1 Introduction

In this technical report, we explain our project for CS 511. Our project is to create a Client-side Transitive Closure Computation web application using Javascript, which can validate propositional logic formulas.

In this project, we realize a data structure to represent directed hypergraphs, and implement transitive closure algorithm on this structure. This application uses the hypergraph data structure to represent propositional logic formulas, and encodes all deductive rules for propositional calculus into the closure algorithm. In short, it takes an propositional formula as input, converts into a hypergraph instance, then after finishing the closure operations on the hypergraph, it will obtain a new hypergraph, and by checking this new hypergraph, it can validate the input formula.

The implementation of this project includes three parts: parsing, hypergraph data structure, propositional logic based closure algorithm. The focus of this project is on the last two parts. The rest of this report is organized as follows. In section 2, we will give the description of the whole project. In section 3, we will explain how to use the web application. In the last part, we will give the resources and references used in the project.

2 Description

2.1 Overview

The implement of this project includes: parsing, hypergraph data structure and propositional logic based closure algorithm. In this section, we will first explain the hypergraph data structure, then how to parse the input propositional formula to generate corresponding hypergraph. In the last part, we will explain the propositional logic based closure algorithm.

The overview of our implementation is shown in Figure 1. We first parse the input propositional formula and generate the corresponding hypergraph, then perform propositional logic based closure algorithm on the hypergraph. By checking the final hypergraph, we can validate the input formula.
2.2 Hypergraph Data Structure

2.2.1 Data structure component

The hypergraph data structure consists of node, edge and methods that operate on nodes and edges.

We use an array of node to store all nodes in the hypergraph. Node only contains one attribute - label, which is unique to identify the node.

We use an array of edge to store all the edges in the hypergraph. Edge contains three attributes: label, nodes and arity. label is used to define a group of edge, and each edge has a unique label. nodes includes all the nodes that belong to this edge, and if two edges have the same label and nodes, but the order of the nodes are different, then they are two different edges. arity is the number of nodes in this edge. Each edge is identified by a unique edge key, which will be explained later.

Methods in this hypergraph are as follows:

1. checkNode(node): Check whether a certain node exists or not. The complexity of this method is $O(1)$.
2. checkEdge(edge): Check whether a certain edge exists or not. The complexity of this method is $O(1)$.
3. insertNode(node): When inserting a node, we will first check whether this node exists, if not we add the new node to this hypergraph.
4. InsertEdge(edge): When inserting a node, we will first check whether this edge exists and whether it is a valid edge(arity is consistent). If so we add the new edge. The complexity of this method is $O(1)$.
5. getArity(label): It will return the arity of a edge with a certain label. The complexity of this method is $O(n)$.
6. getEdge(label): It will return all the edges with the same label. The complexity of this method is $O(n)$.

The reason why the complexity of first four methods is $O(1)$, is as follows. When we insert a
node or edge into the hypergraph, we will first make a key (string type) for that node or edge, and put it in a separate array. Such array is kind of index for our nodes and edges. So when we want to insert a new node or edge, we first use the search method provided in Javascript to check whether the key exist, since the complexity of this internal method is O(1), then the complexity for the checkNode and checkEdge is O(1). So if this key does not exist, we will insert the new node or edge, thus the complexity is O(1). As for the last two methods getArity and getEdge, because we go through all edges, thus its complexity is O(n).

2.2.2 Hypergraph for propositional logic

We use the hypergraph data structure to represent propositional formulas. Each labeled node represents an atom or a subformula. In addition, there are six kinds of edge for propositional logic, and the meaning of each edge is as follows:

1. isTrue(A) : A = True.
2. isAndOf(A, B, C) : A = B \land C
3. isOrOf(A, B, C) : A = B \lor C
4. isImply(A, B, C) : A = B \rightarrow C
5. isNeg(A, B) : A = \neg B
6. contradiction() : There is contradiction in the formula.

2.3 Input parsing

The input of the application is a propositional logic formula, and we first parse the input and create the corresponding hypergraph. Stack structure is used to parse the input. The parsing algorithm is shown in algorithm 1, and it contains another algorithm to process the data as shown in algorithm 2.

<table>
<thead>
<tr>
<th>Algorithm 1: Parsing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input</strong>: input</td>
</tr>
<tr>
<td><strong>output</strong>: hypergraph</td>
</tr>
<tr>
<td>1 Construct a stack S;</td>
</tr>
<tr>
<td>2 for i = 0; i &lt; input.length; i += do</td>
</tr>
<tr>
<td>3 push input[i] into S;</td>
</tr>
<tr>
<td>4 if input[i] = ')' then</td>
</tr>
<tr>
<td>5 repeat pop S and append to an array N until ')' is popped;</td>
</tr>
<tr>
<td>6 UpdateHypergraph(N, hypergraph)(in Algorithm 2), push returned value into S;</td>
</tr>
</tbody>
</table>
Algorithm 2: UpdateHypergraph

<table>
<thead>
<tr>
<th>input : input, hypergraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>output : C</td>
</tr>
<tr>
<td>1 create node C;</td>
</tr>
<tr>
<td>2 if input is in the form of A opt B then</td>
</tr>
<tr>
<td>3   if A does not exist then add node A to hypergraph;</td>
</tr>
<tr>
<td>4   if B does not exist then add node B to hypergraph;</td>
</tr>
<tr>
<td>5   if opt = ( \land ) then create edge e: isAndOf(C, A, B);</td>
</tr>
<tr>
<td>6   else if opt = ( \lor ) then create edge e: isOrOf(C, A, B);</td>
</tr>
<tr>
<td>7   else if opt = ( \rightarrow ) then create edge e: isImply(C, A, B);</td>
</tr>
<tr>
<td>8 else if input is in the form of ( \neg B ) then</td>
</tr>
<tr>
<td>9   if B does not exist then add node B to hypergraph;</td>
</tr>
<tr>
<td>10  create edge e: isNeg(C, B);</td>
</tr>
<tr>
<td>11 if e does not exist then add edge e and node C to hypergraph;</td>
</tr>
<tr>
<td>12 else C ← first node of e ;</td>
</tr>
<tr>
<td>13 return C;</td>
</tr>
</tbody>
</table>

2.4 Propositional logic based closure algorithm

2.4.1 Transitive closure

The closure algorithm is shown in algorithm 3. It will keep adding new edge to hypergraph until no new edge is added.

Algorithm 3: Transitive closure

<table>
<thead>
<tr>
<th>input : hypergraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeat</td>
</tr>
<tr>
<td>2 hypergraph = transit(hypergraph);</td>
</tr>
<tr>
<td>3 until no new edge is added;</td>
</tr>
</tbody>
</table>

Next we will explain the transit algorithm. We use substitution to compute the transit operation. For each rule in transit algorithm, we will first find all possible substitutions that are consistent for every variable in the method. If the substitution satisfies the rule, we will add new edge to the hypergraph.

Generally for a rule in the following form:

\[
\text{for all } x_1, x_2, x_3, x_4...x_n, \ label_1(x_1, x_2, x_3, x_4...x_n), label_2(x_1, x_2, x_3, x_4...x_n), \ldots \text{ and } label_k(x_1, x_2, x_3, x_4...x_n) \text{ implies } label_x(x_1, x_2, x_3, x_4...x_n),
\]

our corresponding algorithm of this rule is shown in algorithm 4.
Algorithm 4: Deductive rule

input: hypergraph
1 $S_1 \leftarrow \text{getEdge}(\text{label}_1)$;
2 $S_2 \leftarrow \text{getEdge}(\text{label}_2)$;
3 ...;
4 $S_n \leftarrow \text{getEdge}(\text{label}_n)$;
5 forall the element $s_1 \in S_1$ do
6       forall the element $s_2 \in S_2$ do
7           ...;
8       forall the element $s_n \in S_n$ do
9           if label$_x(s_1, s_2 \ldots s_n)$ is a valid edge then insertEdge label$_x(s_1, s_2 \ldots s_n)$

2.4.2 Deductive rules in propositional logic

Deductive rules used in this project are represented as follows:

(1) "∧" introduction: $\phi, \psi \vdash (\phi \land \psi)$
for all x, y. isTrue(x), isTrue(y) and isAndOf(z, x, y) implies isTrue(z).

(2) "∧" elimination: $\phi \land \psi \vdash \phi, \psi$
for all x, y. isTrue(z) and isAndOf(z, x, y) implies isTrue(x) and isTrue(y).

(3) "∨" introduction: $\phi \vdash \phi \lor \psi$ and $\psi \vdash \phi \lor \psi$
for all x. isTrue(x), and isOrOf(z, x, y) implies isTrue(z).
for all y. isTrue(y), and isOrOf(z, x, y) implies isTrue(z).

(4) "∨" elimination: $\phi \lor \psi, \phi \rightarrow \chi, \psi \rightarrow \chi \vdash \chi$
for all x, y, z. p, q, w. isTrue(z), isOrOf(z, x, y), isImply(p, x, w) and isImply(q, y, w) implies isTrue(w).

(5) "→" elimination: $\phi, \phi \rightarrow \psi \vdash \psi$
for all x, y, z. isTrue(x), isTrue(z) and isImply(z, x, y) implies isTrue(y).

(6) "→" introduction: $\phi, \psi \vdash \phi \rightarrow \psi$
for all x, y, z. isTrue(x), isTrue(y) and isImply(z, x, y) implies isTrue(x).

(7) "⊥" introduction: $\phi, \neg\phi \vdash \bot$
for all x, y. isTrue(x), isTrue(y) and isNeg(y, x) implies contradiction().

(8) "¬¬" introduction: $\phi \vdash \neg\neg\phi$
for all x, y, z. isTrue(x), isNeg(y, x) and isNeg(z, y) implies isTrue(z).

(9) "¬¬" elimination: $\neg\neg\phi \vdash \phi$
for all x, y, z. isTrue(x), isNeg(y, x) and isNeg(z, y) implies isTrue(x).

We encode all the deductive rules above into transit algorithm, and we also provide their complexity as follows. Each time when transit algorithm is called, all these nine rules will be executed
3 Usage

In this section, we provide an example to demonstrate how our project works. We will go through the project by using this example and explain all restrictions in our project.

3.1 Input

3.1.1 From sequent to formula

The input of our system is a propositional formula, but our project aims to validate a sequent in the form of $\varphi_1, \varphi_2, \ldots, \varphi_n \vdash \psi$, so first we need to convert the sequent to a propositional formula.

For example, a sequent like $\varphi_1, \varphi_2, \ldots, \varphi_n \vdash \psi$ will be convert to $\varphi_1 \land \varphi_2 \land \ldots \land \varphi_n \rightarrow \psi$.

Then we assume the left part of the formula is $True$, if we could prove the right part is also $True$, then according to complete and soundness of propositional logic, we can say the sequent is valid.

In this section we want to check the validity of the following sequent:

$$A \lor B, B \rightarrow C, A \rightarrow C, C \rightarrow \neg\neg D \vdash (C \lor A) \land D.$$

First we convert it to formula as follows:

$$((A \lor B) \land (B \rightarrow C) \land (A \rightarrow C) \land (C \rightarrow \neg\neg D) \rightarrow ((C \lor A) \land D))$$

Then we assume the left part is $True$, and try to figure out whether the right part is $True$. If so, the sequent is valid.

3.1.2 Format of input formulas

Although we have conventions about the binding priorities of symbols such as $\land, \lor, \rightarrow, \neg$. In our system, we require users to insert parenthesis so that it is easier for parsing process.

The format of all binary relations, such as $\land, \lor, \rightarrow$, should be like $(A \ast B)$. For example, $(A \land B), (A \lor B)$ and $(A \rightarrow B)$. The format of all unary relations, such as $\neg$, should be like $(\ast A)$. For example, $(\neg B)$.

Using example above, if we have a formula:

$$A \lor B \land B \rightarrow C \land A \rightarrow C \land C \rightarrow \neg\neg D \rightarrow (C \lor A) \land D$$

The correct format of this formula as an input should be:

$$(((A \lor B) \land (B \rightarrow C)) \land ((A \rightarrow C) \land (C \rightarrow \neg\neg D))) \rightarrow ((C \lor A) \land D))$$

Another constriction is that we require the propositional atoms should be uppercase characters. When we create new nodes, we use non-negative integer numbers as labels.
3.2 Construct hypergraph

3.2.1 From formulas to trees

Before we construct the hypergraph, we need to parse the formula to verify whether the format of the formula is correct and to analyze what kind of nodes and edges should be inserted into the hypergraph. A parse tree of the propositional formula could be better to illustrate the results of parsing.

Following is the parse tree of formula:

\[((((A \lor B) \land (B \rightarrow C)) \land ((A \rightarrow C) \land (C \rightarrow (\neg D)))))) \rightarrow ((C \lor A) \land D)).\]

3.2.2 Using stack to create hypergraph

After parsing the formula, we need to insert nodes and edges into the hypergraph.

For each binary relation, such as \(\land, \lor, \rightarrow\), we will insert three nodes and one edge. For example,
if we have a relation \((A \land B)\), we will create a new node with label 0 for \((A \land B)\), two nodes labeled with \(A, B\), and the new edge will be \(isAndOf(0, A, B)\).

For each unary relation, such as \(\neg\), we will insert two nodes and one edge. For example, if we have a relation \((\neg C)\), we will create a new node with label 1 for \((\neg C)\), another node labeled with \(C\), and the new edge is \(isNeg(1, C)\).

Before insertion, we check whether the nodes or edges already exist in our hypergraph. If some already exists, insertion for that node or edge will be ignored. In our hypergraph, each node is an propositional atom or subformula.

In our system, we use stack to parse input formula and generate hypergraph. In parsing process we push one character at one time into the stack and if the character is ”)”, we pop characters already in the stack until we get the first ”(”. Then push a new character into the stack after processing the popped characters. We create new nodes and edges by processing the characters just popped out of the stack.

We use above formula \((((((A \lor B) \land (B \rightarrow C)) \land ((A \rightarrow C) \land (C \rightarrow (\neg(\neg D)))))) \rightarrow ((C \lor A) \land D))\) to illustrate the process of parsing and creating nodes and edges. We will explain the first several steps in stack.

First step:

```
( ( ( ( A \lor B ) ) )
```

Pop \((A \lor B)\) and push 0 into the stack, then create three nodes \(A, B\) and 0, and one new edge \(isOrOf(0, A, B)\).

Second step:

```
( ( ( ( ( B \rightarrow C ) ) ) )
```

Pop \((B \rightarrow C)\) and push 1 into the stack, then create two new nodes \(C\) and 1, and one new edge \(isImply(1, B, C)\).

Third step:

```
( ( ( ( ( 0 \land 1 ) ) ) )
```

Pop \((0 \land 1)\) and push 2 into the stack, then create one new node 2, and one new edge \(isAndOf(2, 0, 1)\).

Fourth step:

```
( ( ( ( ( 2 \land 3 ) ) ) )
```

Pop \((A \rightarrow C)\) and push 3 into the stack, then create one new node 3, and one new edge \(isImply(3, A, C)\).

Repeat popping and pushing, finally, we get the whole hypergraph after finishing parsing process.

The edges in our hypergraph are shown as follows:

<table>
<thead>
<tr>
<th>isOrOf(0, A, B)</th>
<th>isImply(1, B, C)</th>
<th>isAndOf(2, 0, 1)</th>
<th>isImply(3, A, C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>isNeg(4, D)</td>
<td>isNeg(5, 4)</td>
<td>isImply(6, C, 5)</td>
<td>isAndOf(7, 3, 6)</td>
</tr>
<tr>
<td>isAndOf(8, 2, 7)</td>
<td>isOrOf(9, C, A)</td>
<td>isAndOf(10, 9, D)</td>
<td>isImply(11, 8, 10)</td>
</tr>
</tbody>
</table>

The nodes in our hypergraph are labelled shown as follows:
3.3 Hypergraph transitive closure

3.3.1 Validation using proof box and deductive rules

To validate propositional sequent, we can use proof box and deductive rules. The working process of our system is similar to that but automatically.

For completeness, we provide an example of validation using proof box and deductive rules. To validate the sequent: $A \lor B$, $B \rightarrow C$, $A \rightarrow C$, $C \rightarrow \neg\neg D \vdash (C \lor A) \land D$, we can prove as follows:
3.3.2 Validation using transitive closure

Next we will explain how our system validate propositional formula.

First we assume the left part of the formula is True, which means we first add isTrue(8) to the hypergraph. Then out system keeps executing the transit algorithm which includes all deductive rules, until no new isTrue edge is generated, then the validation is finished. In each loop, we check 9 deductive rules: Intro(∧), Elim(∧), Intro(∨), Elim(∨), Intro(→), Elim(→), Intro(¬¬), Elim(¬¬) and Intro(⊥) which are illustrated in Section 2.4.2.

We demonstrate how new isTrue edges are added in each loop as follows:

<table>
<thead>
<tr>
<th>Loop</th>
<th>According to</th>
<th>Inserted new edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>Elim(∧) and isTrue(8)</td>
<td>isTrue(2) and isTrue(7)</td>
</tr>
<tr>
<td>2nd</td>
<td>Elim(∧) and isTrue(2)</td>
<td>isTrue(0) and isTrue(1)</td>
</tr>
<tr>
<td>3rd</td>
<td>Elim(∧) and isTrue(7)</td>
<td>isTrue(3) and isTrue(6)</td>
</tr>
<tr>
<td>4th</td>
<td>Elim(∨), isTrue(0), isTrue(1) and isTrue(3)</td>
<td>isTrue(C)</td>
</tr>
<tr>
<td>5th</td>
<td>Intro(∨) and isTrue(C)</td>
<td>isTrue(9)</td>
</tr>
<tr>
<td>6th</td>
<td>Elim(→) and isTrue(6) and isTrue(C)</td>
<td>isTrue(5)</td>
</tr>
<tr>
<td>7th</td>
<td>Elim(¬¬) and isTrue(5)</td>
<td>isTrue(D)</td>
</tr>
<tr>
<td>8th</td>
<td>Intro(→) and isTrue(8)</td>
<td>isTrue(11)</td>
</tr>
</tbody>
</table>

We list all isTrue nodes in the hypergraph as follows:
8, 2, 7, 0, 1, 3, 6, C, 9, 5, D, 10, 11

Since the root node 11 is True, so we know the input propositional formula is valid.
4 Resources and References

References:

[1] Logic in computer science: modelling and reasoning about systems By Michael Huth, Mark Ryan