Senior Thesis

Application of Tree Contraction to Boolean Formula Evaluation

Runjia(Ricki) Su, Ruochun(Rachel) Wang
Advisor: Professor Steven Lindell (Haverford College)

Submitted in Partial Fulfillment of
The Requirements of the B.A. Degree in Computer Science
Bryn Mawr College
Spring 2020
Abstract

This thesis presents an application of tree contraction to the Boolean Formula Evaluation problem. We have shown that the use of a parallel Tree Contraction algorithm guarantees that the worst case run time of evaluating a Boolean formula is in $U_{E}$-uniform $NC^{1}$. 
Acknowledgements

First and foremost, we wish to express our sincere appreciation to our advisor, Professor Steven Lindell, for his valuable guidance and encouragement throughout this thesis process. We also want to thank Professor Deepak Kumar for his thoughtful feedback on our previous drafts.

Special thanks to our families and friends, especially Eva Liu, Jiayu Zhou, and Tori Dang for their love and support during this special time. Additionally, we want to say thank you to our idols BTS (from Ricki) and Yibo Wang (from Rachel) for the energy and hope they brought us.
Contents

1 Introduction 2

2 Motivation 3

3 Preliminaries 4
  3.1 Boolean Formulas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
  3.2 The $NC^1$ complexity class . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
    3.2.1 Constant-Width Branching Program . . . . . . . . . . . . . . . . . . . . . 4
    3.2.2 Logarithm Depth Boolean Formula . . . . . . . . . . . . . . . . . . . . . 5
    3.2.3 Oblivious Multi-Head Automaton . . . . . . . . . . . . . . . . . . . . . 5
    3.2.4 ALOGTIME . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
  3.3 Tree Contraction Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
    3.3.1 Rake-and-Compress . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
    3.3.2 Prune-and-Bypass . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
    3.3.3 Run-time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
  3.4 Tree Properties . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

4 Related Work 10

5 Methodology 12
  5.1 Parsing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
    5.1.1 Reduce Boolean Formula with Negation . . . . . . . . . . . . . . . . . . . 12
    5.1.2 Parsing a String without Negation . . . . . . . . . . . . . . . . . . . . . 14
  5.2 Approaches on Evaluating Parsed Boolean String . . . . . . . . . . . . . . . . 16
    5.2.1 First Approach - Prune & Bypass with Relabelling . . . . . . . . . . . 16
    5.2.2 Second Approach - Precompute Tree Contraction Sequence . . . . . . 20

6 Results 25

7 Future Work 26
Chapter 1

Introduction

This thesis aims to build an application/algorithm that uses parallel tree contraction to efficiently, in $U_{E^*}$-uniform $NC^1$ time complexity, evaluate Boolean Formulas.

The Boolean Formula Evaluation Problem is to decide if the value of a given Boolean Formula is True or False. A Boolean Formula is a string that consists of $T$ (True), $F$ (False), $C$ (conjunction, $\land$), $D$ (disjunction, $\lor$), $L$ (left parenthesis), $R$ (right parenthesis), and $N$ (not). Without loss of generality, our work starts without including negation because of the fact that Tree Contraction runs on full binary trees, while negation is an unary operator. However, it is not hard to convert any Boolean Formula with negation to one that doesn’t have it, via De Morgan’s Law. We will show that this conversion runs in in $U_{E^*}$-uniform $NC^1$ time, so it doesn’t add complication [4].

Tree Contraction is a parallel algorithm of removing nodes of a full binary tree until there is only one node left.

The Boolean Formula Evaluation problem consists of two parts: parsing and evaluation. In this thesis, we will provide detailed procedures for both parts. With a $NC^1$ parsing procedure and the adoption of a tree contraction algorithm, we will devise a Boolean evaluation algorithm that is guaranteed to have a worst case run time within $O(\log(n))$. 
Chapter 2

Motivation

The Tree Contraction Problem falls into the $NC^1$ complexity class. The $NC^1$ complexity class has been studied in earlier work [2, 3, 5], yet there is no one precise definition of $NC^1$ problems. There are four main definitions of $NC^1$ problems, namely constant-width branching program, the logarithm depth Boolean formula, oblivious multi-head automaton and $ALOGLTIME$.

A recursive bottom-up algorithm on Boolean Formula Evaluation problem could achieve $O(\log(n))$ run-time on balanced binary parse tree with $n$ nodes, but the performance drops severely if the tree is unbalanced. Tree contraction allows an unbalanced (especially long and skinny) tree of $n$ nodes to achieve $O(\log(n))$ performance when the algorithm runs bottom-up and simultaneous at each level. Our goal is to apply Tree Contraction to achieve the same run-time regardless of whether or not the tree is balanced.
Chapter 3

Preliminaries

3.1 Boolean Formulas

A Boolean Formula is essentially a string, and a string could be expressed by a structure, illustrated as follows. Suppose we have a string \( P \). Let \( S = \{1, \ldots, n\} \) be the set of all indices of characters in the string. The associated structure \( \sigma \) of \( P \) contains the set \( S \), an ordering, and a partition of all elements in \( S \). A Boolean formula could be expressed as such a structure with a binary relation \( \leq \) and a specific partition that involves sets of truth value(T, F), conjunction(C), disjunction(D), left parenthesis(L), right parenthesis(R) and negation(N).

For example, given a Boolean formula \( \beta = ((\neg (t \land t)) \lor (\neg f)) \), the set of all indices of characters in \( \beta \) contains integers from 1 to 15.

\[
\begin{array}{cccccccccc}
L & N & L & L & T & C & T & R & R & D \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15
\end{array}
\]

The partition of \( \beta \) is as follows: \( T = \{5, 7\} \), \( F = \{13\} \), \( C = \{6\} \), \( D = \{10\} \), \( L = \{1, 2, 4, 11\} \), \( R = \{8, 9, 14, 15\} \), \( N = \{3, 12\} \). The binary relation \( \leq \) is defined as \( \{(i, j) : i \leq j, \quad i, j \in S\} \). Altogether, we get the structure \( \sigma : (\{1, \ldots, 15\}, \leq, T, F, C, D, L, R, N) \).

3.2 The \( NC^1 \) complexity class

We will expand on the four definitions of \( NC^1 \) mentioned above.

3.2.1 Constant-Width Branching Program

1. Let \([w] = \{1, \ldots, w\}\)

2. Instruction \(< j, f, g > = \) triple, \( j = \) index(input variable \( x_j \)), \( f, g = [w]\rightarrow[w] \)
   \( < j, f, g > \) means "evaluate to \( f \) if \( x_j = 1 \), otherwise evaluate to \( g \)".
3. Width-$w$ Branching Program of length $l = \text{sequence of instructions } \langle j_i, f_i, g_i \rangle$ for $1 \leq i \leq l$.

Let $x_i \in \{0,1\}$, $1 \leq i \leq n$, branching program $P$ gives function $P(x)$, a composition of all functions generated by the instructions in $P$.

In addition to the definitions above, Barrington showed that any language recognized by a fan-in 2, depth $O(\log(n))$ circuit ($NC^1$ circuit) is recognizable by a constant-width polynomial-size branching program [2].

### 3.2.2 Logarithm Depth Boolean Formula

Extending from the last subsection, Barrington et.al showed how to construct an equivalent Boolean formula of polynomial size and the same depth from an $NC^1$ circuit (fan-in 2, depth $O(\log(n))$ circuit). Hence, logarithm depth Boolean formula is another representation of the $NC^1$ complexity class [3].

### 3.2.3 Oblivious Multi-Head Automaton

**Definition 3.2.1.** Holzer et al.: "A $k$-head finite automaton $M$ is data-independent or oblivious if the position of every input-head $i$ after step $t$ in the computation on input $w$ is a function $f_M(|w|, i, t)$ that only depends on $i, t$, and $|w|$" [5].

Furthermore, Holzer et.al showed that an oblivious multi-head automaton captures the $NC^1$ complexity class [5].

### 3.2.4 ALOGTIME

As a special case, $ALOGTIME$, aka “Alternating Logarithm Time”, is not equivalent to $NC^1$. As Russo showed, $ALOGTIME$ is equivalent to $U_{E^*}$-uniform $NC^1$, ”a kind of alternating log time uniformity” [8].

$ALOGTIME$ is based on a multi-tape Turing Machine on which the input tape is randomly accessed, versus scanned sequentially; additionally, an address $i$ can be written in binary onto a special work tape; finally, each state of the Turing machine can read the $i$-th input symbol [4]. Specifically,

$$ALOGTIME = \bigcup_c ATIME(c \cdot \log(n) + c)$$

One important result from $ALOGTIME$ is that the counting function is in $ALOGTIME$, which will be really helpful to our work later on [4].

### 3.3 Tree Contraction Algorithms

Tree contraction is a parallel algorithm for an arbitrary binary tree $T$ that constructs ”an optimal tree contraction sequence for $T”$ [1]. Specifically, it removes nodes of a full binary tree until there is only one node left, the root. There are two state-of-the-art Tree Contraction
Algorithms, namely Prune-and-Bypass and Rake-and-Compress. We will briefly introduce both of them in this section.

### 3.3.1 Rake-and-Compress

There are two main operations that are involved in the Rake-and-Compress Algorithm, defined as follows:

1. **RAKE()**: In parallel, remove all leaves
2. **COMPRESS()**: In parallel, compress each path (chain of nodes with one child) down to one node

The procedure of contraction is to perform RAKE() and COMPRESS() on the tree repeatedly until only the root is left [7]. Although the Rake-and-Compress Algorithm could achieve \(O(\log(n))\) run-time, there is a more precise Algorithm named Prune-and-Bypass.

### 3.3.2 Prune-and-Bypass

Similar to Rake-and-Compress, two basic operations are involved in the Prune-and-Bypass algorithm [1]. Given node \(v, w\), define:

1. **PRUNE**\((v)\): remove \(v\) if \(v\) is a leaf
2. **BYPASS**\((w)\): remove \(w = \text{parent}(v)\) and connect \(w\)’s other child with \(w\)’s parent

The procedure of contraction is given as \(CONTRACT(T)\):

1. Number all leaves from 1 to \(n\)
2. In parallel, for all odd numbered leaves \(v\), **PRUNE**\((v)\) then **BYPASS**(\(\text{parent}(v)\)).
3. Divide remaining leaves by 2
4. Repeat Step 2 until only one node is left

### 3.3.3 Run-time

It was shown that Prune-and-Bypass can run in \(U_E\)-uniform \(NC^1\) (rounds of contraction) on a full binary tree with \(n\) nodes. Rake-and-Compress can run in \(O(\log(n))\) time [4].
3.4 Tree Properties

For two nodes $u, v$ in a tree $T$, we let $u \triangleright v$ denote that $u$ is an ancestor of $v$, and let $u \trianglerighteq v$ to denote $u = v$ or $u \triangleright v$. For any given node $u$ in the tree, define $C(u)$ to be $\{v : v \trianglerighteq u\}$ with the relation $\trianglerighteq$, where $C(u)$ is the set of all ancestors of $u$. We can see that $C(u)$ is chain, as the relation $\trianglerighteq$ is a total order on $C(u)$. Denote $\mu(C(u))$ as the least element in $C(u)$. We can see that $\mu(C(u)) = u$ always holds, as for any node, the least ancestor of that node is itself.

Define a binary operation $u \hat{\ast} v$ as follows:

**Definition 3.4.1.** For $p \in T$, $u \hat{\ast} v = p$ if and only if $p \trianglerighteq u$ and $p \trianglerighteq v$, and for any $q \in T$, if $q \trianglerighteq u$ and $q \trianglerighteq v$, we must have $q \trianglerighteq p$.

In other words, let $u \hat{\ast} v$ denote the least common ancestor of $u$ and $v$. Alternatively, we could interpret $u \hat{\ast} v$ as $\mu(C(u) \cap C(v))$, the least element of the intersection of the chains generated from $u$ and $v$. We then use this property to prove lemmas:

**Lemma 3.4.1.** For any pair of $u, v$ in a tree $T$, there exists a unique node $w$ in $T$ such that $u \hat{\ast} v = w$.

**Proof.** For any $u, v$, $C(u) \cap C(v)$ contains at least one element: the root. Since $C(u) \cap C(v)$ is a chain which is not empty, $\mu(C(u) \cap C(v))$ must exists and there is only one such node $p$ that $\mu(C(u) \cap C(v)) = p$.

We will use this lemma to prove that the least common ancestor operation is commutative, associative and idempotent, but before that, we need another lemma on the property of the chain, as follows:

**Lemma 3.4.2.** For any given node $u$ and its ancestor chain $C(u)$ in the tree $T$, we have $C(u \hat{\ast} v) = C(u) \cap C(v)$.

**Proof.** Since $u \hat{\ast} v$ could be interpreted as $\mu(C(u) \cap C(v))$, it suffices to show that $C(\mu(C(u) \cap C(v))) = C(u) \cap C(v)$. By definition, we know that $C(u) \cap C(v) = C(w)$ for some node $w$ in the tree. We also know that $\mu(C(w)) = w$, thus the equality $C(\mu(C(u) \cap C(v))) = C(w) \cap C(v)$ follows.

**Lemma 3.4.3.** The binary operation $\hat{\ast}$ is commutative, associative and idempotent.

**Proof.**

(i) Commutative:

For any node $u, v$ in tree $T$, since $C(u) \cap C(v) = C(v) \cap C(u)$, we have $u \hat{\ast} v = \mu(C(u) \cap C(v)) = v \hat{\ast} u$.

(ii) Associative:

Given a tree $T$, for any node $u, v, w \in T$, we want to show that $(u \hat{\ast} v) \hat{\ast} w = u \hat{\ast} (v \hat{\ast} w)$.
By definition, we have:

\[ u^{\hat{}}(v^{\hat{}}w) = \mu(C(u) \cap C(v^{\hat{}}w)) \]
\[ = \mu((C(u) \cap C(v)) \cap C(w)) \quad \text{By lemma 3.2} \]
\[ = \mu(C(u^{\hat{}}v) \cap C(w)) \quad \text{By lemma 3.2} \]
\[ = (u^{\hat{}}v)^{\hat{}}w \]

(iii) Idempotent:

For any node \( u \) in a tree, we have \( \mu(C(u) \cap C(u)) = \mu(C(u)) = u \).

This leads to a Theorem which states that trees can be generated by the least common ancestors of pairwise leaves. We first give a formal definition the generator of a tree:

**Definition 3.4.2.** Given a tree \( T \), denote its set of all leaves as \( L \). Define \( \langle L \rangle \) to be \( \{l_1^{\hat{}}...^{\hat{}}l_n : l_i \in L, 1 \leq i \leq n, n \in \mathbb{N}\} \).

In other words, \( \langle L \rangle \) denotes the tree generated by all leaves in \( T \). To prove the below mentioned theorem, we need to prove following lemma:

**Lemma 3.4.4.** Given tree \( T \) and its set of all leaves \( L \), for any \( l_1, l_2, l_3 \in L \), there exists \( m, n \in \{1, 2, 3\} \) such that \( l_1^{\hat{}}l_2^{\hat{}}l_3^{\hat{}} = l_m^{\hat{}}l_n^{\hat{}} \).

![Figure 3.1: Illustration for proof of Lemma 3.4.4](image)

**Proof.** Label \( l_1, l_2, l_3 \) with \( u, v, w \) according to their positions as shown in Figure 3.1. By Lemma 3.4.3, we can label any three leaves in a tree in such a way without loss of generality. Then we have \( u^{\hat{}}v^{\hat{}}w = p = v^{\hat{}}w \).

Finally, we have the following theorem:
Theorem 3.4.5. For any given tree $T$ and the set of all leaves $L$, we have $\langle L \rangle = \{l_1 \hat{\cdot} l_2 : l_1, l_2 \in L\}$.

Proof. It suffices to show that for any $l_1, l_2 \ldots l_n \in L$, there exists $i, j \in \{1 \ldots k\}$ such that $l_1 \hat{\cdot} \ldots \hat{\cdot} l_n = l_i \hat{\cdot} l_j$. We will prove this by induction on $n$.

- **Base case:**
  - $n = 1$:
    By lemma 3.3, we know that $l_1 = l_1 \hat{\cdot} l_1$.
  - $n = 2$:
    Choose $l_1, l_2$ to be $l_i, l_j$, we have $l_1 \hat{\cdot} l_2 = l_i \hat{\cdot} l_j$, as desired.

- **Inductive Hypothesis:**
  Suppose for any $l_1, l_2 \ldots l_k \in L$, there exists $i, j \in \{1 \ldots k\}$ such that $l_1 \hat{\cdot} \ldots \hat{\cdot} l_k = l_i \hat{\cdot} l_j$ for some $k > 1$.

- **Inductive Step:**
  We will show that the statement holds for $k + 1$. Suppose we have $l_1, l_2 \ldots l_k, l_{k+1} \in L$. By inductive hypothesis, $l_1 \hat{\cdot} \ldots \hat{\cdot} l_k = l_i \hat{\cdot} l_j$ for some $l_i, l_j \in L$, thus we know that $l_1 \hat{\cdot} \ldots \hat{\cdot} l_k \hat{\cdot} l_{k+1} = l_i \hat{\cdot} l_j \hat{\cdot} l_{k+1}$. We only need to show that there are $m, n \in \{i, j, k + 1\}$ such that $l_i \hat{\cdot} l_j \hat{\cdot} l_{k+1} = l_m \hat{\cdot} l_n$, which is true by lemma 3.4.4. Hence the $k + 1$ case holds.

\[\square\]
Chapter 4

Related Work

In *A Simple Parallel Tree Contraction Algorithm* (1989), Abrahamson, et. al presented the well-known Prune-and-Bypass parallel algorithm for Tree Contraction, as mentioned in Section 3.3. Additionally, they showed, via reduction to list-ranking problem, that this algorithm runs in $O(\log(n))$ time for a tree with $N$ nodes, with $O(n/\log(n))$ EREW processors. Comparatively, the Rake-and-Compress algorithm published by Miller and Reif in 1985 runs also in $O(\log(n))$ time but uses $O(n)$ processors [7].

Furthermore, Abrahamson, et. al showed some applications of Prune-and-Bypass to other problems, e.g. Algebraic Tree Computation, Size of the Largest Clique in a Given Cograph $G$, as well as Identifying a Clique of Maximum Size in $G$ [1].

In his paper: *Algorithms for Boolean Formula Evaluation and for Tree Contraction* (1993), Samuel R. Buss presented a simpler algorithm for Boolean Formula Evaluation Problem that runs in $\text{ALOGTIME}$, equivalently $U_{E^*}$-uniform $NC^1$. In the 1993 work, he mentioned that he improved his work in 1987 which involved converting "an infix Boolean formula to Post-Fix-Longer-Operand-First (PLOF) form" and used fairly complicated methods to choose breakpoints [4].

Buss also showed that the Tree Contraction algorithm by Abrahamson, et. al runs in $U_{E^*}$-uniform $NC^1$. Moreover, he presented a formula for calculating indices of leaves left at the $i^{th}$ tree contraction round, eliminating the need to compute previous trees [4].

**Theorem 4.0.1.** Let $T_i$ be all nodes of the tree in $i^{th}$ contraction:

$$T_i = \{s \cdot t : 2^i \mid s, \ 2^i \mid t, \ s, t \in T_0\}$$

However, Buss did not present an application of Tree Contraction to the Boolean Formula Evaluation Problem, hence this will become our focus in Methodology Section.

In *Some Subclasses of Context-Free Languages in NC^1* (1988), Ibarra, Jiang and Racikumar proposed and proved the Counting Lemma which states that it is possible to count the number of occurrence of a given symbol in a string of length $n$ in $O(\log(n))$ time [6]. This is extremely useful because to achieve a $O(\log(n))$ parsing procedure, we need to ensure that
every step involved in this procedure is within this run-time, and one important step is to count the parentheses.
Chapter 5

Methodology

5.1 Parsing

The application of tree parallel algorithm to boolean evaluation involves two main parts: parsing a boolean string into a parse tree, and perform evaluation on the parse tree. The parsing procedure involves two steps, where the first step is to reduce the negations in the boolean formula, and the second step is to parse a boolean formula with only conjunction and disjunctions into a full binary tree with an ancestor relation on the nodes. Figure 5.1 shows this procedure in a flow chart.

5.1.1 Reduce Boolean Formula with Negation

As mentioned in section 3.3, tree contraction algorithms could only be performed on binary trees. That is to say, for any given boolean formula, we need to parse it into a binary parse
tree. This is not a problem if we restrict the logical operations to conjunction and disjunction only, but when negation is included, we need to transform the formula to an equivalent one without negation. Note that this step is not parsing yet but merely converting a string to another string. To achieve that, we need to count the occurrence of negations before each connective or truth value. In fact, we only need to know whether the number of occurrence is odd or even. We will calculate the desired number by converting the original string to a heap. First, we need to define $Q()$. Given a string $σ$: $⟨S, ≤, T, F, C, D, L, R, N⟩$, for each $m ∈ S$, we define $Q(m)$ as follows:

1. We first define a subset of $S_m$ of $S$ as $\{1, ..., m\}$. Let $S'_m = \{1, ..., m, m+1, ..., 2m-1\}$, in which $m+1, ..., 2m-1$ represent leaves in the heap, and $1, ..., m$ represent ancestors in the heap.

2. Define a partition on $S'_m$ as follows:
   $$Z(m-1+i) ⇔ \neg N(i),
   W(m-1+i) ⇔ N(i),
   X(j) ⇔ 1 ≤ j ≤ m.$$  

3. Assign 0 or 1 to each $i ∈ \{m+1, ..., 2m-1\}$ as follows:
   $$S(i) = \begin{cases} 0 & \text{if } Z(i) \\ 1 & \text{if } W(i) \end{cases}.$$  

4. Construct a heap $H_m$ of size $2m$ on $S'_m$, with parent-child relations defined by $p()$ as follows: For any $k ∈ S'_m$, use $p(k)$ to indicate the parent of $k$, we have
   $$p(k) = \begin{cases} 1 & \text{if } k = 1 \\ \left\lfloor \frac{k}{2} \right\rfloor & \text{otherwise} \end{cases}.$$  

5. Define $Q(m)$ to be the result of performing exclusive or of $S(i)$ for $i ∈ \{m+1, ..., 2m-1\}$:
   $$Q(m) = \oplus \sum_{i=m+1}^{2m-1} S(i+1) ⊕ S(i+2) ⊕ ... ⊕ S(2m-1)$$
   From construction, we can see that for each $m ∈ S$, $Q(m)$ indicated whether or not the number of occurrence of ‘¬’ is odd. More specifically, we have:
   $$Q(m) = (\text{Number of occurrences of ‘¬’ before } m) \mod 2$$

Note that here 0 and 1 should be interpreted as truth values.

Based on the Value of $Q(m)$, we can construct a new string $σ_{new} = ⟨S, ≤, T', F', C', D', L', R', B⟩$ that is equivalent to $σ$ but with no negation involved, with the following rules ($B(m)$ is defined as a blank space):
The second step is partition, and it involves the following definitions:

\( x \) where \( x \) be an integer that indicates a position in the string, and the leaves only include positions \( x \) in the binary tree. The first step in parsing is to number the leaves. It is described as follows: Let \( F \) can parse the string to a parse tree. In this case, the parse tree is guaranteed to be a full binary tree. Once we get a boolean formula that only includes conjunctive and disjunctive connectives, we can parse the string to a parse tree. In this case, the parse tree is guaranteed to be a full binary tree. The first step in parsing is to number the leaves. It is described as follows: Let \( F \) can parse the string to a parse tree. In this case, the parse tree is guaranteed to be a full binary tree.

\[
\begin{array}{c|c}
\text{Original} & \text{New} \\
L(m) & L'(m) \\
R(m) & R'(m) \\
N(m) & B(m) \\
(T(m) \land \neg Q(m)) \lor (F(m) \land Q(m)) & T'(m) \\
(F(m) \land \neg Q(m)) \lor (T(m) \land Q(m)) & F'(m) \\
(C(m) \land \neg Q(m)) \lor (D(m) \land Q(m)) & C'(m) \\
(D(m) \land \neg Q(m)) \lor (C(m) \land Q(m)) & D'(m) \\
\end{array}
\]

Table 5.1: Rules for reducing negation

For example, given following boolean formula \( M \):

\[
\begin{array}{cccccccccccccccccc}
( & \neg & ( & t & \lor & ( & \neg & ( & t & \land & ( & \neg & f & ) & ) & ) & ) & ) \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18
\end{array}
\]

Table 5.2: Boolean Formula \( M \)

This formula could be expressed as structure \( \sigma: \langle S, \leq, T, F, C, D, L, R, N \rangle \) where \( T = \{4, 9\}, F = \{13\}, C = \{10\}, D = \{5\}, L = \{1, 3, 6, 8, 11\}, R = \{14, 15, 16, 17, 18\}, N = \{2, 7, 12\}. After the above procedure, we get a new structure \( \sigma_{\text{new}} = \langle S, \leq, T', F', C', D', L', R', B \rangle \), where \( T' = \{9, 13\}, F' = \{4\}, C' = \{5, 10\}, D' = \emptyset, L' = \{1, 3, 6, 8, 11\}, R' = \{14, 15, 16, 17, 18\}, B = \{2, 7, 12\}, \) and the formula will be as follows.

\[
\begin{array}{cccccccccccccccccc}
L & B & L & F & C & L & B & L & T & C & L & B & T & R & R & R & R & R \\
( & ( & f & \land & ( & ( & t & \land & ( & t & ) & ) & ) & ) & ) & ) \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18
\end{array}
\]

Table 5.3: Boolean Formula \( M \) after negation reduction

### 5.1.2 Parsing a String without Negation

Once we get a boolean formula that only includes conjunctive and disjunctive connectives, we can parse the string to a parse tree. In this case, the parse tree is guaranteed to be a full binary tree. The first step in parsing is to number the leaves. It is described as follows: Let \( x \) be an integer that indicates a position in the string, and the leaves only include positions where \( x \in T \) or \( x \in F \). Define:

- The index of \( x \), \( I(x) = i \) if and only if \( \exists y \leq x[T(y) \lor F(y)] \). In other words, index of \( x \) is determined by the number of truth value variables inclusively before \( x \).

The second step is partition, and it involves the following definitions:

- Let \( L(u, v) \) represents the set of the positions of all left parenthesis within the \( u, v \) interval exclusively. We define \( L(u, v) = \{ z : L(z) \land (u < z < v) \} \).

- Similarly, let \( R(u, v) \) represents the set of the positions of all right parenthesis within the \( u, v \) interval exclusively. We define \( R(u, v) = \{ z : R(z) \land (u < z < v) \} \).
• Left endpoint of $x \lambda(x) = u$ if and only if $(|L(u, x)| = |R(u, x)|) \land L(u) \lor ((T(x) \lor F(x)) \land x = u)$ and for any $v$ such that $(|L(v, x)| = |R(v, x)|) \land L(v) \lor ((T(v) \lor F(v)) \land x = v)$, $u \geq v$. In other words, the left endpoint is the rightmost, and therefore, the innermost such $u$.

• Right endpoint of $x \rho(x) = v$ if and only if $(|L(x, v)| = |R(x, v)|) \land R(v) \lor ((T(x) \lor F(x)) \land x = u)$ and for any $u$ such that $(|L(u, x)| = |R(u, x)|) \land R(u) \lor ((T(u) \lor F(u)) \land x = u)$, $v \leq u$. In other words, the right endpoint is the leftmost, and therefore, the innermost such $v$.

• Interval $I_x = [\lambda(x), \rho(x)]$. The left and right endpoint defines the scope of the current position, in other words, for a given position $x$, all elements within the range $I_x$ form a subtree whose root is $x$.

• $A(x, y)$ if and only if $I_y \subseteq I_x$. $A(x, y)$ is a binary relation on $x$ and $y$, and $A(x, y)$ indicates $x$ is the ancestor of $y$ in the resulting parse tree.

Given any valid boolean formula with only conjuctive and disjuctive connectives, using the above definition to find $A(x, y)$ for any $x, y$ that are truth values or connectives will give us a parse tree, as desired.

Take the above mentioned formula $M$ (Table 5.3) as an example. We first index the leaves in the result parsing tree. To do that, we need to decide $I(x)$ for each $x$ such that $T(x)$ or $F(x)$. By counting the occurrence of truth value variables inclusive before each $x$, we have:

|   | L | B | L | F | C | L | B | L | T | C | L | B | T | R | R | R | R | R | R |
|   | ( | f | \& | ( | t | \& | ( | t | ) | ) | ) | ) | ) | ) | ) | ) | ) | ) | ) |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |

Table 5.4: Boolean Formula $M$ with indexed leaves

Now we have the indices of leaves, we will then illustrate on how the ancestor relations between connectives and truth values in the parse tree is determined. Take $x = 5, y = 10$ as example. To determine the value of $A(5, 10)$, we need to find out $I_5$ and $I_{10}$ in $M$. We will first illustrate on the process of finding $I_5$. To determine $I_5$, we need to know the left and right endpoints of index 5. From definition, we know that the value of left endpoint $\lambda(5)$ is determined by the rightmost $u \in L$ such that $|L(u, 5)| = |R(u, 5)|$. The values of $|L(u, 5)|$ and $|R(u, 5)|$ are listed as follows:

<table>
<thead>
<tr>
<th>$u$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>L(u, 5)</td>
<td>$</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.5: Value of $|L(u, 5)|$

<table>
<thead>
<tr>
<th>$u$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>R(u, 5)</td>
<td>$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.6: Value of $|R(u, 5)|$
Note that $L(u,v)$ and $R(u,v)$ is only defined for $u < v$, thus we only need to list $u$ that is smaller than 5. The candidate sets for $\lambda(5)$ is $\{1, 2, 3, 4\}$, among which only 3 satisfied both $3 \in L$ and $|L(3, 5) = R(3, 5)| = 0$. Hence we have $\lambda(5) = 3$. By similar approach, we can determine the other desired values as follows: $\rho(5) = 17, \lambda(10) = 8, \rho(10) = 15$. Then by definition of interval $I_x$, we have $I_5 = [\lambda(5), \rho(5)] = [3, 17]$ and $I_10 = [\lambda(10), \rho(10)] = [8, 15]$. Since $[8, 15] \subseteq [3, 17]$, i.e. $I_5 \subseteq I_{10}$, we can conclude that 5 is an ancestor of 10, in other words, $A(5, 10)$ is true.

![Figure 5.2: Result Parse Tree of formula $M$](image)

Apply this procedure to determine the value of $A(x, y)$ for all $x, y \in T, F, C, D$, the resulting ancestor relationships will give us the parse tree of formula $M$, as shown in figure 5.2.

### 5.2 Approaches on Evaluating Parsed Boolean String

#### 5.2.1 First Approach - Prune & Bypass with Relabelling

**5.2.1.1 Description**

This algorithm is adapted from the well-known Prune & Bypass algorithm by Abrahamson, et al. [1]

The input consists of: A Parse Tree $T$ with indexed leaves: a full binary tree, represented by an ordered array of vertices each of which has associated with it a parent pointer and a list of 0/2 children, where

- each internal node has value (label) of either $C(\land)$ or $D(\lor)$
- each leaf has value (label) of either $T(true)$ or $F(false)$.

For future reference,

- $parent(u) =$ parent of $u$, accessed through the parent pointer
- $siblings(u) = v, \forall v \in children(parent(u)) \land v \neq u$
- $V_i(u)$, value (label) of node $u$ in $i$-th round of contraction.
5.2.1.2 Pseudo-code and Labeling Tables

Helper Functions:

1. \textit{prune}(u): remove u if u is a leaf

2. \textit{bypass}(w): save pointer to \( p = parent(w) \), remove \( w = parent(u) \) and connect \( sibling(u) \) with \( p \)

3. \textit{detect}(u): detects if double bypass will happen

\[
\text{detect}(u) = \begin{cases} 
U & \text{if } \exists u' = \text{child(parent}(u))), index(u') = \text{odd} \\
L & \text{if } \exists u' = \text{child(parent}(u))), index(u') = \text{odd} \\
N & \text{otherwise} 
\end{cases}
\]

\( U - u \) is the upper leaf \( u1 \) in Figure 5.4

\( L - u \) is the lower leaf \( u2 \) in Figure 5.4

\( N - \) otherwise, \( u \) is just a normal leaf whose removal will not lead to a double bypass

4. \textit{relabel}_{\text{simple}}(v, V_i(u), V_i(w)):

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{simple_bypass.png}
\caption{Simple Bypass}
\end{figure}

**Simple case:** In the simple case: Figure 5.3, we prune an leaf node \( u \), bypass its parent \( w \), and relabel its sibling \( v \) using the function \( \text{relabel}(V_i(u), V_i(w)) \) defined below. \( w \) could have the value: \( t \) (True), \( f \) (False), \( \land \) (Conjunction), \( \lor \) (Disjunction). On the other hand, \( u \) could have the value: \( t \) (True), \( f \) (False), because \( u \) is a leaf node.
\[ V_{i+1}(v) = relabel(V_i(u), V_i(w)) = \]

\[
\begin{array}{cc|ccc}
V_i(u) \setminus V_i(w) & t & f & \lor & V_i(v) \\
   t & t & f & t & V_i(v) \\
   f & t & f & V_i(v) & f
\end{array}
\]

Table 5.7: Relabel Table

5. \( relabel_{special}(v, V_i(u2), V_i(w2), V_i(u1), V_i(w1)) : \)

![Figure 5.4: Double Bypass](image)

**Special Case** As shown in Figure 5.4, We deal with a special case during which double bypass happens, i.e. bypass is performed on both parent and grandparent of a leaf node. We will show that this is the only special case in proof of correctness section by showing that we cannot bypass three connected parents in a row.

\[
V_{i+1}(v) = \begin{cases} 
relabel(V_i(u1), V_i(w1)) & \text{if } relabel(V_i(u1), V_i(w1)) \neq V_i(v) \\
relabel(V_i(u2), V_i(w2)) & \text{otherwise}
\end{cases}
\]

**Pseudo-code**

18
Function \text{Evaluation}(T)

for \( i \) from 0 to \( \lceil \log(n) \rceil - 1 \) do  // repeat for \( \lceil \log(n) \rceil \) times

\hspace{1em} \text{for each leaf } u \text{ in parallel do}

\hspace{2em} \text{if } \text{index}(u) \text{ is odd then}

\hspace{3em} \text{if } \text{detect}(u) == U \text{ then }  // \text{special case: upper leaf, do nothing}

\hspace{4em} \text{else if } \text{detect}(u) == L \text{ then }  // \text{special case: lower leaf}

\hspace{5em} w_2 = \text{parent}(u), w_1 = \text{parent}(w_2)

\hspace{5em} v = \text{sibling}(u), u_1 = \text{sibling}(w_2)

\hspace{5em} u = V_i(u), w_2 = V_i(w_2), u_1 = V_i(u_1), w_1 = V_i(w_1)  // \text{store values}

\hspace{5em} \text{prune}(u), \text{bypass}(w_2)  // \text{handle special case}

\hspace{5em} \text{prune}(u_1), \text{bypass}(w_1)

\hspace{5em} \text{relabel}_{\text{special}}(v, u, w_2, u_1, w_1)

\hspace{2em} \text{else}

\hspace{3em} w = \text{parent}(u), v = \text{sibling}(u)

\hspace{3em} w = V_i(w), u = V_i(u)  // \text{store values}

\hspace{3em} \text{prune}(u)

\hspace{3em} \text{bypass}(w)

\hspace{3em} \text{relabel}_{\text{simple}}(v, u, w)

\hspace{2em} \text{else}

\hspace{3em} \text{index}(u) = \text{index}(u)/2

Figure 5.5: Algorithm for Evaluation

5.2.1.3 Time Analysis

We have one processor per leaf, since we’re only relabeling the leaves. So we use \( O(n) \) number of processors. For each processor, the run-time is bounded by the Prune & Bypass algorithm, which is known to run in \( NC^1 \).

5.2.1.4 Proof of Correctness

Lemma 5.2.1. \text{We can’t have more than 2 consecutive parents that are bypassed}

Proof. Assume for contradiction and without loss of generality that \( a, b, c \) are to be bypassed in one iteration of the algorithm. Let \( f \) be the left most leaf under \( c \), and let \( g \) be the right most leaf under \( c \).

Notice that \( d \) and \( e \) must be on different side of the tree, i.e. they must not both be left child, or both be the right child, otherwise \( d \) and \( e \) would be adjacent to each other and \( b \) doesn’t need to be bypassed. Additionally, we could observe that at least one of \( f \) or \( g \) must be a direct child of \( c \), so that \( c \) needs to be bypassed.

1. \( \text{parent}(f) = c \land \text{parent}(g) \neq c \)

   In this case, one of \( d \) and \( f \) will have an even index, which means one of \( a \) and \( c \) doesn’t need to be bypassed.
2. \( \text{parent}(f) \neq c \land \text{parent}(g) = c \)

Similarly, one of \( e \) and \( g \) will have an even index, which means one of \( b \) and \( c \) doesn’t need to be bypassed.

3. \( \text{parent}(f) = c \land \text{parent}(g) = c \)

In this case, either \( d \) and \( g \) have odd index, or \( f \) and \( e \) have odd index. In the former one, only \( a \) and \( c \) need to be bypassed; in the latter one, only \( b \) and \( c \) need to be bypassed.

Therefore, we cannot bypass more than 2 consecutive parents in one iteration, hence the double bypass case is the only special case we need to handle.

\[ \square \]

5.2.2 Second Approach - Precompute Tree Contraction Sequence

5.2.2.1 Description

Contrary to the the first approach where we contract the tree and relabel during each iteration of the algorithm, in this approach we will pre-compute the entire sequence of tree contraction using Buss’s Theorem: Theorem 4.0.1. Given a tree with \( n \) nodes, we will compute all trees in this tree contraction sequence, which consists of \( \log(n) \) trees. The largest tree is the original tree, and the final tree consists of only 1 node (not necessarily the original root). Next, we will label all trees in sequence, during which each tree is labelled in parallel, i.e.
with $O(n)$ processors working on each node of the tree. The number of processors needed to relabel a tree decreases by a factor of 2 each time we move on to the next tree. Finally, the node left will have the result of the original Boolean formula.

5.2.2.2 Definitions

Let $S = \{T_0, T_1, \ldots, T_i, \ldots, T_{\log(n)}\}$ be the entire tree contraction sequence.

**Figure 5.7: Example of Tree Contraction Sequence**

**Definition 5.2.1.** Given a leaf $u$, $\text{rank}(u) = i$ if and only if $u \in T_i \land u \notin T_{i+1}$.

By Theorem 4.0.1, we can interpret the rank of leaf $u$ algebraically, as follows:

**Definition 5.2.2.** Given a leaf $u$, $\text{rank}(u) = i$ if and only if $u = o \cdot 2^i$ where $o$ is odd.

In other words, the rank of $u$ is the highest power of 2 in its label.

Given the definition of rank, we found a one-to-one correspondence between leaves (except for the one with highest rank among leaves) and internal nodes. In general, the corresponding internal node of any given leaf $u$ is the parent of the leaf $u$ when $u$ is pruned. The correspondence could be represented by the concept of Ultimate Parent, as follows:
Definition 5.2.3. For a given leaf $u$ with rank $i$, Ultimate Parent $UP(u) = w$ if and only if $w = \text{parent}(u)$ in $T_i$.

Since only the leaf with highest rank will remain as a single node in the last contraction tree in the sequence, we could find a Ultimate Parent for every other leaves. We find the formula as follows:

**Theorem 5.2.2.** $UP(u) = \min\{u^*(u \pm 2^{\text{rank}(u)})\}$

Therefore, we could use the idea of Ultimate Parent to redefine the relabel process described in Table 4.

Let $V_i(v)$ be the value of $v$ in tree $T_i$.
Let $P_i(v)$ be the parent of $v$ in tree $T_i$.
Hence, $\exists$ leaf $u$, s.t. $UP(u) = P_i(v)$.

$$V_{i+1}(v) = \begin{cases} 
  t & \text{if } (V_i(P_i(v)) = t) \lor ((V_i(P_i(v)) = \lor) \land (u = t)) \\
  f & \text{if } (V_i(P_i(v)) = f) \lor ((V_i(P_i(v)) = \land) \land (u = f)) \\
  V_i(v) & \text{otherwise}
\end{cases}$$
Note that the definition above can be incorporated into the double bypass easily.

5.2.2.3 Time Analysis

Similar to the first algorithm, we will also need $O(n)$ processors to achieve $O(\log(n))$ parallel time.

5.2.2.4 Proof of Correctness

We need to show the correctness of the proposed one-to-one correspondence between internal nodes and leaves, except for the leaf with highest rank, i.e. to prove the correctness of Theorem 5.2.2.

Proof. Suppose $u$ is a leaf of rank $i$ in tree $T$ of $n$ leaves, and $i$ is not the highest rank in $T$. Suppose that the parent and sibling of $u$ in $T_i$ are $w$ and $v$, respectively. In other words, suppose $UP(u) = w$. Let $p_+ = u + 2^i$ and $p_- = u - 2^i$. By Theorem 4.0.1, we know that $p_+$ and $p_-$ are the closest leaves to $u$ in $T_i$.

- If $u$ is the left child of $w$:

![Figure 5.10: $p_+$ is the leftmost leaf in $v$ when $u$ is the left child of $w$](image)

By definition, we know that $p_+$ is the minimum leaf in $T_i$ that is greater than $u$. If $p_+ \leq n$, we know that $p_+$ is the leftmost leaf in subtree $v$. Hence we have $u^*p_+ = w$. If $p_+ > n$, then $u$ would be the maximum leaf in $T_i$, but since $T_i$ is a full binary tree, $u$ cannot be the left child of $w$.  


• If $u$ is the right child of $w$:

![Diagram of a binary tree with nodes $w$, $u$, $v$, and $p$. $p$ is shaded and is the rightmost leaf in $v$ when $u$ is the right child of $w$.]

Figure 5.11: $p_-$ is the rightmost leaf in $v$ when $u$ is the right child of $w$

By definition, we know that $p_-$ is the maximum leaf in $T_i$ that is less than $u$. If $p_- \geq 1$, we know that $p_-$ is the rightmost leaf in subtree $v$. Hence we have $u^{-} p_- = w$. If $p_- < 1$, then $u$ would be the minimum leaf in $T_i$, but since $T_i$ is a full binary tree, $u$ cannot be the right child of $w$.

As $p_+ \neq u$ and $p_- \neq u$, we know that $u^{-} p_\pm$ must be higher than $u$. Therefore, we have $w = \min\{u^{-} p_\pm\}$. 

\[ \square \]
Chapter 6

Results

In this thesis, we built applications of parallel tree contraction algorithm to efficiently evaluate Boolean Formulas.

From the idea of tree contraction and counting lemma proposed by other computer scientists, we found an efficient way of parsing Boolean Formulas with logic, and we developed several methods of evaluating such full binary tree representation of a Boolean Formula to obtain its final result. Furthermore, we could achieve run-time in $NC^1$ using $O(n)$ number of processors. Hence, the parallel run-time of our algorithm is the same as the parallel tree contraction algorithm.
Chapter 7

Future Work

Although our parallel run-time is in $NC^1$, one area of improvement could be the number of processors needed. Currently we need $O(n)$ processors, but we could work on achieving the same run-time with fewer processors. We are also looking to expand our application of tree contraction beyond Boolean Formula evaluation problems.
Bibliography


