the s_{abc} strategy responses that are necessary (where $\{a, b, c\} = \{i, j, k\}$). We can construct P_1 in the following way:

$$P_1 = N$$

$$\forall (a, b, c) \in \{(i, j, k), (i, k, j), (j, i, k), (j, k, i), (k, i, j), (k, j, i)\}$$

such that s_{abc} requires a value

$$P_1 = P_1 - \{s_0, a, s_a, b, s_{ab}\}.$$

If P_1 is nonempty after this process then each of the s_{abc} 's can take on any of the values in P_1 so long as they all agree. If, however, P_1 is empty, then we must attempt to find a move hiding solution for each of the s_{abc} 's. This will require us to perform the following construction of Q to determine if such a solution exists:

$$Q = \{\}$$

$$\forall (a, b, c) \in \{(i, j, k), (i, k, j), (j, i, k), (j, k, i), (k, i, j), (k, j, i)\}$$

Chapter 5. Tic-Tac-Toe: A Case Study

such that s_{abc} requires a value

$$Q = Q \cup \{s_0, s_a, s_{ab}\}$$

if $|Q| = 4$

then there exists a move hiding solution

else there does not exist a move hiding solution.

If there exists a move hiding solution, then we can use the set Q we created to find the value to assign to each of the s_{abc} 's (namely, that value which does not appear in $\{s_0, s_a, s_{ab}\}$).

Some Examples

Let $s = 4$, $s_0 = 2$, $s_1 = 3$. We need a value for s_{01} and s_{10} . If we follow the methods described above, we can build sets P_1 and P_2 . Doing so, we see that

$$P_1 = \{5, 6, 7, 8\}, \quad P_2 = \{3\}.$$

This means that we can assign any of the elements in P_1 to s_{01} so long as we also assign that value to s_{10} . Since P_2 is not null, we can assign s_1 to s_{01} so long as we also assign s_0 to s_{10} .

Let $s = 4$, $s_0 = 2$, $s_1 = 3$, $s_7 = 6$, $s_{01} = 5$, $s_{07} = 5$, $s_{10} = 5$, $s_{17} = 6$, $s_{70} = 5$, $s_{71} = 3$. We need values for s_{017} , s_{071} , s_{170} , s_{701} , and s_{710} . We can build the set P_1 using the method described above to see that $P_1 = \{8\}$. Since P_1 is not empty, we can apply this value to each of s_{017} , s_{071} , s_{170} , s_{701} , and s_{710} . We can build Q to see if there is a move hiding solution. Since $|Q| > 4$, there is no move hiding solution.

Let $s = 4$, $s_0 = 2$, $s_1 = 3$, $s_8 = 2$, $s_{01} = 5$, $s_{08} = 3$, $s_{80} = 3$, $s_{81} = 5$. We need values for s_{018} , s_{081} , s_{801} , and s_{810} . We can build the set P_1 using the method described above to see that $P_1 = \{6, 7\}$. But constructing $Q = \{2, 3, 4, 5\}$ informs us that there is a move hiding solution. Since $s = 4$, $s_0 = 2$, and $s_{01} = 5$, the correct value for s_{018} is 3.

5.4.3 Generating Feasible, Favorable Strategies

A strategy that is feasible and favorable can be constructed in a manner very similar to that used to construct a feasible strategy. The only change necessary is to check for the ability of the opponent to win before making any strategy choices. Generating P_1 , P_2 , and Q (from Section 5.4.2) proceeds as before. After generating the candidate move sets, we must check to make sure none of the candidate moves will allow the opponent to win.

5.5 Generating Boolean Formulas from Strategies

As described in Section 1.3, there is a predefined notion of what our Boolean functions are “supposed to do.” Each function will represent a field on a tic-tac-toe board. The inputs to the functions will be the opponent’s plays. Consider representing the strategy S as a set of Boolean formulas (with behavioral caveats as described in Section 1.2).

At the beginning of the game, the strategy must play in a field without waiting for the opponent to play. This means that there must be one function that is defined to be the constant true. This function will correspond to the field denoted by $s_0 \in S$.

Let us consider the situation in which the opponent has played in field i . We will construct a disjunction, D , and append it to a function. The disjunction will consist of nine variables (one for each of the fields). Eight of the variables will be negated. The ninth (representing i) will be positive. The response to this move is represented by $s_i \in S$. We will append (using a Boolean OR) D to the function representing s_i .

Let us consider the situation in which the opponent has played in field i then in field j (and requires a strategy response). As before, we will construct a disjunction, D , and append it to a function. The disjunction will consist of nine variables (one for each of the fields). Seven of the variables will be negated. The eighth and ninth (representing i and

Chapter 5. Tic-Tac-Toe: A Case Study

j) will be positive. The response to this move is represented by $s_{ij} \in S$. We will append (sum) D to the function representing s_{ij} .

After this fashion, we will append disjuncts for the strategy elements of the type s_{ijk} and s_{ijkm} .

Some Examples

The process of constructing Boolean formulas from a strategy appends a disjunct to a formula for each strategy element. For example, the strategy element $s_{01} = 5$ will append (Boolean OR)

$$i_0 \wedge i_1 \wedge \neg i_2 \wedge \neg i_3 \wedge \neg i_4 \wedge \neg i_5 \wedge \neg i_6 \wedge \neg i_7 \wedge \neg i_8$$

to the formula representing field 5. The strategy element $s_{0615} = 7$ will append (Boolean OR)

$$i_0 \wedge i_1 \wedge \neg i_2 \wedge \neg i_3 \wedge \neg i_4 \wedge i_5 \wedge i_6 \wedge \neg i_7 \wedge \neg i_8$$

to the formula representing field 7.

5.5.1 A Defense of Redundancy

It can be argued after a detailed reading of the previous examples that our generated Boolean formulas are “filled” with redundant information. Since we are passing these formulas through Minimo³, this injected redundancy has no deleterious effect on the final formulas.

³See Chapter 4.

5.5.2 Checking the Formulas

Checking a set of formulas for consistency (and, possibly, favorability) is a simple matter of simulating the “use” of the formulas. We will attempt to construct a strategy tree from the formulas. This construction will be breadth-first. The first level of the tree should be a single node. There should be one function (representing $s_0 \in S$) that evaluates to true when all of the inputs are set to false.

For each of the eight possible first moves by the opponent, i , we must evaluate all nine of the functions with all the input variables set to false except for the variable representing field i . For each set of inputs, one or two functions should evaluate to true. One of those two functions must represent $s_i \in S$ while the other represents $s_0 \in S$.

For each of the possible two move combinations from the opponent, in field i then j , we must evaluate all nine of the functions with all the input variables set to false except for the variables representing fields i and j . For each set of inputs, up to three functions should evaluate to true representing $s_0, s_i, s_{ij} \in S$. At the least, the function representing $s_{ij} \in S$ must evaluate to true. Completing the construction requires checking three- and four-move combinations from the opponent after this fashion.

5.6 Our Tools, Their Locations, and Their Uses

All of the tools described in this thesis may be found on the author’s home page⁴, as well as on the CD-ROM that contains the complete thesis.

⁴<http://www.cs.unm.edu/~bandrews/>

5.6.1 File Formats

Strategy Tree Output Formats

A strategy tree for the game of tic-tac-toe can be represented in several ways: graphically, as a set of Boolean formulas (feasible strategies), and as an input to Minimo. Our strategy generation software can print strategies for any of these representational styles. Specifically, the tool can generate a dot file that can be used to produce a graphical representation of the strategy. When used to produce a set of Boolean formulas, the output is designed to fit into our checker program. This output builds C code that represents the formulas as variables being assigned values according to inputs in a DNF formula.

Minimo Input File

A Minimo strategy input file contains some number of strategies expressed as Boolean formulas (with a blank line separating two strategies). Every line in a strategy must contain the same number of characters. Every line begins with some number of zeroes and ones representing an input vector. The input vector is followed by a single space and an output vector. An output vector consists of the characters zero, one, and hyphen (representing that an input vector is in the DON'T CARE-set of a particular function).

A Minimo candidates input file contains some number of lines, each representing a possible disjunct to be considered for covering a function. Every line must contain the same number of characters. Every line is composed of some number of zeroes, ones, and hyphens. A hyphen as the i th character in a line indicates that the i th input does not appear in that conjunction.

Chapter 6

Conclusions

This thesis saw developments in fields both applied and abstract. We found that we could use a very simple procedure to convert strategies for games into Boolean formulas. We developed a method to convert disjunctive-normal form Boolean formulas into equivalent formulas with predefined disjunct shapes. We developed a counting method for strategies for the game of tic-tac-toe. We constructed feasible, favorable strategies for the game of tic-tac-toe.

6.1 Strategy Conversions

This research has demonstrated a method to convert strategies for a certain type of game into Boolean formulas. When this research began, we were in possession of a simple example of a strategy implemented in MAYA. At the conclusion of this research, we can now describe the mechanisms behind the correct functioning of MAYA, and (more importantly) a general method for generating Boolean formulas from a game strategy.

By generating and converting strategies for the game of tic-tac-toe into Boolean formu-

las, we have demonstrated that our simple method of strategy conversion is robust enough to handle somewhat complex strategies.

6.2 Boolean Manipulation

We have developed tools that are useful in determining whether the CARE-set of a Boolean function admits a form of a given shape. For instances of games in which the number of potential moves by the opponent is rather small, we can enumerate all the possible disjuncts with those inputs. With this list of all disjuncts, we can try to fit them to the function.

6.3 Tic-Tac-Toe Results

We have developed C programs that can generate strategies for the game of tic-tac-toe that are feasible, favorable, both feasible and favorable, and neither feasible nor favorable. At the outset of this research we were not sure whether the set of all feasible, favorable strategies for the game of tic-tac-toe was empty or not. We now know that there exist strategies that are both feasible and favorable. In performing this research we developed a method to count the number of strategies for the game of tic-tac-toe. This counting method was applied, almost trivially, to counting the number of favorable strategies.

At the outset of this research, the goal was to produce a feasible, favorable strategy for the game of tic-tac-toe and convert this strategy into Boolean formulas that can then be simulated in chemistry. To this end, we used a technology loosely designed around the original MAYA chemistry. With MAYA, conjunctions could be simulated with up to two positive and one negative input. We tested a technology that could be described as “3-anded MAYA”. We tried to map our formulas to conjunctions with at most six posi-

Chapter 6. Conclusions

tive inputs and three negative inputs. We tested this technology against over fifty million strategies to no avail. None of the generated formulas could be manipulated into a form that would fit with our technological limitation.

We also tested against a technology with no real basis in the demonstrated chemistry. After working with the formulas for a while, we hypothesized that it might be possible to convert formulas to formulas with conjunctions of (at most) five literals. Our hypothesis was confirmed almost immediately as one of the first one hundred thousand strategies we tested proved to be mappable to a set of formulas with conjunctions of five or fewer literals.¹ Similar tests with a technology including only four literals were not successful. We conjecture that to represent the full game of tic-tac-toe, it is necessary to have conjunctions of five inputs of arbitrary polarity.

¹An example of such a strategy may be found in Appendix A.

Chapter 7

Future Work

The research which produced this thesis can easily be continued in many different avenues.

7.1 Tic-Tac-Toe

While there are other versions of tic-tac-toe to be explored, there are still questions about the original game that remain unanswered. Among them: how many feasible strategies are there? Attempts were made at answering this question during the development of this thesis. None were fruitful enough to merit consideration.

We did, however, find some strategies that map to relatively simple technologies. This makes us wonder as to what technologies are sufficient to represent a strategy for the game of tic-tac-toe (or any other game for that matter).

7.2 Other Games

There are several games whose strategies may be converted to Boolean formulas using a method similar to that described in this thesis. Among them are Connect Four and Othello.

7.2.1 Connect Four

Connect Four is a game in which players permanently claim fields as their own by dropping discs into vertical columns. Traditionally the game is played with seven columns (each with a height of five fields). The vertical nature of the columns implies that no player may claim a field until the field below it has been claimed (with the exception of the fields at the bottoms of the columns). Connect Four would be an interesting application for our strategy conversion method.

7.2.2 Othello

Othello is a game in which players select fields but ownership of fields is not permanent. The ability of the results of this research to map to the game of Othello is very closely tied to the amount of “realism” one wants from the performance of the Boolean formulas.

As the work stands, the claiming of fields is permanent while the claim is not staked throughout the game. A perfect simulation of the game of Othello (with field ownership changes) is well outside the scope of the current abilities of this technology.

Appendix A

A Strategy

The following dot file was used to prepare Figure 5.2. Every game state node is labelled with the move sets by the two players. Every edge between game state nodes is labelled with a move, response pair.

```
digraph foo
{
    size="5,5"
    ratio=fill;

    node [shape=ellipse];
    node [fontname="Helvetica"];
    node [fontsize=30];

    n0 [label="START"]
    n1 [label=" / 4"]
    n2 [label="0 / 4 8"]
    n3 [label="1 / 4 8"]
    n4 [label="2 / 4 8"]
    n5 [label="3 / 4 8"]
    n6 [label="5 / 4 8"]
```

```
    n7 [label="6 / 4 3"]
    n8 [label="7 / 4 8"]
    n9 [label="8 / 4 6"]
    n10 [label="0 1 / 4 8 2"]
    n11 [label="0 2 / 4 8 1"]
    n12 [label="0 3 / 4 8 6"]
    n13 [label="0 5 / 4 8 1"]
    n14 [label="0 6 / 4 8 3"]
    n15 [label="0 7 / 4 8 5"]
    n16 [label="1 0 / 4 8 2"]
    n17 [label="1 5 / 4 8 2"]
    n18 [label="1 6 / 4 8 3"]
    n19 [label="1 7 / 4 8 3"]
    n20 [label="2 0 / 4 8 1"]
    n21 [label="2 5 / 4 8 1"]
    n22 [label="2 6 / 4 8 3"]
    n23 [label="2 7 / 4 8 1"]
    n24 [label="3 0 / 4 8 6"]
    n25 [label="5 0 / 4 8 1"]
    n26 [label="5 1 / 4 8 2"]
    n27 [label="5 2 / 4 8 1"]
    n28 [label="5 6 / 4 8 3"]
    n29 [label="5 7 / 4 8 2"]
    n30 [label="6 0 / 4 3 8"]
    n31 [label="6 1 / 4 3 8"]
    n32 [label="6 2 / 4 3 8"]
    n33 [label="6 5 / 4 3 8"]
    n34 [label="6 7 / 4 3 8"]
    n35 [label="6 8 / 4 3 7"]
    n36 [label="7 0 / 4 8 5"]
    n37 [label="7 1 / 4 8 3"]
    n38 [label="7 2 / 4 8 1"]
    n39 [label="7 5 / 4 8 2"]
    n40 [label="7 6 / 4 8 3"]
    n41 [label="8 1 / 4 6 5"]
    n42 [label="8 2 / 4 6 5"]
    n43 [label="8 3 / 4 6 1"]
    n44 [label="8 7 / 4 6 0"]
    n45 [label="0 1 5/ 4 8 2 3"]
    n46 [label="0 2 3/ 4 8 1 5"]
    n47 [label="0 2 5/ 4 8 1 3"]
    n48 [label="0 2 6/ 4 8 1 3"]
    n49 [label="0 2 7/ 4 8 1 5"]
```

Appendix A. A Strategy

```
n50 [label="0 3 2/ 4 8 6 5"]
n51 [label="0 3 7/ 4 8 6 1"]
n52 [label="0 5 2/ 4 8 1 3"]
n53 [label="0 5 3/ 4 8 1 2"]
n54 [label="0 5 6/ 4 8 1 3"]
n55 [label="0 5 7/ 4 8 1 3"]
n56 [label="0 6 2/ 4 8 3 1"]
n57 [label="0 6 5/ 4 8 3 1"]
n58 [label="0 7 1/ 4 8 5 6"]
n59 [label="0 7 2/ 4 8 5 1"]
n60 [label="0 7 3/ 4 8 5 1"]
n61 [label="1 0 5/ 4 8 2 3"]
n62 [label="1 5 0/ 4 8 2 3"]
n63 [label="1 5 7/ 4 8 2 3"]
n64 [label="1 7 0/ 4 8 3 6"]
n65 [label="1 7 5/ 4 8 3 2"]
n66 [label="2 0 3/ 4 8 1 5"]
n67 [label="2 0 5/ 4 8 1 3"]
n68 [label="2 0 6/ 4 8 1 3"]
n69 [label="2 0 7/ 4 8 1 5"]
n70 [label="2 5 0/ 4 8 1 3"]
n71 [label="2 6 0/ 4 8 3 1"]
n72 [label="2 7 0/ 4 8 1 5"]
n73 [label="3 0 2/ 4 8 6 5"]
n74 [label="3 0 7/ 4 8 6 1"]
n75 [label="5 0 2/ 4 8 1 3"]
n76 [label="5 0 3/ 4 8 1 2"]
n77 [label="5 0 6/ 4 8 1 3"]
n78 [label="5 0 7/ 4 8 1 3"]
n79 [label="5 1 0/ 4 8 2 3"]
n80 [label="5 1 7/ 4 8 2 3"]
n81 [label="5 2 0/ 4 8 1 3"]
n82 [label="5 6 0/ 4 8 3 1"]
n83 [label="5 6 7/ 4 8 3 2"]
n84 [label="5 7 0/ 4 8 2 3"]
n85 [label="5 7 1/ 4 8 2 3"]
n86 [label="5 7 6/ 4 8 2 3"]
n87 [label="6 0 2/ 4 3 8 1"]
n88 [label="6 0 5/ 4 3 8 1"]
n89 [label="6 2 0/ 4 3 8 1"]
n90 [label="6 5 0/ 4 3 8 1"]
n91 [label="6 5 7/ 4 3 8 2"]
n92 [label="6 7 5/ 4 3 8 2"]
n93 [label="6 8 1/ 4 3 7 0"]
n94 [label="6 8 2/ 4 3 7 0"]
n95 [label="6 8 5/ 4 3 7 0"]
n96 [label="7 0 1/ 4 8 5 6"]
n97 [label="7 0 2/ 4 8 5 1"]
n98 [label="7 0 3/ 4 8 5 1"]
n99 [label="7 1 0/ 4 8 3 6"]
n100 [label="7 1 5/ 4 8 3 2"]
n101 [label="7 2 0/ 4 8 1 5"]
n102 [label="7 5 0/ 4 8 2 3"]
n103 [label="7 5 1/ 4 8 2 3"]
n104 [label="7 5 6/ 4 8 2 3"]
n105 [label="7 6 5/ 4 8 3 2"]
n106 [label="8 1 3/ 4 6 5 0"]
n107 [label="8 2 0/ 4 6 5 1"]
n108 [label="8 2 3/ 4 6 5 0"]
n109 [label="8 2 7/ 4 6 5 1"]
n110 [label="8 3 2/ 4 6 1 0"]
n111 [label="8 7 2/ 4 6 0 1"]
n112 [label="WIN"]
n113 [label="DRAW"]
n114 [label="LOSE"]
    edge [fontname="Helvetica-Bold"];
    edge [fontsize=30];
    edge [style=bold];
n0 -> n1 [label="1/4"];
n1 -> n2 [label="0/8"];
n2 -> n10 [label="1/2"];
n10 -> n112 [label="3/6"];
n10 -> n45 [label="5/3"];
n45 -> n113 [label="6/7"];
n45 -> n112 [label="7/6"];
n10 -> n112 [label="6/5"];
n10 -> n112 [label="7/6"];
n2 -> n11 [label="2/1"];
n11 -> n46 [label="3/5"];
n46 -> n112 [label="6/7"];
n46 -> n113 [label="7/6"];
n11 -> n47 [label="5/3"];
n47 -> n112 [label="6/7"];
n47 -> n113 [label="7/6"];
n11 -> n48 [label="6/3"];
n48 -> n112 [label="5/7"];
```

Appendix A. A Strategy

```
n48 -> n112 [label="7/5"];
n11 -> n49 [label="7/5"];
n49 -> n113 [label="3/6"];
n49 -> n112 [label="6/3"];
n2 -> n12 [label="3/6"];
n12 -> n112 [label="1/2"];
n12 -> n50 [label="2/5"];
n50 -> n112 [label="1/7"];
n50 -> n113 [label="7/1"];
n12 -> n112 [label="5/2"];
n12 -> n51 [label="7/1"];
n51 -> n113 [label="2/5"];
n51 -> n112 [label="5/2"];
n2 -> n13 [label="5/1"];
n13 -> n52 [label="2/3"];
n52 -> n112 [label="6/7"];
n52 -> n113 [label="7/6"];
n13 -> n53 [label="3/2"];
n53 -> n112 [label="6/7"];
n53 -> n112 [label="7/6"];
n13 -> n54 [label="6/3"];
n54 -> n112 [label="2/7"];
n54 -> n113 [label="7/2"];
n13 -> n55 [label="7/3"];
n55 -> n113 [label="2/6"];
n55 -> n113 [label="6/2"];
n2 -> n14 [label="6/3"];
n14 -> n112 [label="1/5"];
n14 -> n56 [label="2/1"];
n56 -> n112 [label="5/7"];
n56 -> n112 [label="7/5"];
n14 -> n57 [label="5/1"];
n57 -> n112 [label="2/7"];
n57 -> n113 [label="7/2"];
n14 -> n112 [label="7/5"];
n2 -> n15 [label="7/5"];
n15 -> n58 [label="1/6"];
n58 -> n112 [label="2/3"];
n58 -> n112 [label="3/2"];
n15 -> n59 [label="2/1"];
n59 -> n113 [label="3/6"];
n59 -> n112 [label="6/3"];
n15 -> n60 [label="3/1"];
n60 -> n113 [label="2/6"];
n60 -> n112 [label="6/2"];
n15 -> n112 [label="6/3"];
n1 -> n3 [label="1/8"];
n3 -> n16 [label="0/2"];
n16 -> n112 [label="3/6"];
n16 -> n61 [label="5/3"];
n61 -> n113 [label="6/7"];
n61 -> n112 [label="7/6"];
n16 -> n112 [label="6/5"];
n16 -> n112 [label="7/6"];
n3 -> n112 [label="2/0"];
n3 -> n112 [label="3/0"];
n3 -> n17 [label="5/2"];
n17 -> n62 [label="0/3"];
n62 -> n113 [label="6/7"];
n62 -> n112 [label="7/6"];
n17 -> n112 [label="3/0"];
n17 -> n112 [label="6/0"];
n17 -> n63 [label="7/3"];
n63 -> n112 [label="0/6"];
n63 -> n112 [label="6/0"];
n3 -> n18 [label="6/3"];
n18 -> n112 [label="0/5"];
n18 -> n112 [label="2/0"];
n18 -> n112 [label="5/0"];
n18 -> n112 [label="7/0"];
n3 -> n19 [label="7/3"];
n19 -> n64 [label="0/6"];
n64 -> n112 [label="2/5"];
n64 -> n112 [label="5/2"];
n19 -> n112 [label="2/0"];
n19 -> n65 [label="5/2"];
n65 -> n112 [label="0/6"];
n65 -> n112 [label="6/0"];
n19 -> n112 [label="6/0"];
n1 -> n4 [label="2/8"];
n4 -> n20 [label="0/1"];
n20 -> n66 [label="3/5"];
n66 -> n112 [label="6/7"];
n66 -> n113 [label="7/6"];
n20 -> n67 [label="5/3"];
n67 -> n112 [label="6/7"];
```

Appendix A. A Strategy

n67 -> n113 [label="7/6"];
n20 -> n68 [label="6/3"];
n68 -> n112 [label="5/7"];
n68 -> n112 [label="7/5"];
n20 -> n69 [label="7/5"];
n69 -> n113 [label="3/6"];
n69 -> n112 [label="6/3"];
n4 -> n112 [label="1/0"];
n4 -> n112 [label="3/0"];
n4 -> n21 [label="5/1"];
n21 -> n70 [label="0/3"];
n70 -> n112 [label="6/7"];
n70 -> n113 [label="7/6"];
n21 -> n112 [label="3/0"];
n21 -> n112 [label="6/0"];
n21 -> n112 [label="7/0"];
n4 -> n22 [label="6/3"];
n22 -> n71 [label="0/1"];
n71 -> n112 [label="5/7"];
n71 -> n112 [label="7/5"];
n22 -> n112 [label="1/0"];
n22 -> n112 [label="5/0"];
n22 -> n112 [label="7/0"];
n4 -> n23 [label="7/1"];
n23 -> n72 [label="0/5"];
n72 -> n113 [label="3/6"];
n72 -> n112 [label="6/3"];
n23 -> n112 [label="3/0"];
n23 -> n112 [label="5/0"];
n23 -> n112 [label="6/0"];
n1 -> n5 [label="3/8"];
n5 -> n24 [label="0/6"];
n24 -> n112 [label="1/2"];
n24 -> n73 [label="2/5"];
n73 -> n112 [label="1/7"];
n73 -> n113 [label="7/1"];
n24 -> n112 [label="5/2"];
n24 -> n74 [label="7/1"];
n74 -> n113 [label="2/5"];
n74 -> n112 [label="5/2"];
n5 -> n112 [label="1/0"];
n5 -> n112 [label="2/0"];
n5 -> n112 [label="5/0"];

n5 -> n112 [label="6/0"];
n5 -> n112 [label="7/0"];
n1 -> n6 [label="5/8"];
n6 -> n25 [label="0/1"];
n25 -> n75 [label="2/3"];
n75 -> n112 [label="6/7"];
n75 -> n113 [label="7/6"];
n25 -> n76 [label="3/2"];
n76 -> n112 [label="6/7"];
n76 -> n112 [label="7/6"];
n25 -> n77 [label="6/3"];
n77 -> n112 [label="2/7"];
n77 -> n113 [label="7/2"];
n25 -> n78 [label="7/3"];
n78 -> n113 [label="2/6"];
n78 -> n113 [label="6/2"];
n6 -> n26 [label="1/2"];
n26 -> n79 [label="0/3"];
n79 -> n113 [label="6/7"];
n79 -> n112 [label="7/6"];
n26 -> n112 [label="3/0"];
n26 -> n112 [label="6/0"];
n26 -> n80 [label="7/3"];
n80 -> n112 [label="0/6"];
n80 -> n112 [label="6/0"];
n6 -> n27 [label="2/1"];
n27 -> n81 [label="0/3"];
n81 -> n112 [label="6/7"];
n81 -> n113 [label="7/6"];
n27 -> n112 [label="3/0"];
n27 -> n112 [label="6/0"];
n27 -> n112 [label="7/0"];
n6 -> n112 [label="3/0"];
n6 -> n28 [label="6/3"];
n28 -> n82 [label="0/1"];
n82 -> n112 [label="2/7"];
n82 -> n113 [label="7/2"];
n28 -> n112 [label="1/0"];
n28 -> n112 [label="2/0"];
n28 -> n83 [label="7/2"];
n83 -> n113 [label="0/1"];
n83 -> n112 [label="1/0"];
n6 -> n29 [label="7/2"];

Appendix A. A Strategy

```
n29 -> n84 [label="0/3"];
n84 -> n112 [label="1/6"];
n84 -> n113 [label="6/1"];
n29 -> n85 [label="1/3"];
n85 -> n112 [label="0/6"];
n85 -> n112 [label="6/0"];
n29 -> n112 [label="3/0"];
n29 -> n86 [label="6/3"];
n86 -> n113 [label="0/1"];
n86 -> n112 [label="1/0"];
n1 -> n7 [label="6/3"];
n7 -> n30 [label="0/8"];
n30 -> n112 [label="1/5"];
n30 -> n87 [label="2/1"];
n87 -> n112 [label="5/7"];
n87 -> n112 [label="7/5"];
n30 -> n88 [label="5/1"];
n88 -> n112 [label="2/7"];
n88 -> n113 [label="7/2"];
n30 -> n112 [label="7/5"];
n7 -> n31 [label="1/8"];
n31 -> n112 [label="0/5"];
n31 -> n112 [label="2/0"];
n31 -> n112 [label="5/0"];
n31 -> n112 [label="7/0"];
n7 -> n32 [label="2/8"];
n32 -> n89 [label="0/1"];
n89 -> n112 [label="5/7"];
n89 -> n112 [label="7/5"];
n32 -> n112 [label="1/0"];
n32 -> n112 [label="5/0"];
n32 -> n112 [label="7/0"];
n7 -> n33 [label="5/8"];
n33 -> n90 [label="0/1"];
n90 -> n112 [label="2/7"];
n90 -> n113 [label="7/2"];
n33 -> n112 [label="1/0"];
n33 -> n112 [label="2/0"];
n33 -> n91 [label="7/2"];
n91 -> n113 [label="0/1"];
n91 -> n112 [label="1/0"];
n7 -> n34 [label="7/8"];
n34 -> n112 [label="0/5"];
n34 -> n112 [label="1/0"];
n34 -> n112 [label="2/0"];
n34 -> n92 [label="5/2"];
n92 -> n113 [label="0/1"];
n92 -> n112 [label="1/0"];
n7 -> n35 [label="8/7"];
n35 -> n112 [label="0/1"];
n35 -> n93 [label="1/0"];
n93 -> n112 [label="2/5"];
n93 -> n113 [label="5/2"];
n35 -> n94 [label="2/0"];
n94 -> n112 [label="1/5"];
n94 -> n112 [label="5/1"];
n35 -> n95 [label="5/0"];
n95 -> n113 [label="1/2"];
n95 -> n112 [label="2/1"];
n1 -> n8 [label="7/8"];
n8 -> n36 [label="0/5"];
n36 -> n96 [label="1/6"];
n96 -> n112 [label="2/3"];
n96 -> n112 [label="3/2"];
n36 -> n97 [label="2/1"];
n97 -> n113 [label="3/6"];
n97 -> n112 [label="6/3"];
n36 -> n98 [label="3/1"];
n98 -> n113 [label="2/6"];
n98 -> n112 [label="6/2"];
n36 -> n112 [label="6/3"];
n8 -> n37 [label="1/3"];
n37 -> n99 [label="0/6"];
n99 -> n112 [label="2/5"];
n99 -> n112 [label="5/2"];
n37 -> n112 [label="2/0"];
n37 -> n100 [label="5/2"];
n100 -> n112 [label="0/6"];
n100 -> n112 [label="6/0"];
n37 -> n112 [label="6/0"];
n8 -> n38 [label="2/1"];
n38 -> n101 [label="0/5"];
n101 -> n113 [label="3/6"];
n101 -> n112 [label="6/3"];
n38 -> n112 [label="3/0"];
n38 -> n112 [label="5/0"];
```

Appendix A. A Strategy

```
n38 -> n112 [label="6/0"];
n8 -> n112 [label="3/0"];
n8 -> n39 [label="5/2"];
n39 -> n102 [label="0/3"];
n102 -> n112 [label="1/6"];
n102 -> n113 [label="6/1"];
n39 -> n103 [label="1/3"];
n103 -> n112 [label="0/6"];
n103 -> n112 [label="6/0"];
n39 -> n112 [label="3/0"];
n39 -> n104 [label="6/3"];
n104 -> n113 [label="0/1"];
n104 -> n112 [label="1/0"];
n8 -> n40 [label="6/3"];
n40 -> n112 [label="0/5"];
n40 -> n112 [label="1/0"];
n40 -> n112 [label="2/0"];
n40 -> n105 [label="5/2"];
n105 -> n113 [label="0/1"];
n105 -> n112 [label="1/0"];
n1 -> n9 [label="8/6"];
n9 -> n112 [label="0/2"];
n9 -> n41 [label="1/5"];
n41 -> n112 [label="0/2"];
n41 -> n112 [label="2/3"];
n41 -> n106 [label="3/0"];
n106 -> n113 [label="2/7"];
n106 -> n112 [label="7/2"];
n41 -> n112 [label="7/2"];
n9 -> n42 [label="2/5"];
n42 -> n107 [label="0/1"];
n107 -> n112 [label="3/7"];
n107 -> n112 [label="7/3"];
n42 -> n112 [label="1/3"];
n42 -> n108 [label="3/0"];
n108 -> n113 [label="1/7"];
n108 -> n113 [label="7/1"];
n42 -> n109 [label="7/1"];
n109 -> n112 [label="0/3"];
n109 -> n113 [label="3/0"];
n9 -> n43 [label="3/1"];
n43 -> n112 [label="0/2"];
n43 -> n110 [label="2/0"];

n110 -> n112 [label="5/7"];
n110 -> n113 [label="7/5"];
n43 -> n112 [label="5/2"];
n43 -> n112 [label="7/2"];
n9 -> n112 [label="5/2"];
n9 -> n44 [label="7/0"];
n44 -> n112 [label="1/2"];
n44 -> n111 [label="2/1"];
n111 -> n113 [label="3/5"];
n111 -> n112 [label="5/3"];
n44 -> n112 [label="3/2"];
n44 -> n112 [label="5/2"];
{
rank = same;
n10;
n11;
n12;
n13;
n14;
n15;
n16;
n17;
n18;
n19;
n20;
n21;
n22;
n23;
n24;
n25;
n26;
n27;
n28;
n29;
n30;
n31;
n32;
n33;
n34;
n35;
n36;
n37;
n38;
```

Appendix A. A Strategy

```
n39;
n40;
n41;
n42;
n43;
n44;
}
{ rank = same;
n45;
n46;
n47;
n48;
n49;
n50;
n51;
n52;
n53;
n54;
n55;
n56;
n57;
n58;
n59;
n60;
n61;
n62;
n63;
n64;
n65;
n66;
n67;
n68;
n69;
n70;
n71;
n72;
n73;
n74;
n75;
n76;
n77;
n78;
n79;

n80;
n81;
n82;
n83;
n84;
n85;
n86;
n87;
n88;
n89;
n90;
n91;
n92;
n93;
n94;
n95;
n96;
n97;
n98;
n99;
n100;
n101;
n102;
n103;
n104;
n105;
n106;
n107;
n108;
n109;
n110;
n111;
}
{ rank = same;
n112;
n114;
n113;
}
}
/*loser = 0*/
```

Appendix A. A Strategy

This strategy is special. It can be converted into Boolean formulas (using our conversion method) that can then be “simplified” or “manipulated” into formulas that have conjunctions with no more than five literals. The result of this conversion process (expressed in PLA form) is the following set of formulas.

```

100000000 0000-0001
110000000 0010-000-
110100000 00-0-010-
110001000 00-1-000-
110001100 00--001-
110001010 00--010-
110000100 00-0-100-
110000010 00-0-010-
101000000 0100-000-
101100000 0-00-100-
101100100 0-00--01-
101100010 0-00--10-
101001000 0-01-000-
101001100 0-0--001-
101001010 0-0--010-
101000100 0-01-000-
101001100 0-0--001-
101000110 0-0--100-
101000010 0-00-100-
101100010 0-00--10-
101000110 0-01--00-
100100000 0000-010-
110100000 0010-0-0-
101100000 0000-1-0-
111100000 0000--1-
101100010 0100--0-
100101000 0010-0-0-
100100010 0100-0-0-
101100010 0-00-1-0-
100101010 0-10-0-0-

```

```

100001000 0100-000-
101001000 0-01-000-
101001100 0-0--001-
101001010 0-0--010-
100101000 0-10-000-
100101100 0--0-001-
100101010 0--0-010-
100001100 0-01-000-
101001100 0-0--001-
100001110 0-1--000-
100001010 0-01-000-
101001010 0-0--010-
100001110 0-1--000-
100000100 0001-000-
110000100 000--100-
101000100 010--000-
101001100 0-0--001-
101000110 0-0--100-
100001100 010--000-
101001100 0-0--001-
100001110 0-1--000-
100000110 000--100-
100000010 0000-100-
110000010 0000--10-
111000010 0001--0-
110100010 0010--0-
101000010 0100--00-
101100010 0-00--10-
101000110 0-01--00-
100100010 0100--00-
101100010 0-00--10-
100100110 0-10--00-
100000110 0001--00-
010000000 0000-0001
110000000 0010-000-
110100000 00-0-010-
110001000 00-1-000-
110001100 00--001-
110001010 00--010-
110000100 00-0-100-
110000010 00-0-010-
011000000 1000-000-
010100000 1000-000-

```

Appendix A. A Strategy

010001000 0010-000-	001101000 1-00-000-
110001000 00-1-000-	001001100 1-00-000-
110001100 00---001-	001001010 1-00-000-
110001010 00---010-	001000100 0001-000-
010101000 10-0-000-	101000100 010--000-
010001100 10-0-000-	101001100 0-0--001-
010001010 00-1-000-	101000110 0-0--100-
110001010 00---010-	011000100 100--000-
010001110 10---000-	001001100 100--000-
010000100 0001-000-	001000110 100--000-
110000100 000--100-	001000010 0100-000-
011000100 100--000-	101000010 0-00-100-
010001100 100--000-	101100010 0-00--10-
010000110 100--000-	101000110 0-01--00-
010000010 0001-000-	001100010 1-00-000-
110000010 000--010-	001001010 1-00-000-
111000010 000--1-0-	001000110 1-00-000-
110001010 001--0-0-	000100000 0000-0001
011000010 100--000-	100100000 0000-010-
010001010 001--000-	110100000 0010-0-0-
110001010 00---010-	101100000 0000-1-0-
010001110 10---000-	111100000 0000---1-
010000110 100--000-	101100010 0100---0-
001000000 0000-0001	100101000 0010-0-0-
101000000 0100-000-	100100010 0100-0-0-
101100000 0-00-100-	101100010 0-00-1-0-
101100100 0-00--01-	100101010 0-10-0-0-
101100010 0-00--10-	010100000 1000-000-
101001000 0-01-000-	001100000 1000-000-
101001100 0-0--001-	000101000 1000-000-
101001010 0-0--010-	000100100 1000-000-
101000100 0-01-000-	000100010 1000-000-
101001100 0-0--001-	000001000 0000-0001
101000110 0-0--100-	100001000 0100-000-
101000010 0-00-100-	101001000 0-01-000-
101100010 0-00--10-	101001100 0-0--001-
101000110 0-01--00-	101001010 0-0--010-
011000000 1000-000-	100101000 0-10-000-
001100000 1000-000-	100101100 0--0-001-
001001000 0100-000-	100101010 0--0-010-
101001000 0-01-000-	100001100 0-01-000-
101001100 0-0--001-	101001100 0-0--001-
101001010 0-0--010-	100001110 0-1--000-

Appendix A. A Strategy

100001010 0-01-000-	101000100 010--000-
101001010 0-0--010-	101001100 0-0--001-
100001110 0-1--000-	101000110 0-0--100-
010001000 0010-000-	100001100 010--000-
110001000 00-1-000-	101001100 0-0--001-
110001100 00---001-	100001110 0-1--000-
110001010 00---010-	100000110 000--100-
010101000 10-0-000-	010000100 000--0001
010001100 10-0-000-	110000100 000--100-
010001010 00-1-000-	011000100 100--000-
110001010 00---010-	010001100 100--000-
010001110 10---000-	010000110 100--000-
001001000 0100-000-	001000100 000--0001
101001000 0-01-000-	101000100 010--000-
101001100 0-0--001-	101001100 0-0--001-
101001010 0-0--010-	101000110 0-0--100-
001101000 1-00-000-	011000100 100--000-
001001100 1-00-000-	001001100 100--000-
001001010 1-00-000-	001000110 100--000-
000101000 1000-000-	000001100 000--0001
000001100 0001-000-	100001100 010--000-
100001100 010--000-	101001100 0-0--001-
101001100 0-0--001-	100001110 0-1--000-
100001110 0-1--000-	010001100 100--000-
010001100 100--000-	001001100 100--000-
001001100 100--000-	000001110 001--000-
000001110 001--000-	100001110 01--000-
100001110 01--000-	010001110 10--000-
010001110 10--000-	000000110 000--0001
000001010 0010-000-	100000110 000--100-
100001010 00-1-000-	010000110 100--000-
110001010 00---010-	001000110 100--000-
100001110 01---000-	000001110 001--000-
010001010 00-1-000-	100001110 01--000-
110001010 00---010-	010001110 10--000-
010001110 10--000-	000000101 000--0010
000101010 10-0-000-	100000101 010--00-0
000001110 00-1-000-	010000101 100--00-0
100001110 01--000-	011000101 -00--10-0
010001110 10--000-	010001101 -01--00-0
000000100 0001-0000	001000101 100--00-0
100000100 000--0001	011000101 -00--10-0
110000100 000--100-	001001101 -10--00-0

Appendix A. A Strategy

000001101 100--00-0	000000110 0001-000-
010001101 -01--00-0	100000110 000--100-
001001101 -10--00-0	010000110 100--000-
000000010 0000-0001	001000110 100--000-
100000010 0000-100-	000001110 001--000-
110000010 0000--10-	100001110 01--000-
111000010 0001---0-	010001110 10--000-
110100010 0010---0-	000000001 0000-0100
101000010 0100--00-	100000001 0010-0-00
101100010 0-00--10-	010000001 0000-1-00
101000110 0-01--00-	110000001 0010--00
100100010 0100--00-	011000001 0001--00
101100010 0-00--10-	010100001 1000--00
100100110 0-10--00-	011100001 -000--10
100000110 0001--00-	010100011 -010--00
010000010 0001-000-	010000011 0010--00
110000010 000--010-	001000001 0000-1-00
111000010 000--1-0-	101000001 0100--00
110001010 001--0-0-	101100001 0-00--10
011000010 100--000-	101000011 0-01--00
010001010 001--000-	011000001 0001--00
110001010 00--010-	001100001 1000--00
010001110 10---000-	011100001 -000--10
010000110 100--000-	001100011 -100--00
001000010 0100-000-	001000011 0100--00
101000010 0-00-100-	101000011 0-01--00
101100010 0-00--10-	001100011 1-00--00
101000110 0-01--00-	000100001 0100-0-00
001100010 1-00-000-	100100001 0-10-0-00
001001010 1-00-000-	001100001 1-00-0-00
001000110 1-00-000-	001101001 --00-0-10
000100010 1000-000-	001100011 --00-1-00
000001010 0010-000-	000101001 0-10-0-00
100001010 00-1-000-	000100011 0-10-0-00
110001010 00--010-	000001001 0010-0-00
100001110 01--000-	000000011 1000-0-00
010001010 00-1-000-	010000011 -010-0-00
110001010 00--010-	001000011 -100-0-00
010001110 10---000-	001100011 --00-1-00
000101010 10-0-000-	001001011 --01-0-00
000001110 00-1-000-	000100011 -010-0-00
100001110 01--000-	000001011 -010-0-00
010001110 10---000-	000000000 000010000

Appendix A. A Strategy

While the PLA-form fully describes a set of formulas, there are more readable formats. What follows is the formulas expressed in C.

```
output[0] =
(!input[0] && input[1] && input[2] && !input[8] && TRUE) ||
(!input[0] && input[1] && input[3] && TRUE) ||
(!input[0] && input[1] && input[5] && input[6] && TRUE) ||
(input[1] && input[6] && input[7] && TRUE) ||
(input[1] && input[2] && input[6] && TRUE) ||
(!input[0] && input[2] && input[3] && TRUE) ||
(!input[0] && input[2] && input[5] && input[6] && TRUE) ||
(!input[0] && input[2] && input[5] && input[7] && TRUE) ||
(!input[0] && input[2] && input[6] && input[7] && TRUE) ||
(!input[0] && input[3] && input[5] && !input[8] && TRUE) ||
(!input[0] && input[3] && input[6] && TRUE) ||
(!input[0] && input[3] && input[7] && !input[8] && TRUE) ||
(input[1] && input[6] && input[8] && TRUE) ||
(input[2] && input[6] && input[8] && TRUE) ||
(input[5] && input[6] && input[8] && TRUE) ||
(!input[1] && !input[2] && !input[3] && input[7] && input[8] && TRUE) ||
FALSE;

output[1] =
(input[0] && !input[1] && input[2] && !input[3] && TRUE) ||
(input[2] && input[3] && input[7] && TRUE) ||
(input[0] && !input[1] && input[3] && input[7] && TRUE) ||
(input[0] && !input[1] && !input[3] && input[5] && !input[7] && TRUE) ||
(input[0] && input[2] && input[6] && TRUE) ||
(input[0] && !input[1] && input[5] && input[6] && TRUE) ||
(input[2] && input[5] && !input[6] && TRUE) ||
(!input[1] && input[2] && !input[6] && input[7] && TRUE) ||
(input[0] && input[6] && input[8] && TRUE) ||
(input[2] && input[5] && input[8] && TRUE) ||
(input[0] && input[2] && input[8] && TRUE) ||
(input[2] && input[7] && input[8] && TRUE) ||
(!input[1] && !input[2] && input[3] && !input[7] && input[8] && TRUE) ||
FALSE;

output[2] =
(input[0] && input[1] && !input[2] && !input[6] && !input[7] && TRUE) ||
(input[0] && input[1] && !input[2] && input[3] && TRUE) ||
(input[0] && input[3] && input[5] && TRUE) ||
```

Appendix A. A Strategy

```
(input[5] && input[6] && input[7] && TRUE) ||
(input[1] && input[3] && input[7] && TRUE) ||
(input[3] && input[6] && input[7] && TRUE) ||
(input[1] && input[5] && !input[6] && TRUE) ||
(input[0] && input[1] && input[5] && TRUE) ||
(!input[0] && !input[2] && input[5] && input[7] && TRUE) ||
(input[1] && input[5] && input[8] && TRUE) ||
(input[0] && !input[2] && !input[6] && input[8] && TRUE) ||
(input[0] && input[1] && input[8] && TRUE) ||
(input[1] && input[7] && input[8] && TRUE) ||
(input[0] && !input[2] && input[3] && input[8] && TRUE) ||
(!input[2] && input[3] && input[5] && input[8] && TRUE) ||
(!input[2] && input[3] && input[7] && input[8] && TRUE) ||
(!input[2] && input[5] && !input[6] && input[8] && TRUE) ||
FALSE;

output[3] =
(input[0] && input[1] && input[5] && TRUE) ||
(input[0] && input[2] && input[5] && TRUE) ||
(input[0] && input[2] && !input[3] && input[6] && TRUE) ||
(input[0] && !input[3] && input[5] && input[6] && TRUE) ||
(input[0] && !input[3] && input[5] && input[7] && TRUE) ||
(input[0] && !input[1] && !input[3] && input[6] && TRUE) ||
(input[1] && input[2] && input[7] && TRUE) ||
(input[1] && input[5] && input[7] && TRUE) ||
(!input[0] && input[1] && !input[5] && input[6] && TRUE) ||
(!input[0] && input[1] && input[7] && !input[8] && TRUE) ||
(!input[0] && input[2] && !input[5] && input[6] && !input[7] && TRUE) ||
(!input[1] && !input[2] && !input[3] && input[6] && TRUE) ||
(input[5] && input[6] && input[7] && TRUE) ||
(input[1] && input[2] && !input[3] && input[8] && TRUE) ||
(input[0] && input[7] && input[8] && TRUE) ||
(input[2] && input[5] && input[7] && input[8] && TRUE) ||
FALSE;

output[4] =
TRUE;

output[5] =
(input[0] && input[1] && !input[5] && input[6] && TRUE) ||
(input[0] && input[2] && input[3] && TRUE) ||
(input[0] && input[2] && input[6] && input[7] && TRUE) ||
(input[0] && input[2] && !input[5] && input[7] && TRUE) ||
```

Appendix A. A Strategy

```
(input[0] && !input[5] && input[6] && input[7] && TRUE) ||
(input[0] && !input[1] && !input[3] && !input[5] && input[7] && TRUE) ||
(input[0] && input[1] && input[2] && TRUE) ||
(input[1] && input[2] && input[8] && TRUE) ||
(input[1] && !input[6] && !input[7] && input[8] && TRUE) ||
(input[2] && !input[3] && !input[6] && !input[7] && input[8] && TRUE) ||
(input[2] && input[3] && input[7] && input[8] && TRUE) ||
FALSE;

output[6] =
(input[0] && input[1] && input[3] && TRUE) ||
(input[0] && input[1] && input[7] && TRUE) ||
(input[0] && input[2] && input[3] && input[7] && TRUE) ||
(input[0] && input[2] && input[5] && input[7] && TRUE) ||
(input[0] && !input[2] && input[3] && !input[5] && !input[7] && TRUE) ||
(input[0] && input[3] && input[5] && input[7] && TRUE) ||
(!input[6] && input[8] && TRUE) ||
FALSE;

output[7] =
(input[0] && input[1] && input[5] && input[6] && TRUE) ||
(input[2] && input[3] && input[6] && TRUE) ||
(input[0] && input[2] && input[5] && input[6] && TRUE) ||
(input[1] && input[2] && input[3] && TRUE) ||
(input[3] && input[5] && input[6] && TRUE) ||
(input[6] && input[8] && TRUE) ||
(input[0] && input[2] && input[3] && input[8] && TRUE) ||
(input[2] && input[3] && input[5] && input[8] && TRUE) ||
FALSE;

output[8] =
(input[0] && !input[8] && TRUE) ||
(input[1] && !input[8] && TRUE) ||
(input[2] && !input[8] && TRUE) ||
(input[3] && !input[8] && TRUE) ||
(input[5] && !input[8] && TRUE) ||
(input[6] && input[7] && TRUE) ||
(input[7] && !input[8] && TRUE) ||
FALSE;
```

The formulas can also be expressed in mathematical terms as follows.

Appendix A. A Strategy

$$\begin{aligned}
o_0 = & (\neg i_0 \wedge i_1 \wedge i_2 \wedge \neg i_8) \vee \\
& (\neg i_0 \wedge i_1 \wedge i_3) \vee \\
& (\neg i_0 \wedge i_1 \wedge i_5 \wedge i_6) \vee \\
& (i_1 \wedge i_6 \wedge i_7) \vee \\
& (i_1 \wedge i_2 \wedge i_6) \vee \\
& (\neg i_0 \wedge i_2 \wedge i_3) \vee \\
& (\neg i_0 \wedge i_2 \wedge i_5 \wedge i_6) \vee \\
& (\neg i_0 \wedge i_2 \wedge i_5 \wedge i_7) \vee \\
& (\neg i_0 \wedge i_2 \wedge i_6 \wedge i_7) \vee \\
& (\neg i_0 \wedge i_3 \wedge i_5 \wedge \neg i_8) \vee \\
& (\neg i_0 \wedge i_3 \wedge i_6) \vee \\
& (\neg i_0 \wedge i_3 \wedge i_7 \wedge \neg i_8) \vee \\
& (i_1 \wedge i_6 \wedge i_8) \vee \\
& (i_2 \wedge i_6 \wedge i_8) \vee \\
& (i_5 \wedge i_6 \wedge i_8) \vee \\
& (\neg i_1 \wedge \neg i_2 \wedge \neg i_3 \wedge i_7 \wedge i_8) \\
o_1 = & (i_0 \wedge \neg i_1 \wedge i_2 \wedge \neg i_3) \vee \\
& (i_2 \wedge i_3 \wedge i_7) \vee \\
& (i_0 \wedge \neg i_1 \wedge i_3 \wedge i_7) \vee \\
& (i_0 \wedge \neg i_1 \wedge \neg i_3 \wedge i_5 \wedge \neg i_7) \vee \\
& (i_0 \wedge i_2 \wedge i_6) \vee \\
& (i_0 \wedge \neg i_1 \wedge i_5 \wedge i_6) \vee \\
& (i_2 \wedge i_5 \wedge \neg i_6) \vee
\end{aligned}$$

Appendix A. A Strategy

$$\begin{aligned}
& (\neg i_1 \wedge i_2 \wedge \neg i_6 \wedge i_7) \vee \\
& (i_0 \wedge i_6 \wedge i_8) \vee \\
& (i_2 \wedge i_5 \wedge i_8) \vee \\
& (i_0 \wedge i_2 \wedge i_8) \vee \\
& (i_2 \wedge i_7 \wedge i_8) \vee \\
& (\neg i_1 \wedge \neg i_2 \wedge i_3 \wedge \neg i_7 \wedge i_8) \\
o_2 = & (i_0 \wedge i_1 \wedge \neg i_2 \wedge \neg i_6 \wedge \neg i_7) \vee \\
& (i_0 \wedge i_1 \wedge \neg i_2 \wedge i_3) \vee \\
& (i_0 \wedge i_3 \wedge i_5) \vee \\
& (i_5 \wedge i_6 \wedge i_7) \vee \\
& (i_1 \wedge i_3 \wedge i_7) \vee \\
& (i_3 \wedge i_6 \wedge i_7) \vee \\
& (i_1 \wedge i_5 \wedge \neg i_6) \vee \\
& (i_0 \wedge i_1 \wedge i_5) \vee \\
& (\neg i_0 \wedge \neg i_2 \wedge i_5 \wedge i_7) \vee \\
& (i_1 \wedge i_5 \wedge i_8) \vee \\
& (i_0 \wedge \neg i_2 \wedge \neg i_6 \wedge i_8) \vee \\
& (i_0 \wedge i_1 \wedge i_8) \vee \\
& (i_1 \wedge i_7 \wedge i_8) \vee \\
& (i_0 \wedge \neg i_2 \wedge i_3 \wedge i_8) \vee \\
& (\neg i_2 \wedge i_3 \wedge i_5 \wedge i_8) \vee \\
& (\neg i_2 \wedge i_3 \wedge i_7 \wedge i_8) \vee \\
& (\neg i_2 \wedge i_5 \wedge \neg i_6 \wedge i_8) \\
o_3 = & (i_0 \wedge i_1 \wedge i_5) \vee \\
& (i_0 \wedge i_2 \wedge i_5) \vee
\end{aligned}$$

Appendix A. A Strategy

$$\begin{aligned}
& (i_0 \wedge i_2 \wedge \neg i_3 \wedge i_6) \vee \\
& (i_0 \wedge \neg i_3 \wedge i_5 \wedge i_6) \vee \\
& (i_0 \wedge \neg i_3 \wedge i_5 \wedge i_7) \vee \\
& (i_0 \wedge \neg i_1 \wedge \neg i_3 \wedge i_6) \vee \\
& (i_1 \wedge i_2 \wedge i_7) \vee \\
& (i_1 \wedge i_5 \wedge i_7) \vee \\
& (\neg i_0 \wedge i_1 \wedge \neg i_5 \wedge i_6) \vee \\
& (\neg i_0 \wedge i_1 \wedge i_7 \wedge \neg i_8) \vee \\
& (\neg i_0 \wedge i_2 \wedge \neg i_5 \wedge i_6 \wedge \neg i_7) \vee \\
& (\neg i_1 \wedge \neg i_2 \wedge \neg i_3 \wedge i_6) \vee \\
& (i_5 \wedge i_6 \wedge i_7) \vee \\
& (i_1 \wedge i_2 \wedge \neg i_3 \wedge i_8) \vee \\
& (i_0 \wedge i_7 \wedge i_8) \vee \\
& (i_2 \wedge i_5 \wedge i_7 \wedge i_8) \\
o_4 &= \text{TRUE} \\
o_5 &= (i_0 \wedge i_1 \wedge \neg i_5 \wedge i_6) \vee \\
& (i_0 \wedge i_2 \wedge i_3) \vee \\
& (i_0 \wedge i_2 \wedge i_6 \wedge i_7) \vee \\
& (i_0 \wedge i_2 \wedge \neg i_5 \wedge i_7) \vee \\
& (i_0 \wedge \neg i_5 \wedge i_6 \wedge i_7) \vee \\
& (i_0 \wedge \neg i_1 \wedge \neg i_3 \wedge \neg i_5 \wedge i_7) \vee \\
& (i_0 \wedge i_1 \wedge i_2) \vee \\
& (i_1 \wedge i_2 \wedge i_8) \vee \\
& (i_1 \wedge \neg i_6 \wedge \neg i_7 \wedge i_8) \vee \\
& (i_2 \wedge \neg i_3 \wedge \neg i_6 \wedge \neg i_7 \wedge i_8) \vee
\end{aligned}$$

Appendix A. A Strategy

$$\begin{aligned}
& (i_2 \wedge i_3 \wedge i_7 \wedge i_8) \\
o_6 = & (i_0 \wedge i_1 \wedge i_3) \vee \\
& (i_0 \wedge i_1 \wedge i_7) \vee \\
& (i_0 \wedge i_2 \wedge i_3 \wedge i_7) \vee \\
& (i_0 \wedge i_2 \wedge i_5 \wedge i_7) \vee \\
& (i_0 \wedge \neg i_2 \wedge i_3 \wedge \neg i_5 \wedge \neg i_7) \vee \\
& (i_0 \wedge i_3 \wedge i_5 \wedge i_7) \vee \\
& (\neg i_6 \wedge i_8) \\
o_7 = & (i_0 \wedge i_1 \wedge i_5 \wedge i_6) \vee \\
& (i_2 \wedge i_3 \wedge i_6) \vee \\
& (i_0 \wedge i_2 \wedge i_5 \wedge i_6) \vee \\
& (i_1 \wedge i_2 \wedge i_3) \vee \\
& (i_3 \wedge i_5 \wedge i_6) \vee \\
& (i_6 \wedge i_8) \vee \\
& (i_0 \wedge i_2 \wedge i_3 \wedge i_8) \vee \\
& (i_2 \wedge i_3 \wedge i_5 \wedge i_8) \\
o_8 = & (i_0 \wedge \neg i_8) \vee \\
& (i_1 \wedge \neg i_8) \vee \\
& (i_2 \wedge \neg i_8) \vee \\
& (i_3 \wedge \neg i_8) \vee \\
& (i_5 \wedge \neg i_8) \vee \\
& (i_6 \wedge i_7) \vee \\
& (i_7 \wedge \neg i_8)
\end{aligned}$$

Appendix B

Our Tools, Their Locations, and Their Uses

All of the tools described in this thesis may be found on the author's home page¹, as well as on the CD-ROM that contains the complete thesis.

B.1 Stratgen

Stratgen is our strategy generation toolset. There are actually four tools in the family:

- `treegen` - generates strategy trees for the game of tic-tac-toe
- `favgen` - generates favorable strategy trees for the game of tic-tac-toe
- `feasgen` - generates feasible strategy trees for the game of tic-tac-toe
- `feasfavgen` - generates feasible, favorable strategy trees for the game of tic-tac-toe

¹<http://www.cs.unm.edu/~bandrews/thesis-research>

B.1.1 Usage and Output

Running any one of these tools with no arguments will result in a “usage” message. All of these tools take one command line parameter. The tools when run with an integer parameter i will produce strategies until they have produced i strategies.

All of the strategy generation tools are set up to produce strategies as `Minimo` input files. The tools print their output to stdout.

B.2 Stratcounter

`Stratcounter` is our toolset for counting the number of strategies for the game of tic-tac-toe. From one C file, we produce four executables:

- `counter` - counts the number of tic-tac-toe strategies after a given starting move
- `counter-debug` - counts the number of tic-tac-toe strategies after a given starting move and prints debugging information
- `favcounter` - counts the number of favorable tic-tac-toe strategies after a given starting move
- `favcounter-debug` - counts the number of favorable tic-tac-toe strategies after a given starting move and prints debugging information

B.2.1 Usage and Output

Running any of the executables with no arguments produces no results. Running the executables with an argument between zero and eight calculates and prints to stdout the

desired number. In the case of the “-debug” variations, there will be some amount of extra calculations printed.

B.3 Checker

`Checker` is our strategy-checking tool. After a formula is converted to PLA-form, it is compiled into this tool. This tool plays out all the possible games against a strategy.

B.3.1 Usage and Output

This tool must be recompiled for every strategy to be checked. Once compiled, it is run with “`./cheat-checker`”. If the strategy being checked is a valid strategy, then the tool will print a `dot` representation of the strategy to `stdout`. If the strategy is not a valid strategy, then the tool prints debugging information to `stdout`.

B.4 CandBuild

`CandBuild` builds candidate conjunctions for technology mapping. The easiest way to use this tool is to build candidates lists recursively. First, generate the list of candidates that fit the desired shape that have one literal. Next, generate the list of candidates that fit the desired shape that have two literals and append this to the list of candidates with one literal. Using this method, you are assured to have a list of candidates with increasing numbers of literals.

B.4.1 Usage and Output

Running “java CandBuild” produces a candidates file that can be used with `Minimo`.

B.5 Minimo

`Minimo` is our Boolean formula mapping tool.

B.5.1 Usage

`Minimo` requires two input files in order to run: a file containing strategies (in PLA form), and a file containing candidate conjunction shapes. If the candidates list is ordered by increasing numbers of literals, then the output from `Minimo` is guaranteed to produce the minimal representation within the candidates list.

B.5.2 Output

`Minimo` is constantly trying to map strategies to the candidates. The output from `Minimo` is somewhat sporadic. As it attempts to map strategies to candidates, it prints the generated formula disjuncts until the tool fails to match for the strategy. If and when the word “found” is printed, then the tool has found a mapping, and the formulas printed just before the “found” represent that set of formulas.

The output formulas are printed in C. They are written in such a way that they can compile with the `checker` tool.

B.6 Minimo Strategy Input File

A `Minimo` strategy input file contains some number of strategies expressed as Boolean formulas (with a blank line separating two strategies). Every line in a strategy must contain the same number of characters. Every line begins with some number of zeroes and ones representing an input vector. The input vector is followed by a single space and an output vector. An output vector consists of the characters zero, one, and hyphen (representing that an input vector is in the DON'T CARE-set of a particular function).

A `Minimo` candidates input file contains some number of lines, each representing a possible disjunct to be considered for covering a function. Every line must contain the same number of characters. Every line is composed of some number of zeroes, ones, and hyphens. A hyphen as the i th character in a line indicates that the i th input does not appear in that conjunction.

Appendix C

Computing Values For f

The printed version of this thesis document omits Sections C.1, C.1, and C.3. They can be found on the enclosed CD-ROM or on the author's web page¹.

The following was stated in Section 5.3.2:

$$\begin{aligned} f([], []) &\approx (1.90478 \cdot 10^{123} \cdot 4) + (7.45027 \cdot 10^{122} \cdot 4) + (3.6333 \cdot 10^{123}) \\ &\approx 1.4233 \cdot 10^{124}. \end{aligned}$$

This chapter is meant to provide a demonstration of how this figure was derived. From Section 5.3.2, we know that $f([], []) = \sum_{i=0}^8 f([], [i])$. Surely the number of strategies that begin with a move in one of the “corner” positions on the board should not depend on *which* corner the move is in. Similarly, the number of strategies that begin with a move in one of the “side” positions should be equal to the number of strategies that begin in any of the other side positions. With this in mind, we only really need to perform three computations to compute $f([], [])$. Specifically, we need to compute $f([], [0])$, $f([], [1])$, and $f([], [4])$.

¹<http://www.cs.unm.edu/~bandrews/thesis-research>

Appendix C. Computing Values For f

C.1 Computing the Value of $f([], [0])$

C.2 Computing the Value of $f([], [1])$

C.3 Computing the Value of $f([], [4])$

Appendix D

Computing Values For g

The printed version of this thesis document omits Sections D.1, D.1, and D.3. They can be found on the enclosed CD-ROM or on the author's web page¹.

Similarly, we will demonstrate the derivation of the function g .

D.1 Computing the Value of $g([], [0])$

D.2 Computing the Value of $g([], [1])$

D.3 Computing the Value of $g([], [4])$

¹<http://www.cs.unm.edu/~bandrews/thesis-research>

References

- [1] Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(5187):1021–1024, November 1994.
- [2] Philip Ball. Chemistry meets computing. *Nature*, 406:118–120, July 2000.
- [3] R. R. Breaker and G. F. Joyce. A DNA enzyme that cleaves RNA. *Chemistry and Biology*, 1:223–229, 1994.
- [4] Ronald R Breaker and Gerald F Joyce. A DNA enzyme with Mg^{2+} -dependent RNA phosphoesterase activity. *Chemistry & Biology*, 2:655–660, 1995.
- [5] Bernard Cuenoud and Jack W. Szostak. A DNA metalloenzyme with DNA ligase activity. *Nature*, 375:611–614, June 1995.
- [6] Jan Hlavička and Petr Fišer. BOOM - a heuristic boolean minimizer. In *International Conference on Computer Aided Design*, pages 439–442, 2001.
- [7] Jesper Juul. 255,168 ways of playing tic tac toe. From the author’s web page: <http://www.jesperjuul.dk/ludologist/?p=55>.
- [8] E. McCluskey. Minimization of Boolean functions. *Bell System Technical Journal*, 35:437–457, 1956.
- [9] A. Meyer and L. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proceedings of the 13th IEEE Symposium on Switching and Automata Theory*, pages 125–129, 1972.
- [10] Edward W. Packel. *The Mathematics of Games and Gambling*, volume 28 of *The Mathematical Association of America New Mathematical Library*. The Mathematical Association of America, Washington, D.C., 1st edition, 1981.
- [11] Milan N. Stojanovic, Paloma de Prada, and Donald W. Landry. Catalytic molecular beacons. *ChemBioChem*, 2(6):411–415, 2001.

References

- [12] Milan N. Stojanovic, Tiffany Elizabeth Mitchell, and Darko Stefanovic. Deoxyribozyme-based logic gates. *Journal of the American Chemical Society*, 124(14):3555–3561, April 2002.
- [13] Milan N. Stojanovic and Darko Stefanovic. Deoxyribozyme-based half adder. *Journal of the American Chemical Society*, 125(22):6673–6676, 2003.
- [14] Milan N. Stojanovic and Darko Stefanovic. A deoxyribozyme-based molecular automaton. *Nature Biotechnology*, 21:1069–1074, 2003.
- [15] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.