# Strategies to Fast Evaluation of Expression Trees

**Raed Yousef Mohammed Basbous**

Submitted to the
Institute of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Applied Mathematics and Computer Science

Eastern Mediterranean University
June 2016
Gazimağusa, North Cyprus

Approval of the Institute of Graduate Studies and Research

Prof. Dr. Cem Tanova
Acting Director

I certify that this thesis satisfies the requirements as a thesis for the degree of Doctor of Philosophy in Applied Mathematics and Computer Science.

Prof. Dr. Nazım Mahmudov
Chair, Department of Mathematics

We certify that we have read this thesis and that in our opinion it is fully adequate in scope and quality as a thesis for the degree of Doctor of Philosophy in Applied Mathematics and Computer Science.

Assoc. Prof. Dr. Benedek Nagy
Supervisor

Examining Committee

1. Prof. Dr. Rahib H. Abiyev
2. Prof. Dr. Rza Bashirov
3. Prof. Dr. Robert Elsässer
4. Assoc. Prof. Dr. Cüneyt Bazlamaçcı
5. Assoc. Prof. Dr. Benedek Nagy

# ABSTRACT

Expression trees are well-known tools to visualize the syntactical structure of the expressions. They are helpful also in evaluations, e.g., decision trees are widely used. Games and game theory form an important field in Artificial Intelligence and it has several connections to Business and Economy. Short circuit, short cut, or by other name, lazy evaluations play important roles in various fields of computer science including logic, hardware design, programming, decision making.

In this thesis, different types of trees are considered including extensions of game trees using operations, e.g., multiplication, (constrained) addition and the usual minimum and maximum, and three of the best known and used fuzzy logic systems, (Gödel, Lukasiewicz, and product logics).

The evaluation of lots of formulae can be speeded up by various pruning techniques by discovering which remaining part of the formulae has no influence on the final result for various reasons. The presented techniques can be seen as generalizations of short circuit evaluations in Boolean logic and also of alpha-beta pruning of game trees. Simulation results show the efficiency of the presented techniques.

**Keywords:** expression trees, game trees, formula trees, fast evaluation, fuzzy logic, many valued logic, pruning techniques, short circuit evaluation, lazy evaluations.

# ÖZ

İfade ağaçları, ifadelerin sözdizimsel yapılarını görselleştirmek için kullanılan araçlardır. Onlar değerlendirmede oldukça yardımcıdırlar, örneğin, karar ağaçları yaygın olarak kullanılanlardandır. Oyun ve oyun teorisi yapay zekada önemli bir alan olup, işletme ve ekonomide çeşitli bağlantılara sahiptir. Kısa devre, kısa yol, ya da diğer bir adıyla tembel değerlendirmeler mantık, donanım tasarımı, programlama, karar verme gibi bilgisayar bilimlerinin çeşitli alanlarında önemli rol oynar.

Bu tezde, oyun ağaçlarının uzantıları dahil farklı ağaç modelleri düşünülmüştür. Örneğin, çarpma, toplama(sınırlandırılmış) ve olağan minimum ve maksimum işlemleri ile, en iyi bilinen ve en çok kullanılan bulanık mantık sistemlerinden üç tanesi, çarpma mantığı işlenmiştir.

**Anahtar kelimeler**: ifade ağaçları, oyun ağaçları, formül ağaçları, hızlı hesaplamalar, bulanık mantık, birçok değerli mantık, budama teknikleri, kısa devre hesaplamaları, tembel hesaplamalar.

# DEDICATION

*To my wife, son, daughters, and parents*

# ACKNOWLEDGMENT

I would like to thank my supervisor, Assoc. Prof. Dr. Benedek Nagy for his encouragement and support all the time. He always guided me into the right directions by his extensive knowledge and experience on the subject, encouraged, motivated and trusted me while doing this research.

I am thankful to my director Dr. Marwan Darwish for his continuous help and support that was always given to me always with a great politeness and smiling faces.

My appreciations are also to the president of AlQuds Open University, Prof. Younes Amro for his encouragement and support to complete my PhD study.

I am grateful to Mr. Tibor Tajti from Hungary for his help and collaboration in programming the pseudo codes for the proposed pruning algorithms.

I owe too much to my wife Shireen for her endless support, understanding, patience, and also for her great help and encouragement.

The last but not the least, I would like to thank my parents, my son Yousef, and my lovely daughters Talah, Daniah and Sarah who instilled me the free thinking and the joy of making researches.

# TABLE OF CONTENTS

# LIST OF FIGURES

# Chapter 1

# INTRODUCTION

## 1.1 Motivations

The classical, also known as Boolean, logic is well known and can be found in thousands of text books, including books on (discrete) mathematics, programming technologies, software engineering, hardware design, mathematical and philosophical logics, linguistics, etc., since it is known as a base of mathematics, computer science and information technology, electrical engineering and other theoretical, scientific and technical fields [1]. Apart from the Boolean logic there are several other branches developed for various purposes. In first order logics quantifiers allow more structured formulae. In modal and temporal logics new operations (such as necessity, possibility and always, sometime in the future) are introduced. Many valued and fuzzy logics are motivated by the fact that in the real word usually there is no strict border of concepts. Good examples are the adjectives, they do not have a 'crisp' meaning: what does it mean large? What if a mouse is large, or a city is large? What if another one is larger? etc.. In these systems the set of truth values is extended from the classical two values to a larger, even to an infinite set. Various paradoxes of the classical logic can also be avoided by using many valued or fuzzy logic [2]. A well-known, ancient example is the liar paradox [3]. The sentence "This sentence is false" cannot be true and cannot be false, but in Boolean logic there is no other possible truth value. A third, additional truth-value can solve this paradox.

From the 20s of the last century various fuzzy logics were developed. The most important such logical systems are the Gödel type logic, the Lukasiewicz type logic and the product logic [2,4,5,6,7]. In Gödel logic the classical law of double negation does not work as a logical law [4]. Gödel logic can be used in an optimist environment, when partners are friends and they want to cooperate to gain maximal profit. The product logic is perfect to model tolerant and realistic environments with non-expert, independent partners. The Lukasiewicz type logic (especially, the Lukasiewicz type conjunction and disjunction) refers to pessimistic, unfriendly environment, where the aim of the partners is to have minimal loss in their competition [2]. While both the Gödel and the Lukasiewicz logics work well with any finite number of truth values (in case of 2 values they give back the classical logic, with larger sets they are real many valued logics), the product logic has only variants with infinitely many truth values, e.g., all rational numbers or all real numbers of the unit interval [0,1] are truth values. There are various studies on the product logic including its relation to other fuzzy logic systems [8,9], axiomatization and proof systems [10,11,12]. We note also that fuzzy sets and fuzzy logics are also used in several applications [13].

Evaluating a logical expression is an important task when one is working with logical formulae. These expressions can be evaluated in a fast way based on the members of a conjunctive and disjunctive formulae [14]. This is the case in various programming languages at, e.g., conditional statements. Different pruning strategies to fast evaluation of Gödel and product logics are presented in [16,28], they are closely related to pruning techniques in game theory [17,18,19] and in generalized game trees [20,21].

## 1.2 Contributions

In this study, various pruning techniques are presented to speed up the evaluations of a special formula/expression trees that can be considered as a type of extension of the usual game trees/logical expression trees, and the evaluations of logical formulae in the Gödel type logic, the Lukasiewicz type logic and the product logic.

Experimental/simulation results are presented to show the efficiency of the proposed techniques. Some of the proposed algorithms have been programmed on Python language and hundreds thousands of tests on formulae with various sizes are conducted.

## 1.3 Thesis Structure

The following chapters of this research can be mainly divided into 5 parts; namely, literature review and preliminaries, the strategies to fast evaluation of Boolean expressions and extended game trees, strategies to fast evaluation of many-valued logic formulae, simulation results, and finally the conclusions.

The research is organized as follows:

In chapter two, some preliminaries are recalled, including the semantics of the game theory, expression trees, Boolean logic, Gödel logic, Lukasiewicz logic, product logic, well known short circuit evaluations techniques used in Boolean logic, and pruning techniques in artificial intelligence and game theory.

Chapter three and four present various lazy evaluation techniques for the expressions and logic trees mentioned in chapter two. Chapter five shows the simulation results

for the proposed pruning strategies. Finally, chapter six provides the conclusions for this study.

# Chapter 2

# LITERATURE REVIEW AND PRELIMINARIES

## 2.1 Statement of Problem

Evaluating a logical expression is an important task when one is working with logical formulae. The relation between the length of the expression and the size of the required memory and the time of the computation is proportional. This issue can be solved by applying the lazy evaluations.

Lazy evaluation techniques are used in several places in science, engineering and technology. These techniques are used to reduce the size of a circuit connected to "don't care" values in hardware design. They could also reduce the working memory and/or the time of the computation. It is based on the fact that sometimes the result can be computed with 100% sure without knowing all the subresults or all parts of the computation. These methods are used to reduce the size of a circuit, the working memory and/or the time of the computation. Also, by these techniques one could effectively reduce the size of the expression trees allowing a much faster method of evaluation.

## 2.2 Literature Review

In [14] the author mentioned that logical expressions can be evaluated in a fast way based on the following facts: if a member of a conjunctive formula is false, then the whole formula is false; if a member of a disjunctive formula is true, then the whole formula is true.

The shortcut evaluation technique is used in several programming languages to have a fast evaluation of logical formulae, e.g., in conditions [22].

In [18] an idea is presented that is very similar to shortcut evaluation which is used in game trees. At the most investigated zero-sum two-player games the minimax algorithm gives the best strategies for the players and it also answers the question who has a winning strategy. To compute the minimax algorithm every leaf of the tree is computed and the whole tree is evaluated. However, in most of the cases, it can be done with a much less effort, using alpha and beta pruning techniques [18,23].

Special extensions, as a mixture of decision and game-trees were already discussed in [19].

Classical two-valued logic was formalized by Boole in the 19th century. It works with algebraic technique: with only two values, 0 (false) and 1 (true) Boolean algebra gives the symbolic framework. It is used in electronic switching circuits, and thus, it gives the base of all our digital machines [1].

In [15] it is presented that the logic used in conditions is very close to the Boolean logic even in some points it is not exactly the classical two-valued logic. Also, it is presented that the lazy evaluation appears in logic in various forms.

The idea of considering intermediate truth values rather than only the classical set of truth values {0,1} has been used extensively by Lukasiewicz and others since the 1920's. Various fuzzy logic were developed, e.g., Gödel type logic, Lukasiewicz logics and the product logic [2,4,5,6].

From historical point of view we mention that Lukasiewicz made three-valued and four-valued systems first. Later he extended the system to arbitrary-many ($n \geq 2$) truth-values up to infinitely many [5,6,9].

In [4] Gödel introduced his logic to obtain an intuitionistic logic. The most important feature of these type of logics that the classical law of double negation does not work in these systems as a logical law.

In [10] the authors described by explicit mathematical way one of the most popular fuzzy logic systems: the product logic. The product logic is perfect to model tolerant and realistic environment with non-expert, independent partners.

In [21] the authors considered simple network models, in which there are only finite communication channels. The children nodes send their data/results to their parents and thus, the whole process is finite and can be modeled by evaluation techniques. They showed pruning strategies: techniques for expressions where sum, product, minimum and maximum modeling various modalities in CogInfoCom networks. These techniques could help in fast evaluations of various expressions (other types of trees and networks).

## 2.3 Preliminaries

In this section we recall some concepts that are related to our study, specially decision trees, game trees, expression trees, minimax algorithm, alpha-beta pruning, and the syntax and semantics of the Lukasiewicz, Gödel and product logics. We start this section by the well-known concept of decision trees.

### 2.3.1 Decision Trees

In this subsection we recall decision trees (see, e.g. in [19]); they can also be used to evaluate games against the "Nature". Let a person be given who has different decision points, and different random events with known probabilities, also let us consider a tree with decision nodes and chance nodes. At decision nodes the person chooses a successor node. At chance nodes: the successor node is chosen randomly. The leaves represent the payoff values. The aim of the person is to maximize the payoff value and for this purpose he/she chooses the branch that leads to the highest expected value at every decision node.

Assume that P(Node) denotes the probability of the event Node. Figure 1, shows an example of a decision tree. It includes decision nodes, where the person chooses among the successor, i.e. children nodes (represented by rectangles). At nodes represented by ellipses random events will determine the successor node (the numbers that are written on the edges represent the probabilities). A numeric value, the expected outcome, is assigned to every node of the tree; it is shown under the rectangle/ellipse.

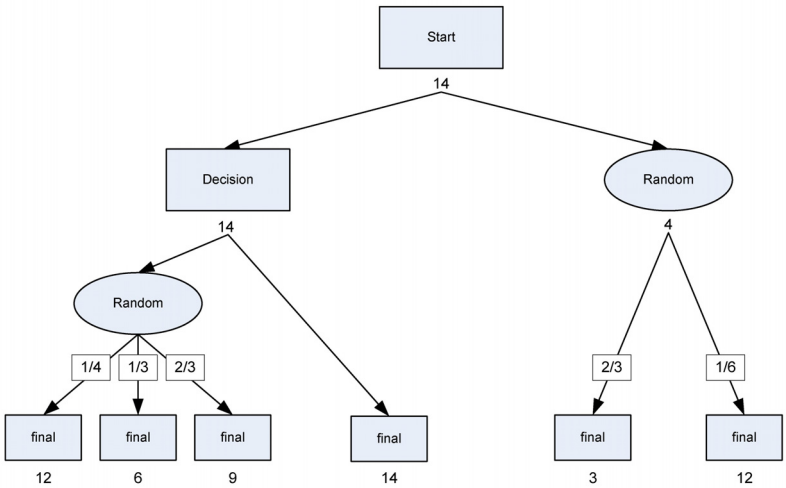


Figure 1: An example of a decision tree.

The values in Figure 1 are computed using the expectimax algorithm which is presented in [19], since the maximum of the expected values of the children (successor nodes) are computed at decision points, while at chance nodes the expected values of the children (successor nodes) are computed by finding the sum of the assigned probabilities multiplied by the value of the given successors. These values are computed by Algorithm 1 (it is from [19]).

*Algorithm 1 (Decision)*

1. function Dec(Node)
2. begin
3. if Node is leaf then
4. return the value of Node
5. end if
6. else
7. let $Node_1$, $Node_2$, …, $Node_m$ be the children of Node
8. if Node is a decision node then
9. return maximum of {$Dec(Node_1)$, $Dec(Node_2)$, …, $Dec(Node_m)$}
10. end if
11. if Node is a chance node then
12. return $P(Node_1)Dec(Node_1)$+ … + $P(Node_m)Dec(Node_m)$
13. end if
14. end else
15. end Dec

### 2.3.2 Game Theory

An important and widely investigated field of game theory is about deterministic, strategic, two-player, zero-sum, finite games with perfect information. An instance of a game begins with the first player's choice from a set of specified alternatives, called moves. After a move, a new state of the game is obtained; the other player is to make the next move from the alternatives available to that player. In some state of the game there is no move possible, the instance of the game has been finished. In such states each player receives a payoff, such that their sum is zero, and therefore it

is enough to know the payoff of the first player. In some typical zero-sum games, the value +1 assigned to the player in case of win, -1 in case of lose, and 0 in case of draw.

Game trees can be used to represent games. The nodes represent the states of the game, while the arcs represent the available moves. In the game tree two kinds of nodes are used to represent the decision situations of the two players (A and B). At the root node player A has decision. The leaves represent last states with their payoff values (for player A).

In every two player, zero sum, deterministic game with perfect information there exists a perfect strategy for each player that guarantees the at least result in every instance of the game. The most fundamental result of game theory is the minimax theorem and the minimax algorithm. The theorem says: If a minimax of one player corresponds to a maximin of the other player, then that outcome is the optimal for both players.

#### 2.3.2.1 Minimax Algorithm

In minimax theorem players adopt strategies which maximize their gains, while minimizing their losses. Therefore, the solution is the optimal for each player that she/he can do for him/herself in the face of opposition of the other player. These optimal strategies and the optimal payoff can be determined by the minimax algorithm. It uses simple recursive functions to compute the minimax values of each successor state [18]. Algorithm 2 represents the way that minimax algorithm works (it is recalled from [18,19]), see also Figure 2 for an example.

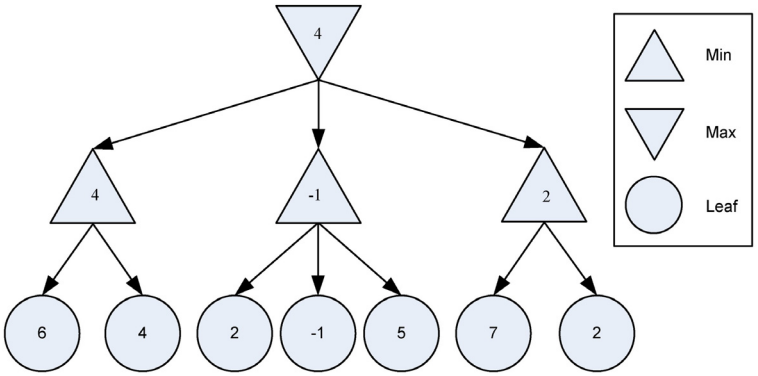

Figure 2: A minimax tree.

*Algorithm 2 (MINIMUM and MAXIMUM)*

```
1.    function MAXIValue(Node)
2.    begin
3.    if Node is leaf then
4.        return the value of Node
5.    end if
6.    else
7.       initiate v = -∞
8.       for every child Node_i of Node do
9.           set v= maximum{v, MINIValue(Node_i)}
10.      end for
11.   end else
12.   return v
13.   end MAXIValue
```

```
1.    function MINIValue(Node)
2.    begin
3.    if Node is a leaf then
4.       return the value of Node
5.    end if
6.    else
7.       initiate v = +∞
8.       for every child Node_i of Node do
9.          set v= minimum{v, MAXIValue(Node_i)}
10.      end for
11.   end else
12.   return v
13.   end MINIValue
```

#### 2.3.2.2 Alpha-Beta Pruning

The minimax algorithm evaluates every possible instance of the game, and thus to compute the value of the game, i.e. its optimal payoff, takes usually exponential time on the length of the instances of the game. To overcome on this issue special cut techniques can be used. The alpha-beta pruning helps find the optimal values without looking at every node of the game tree. While using minimax, some situations may arise when searching of a particular branch can safely be terminated. So, while doing search, these techniques figure out those nodes that do not require to be expanded [18,19].

The way that this algorithm works can be described as below.

- Max-player cuts off search when she/he knows that Min-player can force a clear bad (for the first player, i.e. for Max-player) outcome.

- Min-player cuts off search when she/he knows that Max-player can force a clear good (for Max-player) outcome.

- Applying alpha-pruning (beta-pruning) means the search of a branch is stopped because a better opportunity for Max-player (Min-player) is already known elsewhere. Applying both of them is called alpha-beta pruning technique.

These algorithms, shown in Algorithm 3, are recalled from [18,19]. Figure 3 shows an example of beta-pruning, when $\beta$ becomes smaller than or equal to $\alpha$, we can stop expanding the children of N.

*Algorithm 3 (MINIMUM and MAXIMUM PRUNING)*

```
1.    function ALPHAPrune(Node, α, β)
2.    begin
3.    if Node is leaf then
4.       return the value of Node
5.    end if
6.    else
7.       initiate v = -∞
8.       for every child Node_i of Node do
9.          set v=  maximum{v,  BETAPrune(Node_i, α, β)}
10.         if v is greater or equal to β then return v
11.         set α = maximum{α,v}
12.      end for
13.   end else
14.   return v
15.   end ALPHAPrune
```

```
1.    function BETAPrune(Node, α, β)
2.    begin
3.    if Node is a leaf then
4.       return the value of Node
5.    end if
6.    else
7.       set v = +∞
8.       for every child Node_i of Node do
9.          set v=  minimum{v,  ALPHAPrune(Node_i, α, β)}
10.         if v is less or equal to α then return v
11.         set β = maximum{β,v}
12.      end for
13.   end else
14.   return v
15.   end BETAPrune
```

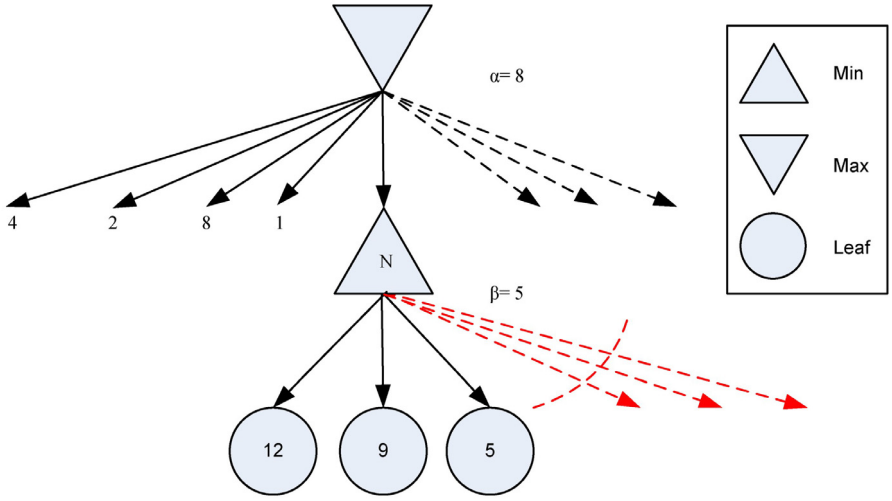


Figure 3: A beta-pruning example.

### 2.3.3 Expression Trees

In every field of mathematics, computer science and also in other sciences, various formulae are used to describe (some phenomena of) the world. In these expressions, usually, binary and unary operators are used. Mathematical, logical etc. expressions are usually displayed by tree graphs. The structure of the expression can easily been understood following the structure of the graph: The main operator is located at the root of the tree (i.e. it is the topmost vertex of the tree graph). Nodes with unary operators have exactly one child, while vertices representing binary operators have exactly two children. To evaluate an expression, in a way learned in school, one should start from the leaves of the tree using a bottom-up strategy. The values given at the leaves are used to evaluate each subformula and, finally, the whole, original formula gets its value. In some cases, e.g., based on associative property of some operations, more than two children of a node are allowed.

### 2.3.4 Boolean Logic

The classical, two-valued logic is well known base of almost all sciences. It was mathematically formalized by Boole in the end of the XIX century, thus the name Boolean logic (and Boolean algebra) is used. There are two values (called truth-

values): true and false, sometimes interpreted as yes and no, denoted also by 1 and 0, or by T and $\perp$, respectively. This mathematical logic is applied in electronic switching circuits, and therefore, it gives the base of all our digital machines including electronic computers (due to a principle of J. von Neumann). Readers not familiar to classical logic are referred to the textbook [1], or similar materials. Here we mention only some parts briefly.

The syntax of Boolean logic is usually defined by an inductive way. The base of this induction goes by the atomic formulae: There are infinitely many Boolean (also called propositional) variables. Each of them is counted as an atomic formula. The signs T and $\perp$, as logic constants, are also belonging to this set. We prefer to use their numerical value: 1 and 0, instead of them, since in this way, they could be more easily to be generalized to fuzzy logic systems.

The inductive step is based on logical connectives. Usually, in Boolean logic the operators conjunction, disjunction and negation are defined (this latter operator is unary, all others are binary). Implication is also frequently defined and used, since it has a strong relation to logical deduction. When A and B are two logical formulae, then each of their conjunction ($A \wedge B$), disjunction ($A \vee B$) and implication ($A \rightarrow B$) is also a logical formulae. The formulae A and B are called the main subformulae of the original formula. The negation $\neg A$ of a logical formula A is also a logical formula.

Moreover, every logical formula can be obtained from atomic formulae by a finite number of inductive steps.

We note that in engineering textbooks the multiplication operator stands for conjunction (logical AND), and addition operator for disjunction (logical OR). The former one already gives an idea how the operation can be extended by a larger class of truth values at, e.g., the product logic.

Logical formula can be represented by its formula tree, building it according to the iterative definition.

Having logical formula and the assigned truth-values to the appearing propositional variables, one can evaluate the formula using the semantic rules of Boolean logic:

- A conjunctive formula ($A \wedge B$) is true if and only if both A and B are true.
- A disjunctive formula ($A \vee B$) is true if and only if at least one of the formulae A and B is true.
- An implication formula ($A \rightarrow B$) is true if and only if A is false or B is true.
- A negation formula $\neg A$ is true if and only if the formula A is false.

In each other case the formula evaluates to false. In Boolean logic there is no other option. However, based on the listed semantic rules, there are cases, when knowing only the values of one of the two main subformulae is enough to know the truth-value of the original formula. This phenomenon leads to the short circuit evaluation technique. In this way, often, we do not need a full evaluation, i.e., some of the vertices of the formula tree may not need to be visited; their values may not have any

effect on the final value of the formula. These types of evaluation techniques are highly used in programming languages helping the computation be (nearly) optimal. In the next part we show the two basic forms of these, also called, short circuit evaluations.

### 2.3.5 Short Circuit Evaluation in Boolean Logic

Fast evaluation based on short circuit technique can be used for time saving and, in some cases; it is also used for safety reasons [15]. This technique of evaluation in logic is closely connected to alpha-beta pruning techniques in game theory [17,18,19,20]. In some programming languages the symbols && and || are used for the logical operations AND and OR, respectively. To evaluate these operations short circuit evaluations can be used as we detail below:

- At AND, if it is known that all the conditions/arguments must be true in order to make it true, it is not necessary to check all the conditions if one of them is already known to be false.

- At OR, if it is known that one condition is true, then there is no need to check the value of the other conditions.

See also Figure 4, where examples are shown for both of these cut techniques (values of the leaves are given, by associativity more than two children are allowed, in this way shortening the formula tree).

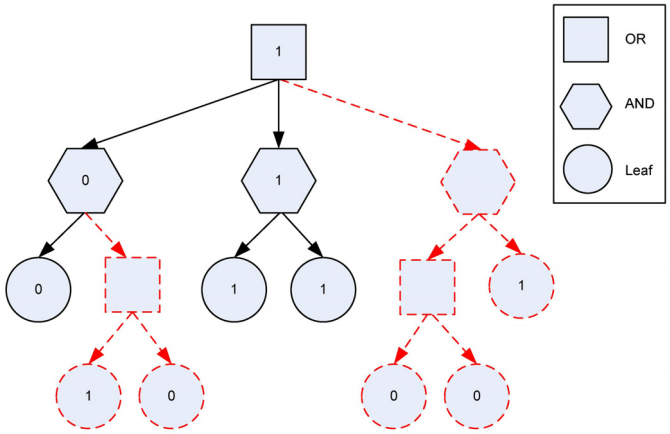

Figure 4: An example of applying short circuit evaluation.

As shown in this example, when evaluating the AND nodes, if a zero value (0) returned back from one of the children nodes, then we cut the next connected nodes and leaves and there is no need to evaluate them since their value will not affect the final result. The same technique is applied in evaluating the OR nodes, the idea here is to cut-off when the value one (1) returned back from a connected node or leaf. The final result of evaluating that node will be one (1) regardless the value of the remaining connected children nodes. See also Algorithm 4 below which describes how the idea of the short circuit evaluation is used. We allow not only binary conjunctions and disjunctions and thus, we may assume that they are alternating by levels.

*Algorithm 4 (OR and AND PRUNING)*

```
1.    function ORPrune (Node)
2.    begin
3.    if Node is leaf then
4.       return the value of Node
5.    end if
6.    else
7.      initiate v = 0
8.      for every child Node_i of Node do
9.        while v not equal to 1  do
```

```
10.             set v= v + ANDPrune(Node_i)
11.          end while
12.       end for
13.    end else
14.    return v
15.    end ORPrune
```

```
1.     function ANDPrune (Node)
2.     begin
3.     if Node is a leaf then
4.        return the value of Node
5.     end if
6.     else
7.        initiate v = 1
8.        for every child Node_i of Node do
9.           while v not equal to  0  do
10.             set v = v · ORPrune(Node_i)
11.          end while
12.       end for
13.    end else
14.    return v
15.    end ANDPrune
```

### 2.3.6 Gödel type Fuzzy Logic

This system has been introduced by Kurt Gödel in 1932 [4]. The possible truth values are real numbers from the closed unit interval [0,1], i.e., numbers $0 \leq x \leq 1$. The designated value, that is the value that is counted as true, is usually only the value 1. Gödel defined four main connectives for this system (implication, negation, disjunction and conjunction denoted to by the symbols $\rightarrow$ , $\neg$, $\vee$,and $\wedge$, respectively). Their syntax is the same as in Boolean logic, and their semantics are defined in the following way [5,9]:

$$|A \rightarrow B| = 1, \text{ if } |A| \leq |B|; \qquad (2.1a)$$

$$|A \rightarrow B| = |B|, \text{ otherwise} \qquad (2.1b)$$

$$|\neg A| = 1, \text{ if } |A|=0; \quad (2.2a)$$

$$|\neg A| = 0, \text{ otherwise} \quad (2.2b)$$

$$|A \vee B| = \max \{|A|, |B|\} \quad (2.3)$$

$$|A \wedge B| = \min \{|A|, |B|\} \quad (2.4)$$

Note that, for simplicity, we use letters A, B for the variables and, also, for their values, without causing any misunderstanding. The system is infinitely many valued, and it fulfills the axioms of intuitionistic logic with one additional law, namely, the law of chain: The formula $((A \rightarrow B) \vee (B \rightarrow A))$ has value 1 independently of the values of subformulae A and B.

Expressions and so, expression trees in Gödel logic are very similar to Boolean expressions. The difference is that here the values at leaves (i.e., the truth values of variables) is not restricted to the Boolean set {0,1}, but any real number between 0 and 1 (inclusively) can be used.

**2.3.7 Lukasiewicz type Fuzzy Logic**

The idea of considering intermediate truth values rather than only the classical set of truth values {0,1} has been used extensively by Lukasiewicz and others since the 1920's. From historical point of view we mention that Lukasiewicz made three-valued and four-valued systems first. Later he extended the system to arbitrary-many ($n \geq 2$) truth-values up to infinitely many [7,11,13]. Among the various many-valued logics, the Lukasiewicz logic with infinitely many truth-values is one of the most attractive candidates of fuzzy logic [24]. In this system all real numbers of the closed interval [0,1] are allowed to be truth-value. The language has two primitive logical connectives, i.e., $\{\rightarrow, \neg\}$, where "$\rightarrow$" is the Lukasiewicz implication and "$\neg$" is the

usual negation operation. Based on those two operations the other connectives, the Lukasiewicz type conjunction (&) and disjunction (+) were also defined.

The syntax of this logic is entirely the same as the syntax of Boolean logic: the negation is unary, the implication ($\rightarrow$), the conjunction (&) and the disjunction (+) are binary operators. Formula trees are also defined and used accordingly.

The semantics of Lukasiewicz logic is defined in the following way. Generally, the variables and the constants may have any values from the interval [0,1] including the classical two values. The truth-values of formulae with connectives can be computed from the value of their main subformulae [6,9,24].

$$|\neg A| = 1 - |A| \tag{2.5}$$

$$|A \rightarrow B| = \min(1 - |A| + |B|, 1) \tag{2.6}$$

$$|A \,\&\, B| = \max(|A| + |B| - 1, 0) \tag{2.7}$$

$$|A + B| = \min(|A| + |B|, 1) \tag{2.8}$$

Lukasiewicz logic with the same semantics can also be used having a finite number of truth values. In these systems the values are (for each integer $k > 2$): $0 = 0/(k-1)$, $1/(k-1)$, ..., $(k-1)/(k-1) = 1$. In the special case $k = 2$, one get back the classical Boolean connectives working on the classical truth values. The Lukasiewicz-type conjunction and disjunction have the names “bounded product” and “bounded sum”, respectively.

### 2.3.8 Product Logic

One of the most popular fuzzy logic systems is the product logic. It was described by explicit mathematical way in [10]. It is a logic with a natural many-valued semantics interpreting conjunction as multiplication on the real unit interval [0, 1]. By restricting the values to the traditional binary set {0,1}, the product, actually, the same as the usual conjunction. This logic is considered, along with the other two most significant fuzzy logics, the Lukasiewicz and the Gödel logics, to be one of the fundamental t-norm based fuzzy logics due to the fact that every continuous t-norm is locally isomorphic to either the Product, Gödel or Lukasiewicz t-norm.

In product logic all real numbers of the closed interval [0,1] are allowed to be a truth-value.

The syntax of this logic is entirely the same as the syntax of Boolean logic: the negation is unary, the implication, the conjunction and the disjunction are binary operators. Formula trees are also defined and used in a similar manner.

The semantics of product logic is defined in the following way. Generally, the variables and the constants may have any values from the interval [0,1] including the classical two values. The truth-values of formulae with connectives can be computed from the value of their main subformulae [9,10].

$$|\neg A| = 1 - |A| \tag{2.9}$$

$$|A \rightarrow B| = 1, \text{ if } |A| \leq |B|; \tag{2.10a}$$

$$|A \rightarrow B| = |B| / |A|, \text{ otherwise} \tag{2.10b}$$

$$|A \wedge B| = |A| \cdot |B| \tag{2.11}$$

$$|A \vee B| = |A| + |B| - |A| \cdot |B| \tag{2.12}$$

One can easily prove that both conjunction and disjunction are associative in this logic, therefore, to make our work more efficient, we allow multiple (more than two) children of conjunction and disjunction nodes in the expression trees, similarly to the case of Boolean logic and Gödel type logic.

In the product logic system, the conjunction is the product of the values of the arguments, and the name of disjunction is "algebraic sum". This type of conjunction and disjunction look more like operations of a probabilistic system [9]. Assuming that the values A and B are independent we get the result of their common occurrence.

$$P(|A| \text{ and } |B|) = P(|A|)\,P(|B|)$$

$$P(|A| \text{ or } |B|) = 1 - P(|\neg A| \text{ and } |\neg B|) = 1 - P(|\neg A|)P(|\neg B|)$$

$$= 1 - (1 - P(|A|))\,(1 - P(|B|)) = P(|A|) + P(|B|) - P(|A|)P(|B|)$$

The value of $A \rightarrow B$ is the maximal probability of B if A is true.

For the many-valued logic systems that are defined in the previous subsections, and in a given (fixed) evaluation, the formula and its leaves are fixed, the leaves have labels from the interval [0,1]. Then, the task is to compute the (truth) value of the whole formula. This can be done with a bottom-up strategy. However, as in the case

of Boolean logic, we may not need to compute the value of each subformula to know the final result. In chapter four we show pruning techniques that can be used in fast evaluations of these formula trees.

# Chapter 3

# STRATEGIES TO FAST EVALUATION OF BOOLEAN EXPRESSIONS AND EXTENDED GAME TREES

## 3.1 Introduction

In this chapter, we consider special formula/expression trees that can be considered as a type of extensions of the game trees. Special extensions, as a mixture of decision and game-trees were discussed in [19]. Here, by a further step, such extensions of game-trees are investigated in which some operations may not be directly connected to games, but with usual (mathematical or logical) expressions. We use a bounded set of payoff values 0 and ±1 at the leaves of the trees. In some cases, it is not necessary to know the value of every descendant to evaluate a node of the tree, these cases leads to various pruning techniques that simplify the evaluation of the tree.

In the following subsections, various algorithms are proposed to speed up the evaluation of these special trees in which there are specific operations used. Apart from the usual min and max operations (that can be seen as AND and OR by restricting the values to the Boolean set, i.e. to {0,1}) we use the operations multiplication (that can also be seen as AND on the Boolean set) and sum (that usually can be seen as binary addition, i.e. OR on the Boolean set), as well. We keep only the sign of the sum to have the result inside the domain {-1, 0, +1}. The proposed modified minimax, sum and product pruning, minimax with sum, and minimax with product algorithms are described in the following sections in detail. As

an enhancement for the proposed algorithms, we also show that reordering the branches of the tree may lead to an even faster exact evaluation.

## 3.2 Modified Alpha-Beta Pruning Algorithm

We start with an obvious modification of Algorithm 3. Since the set of payoff values is bounded, we can modify the alpha-beta pruning algorithm to do a cut when the maximum (+1) or the minimum (-1) value of the set is already found in ALPHAPrune and BETAPrune, respectively, as shown in Algorithm 5. Line 9 of Algorithm 3 is modified in both ALPHAPrune and BETAPrune functions to test if the values +1 and -1 (the possible maximum and minimum, respectively) have been found in the evaluated node, thus there is no need to visit the remaining successors for that node: a cut can be done. Figure 5 shows an example for the usage of the modified pruning algorithm.

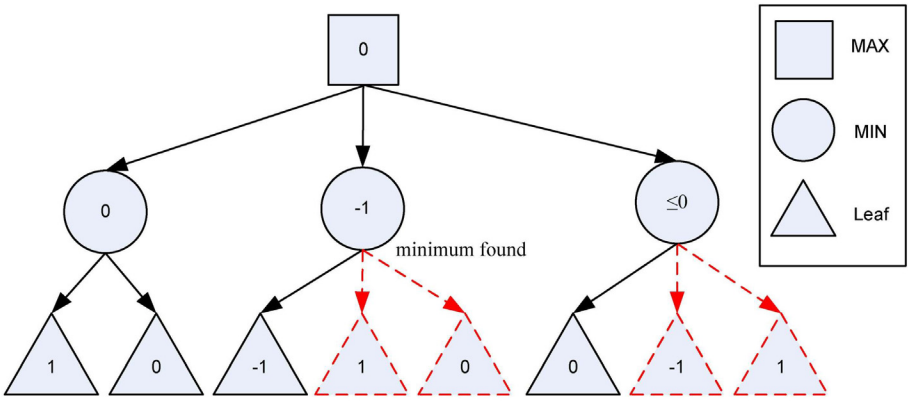


Figure 5: A modified minimax alpha-beta pruning example. Only four leaves out of eight are explored to evaluate the tree.

*Algorithm 5 (Modified MINIMUM and MAXIMUM PRUNING)*

```
1.    function MAXIPrune(Node, α, β)
2.    begin
3.    if Node is leaf then
4.       return the value of Node
5.    end if
6.    else
7.      initiate v = -2
8.      for every child Node_i of Node do
9.        while v is less than +1  do
10.         set v=  maximum{v,  MINIPrune(Node_i, α, β)}
11.         if v is greater than or equal to β then return v
12.         end if
13.         set α = maximum{α,v}
14.       end while
15.     end for
16.   end else
17.   return v
18.   end MAXIPrune
```

```
1.    function MINIPrune(Node, α, β)
2.    begin
3.    if Node is a leaf then
4.       return the value of Node
5.    end if
6.    else
7.      initiate v = +2
8.      for every child Node_i of Node do
9.         while v is greater than -1  do
10.           set v=  minimum{v,  MAXIPrune(Node_i, α, β)}
11.           if v is less than or equal toα then return v
12.           end if
13.           set β = maximum{β,v}
14.        end while
15.     end for
16.   end else
17.   return v
18.   end MINIPrune
```

## 3.3 Sum and Product Pruning Algorithm

Now let us consider a new type of expression in which product operations follow the additions and vice versa, e.g., expressions of the form abc+de+fghi and also more complex expressions with these two operations appearing in the expression tree. Remember that both the possible values of the variables (leaves) and of the expressions (other nodes) are restricted to the set {-1, 0, +1} and thus both the sum and product functions can have these three output values.

Using the fact that both addition and multiplication operations are associative, any positive number of operands are allowed, i.e, any positive number of children of these nodes, including only 1 child. Ideas similar to the short circuit evaluation can be used to cut during the evaluations of sum and product (multiplication) functions as they are described in Algorithm 6, see also, e.g. Figure 6 for examples.

The possible cuts that are applied here are the following: For the sum function, if the absolute value of the actual value is greater than the remaining successors to visit, then pruning is applied since there is no need to evaluate the remaining nodes: the result is already known: the return value is 1 if the sum value greater than zero, and -1 if the sum value is less than zero. This can be done using the while loop in line 9 in the function SUMPrune. At the product operator a zero-cut is done, since 0 is the zero element of the multiplication, as it is written in the function MULPrune.

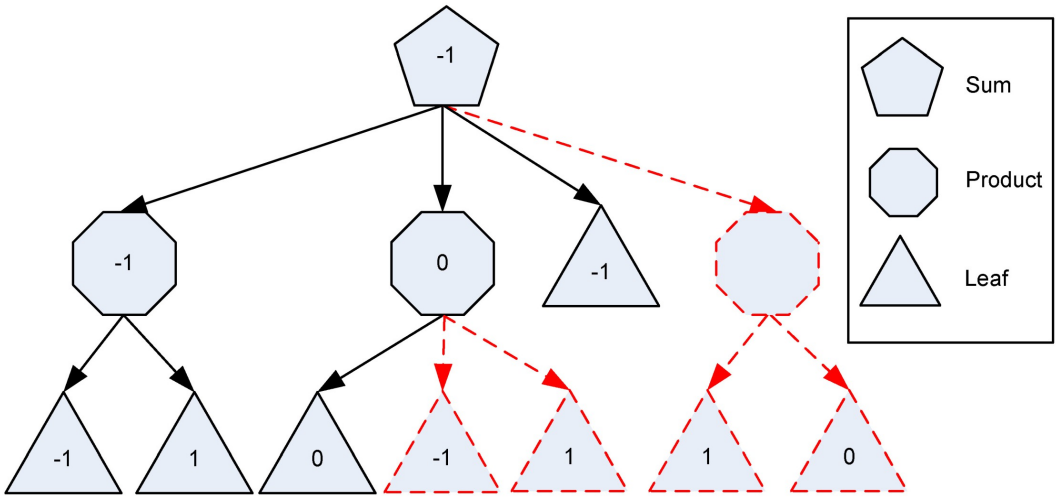


Figure 6: A sum and product pruning example. Only three vertices and four leaves (total 7) out of four vertices and eight leaves (total 12) are explored and evaluated to get the final result at the root.

*Algorithm 6 (MUL and SUM PRUNING)*

```
1.    function SUMPrune (Node)
2.    begin
3.    if Node is leaf then
4.       return the value of Node
5.    end if
6.    else
7.     initiate v = 0
8.     for every child Node_i of Node do
9.       while |v| is less or equal to (the number of remaining nodes to visit do)
10.         set v= v + MULPrune(Node_i)
11.      end while
12.    end for
13.   end else
14.   if v is greater than 0 then
15.       return 1
16.   end if
17.   if v is less than 0 then
18.       return -1
19.   end if
20.   else
21.       return 0
22.   end else
23.   end SUMPrune
```

```
1.    function MULPrune (Node)
2.    begin
3.    if Node is a leaf then
4.       return the value of Node
5.    end if
6.    else
7.      initiate v = 1
8.      for every child Node_i of Node do
9.          while v is not equal to 0  do
10.             set v = v * SUMPrune(Node_i)
11.         end while
12.     end for
13.   end else
14.   return v
15.   end MULPrune
```

## 3.4 Minimax-Product Pruning Algorithm

In minimax-product case, the evaluation of the tree can be speed uped by applying the zero-cut at product layer when a node with zero value is returned from one of the connected successors. In addition to this, similarly to the usual alpha-beta pruning, pruning can be applied in max and min layers. Example for these kinds of pruning are shown in Figure 7 where two cuts are applied. The first one is a zero-cut when the value zero is found in a leaf connected to product node in the product layer. The second one is applied in the second node in min layer when a zero value (β) is returned back from the product layer, which is less or equal to the maximum value (α) that is found in the first node.

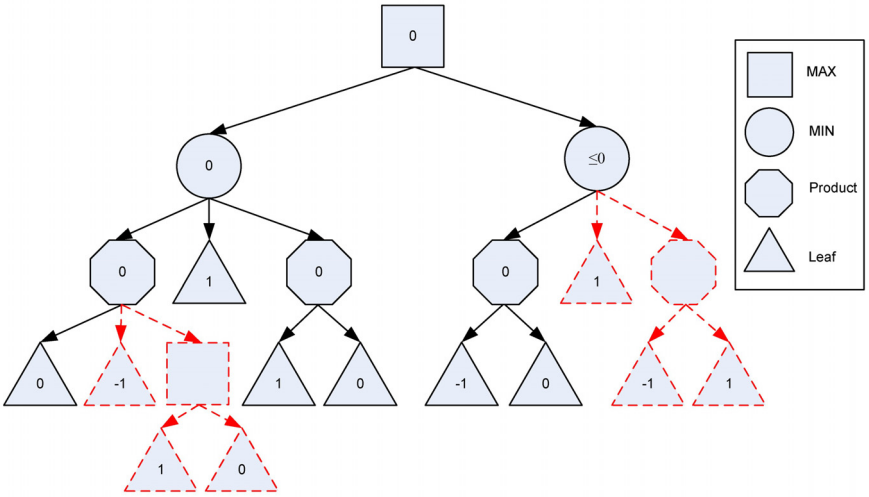


Figure 7: A minimax and product pruning example. Only six vertices and six leaves (total 12) out of eight vertices and twelve leaves (total 20) are explored and evaluated to get the final result at the root.

The proposed process for pruning is shown in Algorithm 7. The same MULPrune function that proposed previously in Algorithm 6 is used here with MAXIPrune and MINIPrune functions. (They can easily be modified having other order of layers in the expression tree.)

*Algorithm 7 (MUL, MININIMUM and MAXIMUM PRUNING)*

```
1.    function MAXIPrune(Node, α, β)
2.    begin
3.    if Node is leaf then
4.        return the value of Node
5.    end if
6.    else
7.        initiate v = -2
8.        for every child Node_i of Node do
9.            while v is less than +1 do
10.               set v= maximum{v, MINIPrune(Node_i, α, β)}
11.           end while
12.           set α = maximum{α,v}
13.       end for
14.   end else
15.   return v
16.   end MAXIPrune
```

```
1.  function MINIPrune(Node, α, β)
2.  begin
3.  if Node is a leaf then
4.      return the value of Node
5.  end if
6.  else
7.     initiate v = +2
8.     for every child Node_i of Node do
9.        while v is greater than -1  do
10.          set v=  minimum{v,  MULPrune(Node_i, α, β)}
11.       end while
12.       if v is less than or equal α then
13.           return v
14.       end if
15.       set β = maximum{β,v}
16.    end for
17. end else
18. return v
19. end MINIPrune
```

```
1.  function MULPrune(Node, α, β)
2.  begin
3.  if Node is a leaf then
4.     return the value of Node
5.  end if
6.  else
7.     initiate v = 1
8.     for every child Node_i of Node do
9.        while v is not equal 0 do
10.          set v = v * MAXIPrune(Node_i, α, β)
11.       end while
12.    end for
13. end else
14. return v
15. end MULPrune
```

## 3.5 Minimax-Sum Pruning Algorithm

In expression trees with three layers (max, min and sum), we use the idea of sum short circuit evaluation initiated in subsection 3.3.

As mentioned before, the sum function has three output values (-1, 0, +1). When the absolute value of the sum of the visited successors is greater than the number of remaining successors, then the cut can be done and if the sum is positive, then +1 is returned; if the sum is negative, then -1 is returned.

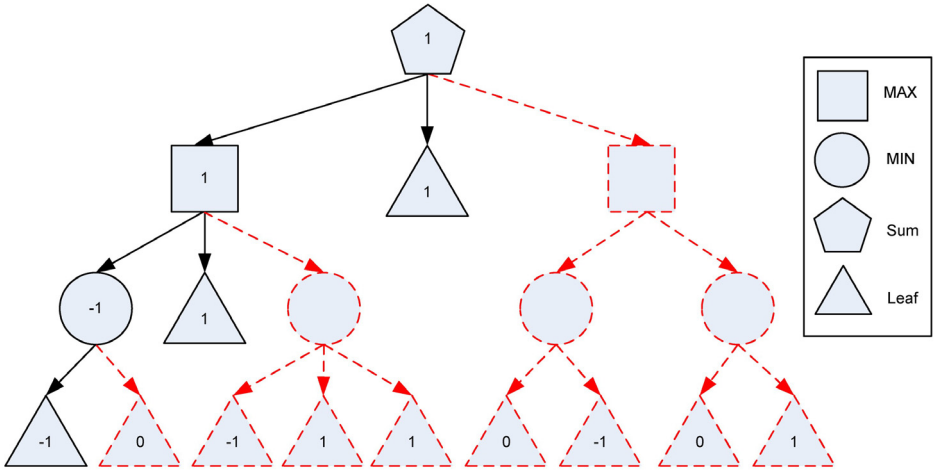


Figure 8: A minimax and sum pruning example. Only three vertices and three leaves (total 6) out of seven vertices and eleven leaves (total 18) are explored and evaluated to get the final result at the root.

Furthermore, similar alpha and beta pruning for min and max functions can be done as in the previous algorithms to cut and speed up the evaluation of the tree (see Algorithm 8). In Figure 8, in the minimum layer a pruning is applied when the minimum value (-1) has been found, and after that in maximum layer again, when the maximum value (+1) has been found. The sum short circuit is applied too; since the absolute value of the sum function (+2) is greater than the remaining nodes to evaluate (we have one node remain to evaluate, which is less than 2).

*Algorithm 8 (ADD, MINIMUM and MAXIMUM PRUNING)*

1. function MAXIPrune(Node, α, β)
2. begin
3. if Node is leaf then

```
4.          return the value of Node
5.      end if
6.      else
7.         initiate v = -2
8.         for every child Node_i of Node do
9.             while v is less than +1  do
10.               set v=  maximum{v,  MINIPrune(Node_i, α, β)}
11.            end while
12.            set α = maximum{α,v}
13.        end for
14.     end else
15.     return v
16.     end MAXIPrune
```

```
1.      function MINIPrune(Node, α, β)
2.      begin
3.      if Node is a leaf then
4.          return the value of Node
5.      end if
6.      else
7.          initiate v = +2
8.          for every child Node_i of Node do
9.              while v is greater than -1  do
10.                set v=  minimum{v,  ADDPrune(Node_i, α, β)}
11.             end while
12.             if v is less than or equal α then return v
13.             end if
14.             set β = maximum{β,v}
15.         end for
16.     end else
17.     return v
18.     end MINIPrune
```

```
1.      function ADDPrune(Node, α, β)
2.      begin
3.      if Node is a leaf then
4.         return the value of Node
5.      end if
6.      else
7.        initiate v = 0
```

```
8.          for every successor Node_i of Node do
9.            while |v| is less than or equal (the number of remaining nodes to visit do)
10.                set v= v + MAXIPrune(Node_i, α, β)
11.           end while
12.         end for
13.       end else
14.       if v is greater than 0 then
15.           return 1
16.       end if
17.       if v is less than 0 then
18.           return -1
19.       end if
20.       else
21.           return 0
22.       end else
23.       end ADDPrune
```

## 3.6 Reordering the Branches of the Trees

Notice that each of the used operations is commutative, and thus, the result of the operand does not depend on the order of the children branches. Therefore, in addition to the above proposed algorithms to speed up the evaluation of the Boolean expression trees with the presented logical and mathematical operations, a reordering technique can be applied on these kinds of trees before starting the evaluation.

The intuitive idea behind the reordering is that shorter branches can be computed faster. Therefore, the aim is to move the node that is the root of a subtree having least depth to the left side to be evaluated first. The process of reordering must be started from the lowest layers and goes up to the root. Then, after the reordering is done, the evaluation process can be started where the previously mentioned pruning techniques can be applied to speed up the evaluation.

### 3.6.1 Reordering the branches of Boolean expressions

In this subsection we show how the evaluation of a Boolean expression tree with two operations can be done in a more efficient way. Figure 9 shows how the evaluation of the tree in Figure 4 is done after reordering its branches. After applying the reordering process, only two vertices (operators) out of six are evaluated to get the final result in the root. While only two of the leaves are also used instead of a total of eight during the evaluation process using the operators AND and OR. The process of reordering the tree branches is described in Algorithm 9. The same idea can be applied if we have different types of operators (e.g., sum).

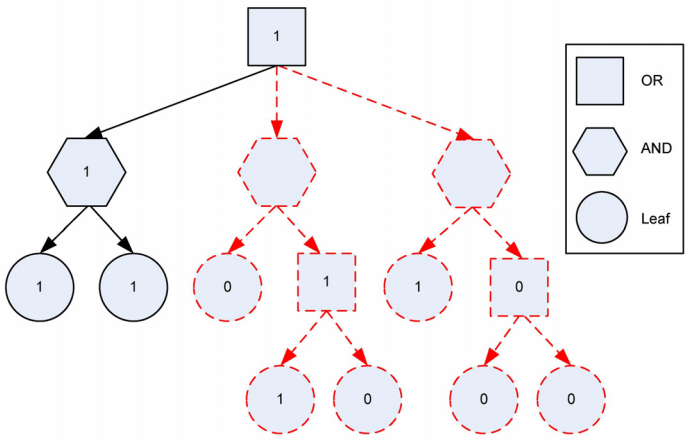


Figure 9: An example of applying reordering and short circuit algorithm. Only two vertices and two leaves (total 4) out of six vertices and eight leaves (total 14) are explored and evaluated after the reordering and pruning process.

*Algorithm 9 (AND, and OR Tree Reordering)*

1. function ORReorder(Node)
2. begin
3. if Node is leaf then
4. move Node to the left side
5. end if
6. else
7. for every child $Node_i$ of Node do
8. ANDReorder($Node_i$)
9. count the number of connected leaves and nodes

```
10.             move the node Node_i with less leaves and nodes to the left side
11.         end for
12.     end else
13.     end ORReorder
```

```
1.      function ANDReorder(Node)
2.      begin
3.      if Node is leaf then
4.          move Node to the left  side
5.      end if
6.      else
7.          for every child Node_i of Node do
8.              ORReorder(Node_i)
9.              count the number of connected leaves and nodes
10.             move the node Node_i with less leaves and nodes to the left side
11.         end for
12.     end else
13.     end ANDReorder
```

### 3.6.2 Reordering and Pruning Complex Trees

In this subsection we use the reordering technique for expression trees with more than two operators and show that it can be used very efficiently in these cases as well.

In the first example of this subsection an expression tree is shown (see Figure 10 below). This tree includes four different operators (sum, multiplication, max, and min) in such a way that in the same level only the same operator is used. Since, in this case, the order of the operators are fixed (as at game trees), the evaluation functions call each other in a predefined order. To evaluate the tree of that example, without applying the reordering technique and pruning algorithms mentioned and detailed in the previous subsections, 22 vertices and 25 leaves must be explored and evaluated.

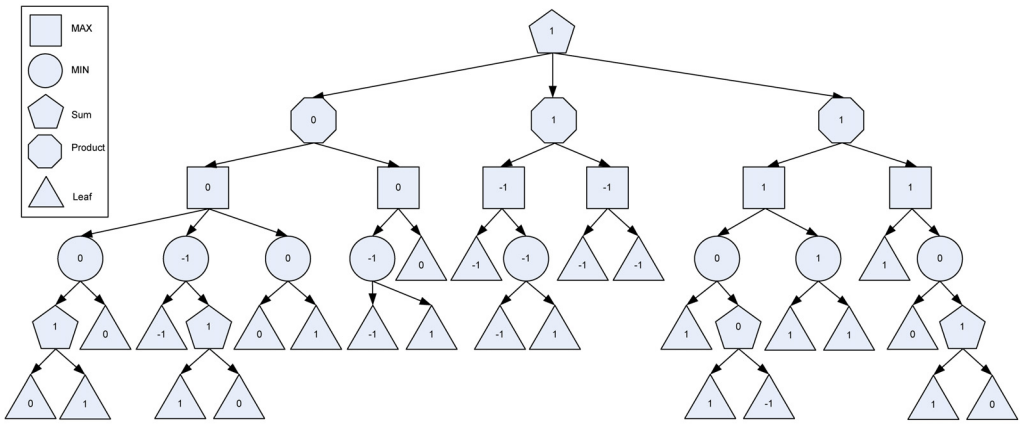


Figure 10: An expression tree with sum, multiplication, max, and min operators. The tree contains twenty two vertices and twenty five leaves (total 47).

Figure 11 shows the same tree evaluated after applying the proposed pruning algorithms without reordering the branches. The same result of evaluation is returned back to the root, but this was by exploring and evaluating only 19 vertices and 17 leaves (total 36) out of 22 vertices and 25 leaves (total 47).

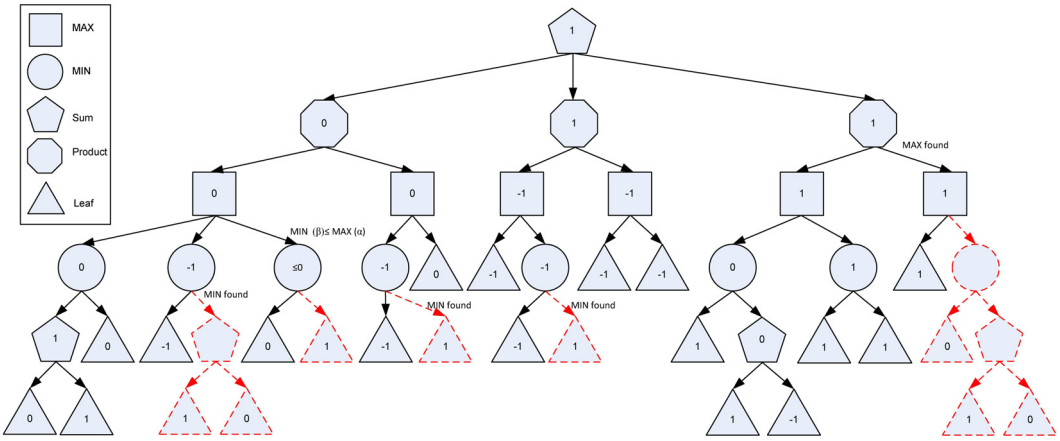


Figure 11: The expression tree of the example of Figure 10 is evaluated by the proposed pruning techniques. After applying the pruning algorithms without reordering the branches, nineteen vertices and seventeen leaves remained (total 36) out of twenty two vertices and twenty five leaves (total 47).

Figure 12 shows the same tree evaluated after reordering the branches and then applying the proposed pruning algorithms. The same result of evaluation is returned back to the root, but this was by exploring and evaluating only 12 vertices and 7

leaves (total 19) out of 22 vertices and 25 leaves (total 47). This result shows how fast the evaluation process can be after applying the proposed technique to evaluate an expression tree that includes various logical and mathematical operators with some order.

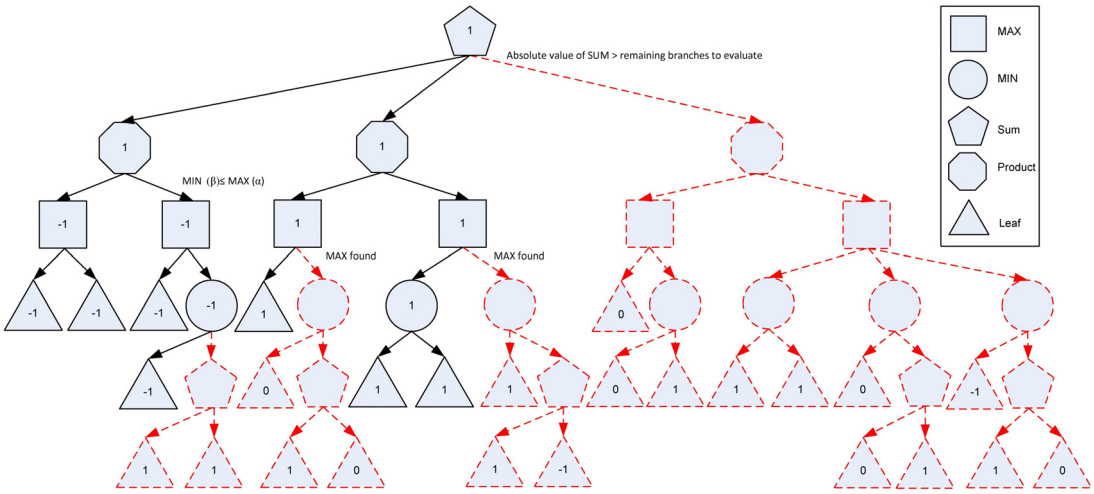


Figure 12: The expression tree of Figures 10 and 11 is evaluated by reordering and pruning. After applying the reordering and then the pruning algorithms, only nine vertices and seven leaves remained (total 16) out of twenty two vertices and twenty five leaves (total 47).

In our other example, shown in Figure 13, the operators have no fixed order in the expression, which maybe much closer to real word applications in some cases. Also, leaves can be found in various levels (i.e., various depths) of the tree. To evaluate the tree without applying the reordering technique and pruning algorithms mentioned above, 11 vertices and 13 leaves must be explored and evaluated (total 24). Using various pruning strategies the number of vertices that must be evaluated is 9, while the number of leaves that must be explored is 7 (total 16 vertices, see Figure 14). The proposed reordering technique together with the pruning algorithms evaluates the expression tree of this example by exploring and evaluating only 5 vertices and 5 leaves (total 9 vertices are visited for the evaluation) as it can be seen in Figure 15.

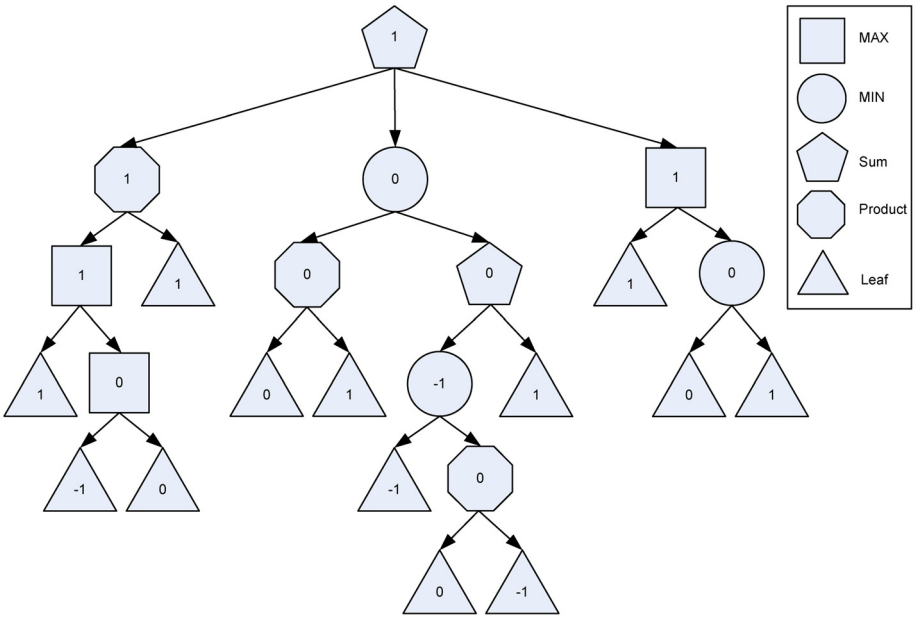


Figure 13: An expression tree with sum, multiplication, max, and min operators in various order. The tree contains eleven vertices and thirteen leaves (total 24).

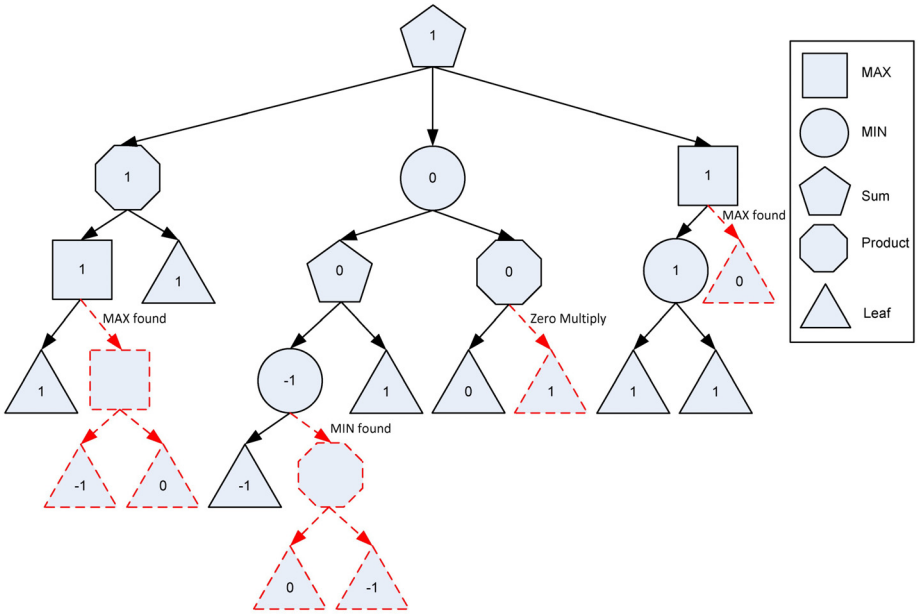


Figure 14: The example of Figure 13 is evaluated by applying the pruning algorithms without reordering the branches: nine vertices and seven leaves evaluated and explored (total 16) out of eleven vertices and thirteen leaves (total 24).

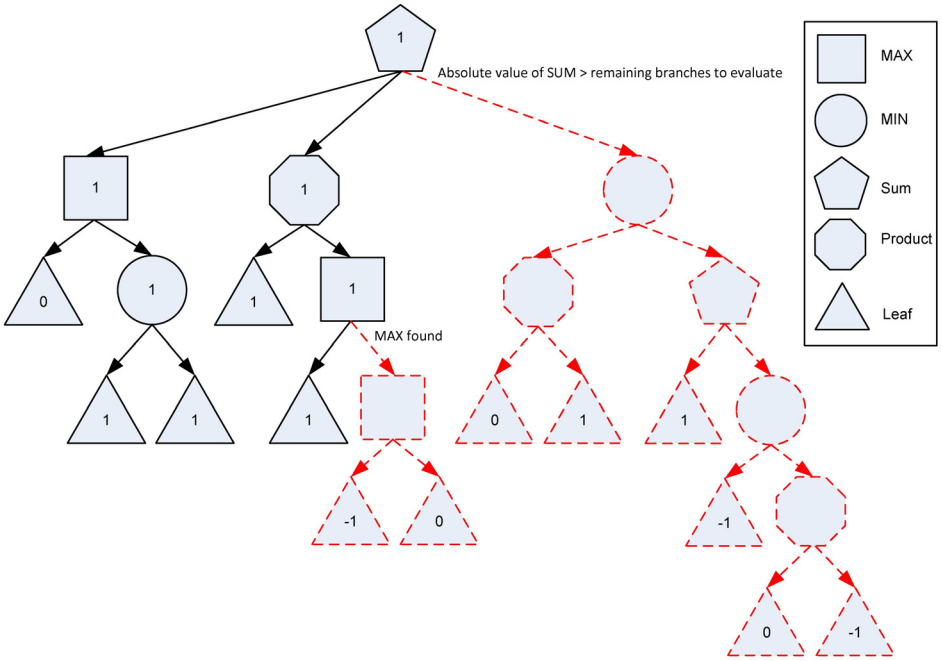


Figure 15: The expression tree of Figures 13 and 14 is evaluated by applying both the reordering and then the pruning algorithms: only five vertices and five leaves remained (total 9) out of eleven vertices and thirteen leaves (total 24).

# Chapter 4

# STRATEGIES TO FAST EVALUATION OF MANY-VALUED LOGIC FORMULAE

## 4.1 Introduction

In the following sections, we show various lazy evaluation techniques to evaluate the expressions in the most known fuzzy logic systems; Gödel, Lukasiewicz, and product logics. By these cut techniques one could effectively reduce the size of the expression trees allowing a much faster method of evaluation. We are dealing with trees with bounded set of truth (or payoff) values: the real numbers of the closed interval [0,1] can be used.

We start this chapter by showing some strategies to fast evaluate the expression trees in Gödel logic system.

## 4.2 Strategies to Fast Evaluation of Gödel Type Logic Formulae

In this section, various algorithms are proposed to speed up the evaluation of Gödel logical connectives previously defined in section 2.3.6. The conjunction and disjunction operations can be seen as AND and OR operators by restricting the values to the Boolean set, i.e., to {0, 1}. In our expression trees, in a similar manner to the Boolean expressions (actually, the syntax of expressions of Gödel logic is the same as the syntax of Boolean expressions), since both AND and OR are associative, without loss of generality we allow multiple children of nodes of these types. Nodes

with negation must have exactly one child, while nodes with implications must have exactly two children, called left child and right child, respectively.

The proposed algorithms to speed up the evaluation of this kind of trees are described below in details in the next subsections.

### 4.2.1 Alpha-Beta Pruning

A form of the classical short circuit evaluation for Gödel logic is exactly the usual alpha-beta pruning. When OR and AND operators are used alternately by levels of the expression tree, alpha-beta pruning can be applied and, actually, by knowing the possible minimal and maximal values, these information can also be used (reaching these values some neighbor branches can be cut without effecting the computed value).

### 4.2.2 Implication Pruning

Evaluating the implication nodes can be speed uped using various techniques. These techniques are based on the possible left and right children of the implication node, and they are described below in details.

#### 4.2.2.1 Conjunction-Disjunction Children Pruning

We start with the case when the left child node is a conjunction and the right child node is a disjunction. The first idea to speed up this evaluation is to evaluate the branches connected to its successors one by one in parallel that is, evaluating the first child of the conjunction node and then the first child of the disjunction node, then the second child of the conjunction node, etc. This technique is very useful in case if conjunction and disjunction nodes are connected to implication node. Figure 16 shows an example of such case.

As shown in Figure 16, conjunction (left child) and disjunction (right child) successors evaluated in parallel (one by one, after each other). After evaluating the second successor of disjunction and conjunction nodes, we have that conjunction is less or equal to 0.3 and disjunction is greater or equal to 0.7 which means that we can cut the other successors and return back the value 1 for implication node. More generally, if it is known that the value of the right child (disjunction in this case) is at least as many as the value of the left child (conjunction in this case), we can eliminate the evaluation of the other brothers and sisters, the implication node has a value 1.

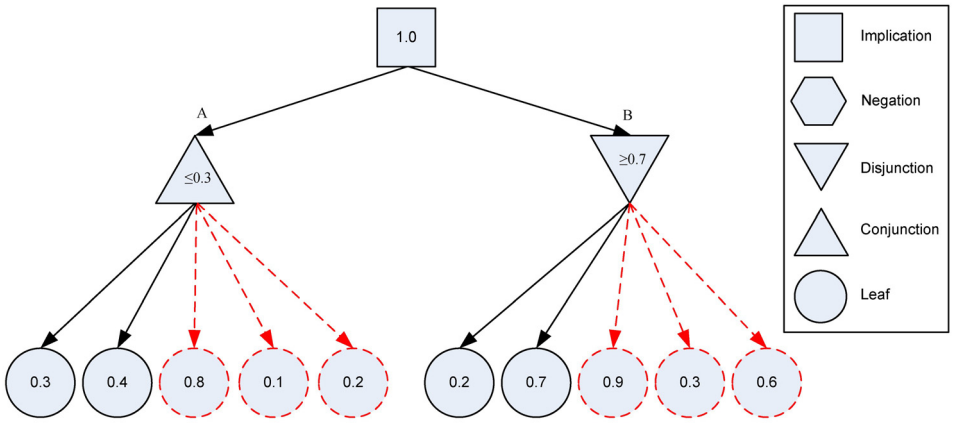


Figure 16: An example of applying the pruning when evaluating an implication node. Two children are connected; a conjunction node to left side and disjunction node to right side. Only four children (out of 10) are explored to get the final result at the root.

#### 4.2.2.2 Disjunction-Conjunction Children Pruning

The second pruning technique is applied when the successors of an implication node are disjunction at left child and conjunction at right child. Figure 17 shows an example of this case.

As shown in Figure 17, the evaluation of the disjunction and conjunction nodes successors starts in parallel (one by one in turns). After evaluating the second successor of disjunction and conjunction nodes we have that disjunction will be always greater or equal to 0.6, while conjunction node will be less or equal to 0.4. Since the value of left child will be less than right child, we can cut all the subsequent children of disjunction node and evaluate only the conjunction node and return back its value to implication node.

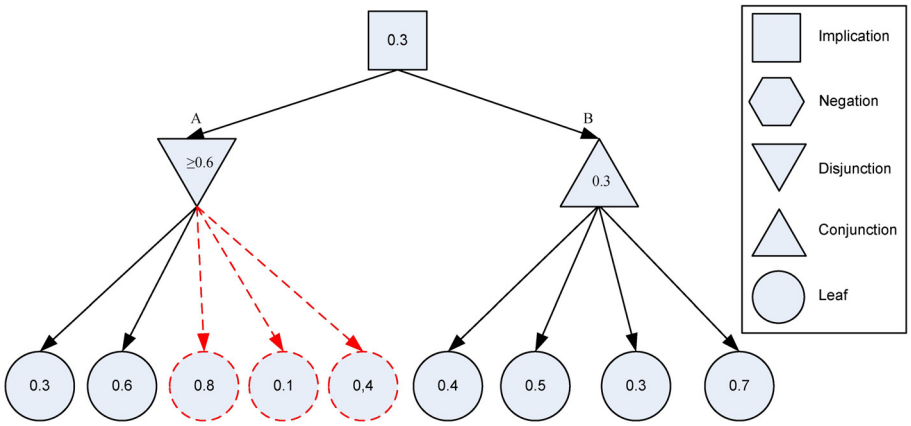


Figure 17: An example of applying the pruning when evaluating an implication node. Two children are connected; a conjunction node to left side and disjunction node to right side. Only six children (out of 9) are explored to get the final result at the root.

#### 4.2.2.3 Negation Pruning

As previously mentioned in section 2.3.6, equations (2.2a,2.2b) show that a negation node in Gödel expression has only two possible resulted values (0 and 1). The idea here is to check all the connected leaves in the lower layers of negation node before evaluating the connected nodes. If all the connected leaves has non-zero values and there is no more negation nodes connected, then cut-off can be applied, the subtree rooted at this negation node can be cut and 0 can be returned back the value to its parent node. Observe that with non-zero values using only conjunction, disjunction and implication the value cannot be zero. Thus, by a simple pattern matching on the

subexpression it can be checked whether it contains other negation or a 0 value in a leaf.

This idea is described in Algorithm 10, see also, e.g., Figure 18 for examples. The Check_Leaves function exploring all the leaves of the subtree rooted at the negation node. If all the connected leaves are non zeros and there is no other negation node in the subtree, then it returns the initiated flag value 1 meaning that the pruning can be done and the negated subexpression has a 0 value.

*Algorithm 10 (Negation Prune)*

```
1.      function Check_Leaves(Node)
2.      begin
3.      initiate Flag=1
4.        if Node is leaf then
5.          if Node = 0 then
6.            Flag=0                %"If at least one leaf equals to zero or"
7.          return Flag
8.        else
9.        if Node type is NEG then      %"if a NEG node is found,"
10.         set Flag=0                   %"then cut the search,    "
11.         return Flag
12.       end if
13.       end else                    %"we cannot prune.      "
14.       else
15.         for every child Node_i of Node do
16.          if Check_Leaves(Node_i) = 0 then
17.                Flag=0
18.                return Flag
19.         end if
20.         end for
21.       end else
22.     return Flag
23.     end Check_Leaves
```

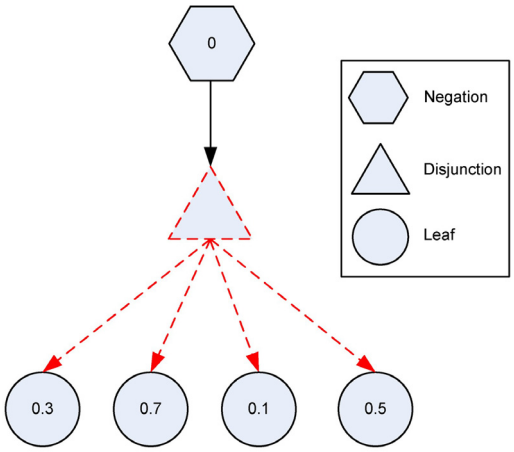


Figure 18: An example of applying the pruning for a negation node when all the connected leaves are non zeros.

#### 4.2.2.4 Implication with Negation Child Pruning.

In addition to the above mentioned pruning technique for negation, more pruning techniques can be done for the cases when an implication node has a child with negation and disjunction or conjunction at the other child. The strategy here is to try first a pruning at the negation node.

Figure 19 shows an example where we have an implication node which has the successors disjunction as left child and negation as right child. First, we can apply the negation pruning algorithm since all the connected leaves are non-zeros and the value 0 is returned back to the negation node. Then, we evaluate the successors of the disjunction node. After evaluating the second successor of disjunction node we have that its value will be always greater than 0.5, while it is greater than the value of negation node (the right successor). A cut can be applied here, and there is no need to explore and evaluate the remaining successors of the disjunction node for the exact evaluation of the implication node. A zero value will returned back as a value of the implication node.

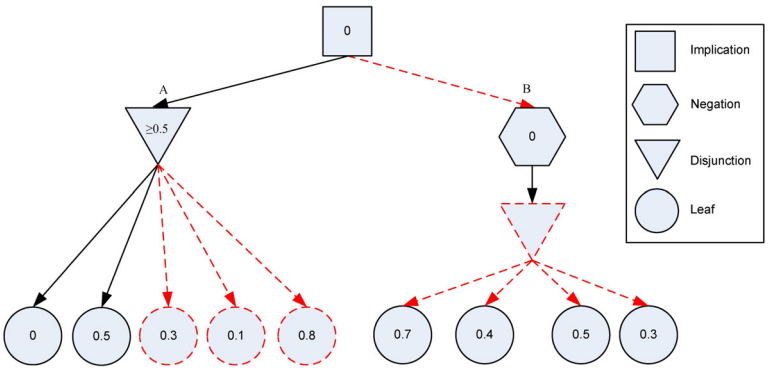


Figure 19: An example of implication pruning when we have an implication node that has the successors disjunction as left child and negation as right child. Only six children (out of 9) are explored to get the final result at the root.

Figure 20 shows an example where we have an implication node which has the successors conjunction as left child and negation as right child. First, as in the previous example, we can apply the negation prune since all the connected leaves are non zeros. Then, we evaluate the conjunction node successors. After evaluating the second successor of conjunction node we have that its value is equal to 0 which is the minimum. The cut can be applied here, and no need to explore and evaluate the remaining successors of conjunction node to evaluate the implication node. The value of the left and right successors are equal, so the value 1 will returned back as a value of the implication node.

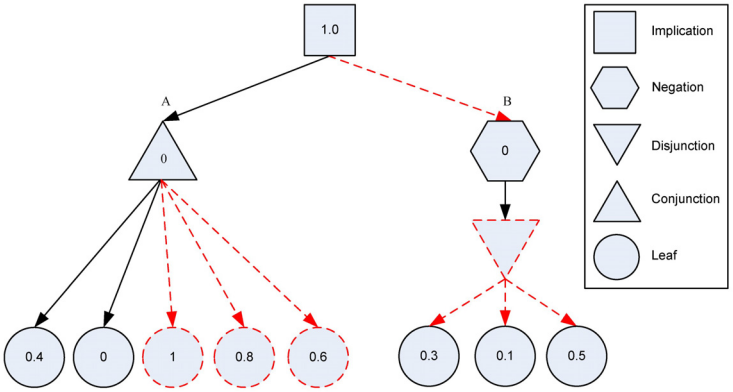


Figure 20: An example of implication pruning when we have an implication node that has the successors conjunction as left child and negation as right child. Only five children (out of 8) are explored to get the final result at the root.

Figure 21 shows an example in which the left child of the implication is a negation. In this case, independently of the type of the right child, if the pruning of the negation can be applied and gives a value 0 for the negation node, then the right child of the implication can be cut off, independently of its value, the implication gets its value 1.

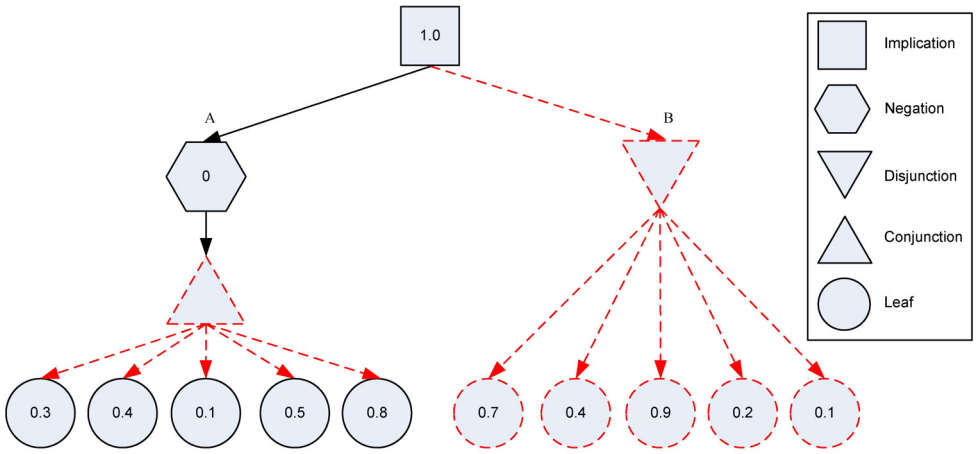


Figure 21: An example of applying the pruning when evaluating an implication node with negation node connected to the left side. Only five children (out of 10) are explored to get the final result at the root

In the next subsection some complex examples are displayed.

### 4.2.3 Complex Examples

In this subsection we show two complex Gödel expression tree examples, and how the proposed pruning techniques can be used to evaluate these trees very efficiently. First, the expression trees are shown (see Figure 22 and Figure 24, respectively). These trees include the four predefined operators (negation, implication, conjunction and disjunction). The order of the operators is various in these expressions, and also, leaves can be found in various levels (i.e., various depths) of the tree.

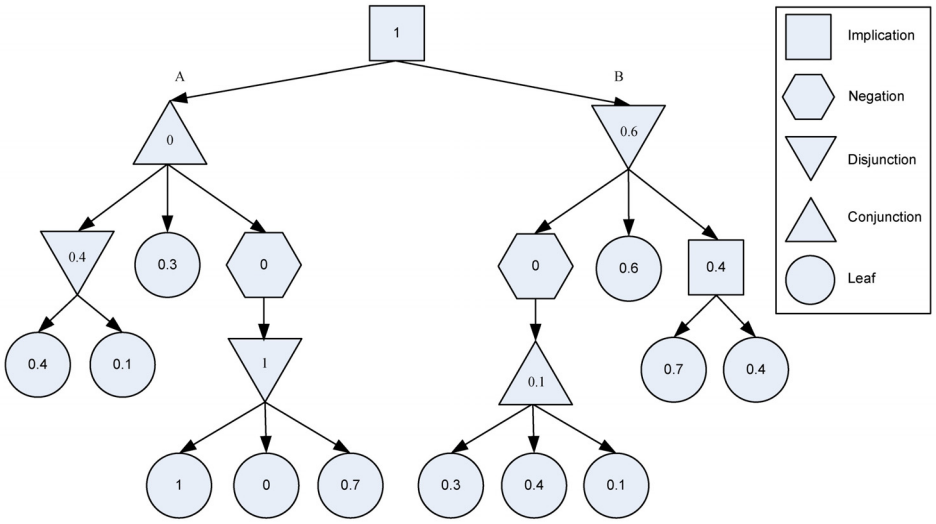


Figure 22: An example of Gödel expression tree without pruning. Nine vertices and twelve leaves (total 21) must be explored and evaluated to get the final result at the root.

To evaluate the tree of the example in Figure 22, without applying our pruning algorithms, 9 vertices and 12 leaves must be explored and evaluated. Figure 23 shows the same tree evaluated after applying the proposed pruning algorithms. The same result of evaluation is provided at the root but by exploring and evaluating only 5 vertices and 7 leaves (total 12) out of 9 vertices and 12 leaves (total 21).

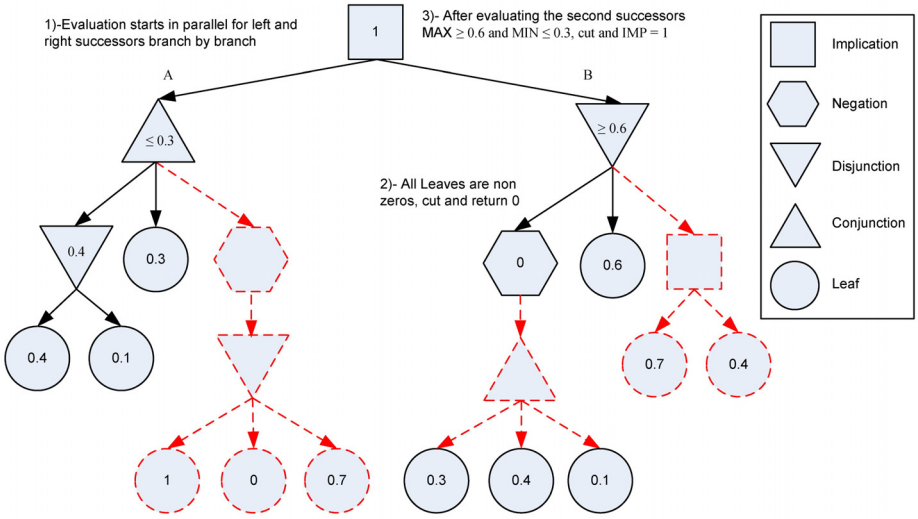


Figure 23: The example of Gödel expression tree in Figure 22 after applying the proposed pruning techniques. Only five vertices and seven leaves explored and evaluated (total 12) out of eight vertices and eleven leaves (total 21) to get the final result at the root.

As shown in Figure 23, the evaluation by applying the proposed pruning techniques has been done as follows:

- The evaluating process is started in parallel to evaluate the conjunction and disjunction nodes (the two children of the implication at the root).

- While evaluating the first successor (negation node) of disjunction node, a negation prune has been applied for the reason that all the connected leaves are non zeros.

- After evaluating the second successor of disjunction and conjunction, the value of disjunction node (right branch) will be always greater than 0.6 and the value of conjunction node (left branch) is less than 0.3. A pruning can be applied here by cutting exploring and evaluating the remaining leaves and nodes (the third branch of disjunction and conjunction nodes) and return back the value 1 to the root (implication node).

In our other example, shown in Figure 24, to evaluate the tree without applying the pruning algorithms mentioned above, 18 vertices and 19 leaves must be explored and evaluated (total 37). By using various pruning strategies the number of vertices that must be evaluated is 11, while the number of leaves that must be explored is 11 (total 22, see Figure 25).

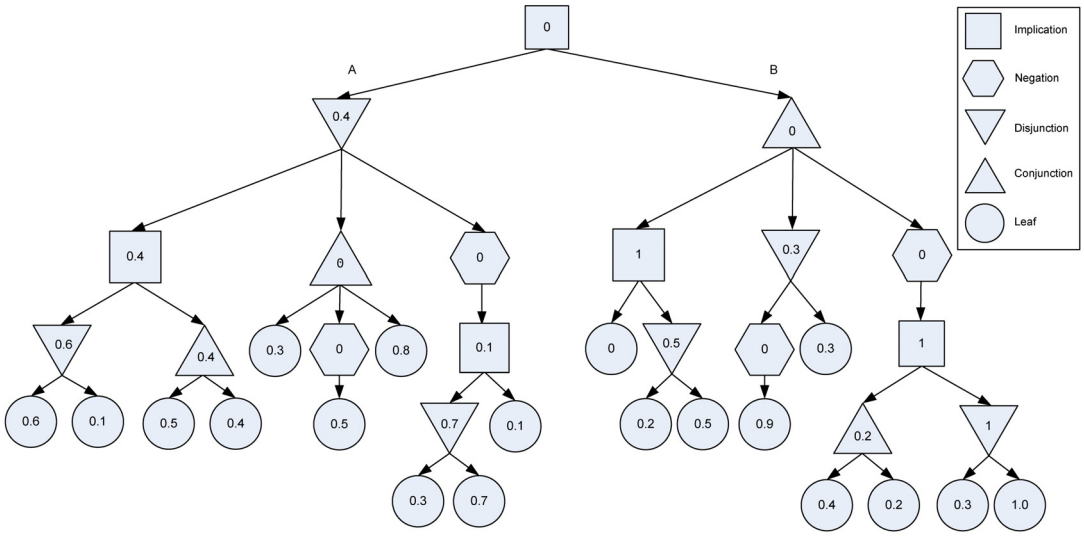


Figure 24: A complex Gödel expression tree. Eighteen vertices and nineteen leaves (total 37) must be explored and evaluated to get the final result at the root.

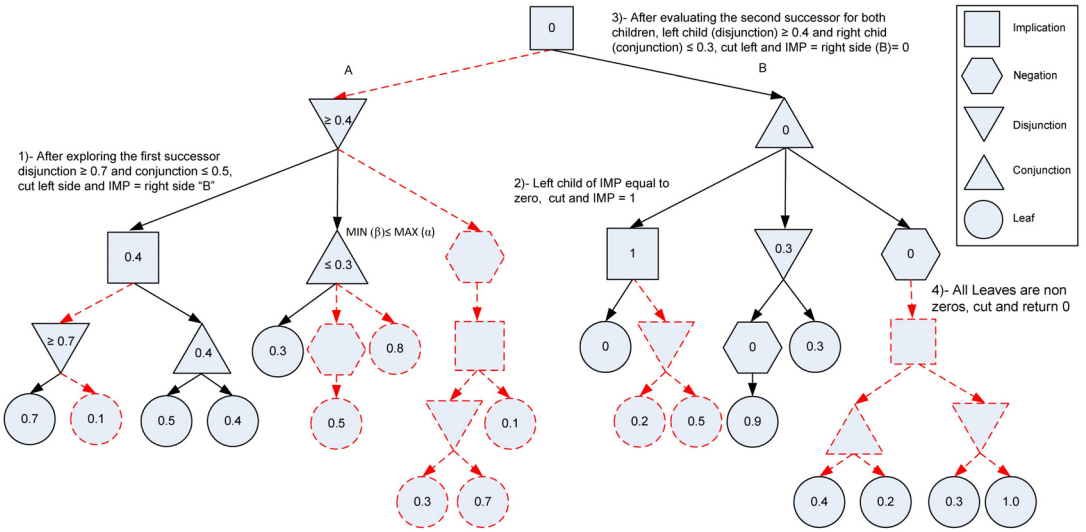


Figure 25: A complex Gödel expression tree after applying the proposed pruning techniques. Only eleven vertices and eleven leaves explored and evaluated (total 22) out of eighteen vertices and nineteen leaves (total 37) to get the final result at the root.

Figure 25 shows how the proposed pruning techniques used to evaluate the expression tree in Figure 24. These techniques have been applied in the following way:

- The evaluating process is started in parallel to evaluate the disjunction and conjunction nodes.

- While evaluating the first successor (implication node) of disjunction node, we have that it's right branch is greater than 0.7 while the left one is less than 0.5. The proposed algorithm cuts the left branch and returns back the value of the node at the right side.

- In the evaluation process of the first successor connected to the conjunction node (which is the right child of the root), a cut-off can be applied to evaluate the connected Implication node since its left branch is a zero leaf and the value 1 is returned back.

- After evaluating the second successor of both the main disjunction and conjunction nodes in the expression, we have that disjunction is greater than 0.4 and conjunction is less than 0.3. The proposed algorithm cut-off all the left branch of the root and continue to evaluate the right branch only. Furthermore, a cut-off has been applied while evaluating the second successor of the main disjunction node. The value of this node is less than 0.3 while the current value of the main disjunction node is greater than 0.4; there is no need to explore the remaining leaves to evaluate it.

- A negation prune can be applied for the third successor of the main conjunction node. This is for the reason that all the connected leaves are non zeros.

- Finally, the value 0 is returned back to the root as the final value of the expression.

These results show how fast the evaluation process can be after applying the proposed technique to evaluate an expression trees.

## 4.3 Strategies to Fast Evaluation of Lukasiewicz Type Logic Formulae

In this section, various techniques are proposed to speed up the evaluation of formula trees in Lukasiewicz logic based on the logical connectives previously defined in section 2.3.7. As in Gödel expression trees, we are dealing with trees with bounded set of truth values: the real numbers of the closed interval [0,1] can be used at the leaves of the tree. At the beginning of the evaluation they are given at the leaves of the tree, and the task is to compute the value at the root. About the forms of these trees we have the restriction: negation vertices must have exactly one child, while the other vertices (operators) must have exactly two children, called left child and right child, respectively.

The proposed pruning techniques to speed up the evaluation of this kind of trees are described in details in the next subsections.

### 4.3.1 Conjunction and Disjunction Pruning

Evaluation of the disjunction (&) and conjunction (+) nodes can be speeded up using various techniques. These techniques are based on the possible left and right children of these nodes. For a vertex associated to a conjunction, cut can be applied if we have a child (maybe a leaf) with value zero which make the result of equation (2.7) equal to zero (the minimum value) whatever the value of the vertex that is connected

to the other side of the given conjunction vertex. To do the evaluation in a faster way, the process of evaluating this kind of nodes can be started by getting the value of the connected leaf if it exists. Figure 26 shows an example for such case.

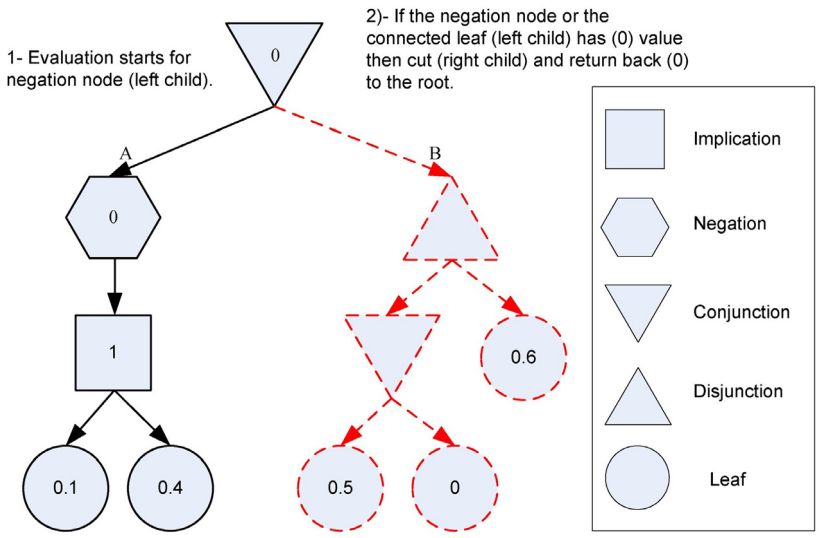


Figure 26: An example for a cut applied when a conjunction (&) vertex has a child (to left or right) with a value equal to 0. Only three vertices with connectives and two leaves (total 5) out of five vertices with connectives and five leaves (total 10) are explored and evaluated to get the final result at the root.

When the root (of the subtree we are working with) is a disjunction vertex, the cut can be applied if we have a child (maybe a leaf) with value 1 which makes the result of (2.8) equal to one (the possible maximum value) whatever the value of the child that connected to the other side of the disjunction node. To speed up the evaluation, the same process can be applied here as at the conjunction vertices.

In addition to the previous techniques that are operating when the minimal/maximal value is reached, another technique can also be applied at conjunction and disjunction vertices. This technique is based on equations (2.6), (2.7) and (2.8) depending on the operators at the children. It is clear that the result of the expression in equation (2.6) is always greater or equal to the negation of A which is equal to 1 –

|A|. Moreover, the minimum value of the expression in equation (2.7) is when the sum of the values of the two connected successors (A and B) is less than 1, and the value of the expression is always less or equal to |A|. In equation (2.8) we can see that its maximum value is equal to 1, and the expression has a final value which is always greater or equal to |A|. This can happen when the sum of the values of the connected successors (A and B) is greater or equal to 1.

The idea of the proposed cuts is the following. Firstly, suppose that the root (of the subtree) is a disjunction vertex, and each of the connected successors (i.e., children) is one of the following types:

- negation vertex with a conjunction child;
- implication vertex; or
- disjunction vertex.

In this case the evaluation of the connected vertices can be started in parallel (e.g., by the left child of both of these vertices), starting by getting the value of the connected leaf if such child exists. After evaluating or getting the value of the first successor of the both connected vertices and as mentioned above, if the sum of both evaluated successors (in case of disjunctions or negated conjunctions vertices are connected), or the sum of the negated successors (1 - |A|'s in case of an implication vertex is connected) is greater or equal to 1, then we can make a cut and return back the value 1 to the root and there is no need to evaluate and explore the remaining connected nodes or leaves (e.g., the right side of these branches). To make it more clear we

provide an example of Figure 27 having disjunction at both children and Figure 28 having an implication and a disjunction child.

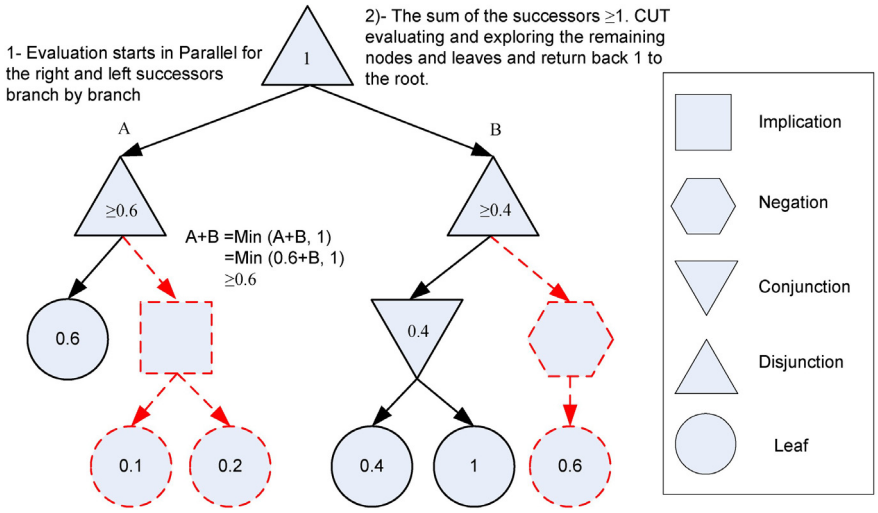


Figure 27: An example for pruning techniques applied at a disjunction vertex having two disjunction children and their sum is greater or equal to 1. Only four vertices with connectives and three leaves (total 7) out of six vertices with connectives and six leaves (total 12) are explored and evaluated to get the final result at the root.

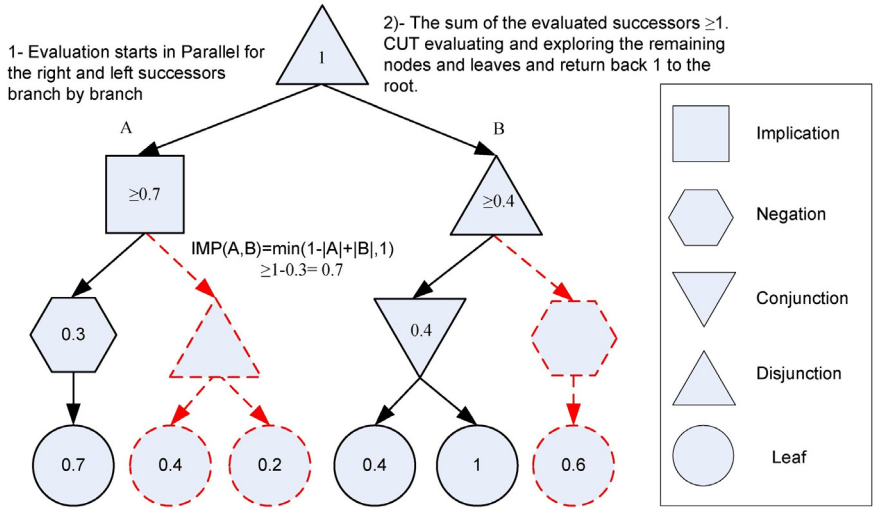


Figure 28: An example for pruning techniques applied at a disjunction vertex having an implication and disjunction nodes as its children and their sum is greater or equal to 1. Only five vertices with connectives and three leaves (total 8) out of seven vertices with connectives and six leaves (total 13) are explored and evaluated to get the final result at the root.

Secondly, suppose that the root is a conjunction vertex, and the connected successors (children) are both either a negated disjunction vertex (i.e., a negation with disjunction child), or conjunction vertex. Similar technique can be applied here to the previously described one. The difference is that the cut is applied when the sum of the values of the two successors (A and B) is less than or equal to 1. The value 0 returned back to the root then. Figure 29 shows an example.

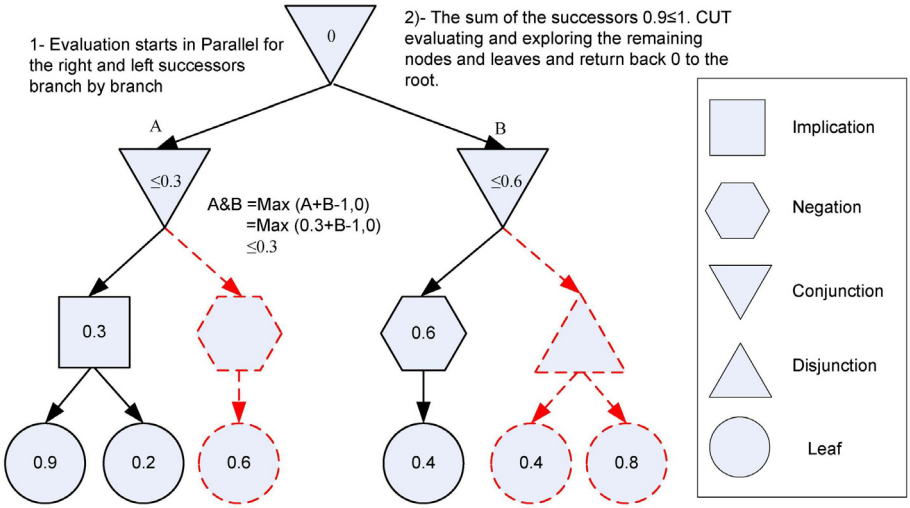


Figure 29: An example for pruning techniques applied at a conjunction vertex having two conjunction nodes as its children and their sum is less or equal to 1. Only five vertices with connectives and three leaves (total 8) out of seven vertices with connectives and six leaves (total 13) are explored and evaluated to get the final result at the root.

### 4.3.2 Implication Pruning

Now let us consider that the root (of the subtree under evaluation) is an implication node. Various cuts can also be applied at these vertices. One can apply a lazy evaluation if the left child of the implication is a node (maybe a leaf) with value equal to zero (the minimum value that it can be). Directly we can cut the right child and return back the value 1 to the implication (root) node. Whatever the value of the right child, it will be greater or equal to the value of the left child. Figure 30 shows an example of this case. It is shown that after evaluating the negation node at the left

child of the implication node, it has the value 0; this means that the right child will be greater or equal to this value. The whole right child can be cut out; the value 1 is returned back to the implication (root) node without evaluating that cut branch.

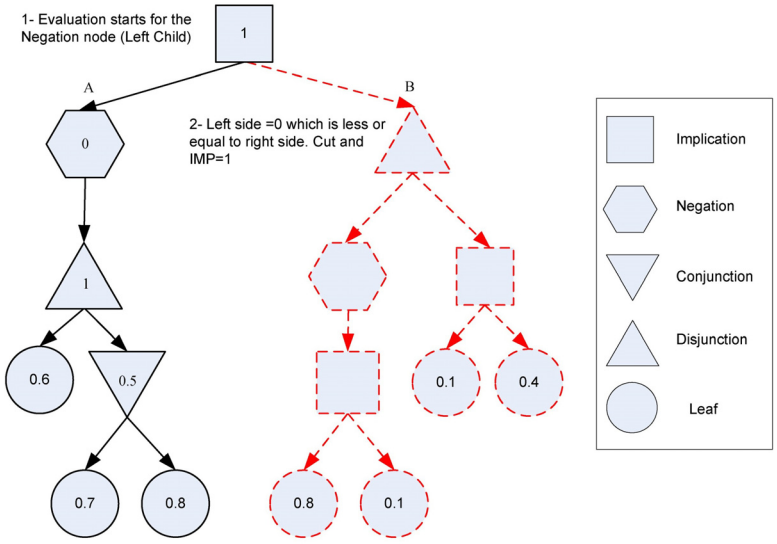


Figure 30: An implication pruning example with negation node as left child. Only four vertices with connectives and three leaves (total 7) out of eight vertices with connectives and seven leaves (total 15) are explored and evaluated to get the final result at the root.

The next possible cut technique that can be applied at evaluation of an implication vertex is the following. The condition for this pruning is that the left child of the implication vertex is either a negated disjunction (i.e., a vertex with negation that's child is a disjunction vertex) or a conjunction; and its right child is either a disjunction or a negated conjunction (that is a negation having its child a conjunction vertex). We can consider the conjunction node as a kind of MINIMUM node, and the disjunction vertex as a kind of MAXIMUM node, while the opposite is true for them in the negated case. Thus, our implication vertex has a left MINIMUM and a right MAXIMUM children. Suppose that A and B are the children of the MINIMUM (e.g., conjunction) node, while C and D are the children of the MAXIMUM node. Then, the evaluation starts in parallel for the children, i.e., for A and for C, then for B and

for D. As a result of the equations (2.7) and (2.8), the value of these two nodes (conjunction or negated disjunction to left and disjunction or negated conjunction to right) is always less or equal to the value of A and greater or equal to the value of C, respectively. This allows us to make a cut in some cases, as we illustrate them by examples.

Figure 31 shows an example of an implication node that has a negated disjunction as the left child and a disjunction vertex as its right child. While evaluating the successors of both the disjunction vertices in parallel, and as explained above, we already know that the negated disjunction (at the negation node) connected to left is less or equal to 0.4, while the disjunction node connected to right is greater or equal to 0.7. This means that right child is greater than left child, already. There is no need to explore and evaluate the right children of any of these disjunction nodes. A cut can be applied and the value 1 is given at the implication (root) node.

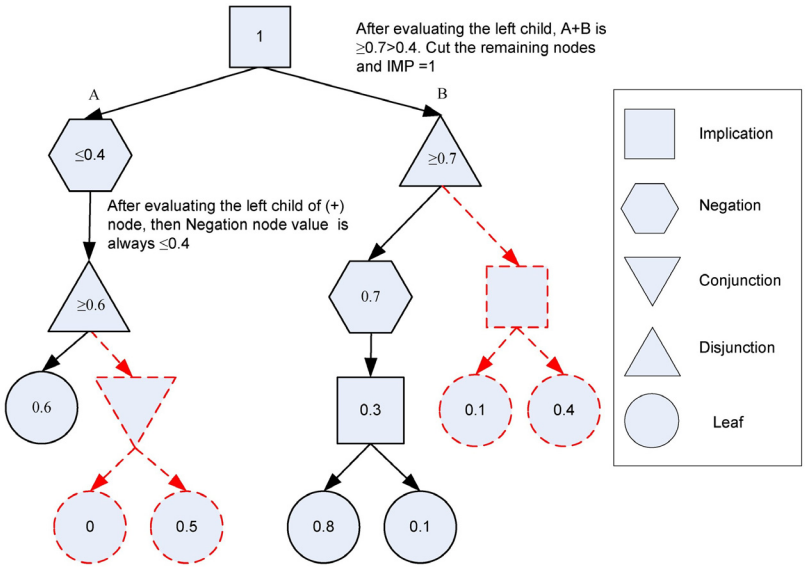


Figure 31: An implication pruning example with negated disjunction (left child) and disjunction (right child). Only eight vertices with connectives and three leaves (total 7) out of eight vertices with connectives and seven leaves (total 15) are explored and evaluated to get the final result at the root.

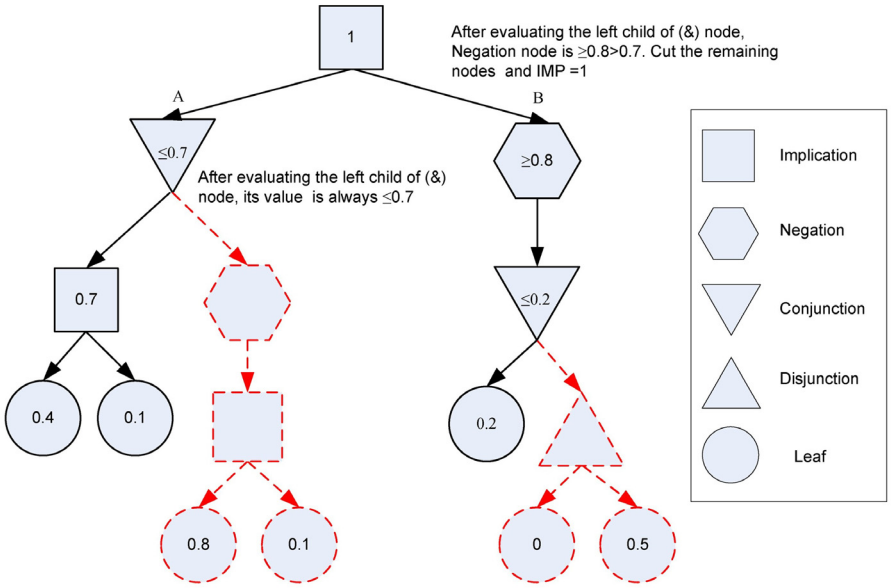


Figure 32: An implication pruning example with conjunction (left child) and negated conjunction (right child). Only five vertices with connectives and three leaves (total 8) out of eight vertices with connectives and seven leaves (total 15) are explored and evaluated to get the final result at the root.

For an example of an implication pruning in case that the left child is a conjunction and the right child is a negated conjunction, see Figure 32. These techniques can be considered as a special case of alpha-beta cut which is widely used in artificial intelligence, and specially, in game theory.

The last proposed example is showing the case when the implication node has conjunction node as a left child and disjunction node as a right child. It represents a special case of alpha-beta pruning, as well. When the value of the disjunction node becomes greater or equal to the value of the conjunction node, a cut is applied and the value 1 is returned back to the implication node (root). See Figure 33 for an example of applying the proposed pruning strategy.

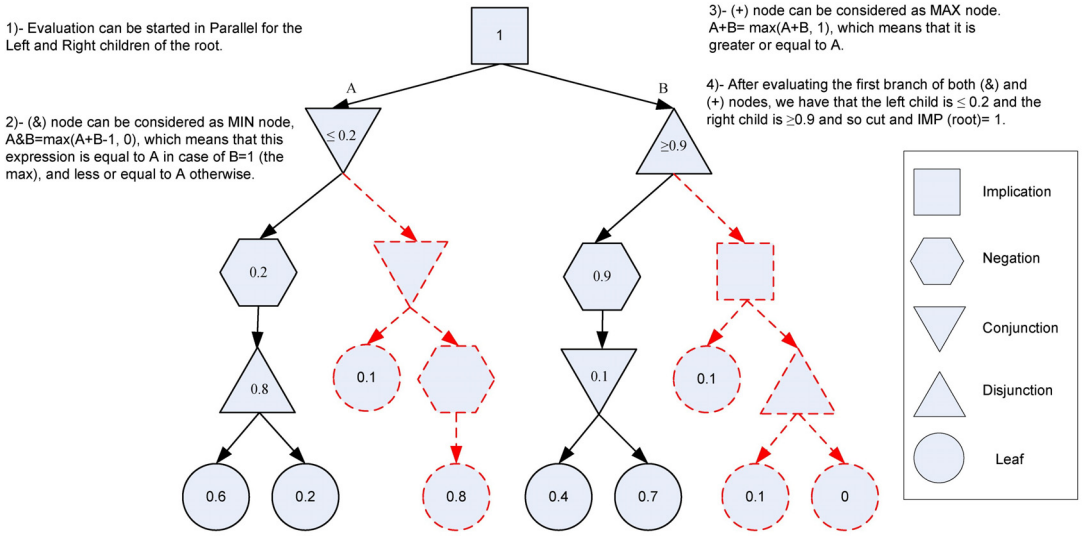


Figure 33: An example of evaluating implication vertex with a special case of alpha-beta pruning (the left child is a conjunction and the right one is a disjunction). Only seven vertices with connectives and four leaves (total 11) out of eleven vertices with connectives and nine leaves (total 20) are explored and evaluated to get the final result at the root.

The last pruning technique that we show is based on the fact that evaluating the implication expression defined in equation (2.6), it has always an output which is greater than or equal to the value 1 – |A|. According to this a pruning can be done when the root of the subtree (implication node) has a right child of the kind (disjunction, negated conjunction, or implication) and a left child of the kind (conjunction, negated disjunction, or negated implication). While evaluating the first successors of the left and right children, and when we have that right child has a greater or equal value to the left child, we can make a cut: we can stop evaluating and exploring the remaining nodes and leaves and return back 1 to the root node. See Figure 34 for an example.

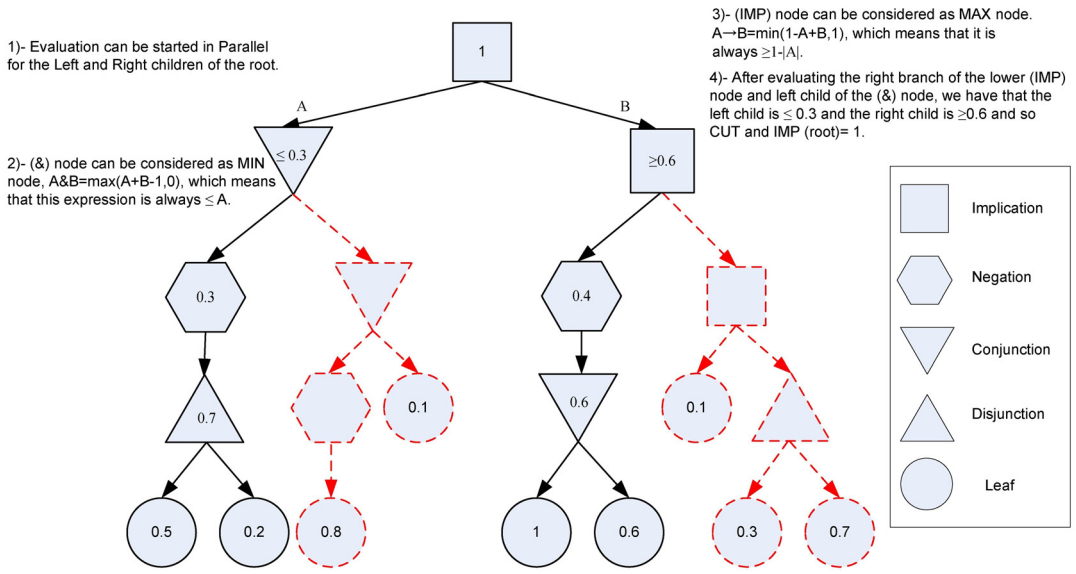


Figure 34: An example of evaluating implication vertex with a special case of alpha-beta pruning (the left child is a disjunction and the right child is an implication). Only seven vertices with connectives and four leaves (total 11) out of eleven vertices with connectives and nine leaves (total 20) are explored and evaluated to get the final result at the root.

## 4.4 Strategies to Fast Evaluation of Product Logic Formulae

In this section, the product fuzzy logic is considered. Various techniques are proposed to speed up the evaluation of formula trees in product logic based on the logical connectives previously defined in section 2.3.8. Similarly to the cases of Gödel and Lukasiewicz fuzzy logic systems shown in previous subsections, we are dealing with trees with bounded set of truth values: the real numbers of the closed interval [0,1] can be used at the vertices of the tree. At the beginning of the evaluation they are given at the leaves of the tree, and the task is to compute the value at the root. About the forms of these trees we have the restriction: nodes assigned to negation must have exactly one child, nodes with implication must have exactly two children (called left and right children), while the other nodes with assigned conjunction and disjunction have at least two children.

The proposed pruning techniques to speed up the evaluation of this kind of trees are described below in details in the next subsections.

### 4.4.1 Conjunction and Disjunction Pruning

Evaluation of the disjunction (AND) and conjunction (OR) nodes can be speed uped using various techniques. These techniques are based on the values of already visited children of these nodes. As an immediate pruning, for a vertex associated to a conjunction, cut can be applied if we have a child (maybe a leaf) with value zero which make the result of equation (2.11) equal to zero (the possible minimum value) whatever the value of the other nodes connected to the given conjunction vertex. To do the evaluation in a faster way, the process of evaluating this kind of nodes can be started from a child which is leaf, if such a child exists. Figure 35 shows an example for such case.

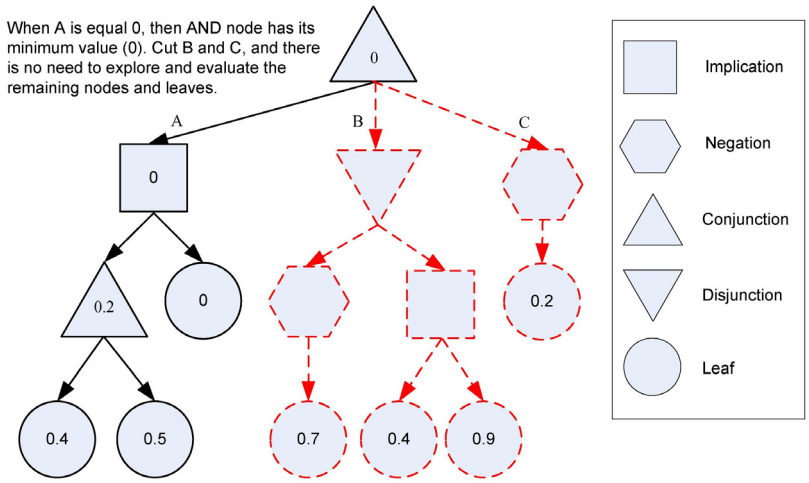


Figure 35: An example for a cut applied when a conjunction (&) vertex has a child (to left or right) with a value equal to 0. Only three vertices with connectives and three leaves (total 6) out of seven vertices with connectives and seven leaves (total 14) are explored and evaluated to get the final result in the root.

When the root (of the subtree we are working with) is a disjunction node, the cut can be applied if we have a child (maybe a leaf) with value 1 which makes the result of equation (2.12) equal to one (the possible maximum value) whatever the value of the

other children of the disjunction node. To speed up the evaluation, a similar cut can be applied here as at the conjunction nodes. For an example, see Figure 36.

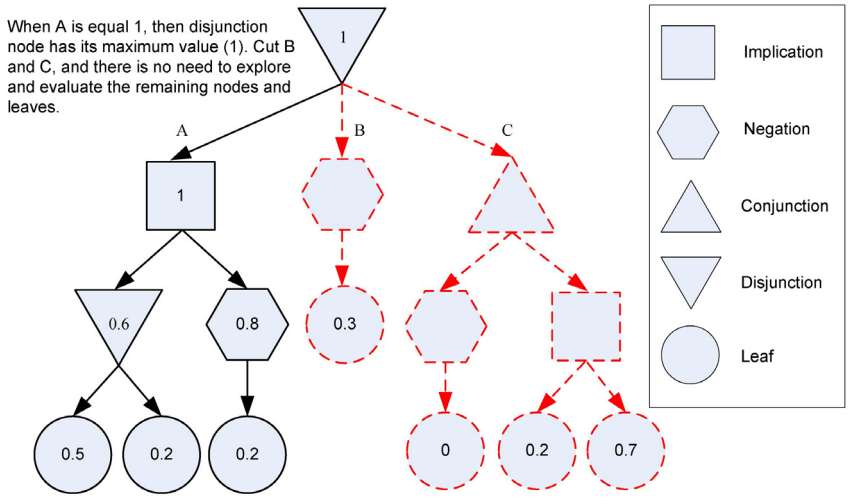


Figure 36: An example for pruning techniques applied at a disjunction node having a child with a value equal to 1. Only three vertices with connectives and three leaves (total 6) out of eight vertices with connectives and seven leaves (total 15) are explored and evaluated to get the final result at the root.

### 4.4.2 Implication Pruning

Now let us consider that the root (of the subtree under evaluation) is an implication node. Implication nodes have exactly two children. In similar way as shown in sections about Gödel and Lukasiewicz, an immediate pruning can be applied if the left child of the implication has a value equals to zero (the minimum value that it can be). Directly we can cut the right child and assign the value 1 to the implication (root) node, since whatever the value of the right child is, it will be greater or equal to the value of the left child. Figure 37 shows an example: After evaluating the negation node at the left child of the implication node, it has the value 0; this means that the right child will be greater or equal to this value. The right child and the whole subtree rooted there can be cut out, the value 1 is guaranteed at the implication (root) node without evaluating the cut branch of the subtree.

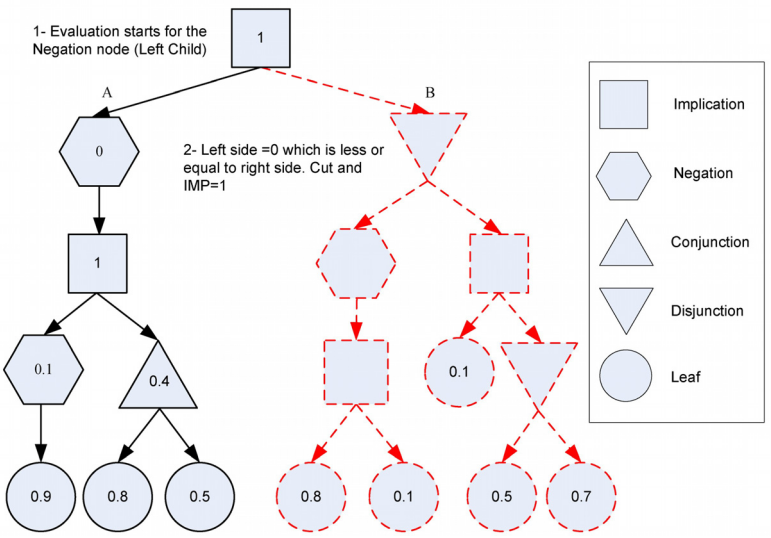


Figure 37: An implication pruning example with negation node as left child. Only five vertices with connectives and three leaves (total 8) out of ten vertices with connectives and eight leaves (total 18) are explored and evaluated to get the final result at the root.

We note here that a kind of dual of the previous technique also exists. In case the right child is evaluated first, (e.g., it could be efficient, if it is a leaf) and its value equals to one (the maximum value that can be), a pruning can be applied: the left child can be directly cut out and the value 1 can be assigned to the implication (root) node. Figure 38 shows an example; the possible cuts are the following ones: if the evaluation starts for the negation node (right child), then while evaluating the connected conjunction node a leaf with value zero has been found. As mentioned in the previous subsection, we can cut the second child in this case without evaluating it giving value 0 to the node with conjunction. In this case the implication node has the value zero. Now, the node with negation will get the value 1 that is greater or equal to the left child of the root (implication node). This will make equation (2.10a) equal to 1, independently of the value of the left child of implication node.

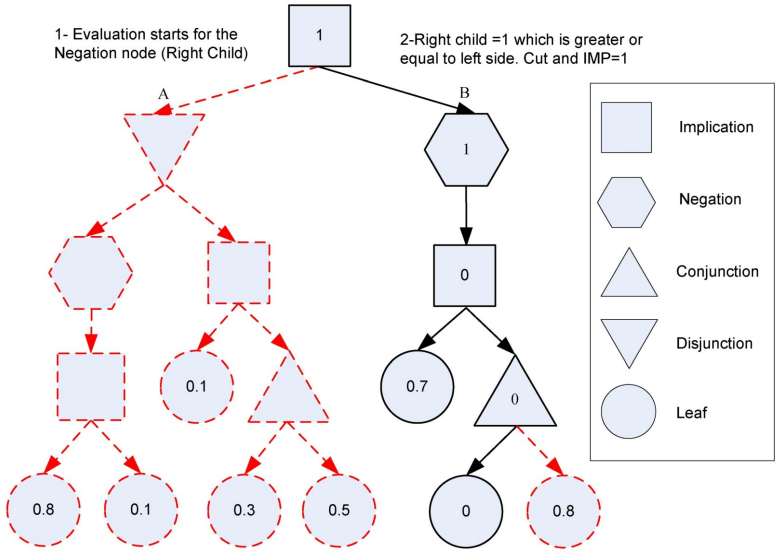


Figure 38: An implication pruning example evaluating its right child first. Only four vertices with connectives and two leaves (total 6) out of nine vertices with connectives and eight leaves (total 17) are explored and evaluated to get the final result at the root.

The next possible cut technique is more applicable than the previously described ones, in the sense that it does not require to have a value 0 or 1 at a node. It can be applied at evaluation of an implication node with the following properties. Its left child is either conjunction or a negated disjunction (i.e., a node with negation that's child is a node with disjunction), while its right child is either a disjunction or a negated conjunction (that is a negation having its child a node with conjunction). We can consider the conjunction and disjunction nodes in somewhat similar roles, as minimum and maximum nodes, respectively (in game theory, or at Gödel logic). In this way, at these nodes, depending on the number of already evaluated children, a value is computed that gives an upper, or a lower bound, respectively, for the real value of the node. (At negated nodes this value $x$ is changes to $1 - x$, and its bound changes to the opposite bound.) Thus, the value of the implication node is already bounded by an upper bound (computed at its left child) and a lower bound (computed at its right child).

As mentioned and explained previously in section 4.3.2 (implication pruning), the same can be applied here. Suppose that A and B are the children of the left branch of the implication, i.e., children of the conjunction or a negated disjunction as a left child, while C and D are the children of the right branch of the implication, i.e., children of the disjunction or a negated conjunction as the right child. Then, the evaluation starts in parallel for the children, i.e., in the following order: starts for A and for C, then for B and for D. (Having more than two children the evaluation should follows alternating for the children, one from the left and one from the right branch.) As a result of (2.11) and (2.12), the value of these two nodes (conjunction or negated disjunction on the left and disjunction or negated conjunction on the right) is always has at most the value of A and at least the value of C, respectively. This allows us to make a cut in some cases, as we illustrate them by examples.

Figure 39 shows an example of an implication node that has a negated disjunction as the left child and a disjunction node as its right child. While evaluating the successors of both the disjunction nodes in parallel, and as explained above, we already know that the negated disjunction (at the negation node) connected to left is less or equal to 0.5, while the disjunction node connected to right is greater or equal to 0.6. This means that right child is greater than left child, already. There is no need to explore and evaluate the left children of any of these disjunction nodes. A cut can be applied and the value 1 is given at the implication (root) node.

Similar examples of an implication pruning can be shown in the other cases, e.g., when the left child is a conjunction and the right child is a negated conjunction.

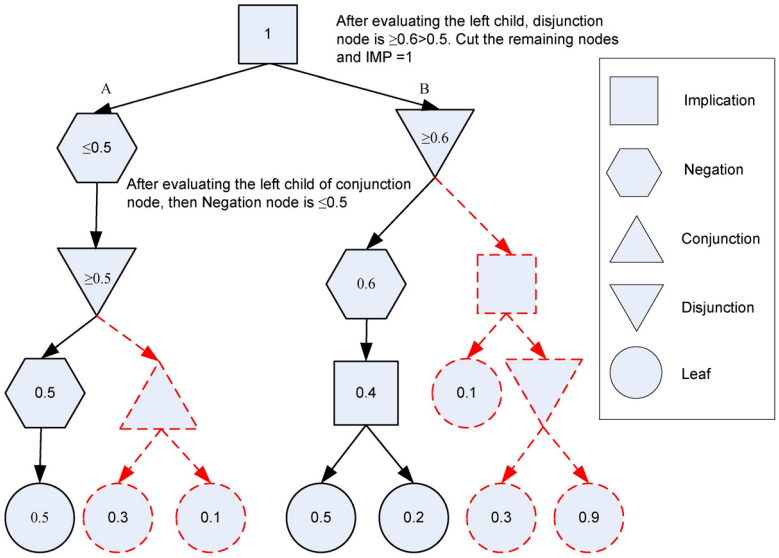


Figure 39: An implication pruning example with disjunction (right child) and negated disjunction (left child). Only seven vertices with connectives and three leaves (total 10) out of ten vertices with connectives and eight leaves (total 18) are explored and evaluated to get the final result at the root.

Figure 40 shows an example where some of the previously described evaluation strategies are applied (when disjunction node has one successor with the value 1 and when an alpha-cut has been applied in the right bottom node of the expression tree). While evaluating the successors of both the disjunction nodes in parallel, and as mentioned in the above subsection, a cut can be applied in disjunction (left child) of implication. The value 1 returned back to disjunction (the maximum). This makes the result of implication expression which is defined in equation (2.10b) equal to the value of the right child. Then, we continue evaluating and exploring the connected nodes and leaves to the right child (conjunction node).

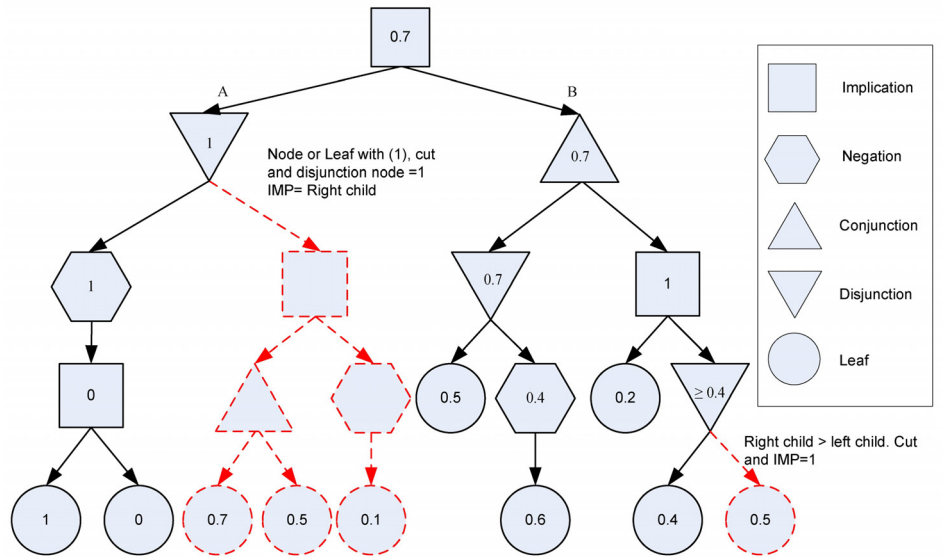


Figure 40: An implication pruning example with a special case of alpha-cut. Only nine vertices with connectives and six leaves (total 15) out of twelve vertices with connectives and ten leaves (total 22) are explored and evaluated to get the final result at the root.

These techniques can be more efficient (could lead to pruning with a higher probability) when there are more than two children of the corresponding conjunction/disjunction nodes. In these cases by the parallel (i.e., alternate) evaluation of the children of both sides, the upper and lower bound is adjusted dynamically. Actually, a bound does not change after evaluating a child if and only if this child has a value 1. An example for such pruning will be shown later in Figure 45.

The next pruning technique we show is based on the fact that evaluating the implication expression in equation (2.10b), it has always an output which is greater than or equal to the value |B|. This technique would require to evaluate the right child of an implication first, and, therefore, it could not be applied automatically, it is a kind of theoretical pruning. According to the condition that the value of the implication is never less than the value of its right child, a pruning could be done when the root of the subtree (implication node) has a right child that one of the

following kinds: disjunction, negated conjunction, or implication; and a left child is one of the following kinds: conjunction, negated disjunction, or negated implication. While evaluating the first successors of the left and right children, and when we have that right child has a greater or equal value to the left child already, we can make a cut: we can stop evaluating and exploring the remaining nodes, the value of the implication node at the root is surely 1. See Figure 41 for an example. Actually, in this example one may observe this pruning technique at the implication at the root.

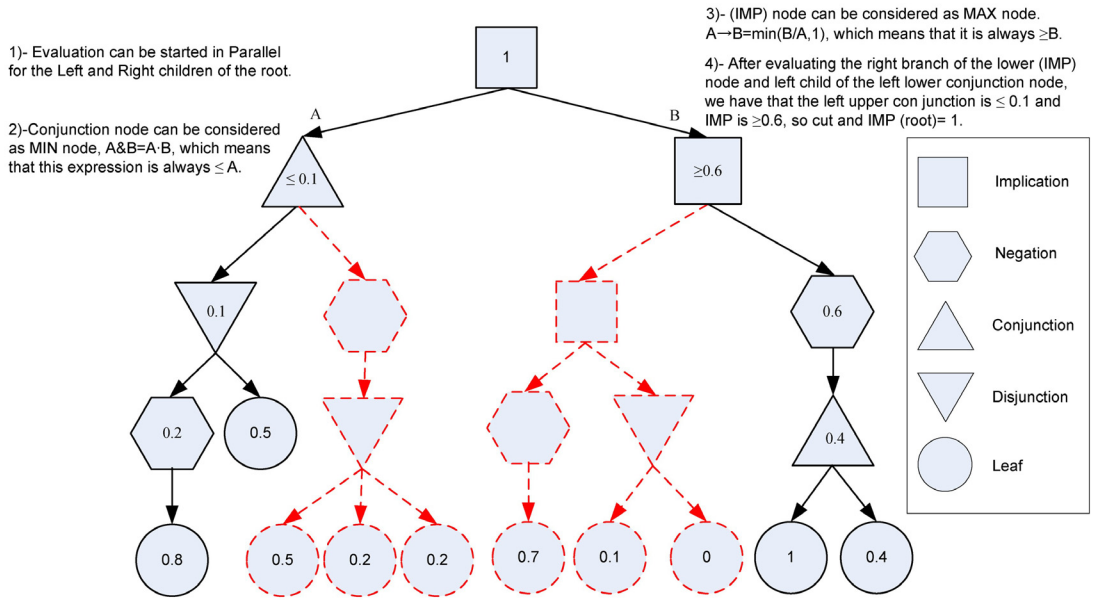


Figure 41: An example of evaluating implication node with a special case of alpha-beta pruning (the left child is a conjunction and the right child is an implication). Only seven vertices with connectives and four leaves (total 11) out of twelve vertices with connectives and ten leaves (total 22) are explored and evaluated to get the final result at the root.

### 4.4.3 Further Techniques

Further enhancement and development can be applied on the previously proposed techniques to make the evaluation faster. This can be done in a similar way that was earlier proposed to speed up the evaluation of Boolean expressions. This enhancement is to reorder the children of conjunction and disjunction nodes in

ascending order by the depth of these subtrees, from left to right. In this way, leaves will automatically be the first children, if there are any of them.

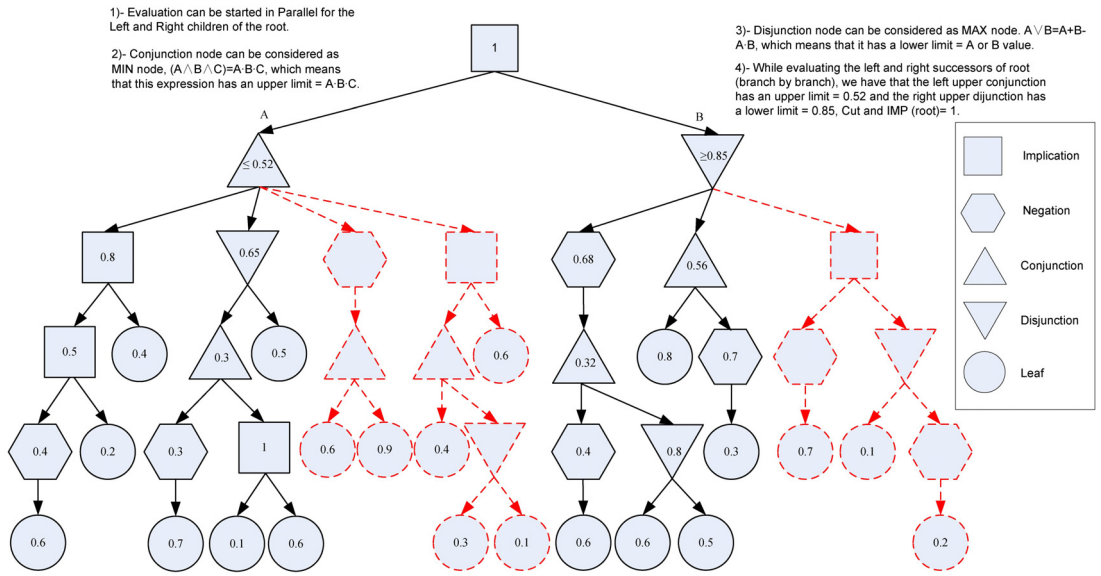


Figure 42: An example of applying the proposed cut techniques in a complex product logic expression tree. Only sixteen vertices with connectives and twelve leaves (total 27) out of twenty five vertices with connectives and twenty leaves (total 45) are explored and evaluated to get the final result at the root.

Using this technique we will have all the nodes with shortest paths to the left and, thus, they will be evaluated earlier than larger subtrees rooted in other children. This technique helps to have cuts earlier, in the average. See, for example, Figure 42 and Figure 43.

Figure 42 shows an example of applying the proposed cut techniques before reordering the branches. The cut is applied after evaluating two branches when the value of disjunction node becomes greater than the value of the conjunction node. In Figure 43, the cut is applied after evaluating one branch only of the both conjunction and disjunction nodes.

Also, one can apply this proposed technique to speed up the evaluation of Gödel and Lukasiewicz expression trees.

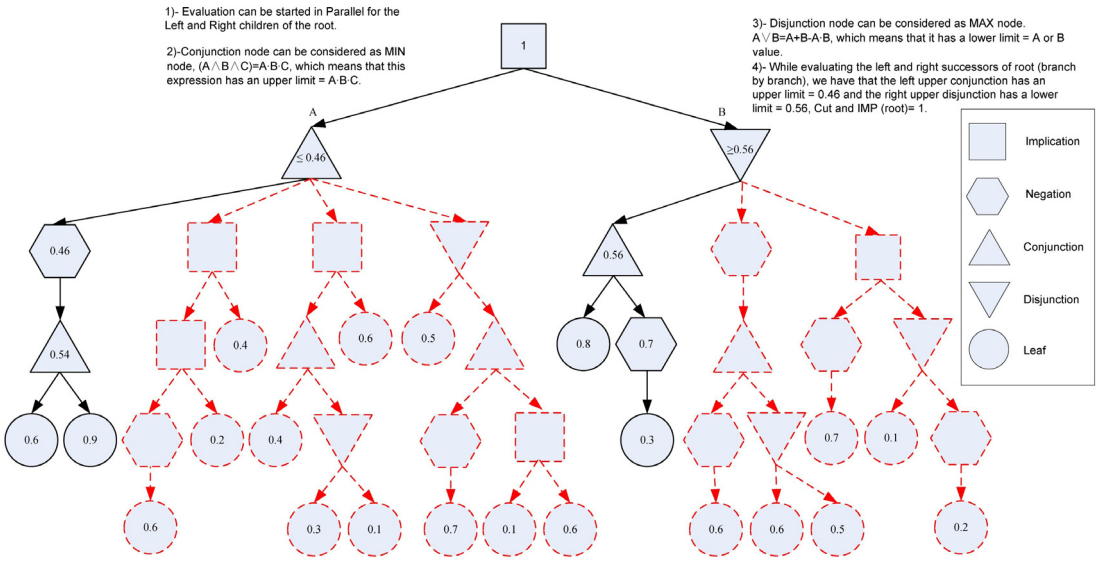


Figure 43: An example of applying the proposed cut techniques in a reordered tree for the complex formula displayed in Figure 42. Only seven vertices with connectives and four leaves (total 11) out of twenty five vertices with connectives and twenty leaves (total 45) are explored and evaluated to get the final result at the root.

### 4.4.4 Comparisons to Similar Techniques Obtained for Gödel Logic

We are closing this chapter by showing some comparisons to similar techniques obtained for Gödel-type logic. Let us consider a pruning at an implication, especially with left child as a conjunction and right child as a disjunction, allowing both of them to have several children.

In Gödel logic, the disjunction and conjunction nodes considered as maximum and minimum nodes, respectively. While in the proposed pruning algorithms for the product logic, we consider the left child has at most value and the right child has at least value. While evaluating more and more children in both sides, the proposed pruning algorithm for the product logic dynamically sharpens the values faster, than

in usual minimax, or Gödel logic. An example to show this difference is follows. Figure 44 shows an example of applying the proposed pruning algorithm in section 4.2, which is actually, very similar to alpha-beta cut at a minimax tree (conjunction and disjunction in Gödel logic is computed as minimum and maximum value of the children, respectively).

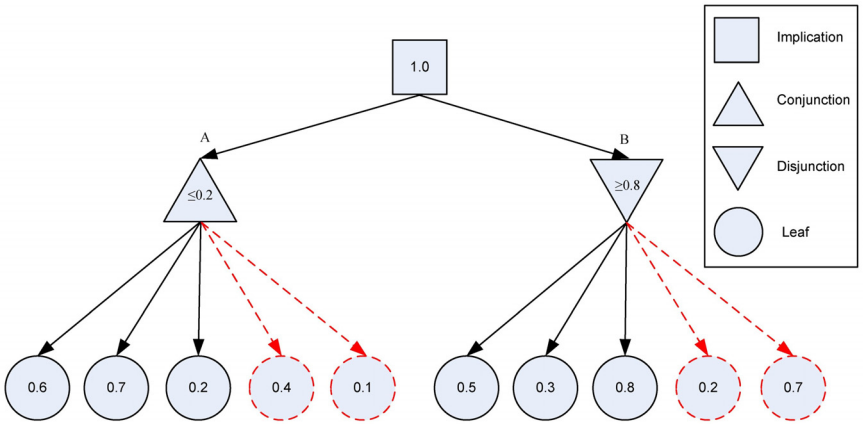


Figure 44: An example of cut applied in evaluating Gödel logic expression. Only six leaves out of ten are explored to get the final result at the root.

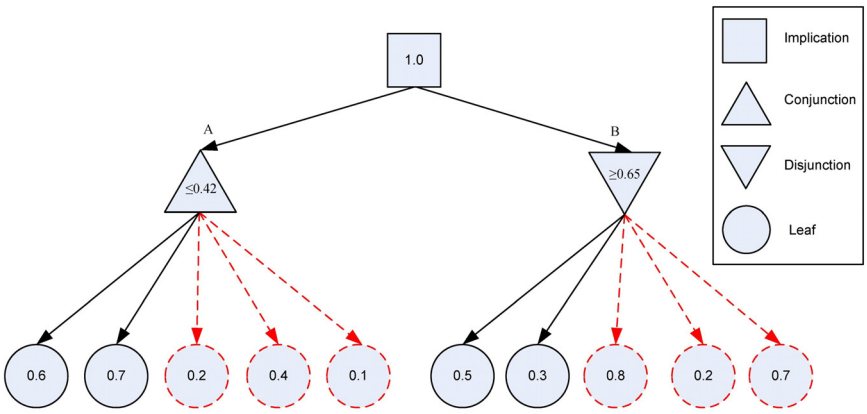


Figure 45: An example of cut applied in evaluating product logic expression. Only four leaves out of ten are explored to get the final result at the root.

Figure 45 shows the same expression tree, but the evaluation goes in product logic: after applying the proposed algorithms to fast evaluation of product logic expressions, it is clear that the cut in product logic happened earlier than in Gödel logic. In Gödel logic the cut is applied after exploring three children for both

conjunction and disjunction (total 6) out of five children for each one of them (total 10), while in product logic, it is applied after exploring two children only for both conjunction and disjunction nodes (total 4).

# Chapter 5

# EXPERIMENTAL RESULTS AND DISCUSSION

## 5.1 Programing the Proposed Algorithms

In this chapter we present some experimental results about the efficiency of the proposed techniques. The proposed algorithms have been programmed on Python language and there were 500 000 tests on formulae with various sizes. Since the tests were run on a normal computer with a non real-time operating system, the measurements for different node counts were run many times in mixed order to eliminate the effect of randomness caused by other software running on the same computer.

The test formulae were generated randomly. A parameter is used that has determined the size of the resulted formula. The four operators were used with equal chance at an operator node (25% each). At conjunction and disjunction nodes the chances to have two, three or four children were also the same. (In the experiment we used only formulae in which each conjunction and disjunction node has two, three or four children.)

At leaves the values 0, 0.05, 0.1, 0,15, ..., 0.9, 0.95 and 1 were assigned with equal chance. In an average test case, the number of leaves was slightly higher than the number of operators. The test formulae were generated randomly. A parameter is used that has determined the size of the resulted formula. The four operators were

used with equal chance at an operator node (25% each). At conjunction and disjunction nodes the chances to have two, three or four children were also the same. (In the experiment we used only formulae in which each conjunction and disjunction node has two, three or four children.)

At leaves the values 0, 0.05, 0.1, 0,15, ..., 0.9, 0.95 and 1 were assigned with equal chance. In an average test case, the number of leaves was slightly higher than the number of operators.

The following sections show the simulation results for the proposed techniques to speed up the evaluation of Gödel and product logic described in sections 4.2 and 4.4. We will start with the results obtained for Gödel type logic. In appendix A, we show the source code of the Python program for evaluating formulae in Gödel type logic.

## 5.2 Simulation Results for Gödel Logic Pruning Strategies

It is clear that the execution time of the simple evaluation algorithm of the expression trees for Gödel logic is linear to the number of nodes of the expression tree, because each node must be counted exactly once. The pruning algorithm we proposed is more complex, therefore in case of small expressions it requires more time to evaluate. However, when it is applied on large expression trees, its performance will be (much) better, because as the tree grows bigger, bigger part of the tree can be pruned in average.

Figure 46 shows the ratio of pruned nodes. The ratio of pruned leaf nodes to the number of leaf nodes has a pretty similar measure (Figure 47 shows some details).

For very small expressions it can happen that nothing or very minor part of the tree can be pruned. As the number of nodes increases the pruned ratio increases as well.

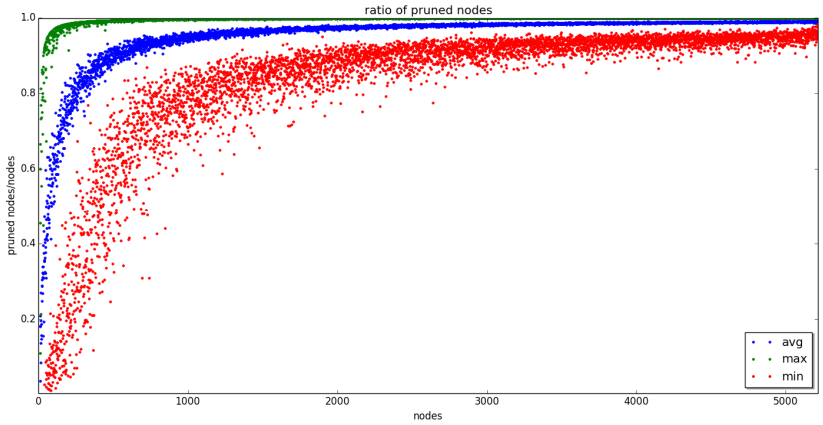


Figure 46: The ratio of the number of pruned nodes with respect to the size of the expression (total number of nodes).

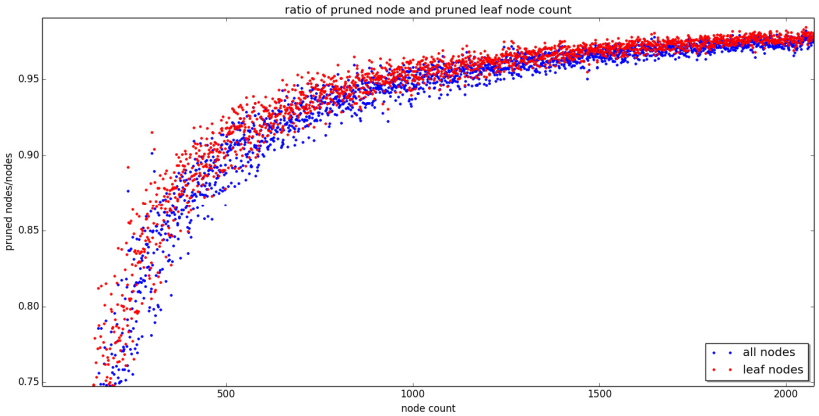


Figure 47: The ratio of pruned nodes and pruned leaves with respect to the total number of nodes and total number of leaves, respectively.

For graphs with thousands of nodes very large portion of nodes of the graph can be pruned in average and even in the worst cases more than 80 or 90 % of nodes will be pruned. It means that the evaluation time in the pruned algorithm is not linear, but much better. Figure 48 shows the execution time (in seconds) of the two algorithms on the same examples. There we can see the difference we sentenced before. It looks as the execution time of the pruning algorithm is more like constant instead of linear to the number of nodes. This is because the number of evaluated nodes that are left

after the pruning is not linear to the number of all nodes. In fact in average the number of these nodes tends to be constant. This is shown in Figure 49.

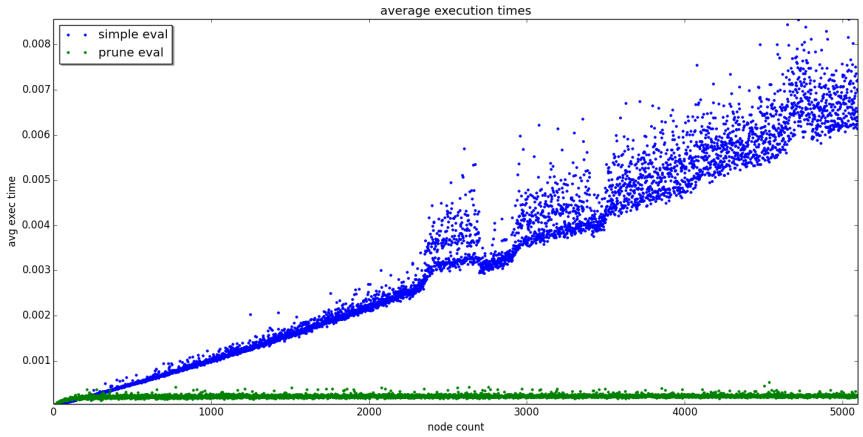


Figure 48: The running time in seconds with respect to the size of the expression (total number of nodes).

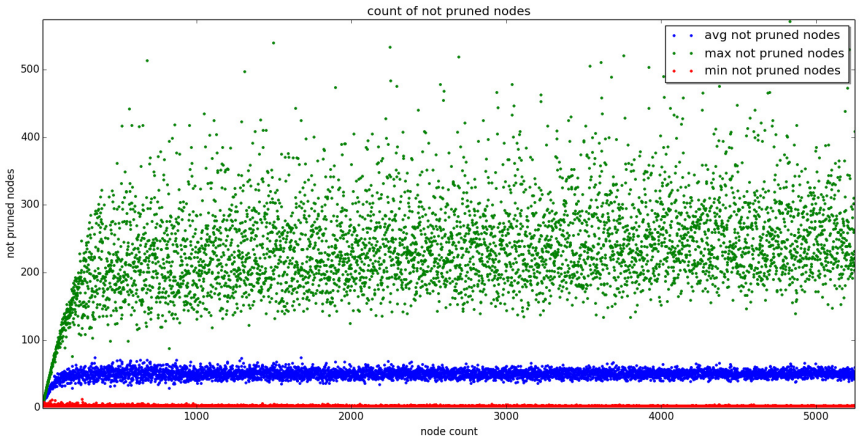


Figure 49: The number of nodes left after pruning with respect to the size of the expression.

Figure 50 shows what is the ratio of the execution times of the pruning and the simple evaluation algorithms (the time of simple evaluation is used as unit). With smaller trees the execution time of the pruning algorithm is still above the execution time of the simple evaluation. Above about 200 nodes the pruning algorithm is faster and it stays so, because the simple evaluation completes in linear time and the pruning algorithm finishes in nearly constant time (in average). The standard

deviation is relatively large on the running time since the amount of the pruned nodes could vary drastically. Also the difference between the measured minimum and maximum values are large. Fortunately the minimum measured pruned ratio values are also very good, with increasing the number of nodes also the minimum tends towards 100%, in fact the measured minimum pruned ratio values are around 90% if the number of nodes is about 5000.

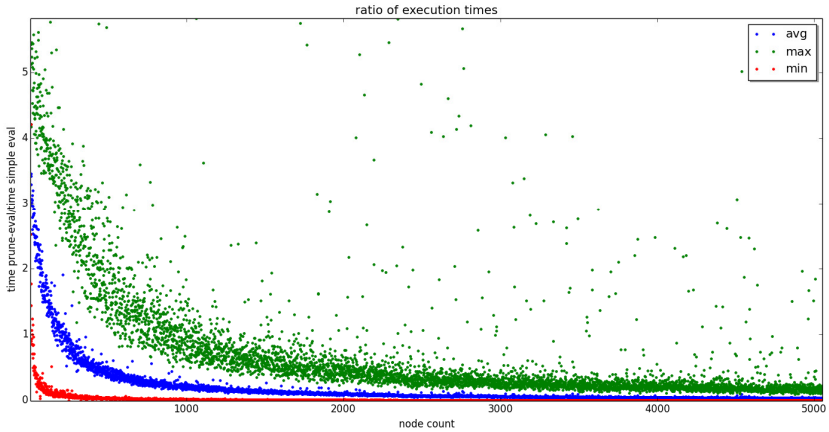

Figure 50: The ratio of running time with respect to the size of the expression.

## 5.3 Simulation Results for Product Logic Pruning Strategies

In the experimental results for product logic, we have run several tests to measure the performance gain of the pruning techniques described in section 4.4. At first we have examined how much ratio of the nodes can be pruned. Figure 51 shows that with the increasing number of the nodes in the expression tree the ratio of the nodes which can be pruned is growing. Actually, in the average it is close to 100% for expressions with, let us say, 1500 nodes (actually the number of nodes of the expression tree including the leaves is exactly the length of the analyzed formula).

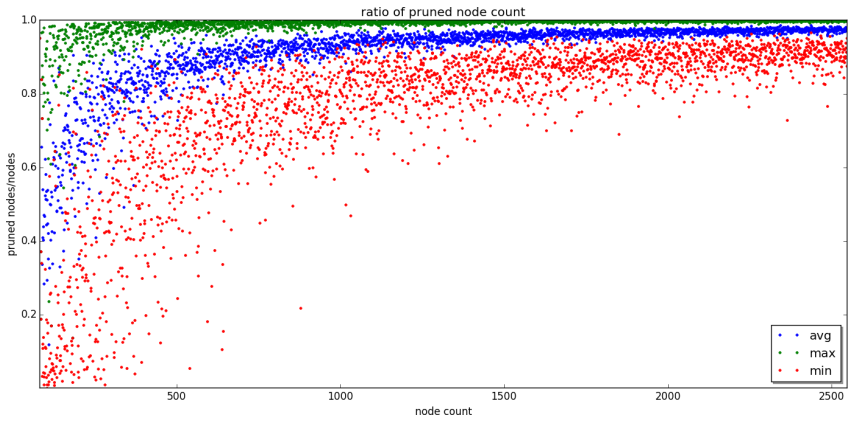

Figure 51: Ratio of pruned nodes for random formulae up to length 2500.

Also, the execution time of the simple evaluation algorithm which does not use any pruning techniques and the execution time for evaluations of the same expressions using the pruning techniques described above has been measured. Figure 52 shows that the execution time of the simple algorithm has a growing tendency with the increasing number of nodes. Contrary, the execution time of the algorithm using pruning strategies seems to be like constant in the average.

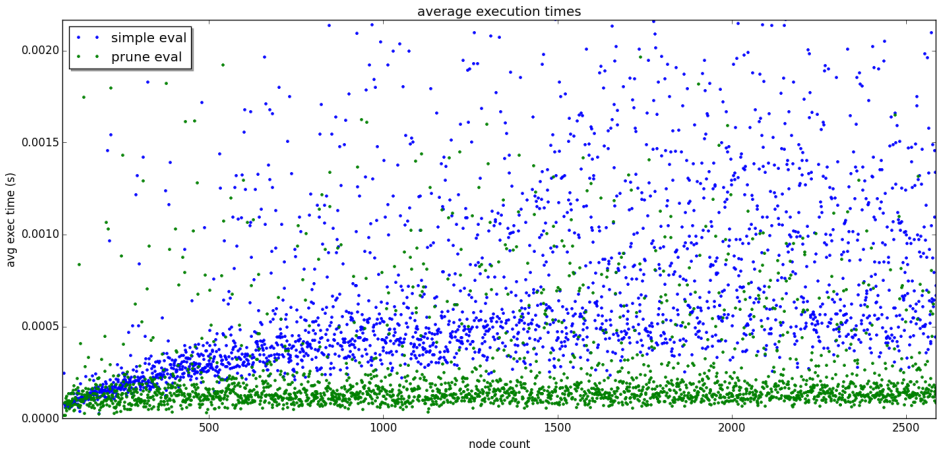

Figure 52: Runtime (in seconds) for evaluation with and without pruning strategies for random formulae up to length 2500.

This could happen because, by pruning algorithms, the number of the nodes in the tree is not equal to the number of the nodes which must be evaluated. If the pruning

algorithm is efficient, then we can hope that only a minor part of the nodes must be evaluated to get the final result of the expression. Figure 53 displays that the number of nodes which cannot be pruned is not growing linearly with the length of the expression, i.e., the total number of nodes in the expression tree. The count of the not pruned nodes is near constant in the average even if the size of the formulae is growing. This explains the near constant execution time of our pruning algorithm.

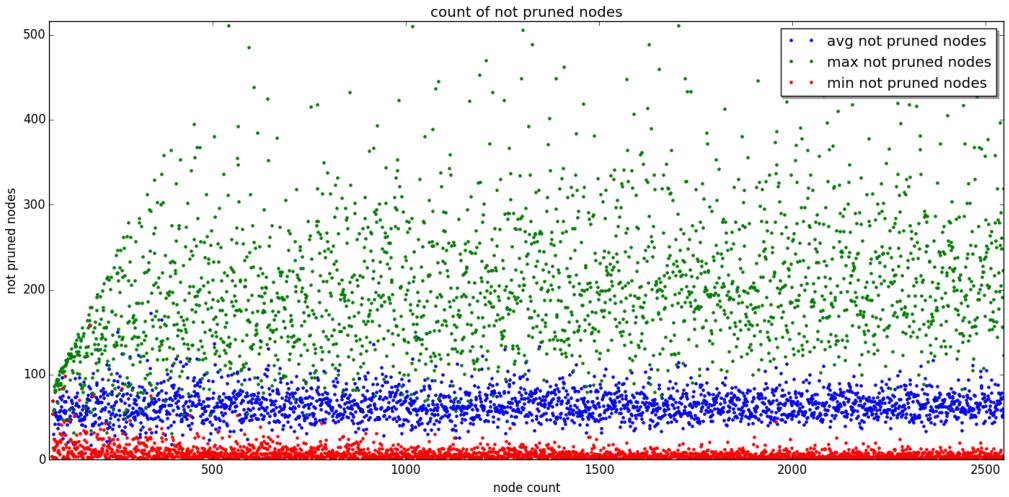

Figure 53: The number of not pruned nodes vs. the total number of nodes.

# Chapter 6

# CONCLUSION

Various logic systems are applied in various fields, e.g., in programming, hardware design, fuzzy technology. There are well-known techniques to speed up the evaluation of Boolean logic expression trees and similar techniques are used at game trees in alpha-beta pruning algorithms.

In this research we considered a kind of extended game trees, Boolean, and three of the most known fuzzy logics (Gödel, Lukasiewicz, and product logics) expression trees. Various pruning techniques have been shown to speed up the evaluation of these expression trees. These techniques can be used to evaluate expressions where sum and product operations can also be found. Similar optimization techniques could help in fast evaluations of various expressions (other types of trees).

Finite expression trees have been considered. In these trees children nodes send their data to their parents and thus, the whole process is finite and can be modeled by evaluation techniques.

Considering a kind of parallel approach in which various processors can work together on the evaluation, these types of techniques can easily reduce the number of required processors and also to reduce their costs. Therefore, the presented algorithms can reduce algorithmic costs (time, space, number of agents/processors) in decision making systems and in various extensions of two-player zero-sum games.

Artificial cognitive systems, CogInfoCom systems [27] and other systems based on artificial/computational intelligence can also be used in a more efficient way.

It is shown that reordering the tree branches, by evaluating first, the shorter ones, may also help a lot to save time and energy, especially, when the evaluation costs of the leaves (or other nodes) consumes a large amount of energy or other limited source.

Since some of the considered many-valued logics (e.g., product logic) are closely related to a system working with probabilities, we believe that our techniques can be used not only in various applications of fuzzy logics and fuzzy systems, but also in decision making and in other disciplines working with probabilities, to make the decisions faster, and to compute results with less efforts. As one could see depending on the length of the formula its evaluation can be done very efficiently, in average.

The experimental results show the efficiency of our proposed techniques. The execution time of the simple algorithm has a growing tendency with the increasing number of nodes. Contrary, the execution time of the algorithm using pruning strategies seems to be like constant in the average. We noticed that the number of nodes which can be pruned is growing linearly with the length of the expression, i.e., the total number of nodes in the expression tree, while the remaining nodes to evaluate is near constant in the average even the size of the formulae is growing. This explains the near constant execution time of our pruning algorithm.

In [9,25] fuzzy logic systems are generalized by using interval-values. It is one of the plans for future work to deal with evaluations in this more general interval-valued

logic. It is also an interesting task to deal with some kinds of generalizations of games, e.g., [19,20,21,26] where various generalizations of decisions and games are studied and work on various evaluation techniques.

# REFERENCES


[1] Bell, J., & Machover, M. (1977). *A Course In Mathematical Logic*, North-Holland, Amsterdam.

[2] Hájek, P. (1998). Metamathematics of fuzzy logic. *Trends in Logic*, 4, Kluwer Academic Publishers, Dordrech.

[3] Barwise, J., & Etchemendy, J. (1987). *The liar: An Essay on Truth and Circularity*. Oxford, Oxford University Press, New York.

[4] Gödel, K. (1932). Zum intuitioonischen aussagenkalkül. Anzeigner Akademie der Wissenschaften im Wien, *Mathematish-Naturwissenschaftliche Klasse*, 69, 65-66; On the Intuitionistic Propositional Calculus, reprinted in *Kurt Gödel, Collected Works* (1986), 1, Oxford University Press, New York.

[5] Gottwald, S. (2015, March 5). Many-valued logic. The Stanford Encyclopedia of Philosophy (Spring 2015 Edition), Edward N. Zalta (ed.), Retrieved from http://plato.stanford.edu/entries/logic-manyvalued/, (First published Tue Apr 25, 2000; substantive revision Thu Mar 5, 2015).

[6] Lukasiewicz, J. (1970). *Selected Works Studies in Logic and The Foundations of Mathematics*. North-Holland, Amsterdam.

[7] Gottwald, S. (2015). Many-valued and fuzzy logics. Kacprzyk, Janusz, Pedrycz, Witold (Eds.), *Springer Handbook of Computational Intelligence*, 7-29.

[8] Baaz, M., Hajek, P., Krajck, J., & Svejda, D. (1998). Embedding logics into product logic. *Studia Logica*, 61, 35–47.

[9] Nagy, B. (2005). A general fuzzy logic using intervals. Proc of. 6th International Symposium of Hungarian Researchers on Computational Intelligence, Budapest, Hungary, pp. 613-624.

[10] Hájek, P., Godo, L., & Esteva, F. (1996). A complete many-valued logic with product-conjunction. *Archive for Mathematical Logic*, 35, 191-208.

[11] Metcalfe, G., N. Olivetti, N., & Gabbay, D. (2004). Analytic calculi for product logics. *Archive for Mathematical Logic*, 43, 859-889.

[12] Adillon, R., & Verdu, V. (1998). On product logic. *Soft Computing*, 2(3), 141-146.

[13] Zadeh, L. (1996). Fuzzy sets, fuzzy logic, and fuzzy systems: selected papers by Lotfi A. Zadeh. (G. J. Klir and B. Yuan, Eds.,) *Advances in Fuzzy Systems – Applications and Theory*, 6, World Scientific, River Edge, NJ, USA.

[14] Shoenfield, J. (2001). *Mathematical Logic*. (2nd ed.), A K Peters.

[15] Nagy, B. (2005). Many-valued logics and the logic of the C programming language. Proc. of ITI 2005: 27th International Conference on Information Technology Interfaces (IEEE), Cavtat, Croatia, pp. 657-662.

[16] Basbous, R., Nagy, B., & Tajti, T. (2015), Short circuit evaluations in Gödel type logic. Proc. of FANCCO 2015: 5th International Conference on Fuzzy and Neuro Computing, Advances in Intelligent Systems and Computing 415 (Springer), Hyderabad, India, pp.119-138, doi:10.1007/978-3-319-27212-2_10.

[17] Rich, E., & Knight, K. (1991). *Artificial Intelligence*. McGraw-Hill Inc., New York.

[18] Russell, S., & Norvig, P. (2003). *Artificial Intelligence, a Modern Approach*. Prentice-Hall, New Jersey.

[19] Melkó, E., & Nagy, B. (2007). Optimal strategy in games with chance nodes. *Acta Cybernetica*, 18, 171-192.

[20] Basbous, R., & Nagy, B. (2014). Generalized game trees and their evaluation. Proc. of CogInfoCom 2014: 5th IEEE International Conference on Cognitive Infocommunications, Vietri sul Mare, Italy, pp. 55-60.

[21] Basbous, R., & Nagy, B. (2015). Strategies to fast evaluation of tree networks. *Acta Polytechnica Hungarica*, 12(6), 127-148, doi: 10.12700/APH.12.6.2015.6.8.

[22] Kernighan, B.W., & Ritchie, D.M. (1988). *The C Programming Language*. (2nd ed.), Prentice Hall, Englewood Cliffs, NJ.

[23] Knuth, D.E., & Moore, R.W. (1975). An analysis of alpha-beta pruning, *Artificial Intelligence*, 6, 293-326.

[24] Kundu, S., & Chen, J. (1998). Fuzzy logic or Lukasiewicz logic: a clarification. *Fuzzy Sets and System*s, 95, 369-379.

[25] Nagy, B.(2006). Reasoning by intervals. Proc. of Diagrams 2006: Fourth International Conference on the Theory and Application of Diagrams, Stanford, CA, USA, LNCS-LNAI 4045, pp. 145-147.

[26] Lakatos, G., & Nagy, B. (2004). Games with few players. Proc. of ICAI'2004: 6th International Conference on Applied Informatics, Eger, Hungary, pp. 187-196.

[27] Baranyi, P., & Csapo, A. (2010). Cognitive infocommunications: CogInfoCom. Proc. of 11th CINTI: IEEE International Symposium on Computational Intelligence and Informatics, Budapest, Hungary, pp. 141-146.

[28] Basbous, R., Tajti, T. & Nagy, B. (2016), Fast evaluations in product logic. The 2016 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 2016), Vancouver, Canada. (Accepted)

# APPENDIX

## Appendix A: The Python Program for Evaluating Formulae in Gödel Type Logic

```
def getNode(nodes, N1):
    nt, nv, nc, na = nodes[N1]
    nodes[N1] = nt, nv, nc, na+1
    return nodes[N1][:3]

def NodeType(nodes, N1):
    nt, nv, nc, na = nodes[N1]
    nodes[N1] = nt, nv, nc, na+1
    return nodes[N1][0]

def NodeVal(nodes, N1):
    nt, nv, nc, na = nodes[N1]
    nodes[N1] = nt, nv, nc, na+1
    return nodes[N1][1]

def NodeSucc(nodes, N1):
    nt, nv, nc, na = nodes[N1]
    nodes[N1] = nt, nv, nc, na+1
    return nodes[N1][2]

def getChild(nodes, N1, n):
    nt, nv, nc, na = nodes[N1]
    nodes[N1] = nt, nv, nc, na+1
    nt, nv, nc, _ = nodes[N1]
    if n < len(nc):
        return nc[n]

def getChildNode(nodes, N1, n):
    nt, nv, nc, na = nodes[N1]
    nodes[N1] = nt, nv, nc, na+1
    nt, nv, nc, _ = nodes[N1]
    if n < len(nc):
        return nodes[nc[n]][:3]

def getSuccessors(nodes, N1, n):
    nt, nv, nc, na = nodes[N1]
    nodes[N1] = nt, nv, nc, na+1
    nt, nv, nc, _ = nodes[N1]
    if n < len(nc):
        return nc[n]
    else:
        return None

def isLeaf(nodes, N1, n):
    nt, nv, nc, na = nodes[N1]
    nodes[N1] = nt, nv, nc, na+1
```

```
    return NodeType(nodes, getChild(nodes, N1, n)) == NODE_TYPE_LEAF

def Check_Leaves(nodes, N1): # "A function to check of all the connected leaves
have non zero values or there no more NEG nodes connected in the successors"
    nt, nv, nc, na = nodes[N1]
    nodes[N1] = nt, nv, nc, na+1
    Flag = 1
    N1Type, N1Val, N1Successors, _ = nodes[N1]

    if NodeType(nodes, N1) == NODE_TYPE_LEAF:
        if N1Val == 0:
            Flag = 0  #"flag = zero means that there is at least one leaf equal zero"
    else:
        if N1Type == NODE_TYPE_NEG:   #"if extra more negation node found in the
lower levels:
            #cut the search and set the flag to zero"
            Flag = 0
            return Flag
        else:
            for N1i in N1Successors:
                if not Check_Leaves(nodes, N1i) == 1:
                    Flag = 0
                    return Flag

    return Flag
#end Check_Leaves

def IMPPrune(nodes, N1, alfa, beta):  #The general function to evaluate IMP nodes"
    if isLeaf(nodes, N1, 0) and  isLeaf(nodes, N1, 1):  #if we have the case left and
right branches are leaves "
        N1LeftVal = NodeVal(nodes, getChild(nodes, N1,0))
        N1RightVal = NodeVal(nodes, getChild(nodes, N1,1))
        if N1LeftVal <= N1RightVal: # tt
            v = 1
        else:
            v = N1RightVal
    elif isLeaf(nodes, N1, 0) or  isLeaf(nodes, N1, 1):        #  "to evaluate IMP node if
one of its successors is a leaf "
        v = IMPLeaf(nodes, N1, alfa, beta)
    else:
        N1leftType = NodeType(nodes, getChild(nodes, N1,0))
        N1rightType = NodeType(nodes, getChild(nodes, N1,1))
        #"to evaluate IMP node if its two successors are nodes"
        if N1leftType == NODE_TYPE_MIN:
            if N1rightType == NODE_TYPE_MAX:
                v = IMPMinMax(nodes, N1, alfa, beta)
            elif N1rightType == NODE_TYPE_MIN:
                v = IMPMinMin(nodes, N1, alfa, beta)
            elif N1rightType == NODE_TYPE_NEG:
                v = IMPMinNeg(nodes, N1, alfa, beta)
```

```
            else:  # if N1rightType == NODE_TYPE_IMP:
                v = IMPMinIMP(nodes, N1, alfa, beta)

        elif N1leftType == NODE_TYPE_MAX:
            if N1rightType == NODE_TYPE_MAX:
                v = IMPMaxMax(nodes, N1, alfa, beta)
            elif N1rightType == NODE_TYPE_MIN:
                v = IMPMaxMin(nodes, N1, alfa, beta)
            elif N1rightType == NODE_TYPE_NEG:
                v = IMPMaxNeg(nodes, N1, alfa, beta)
            else:  #if N1rightType == NODE_TYPE_IMP:
                v = IMPMaxIMP(nodes, N1, alfa, beta)

        elif N1leftType == NODE_TYPE_IMP:
            if N1rightType == NODE_TYPE_MAX:
                v = IMPIMPMax(nodes, N1, alfa, beta)
            elif N1rightType == NODE_TYPE_MIN:
                v = IMPIMPMin(nodes, N1, alfa, beta)
            elif N1rightType == NODE_TYPE_NEG:
                v = IMPIMPNeg(nodes, N1, alfa, beta)
            else:  #if N1rightType == NODE_TYPE_IMP:
                v = IMPIMPIMP(nodes, N1, alfa, beta)

        else:       # if N1leftType == NODE_TYPE_NEG:
            N1left = getChild(nodes, N1,0)
            non_zero_leaves = Check_Leaves(nodes, N1left)
            if non_zero_leaves == 1:
                #cut and
                v = 1  #"since the right side will be ≥ 0 always"
            elif N1rightType == NODE_TYPE_MAX:
                v = IMPNegMax(nodes, N1, alfa, beta)
            elif N1rightType == NODE_TYPE_MIN:
                v = IMPNegMin(nodes, N1, alfa, beta)
            elif N1rightType == NODE_TYPE_NEG:
                v = IMPNegNeg(nodes, N1, alfa, beta)
            else:  #if N1rightType == NODE_TYPE_IMP:
                v = IMPNegIMP(nodes, N1, alfa, beta)

    nt, nv, nc, na = nodes[N1]
    nodes[N1] = (nt, float(v), nc, na)

    return float(v)
#end IMPPrune

def IMPLeaf(nodes, N1, alfa, beta): #" A function to evaluate IMP node if one of its
successors is a leaf"
    N1left, N1right = getChild(nodes, N1,0), getChild(nodes, N1,1)
    N1leftType, N1leftVal, _ = getChildNode(nodes, N1,0)
    N1rightType, N1rightVal, N1rightSuccessors = getChildNode(nodes, N1,1)
    #N1rightType, N1right,  = getChild(N1,1)
```

```
    if  N1leftType is NODE_TYPE_LEAF:   #"we have left side a leaf"
        if  N1leftVal == 0:
            #cut and
            return  1.0   #"since right side will be greater or equal zero"
        else:
            #if left_child  = N1left
            left_child = N1leftVal
            if N1rightType == NODE_TYPE_MAX:
                right_child = -2
                for N1righti in N1rightSuccessors:
                    right_child = max(right_child, Prune(nodes, N1righti, alfa, beta))
                    if right_child >= left_child:
                        #cut and
                        return 1.0          #"since right side will be greater or equal left child
IMP=1"

            elif N1rightType == NODE_TYPE_MIN:
                right_child = +2
                for N1righti in N1rightSuccessors:
                    right_child = min(right_child, Prune(nodes, N1righti, alfa, beta))

            elif N1rightType == NODE_TYPE_IMP:
                right_child = IMPPrune(nodes, N1right, alfa, beta)

            elif N1rightType == NODE_TYPE_NEG:
                non_zero_leaves = Check_Leaves(nodes, N1right)
                if non_zero_leaves == 1:
                    right_child = 0.0              #" 1 means that all the connected successors
are non zeros"
                else:
                    right_child = NEGPrune(nodes, N1right, alfa, beta)

    else:     #"we have right side as a leaf"
        if  N1rightVal == 1:
            #cut and
            return  1.0   #"since left side will be less or equal one"
        else:
            N1rightType, N1rightVal, N1rightSuccessors = getChildNode(nodes, N1, 1)
            right_child = N1rightVal
            if N1leftType == NODE_TYPE_MAX:
                left_child = -2
                left_child = max(left_child, MAXPrune(nodes, N1left, alfa, beta))
            elif N1leftType == NODE_TYPE_MIN:
                left_child = +2
                left_child = min(left_child, MINPrune(nodes, N1left, alfa, beta))
            elif N1leftType == NODE_TYPE_IMP:
                left_child = IMPPrune(nodes, N1left, alfa, beta)
            elif N1leftType == NODE_TYPE_NEG:
                non_zero_leaves = Check_Leaves(nodes, N1left)
```

```
            if non_zero_leaves == 1:      #" 1 means that all the connected successors
are non zeros"
                #cut and
                return 1  #"since right side will be greater or equal zero"
            else:
                left_child = NEGPrune(nodes, N1left, alfa, beta)

    if left_child <= right_child:
        return 1
    else:
        return right_child
#end IMPLeaf

def IMPMinMax(nodes, N1, alfa, beta):    #" A function to evaluate IMP node if it
has Min at left and Max at right"
    left_child = +2
    right_child =  -2
    #"by evaluating the IMP node left and right successors one by one (in Parallel), if
we got max (right side) ≥ min (left side) or if zero found in min (left side) : cut, IMP
=1"
    N1leftType, N1leftVal, N1leftChildren = getChildNode(nodes, N1, 0)
    N1rightType, N1rightVal, N1rightChildren = getChildNode(nodes, N1, 1)
    N1lefti = 0
    N1righti = 0

    while N1lefti < len(N1leftChildren) or N1righti < len(N1rightChildren):
        if N1lefti < len(N1leftChildren):
            left_child = min (left_child, Prune(nodes, N1leftChildren[N1lefti], alfa, beta))
            N1lefti += 1
            if left_child == 0:
                #cut and
                return 1

        if N1righti < len(N1rightChildren):
            right_child = max(right_child, Prune(nodes, N1rightChildren[N1righti], alfa,
beta))
            N1righti += 1
            if left_child <= right_child:
                #cut and
                return 1
    if left_child <= right_child:
        return 1
    else:
        return right_child              #"at this point we have right child  < left child"
#end IMPMinMax

def IMPMinMin(nodes, N1, alfa, beta): #" A function to evaluate IMP node if it has
Min at left and Min at right"
    left_child = +2
    right_child =  +2
```

```
   N1left, N1right = getChild(nodes, N1,0), getChild(nodes, N1,1)
   N1leftType, N1leftVal, _ = getChildNode(nodes, N1,0)
   N1rightType, N1rightVal, _ = getChildNode(nodes, N1,1)
   for N1lefti in NodeSucc(nodes, N1left):
      left_child = min(left_child, Prune(nodes, N1lefti, alfa, beta))
      if left_child == 0:    #"when evaluating left child, if we got min = zero: cut, IMP
=1"
         #cut and
         return 1
   for N1righti in NodeSucc(nodes, N1right):
      right_child = min(right_child, Prune(nodes, N1righti, alfa, beta))

   if left_child <= right_child:
      return 1
   else:
      return right_child
#end IMPMinMin

def IMPMinNeg(nodes, N1, alfa, beta):  #" A function to evaluate IMP node if it has
Min at left and Neg at right"
   left_child = +2
   N1left, N1right = getChild(nodes, N1,0), getChild(nodes, N1,1)
   N1leftType, N1leftVal, _ = getChildNode(nodes, N1,0)
   N1rightType, N1rightVal, _ = getChildNode(nodes, N1,1)
   non_zero_leaves = Check_Leaves(nodes, N1right)
   if non_zero_leaves == 1:          #" all connected leaves are non zeros: NEG =0"
      for N1lefti in NodeSucc(nodes, N1left):
         left_child = min(left_child, Prune(nodes, N1lefti,  alfa, beta))
         if left_child == 0:      #"at this point we check the value of each successor
(MIN) node  if zero value found"
            #cut and
            return 1
         else:
            #cut and
            return 0     #"since right side is equal zero and at this point, the left child is
>0"
   else :
      for N1lefti in NodeSucc(nodes, N1left):
         left_child = min(left_child, Prune(nodes, N1lefti, alfa, beta))
      right_child = NEGPrune(nodes, N1right, alfa, beta)
      nt, nv, nc, na = nodes[N1left]
      nodes[N1left] = (nt, left_child, nc, na+1)
   if left_child <= right_child:
      return 1
   else:
      return right_child
#end IMPMinNeg

def IMPMinIMP(nodes, N1, alfa, beta): #" A function to evaluate IMP node if it has
Min at left and Max at right"
```

```
    left_child = +2
    N1left, N1right = getChild(nodes, N1,0), getChild(nodes, N1,1)
    N1leftType, N1leftVal, _ = getChildNode(nodes, N1,0)
    N1rightType, N1rightVal, _ = getChildNode(nodes, N1,1)
    for N1lefti in NodeSucc(nodes, N1left):
        left_child = min(left_child, Prune(nodes, N1lefti, alfa, beta))
        if left_child == 0:     #"when evaluating left child, if we got min = zero: cut,
IMP =1"
        #cut and
            return 1
    nt, nv, nc, na = nodes[N1left]
    nodes[N1left] = (nt, left_child, nc, na+1)
    right_child =  IMPPrune(nodes, N1right, alfa, beta)

    if left_child <= right_child:
        return 1
    else:
        return right_child

#end IMPMinIMP


def IMPMaxMax(nodes, N1, alfa, beta): #" A function to evaluate IMP node if it has
Max at left and Max at right"
    left_child = -2
    right_child =  -2
    N1left, N1right = getChild(nodes, N1,0), getChild(nodes, N1,1)
    N1leftType, N1leftVal, _ = getChildNode(nodes, N1, 0)
    N1rightType, N1rightVal, _ = getChildNode(nodes, N1, 1)
    for N1lefti  in NodeSucc(nodes, N1left):
        left_child = max(left_child, Prune(nodes, N1lefti, alfa, beta))
    for N1righti in NodeSucc(nodes, N1right):
        right_child = max(right_child, Prune(nodes, N1righti, alfa, beta))
        if right_child == 1:
            #cut and
            return 1    #"Max is found: right ≥ left and no need to evaluate the left side"
        if left_child <= right_child:
            return 1
    return right_child  # TT reorg

#end IMPMaxMax

def IMPMaxMin(nodes, N1, alfa, beta): #" A function to evaluate IMP node if it has
Max at left and Min at right"
    left_child = -2
    right_child =  +2
    N1left, N1right = getChild(nodes, N1,0), getChild(nodes, N1,1)
    N1leftType, N1leftVal, _ = getChildNode(nodes, N1, 0)
    N1rightType, N1rightVal, _ = getChildNode(nodes, N1, 1)
    for N1righti in NodeSucc(nodes, N1right):
```

```
        right_child = min(right_child, Prune(nodes, N1righti, alfa, beta))
    for N1lefti  in NodeSucc(nodes, N1left):
        left_child = max(left_child, Prune(nodes, N1lefti, alfa, beta))
        if left_child > right_child:   #"when we got left  ≥ right: IMP = right side"
            #cut all the left_child and return the value of right_child
            return right_child

    return 1   #"Since at this point left side is less than the right side"

#end IMPMaxMin

def IMPMaxNeg(nodes, N1, alfa, beta): #" A function to evaluate IMP node if it has
Max at left and Neg at right"
    left_child = -2
    N1left, N1right = getChild(nodes, N1,0), getChild(nodes, N1,1)
    N1leftType, N1leftVal, _ = getChildNode(nodes, N1, 0)
    N1rightType, N1rightVal, _ = getChildNode(nodes, N1, 1)
    non_zero_leaves = Check_Leaves(nodes, N1right)
    if non_zero_leaves == 1:
        for N1lefti in NodeSucc(nodes, N1left):
            left_child = max(left_child, Prune(nodes, N1lefti, alfa, beta))
            if left_child > 0:
                #cut and
                return 0   #"since left side will be always greater than 0 (right side"
    else: #"for the case if we have some zero values or more NEG nodes connected"
        for N1lefti in NodeSucc(nodes, N1left):
            left_child = max(left_child, Prune(nodes, N1lefti, alfa, beta))
        right_child = NEGPrune(nodes, N1right, alfa, beta)

    if left_child <= right_child:
        return 1
    else:
        return right_child

#end IMPMaxNeg
def IMPMaxIMP(nodes, N1, alfa, beta): #" A function to evaluate IMP node if it has
Max at left and IMP at right"
    left_child = -2
    N1left, N1right = getChild(nodes, N1,0), getChild(nodes, N1,1)
    N1leftType, N1leftVal, _ = getChildNode(nodes, N1,0)
    N1rightType, N1rightVal, _ = getChildNode(nodes, N1,1)
    for N1lefti in NodeSucc(nodes, N1left):
        left_child = max(left_child, Prune(nodes, N1lefti, alfa, beta))
    right_child = IMPPrune(nodes, N1right, alfa, beta)

    if left_child <= right_child:
        return 1
    else:
        return right_child
#end IMPMaxIMP
```

```
def IMPIMPMax(nodes, N1, alfa, beta): #" A function to evaluate IMP node if it has
IMP at left and Max at right"
   right_child = -2
   N1left, N1right = getChild(nodes, N1,0), getChild(nodes, N1,1)
   N1leftType, N1leftVal, _ = getChildNode(nodes, N1,0)
   N1rightType, N1rightVal, _ = getChildNode(nodes, N1,1)
   for N1righti in NodeSucc(nodes, N1right):
      right_child = max(right_child, Prune(nodes, N1righti, alfa, beta))
      if right_child == 1:
         #cut and
         return 1
   left_child = IMPPrune(nodes, N1left, alfa, beta)
   if left_child <= right_child:
      return 1
   else:
      return right_child
#end IMPIMPMax

def IMPIMPMin(nodes, N1, alfa, beta): #" A function to evaluate IMP node if it has
IMP at left and Min at right"
   right_child = +2.0
   N1left, N1right = getChild(nodes, N1,0), getChild(nodes, N1,1)
   N1leftType, N1leftVal, _ = getChildNode(nodes, N1, 0)
   N1rightType, N1rightVal, _ = getChildNode(nodes, N1, 1)
   for N1righti in NodeSucc(nodes, N1right):
      right_child = min(right_child, Prune(nodes, N1righti, alfa, beta))
      if right_child == 0:                #"to evaluate right side while min not found"
         break  # TT removed potential endless loop !!!!
   left_child = IMPPrune(nodes, N1left, alfa, beta)
   if left_child <= right_child:
      return 1
   else:
      return right_child
#end IMPIMPMin

def IMPIMPNeg(nodes, N1, alfa, beta): #" A function to evaluate IMP node if it has
IMP at left and NEG at right"
   N1left, N1right = getChild(nodes, N1,0), getChild(nodes, N1,1)
   N1leftType, N1leftVal, _ = getChildNode(nodes, N1, 0)
   N1rightType, N1rightVal, _ = getChildNode(nodes, N1, 1)
   non_zero_leaves = Check_Leaves(nodes, N1right)
   if non_zero_leaves == 1:              #"In case all the connected leaves have non zero
values: NEG = 0"
      right_child = 0
   else:
      right_child = NEGPrune(nodes, N1right, alfa, beta)
   left_child = IMPPrune(nodes, N1left, alfa, beta)
   if left_child <= right_child:
      return 1
```

```
    else:
        return right_child
#end IMPIMPNeg

def IMPIMPIMP(nodes, N1, alfa, beta): #" A function to evaluate IMP node with
two connected IMP nodes"
    N1left, N1right = getChild(nodes, N1,0), getChild(nodes, N1,1)
    N1leftType, N1leftVal, _ = getChildNode(nodes, N1,0)
    N1rightType, N1rightVal, _ = getChildNode(nodes, N1,1)
    right_child = IMPPrune(nodes, N1right, alfa, beta)
    left_child = IMPPrune(nodes, N1left, alfa, beta)
    if left_child <= right_child:
        return 1
    else:
        return right_child
#end IMPIMPIMP

def IMPNegMax(nodes, N1, alfa, beta): #" A function to evaluate IMP node if it has
Neg at left and Max at right"
    right_child = -2
    N1left, N1right = getChild(nodes, N1,0), getChild(nodes, N1,1)
    N1leftType, N1leftVal, _ = getChildNode(nodes, N1,0)
    N1rightType, N1rightVal, _ = getChildNode(nodes, N1,1)
    for N1righti in NodeSucc(nodes, N1right):
        right_child = max(right_child, Prune(nodes, N1righti, alfa, beta))
        if right_child == 1:
            #cut and
            return 1     #"Max is found: right ≥ left and no need to evaluate the left side"
    left_child = NEGPrune(nodes, N1left, alfa, beta)
    if left_child <= right_child:
        return 1
    else:
        return right_child
#end IMPNEGMax

def IMPNegMin(nodes, N1, alfa, beta): #" A function to evaluate IMP node if it has
Neg at left and Min at right"
    right_child = +2
    N1left, N1right = getChild(nodes, N1,0), getChild(nodes, N1,1)
    N1leftType, N1leftVal, _ = getChildNode(nodes, N1,0)
    N1rightType, N1rightVal, _ = getChildNode(nodes, N1,1)
    for N1righti in NodeSucc(nodes, N1right):
        right_child = min(right_child, Prune(nodes, N1righti, alfa, beta))
    left_child = NEGPrune(nodes, N1left, alfa, beta)
    if left_child <= right_child:
        return 1
    else:
        return right_child
#end IMPNEGMin
```

```
def IMPNegNeg(nodes, N1, alfa, beta): #" A function to evaluate IMP node if it has
Neg at left and Neg at right"
   N1left, N1right = getChild(nodes, N1,0), getChild(nodes, N1,1)
   N1leftType, N1leftVal, _ = getChildNode(nodes, N1, 0)
   N1rightType, N1rightVal, _ = getChildNode(nodes, N1, 1)
   non_zero_leaves = Check_Leaves(nodes, N1right)
   if non_zero_leaves == 1:              #"In case all the connected leaves have non zero
values: NEG = 0"
      right_child = 0
   else:
      right_child = NEGPrune(nodes, N1right, alfa, beta)
   left_child = NEGPrune(nodes, N1left, alfa, beta)
   if left_child <= right_child:
      return 1
   else:
      return right_child
#end IMPNEGNeg

def IMPNegIMP(nodes, N1, alfa, beta): #" A function to evaluate IMP node if it has
Neg at left and Neg at right"
   N1left, N1right = getChild(nodes, N1,0), getChild(nodes, N1,1)
   N1leftType, N1leftVal, _ = getChildNode(nodes, N1,0)
   N1rightType, N1rightVal, _ = getChildNode(nodes, N1,1)
   right_child = IMPPrune(nodes, N1right, alfa, beta)
   left_child = NEGPrune(nodes, N1left, alfa, beta)
   if left_child <= right_child:
      return 1
   else:
      return right_child
#end IMPNEGIMP

def MAXPrune(nodes, N1, alfa, beta): #"A function to evaluate Max nodes"
   if NodeType(nodes, N1) == NODE_TYPE_LEAF:
       return NodeVal(nodes, N1)
   else:
      v = -2.0
      for N1i in NodeSucc(nodes, N1):
         next_op_type = NodeType(nodes, N1i)
         if next_op_type == NODE_TYPE_MAX:
            v = max(v, MAXPrune(nodes, N1i, alfa, beta))
         elif next_op_type == NODE_TYPE_MIN:
            v = max(v, MINPrune(nodes, N1i , alfa, beta))
         elif next_op_type == NODE_TYPE_IMP:
            v = max(v, IMPPrune(nodes, N1i, alfa, beta))
         elif next_op_type == NODE_TYPE_NEG:
            non_zero_leaves = Check_Leaves(nodes, N1i)
            if non_zero_leaves == 1:
               v = 0
            else:
               v = max(v, NEGPrune(nodes, N1i, alfa, beta))
```

```
            else: # if bextop == NODE_TYPE_LEAVE
               v = max(v, NodeVal(nodes, N1i))
            if v == 1.0 or v >= beta:
               break
            alfa = min(alfa, v)
      nt, nv, nc, na = nodes[N1]
      nodes[N1] = (nt, v, nc, na+1)
      return v
#end MAXPrune

def MINPrune(nodes, N1, alfa, beta): #"A function to evaluate Min nodes"
      if NodeType(nodes, N1) == NODE_TYPE_LEAF:
         return NodeVal(nodes, N1)
      else:
         v = +2.0
         for N1i in NodeSucc(nodes, N1):
            next_op = NodeType(nodes, N1i)
            if next_op == NODE_TYPE_MAX:
               v = min(v, MAXPrune(nodes, N1i, alfa, beta))
            elif next_op == NODE_TYPE_MIN:
               v = min(v,  MINPrune(nodes, N1i, alfa, beta))
            elif next_op == NODE_TYPE_IMP:
               v = min(v, IMPPrune(nodes, N1i, alfa, beta))
            elif next_op == NODE_TYPE_NEG:
               non_zero_leaves = Check_Leaves(nodes, N1i)
               if non_zero_leaves == 1:
                  v = 0
               else:
                  v = min(v, NEGPrune(nodes, N1i, alfa, beta))
            else: # if bextop == NODE_TYPE_LEAVE
               v = min(v, NodeVal(nodes, N1i))
            if v == 0.0 or v <= alfa:
               break
            beta = max(beta, v)
      nt, nv, nc, na = nodes[N1]
      nodes[N1] = (nt, v, nc, na+1)
      return v
#end MINPrune

def NEGPrune(nodes, N1, alfa, beta): #"A function to evaluate Neg nodes"
      Child = NodeSucc(nodes, N1)[0]
      ChildType = NodeType(nodes, Child)

      leav = Check_Leaves(nodes, Child)

      if leav == 1:
         v = 0.0
      else:
         if ChildType == NODE_TYPE_LEAF:
            v = 0.0
```

```
        elif ChildType == NODE_TYPE_MAX:
            v = MAXPrune(nodes, Child, alfa, beta)
        elif ChildType == NODE_TYPE_MIN:
            v = MINPrune(nodes, Child, alfa, beta)
        elif ChildType == NODE_TYPE_IMP:
            v = IMPPrune(nodes, Child, alfa, beta)
        else:    # if ChildType == NODE_TYPE_NEG:  by TT
            v = NEGPrune(nodes, Child, alfa, beta)
        if v == 0:
            v = 1.0
        else:
            v = 0.0

    nt, nv, nc, na = nodes[N1]
    nodes[N1] = (nt, v, nc, na+1)
    return v
#end NEGPrune

def Prune(nodes, N1, alfa, beta):
    nt, nv, nc = getNode(nodes, N1)
    if nt == NODE_TYPE_NEG:
        v = NEGPrune(nodes, N1, alfa, beta)

    elif nt == NODE_TYPE_MIN:
        v = MINPrune(nodes, N1, alfa, beta)

    elif nt == NODE_TYPE_MAX:
        v = MAXPrune(nodes, N1, alfa, beta)

    elif nt == NODE_TYPE_IMP:
        v = IMPPrune(nodes, N1, alfa, beta)

    else: # NODE_TYPE_LEAF
        v = NodeVal(nodes, N1)
    nt, nv, nc, na = nodes[N1]
    nodes[N1] = (nt, v, nc, na+1)
    return v
# end Prune
```