Implementing Order Relations on Graphs

Muhsin Atto
Abstract

Implementing Order Relations on Graphs

Muhsin Atto

The design of automatic verification methods for programs manipulating dynamic data structures is a challenging problem. We consider programs that operate on data structures with one next-pointer, such as singly linked lists and circular lists. We refer to such data structures as heaps. We represent a heap as a graph, where vertices in the graph represent cells in the heap. To test that a program is safe, we should check that the output of this program is well formed and is sorted. The aim of this project is to design a number of algorithms to test the different orderings between graphs which arise when including heaps. More precisely, given two graphs, G1 and G2, we check that G1 can be included in G2, using different type of orderings.
Acknowledgments

First of all I would like to thank Parosh Aziz Abdulla for very useful comments and feedback in my work. I am deeply appreciating his willingness to assist me in writing my report, by providing me valuable suggestions and very helpful comments. I would like to thank my supervisor, Jonathan Cederberg, for his involvement, and guided me through this work especially at the implementation part. I would like to thank my friend and room mate Ran Ji, we have a good time during our work. I wish for him all the best in the future. I would also like to thank Fridrik Aziza for enormous help which I have from him, for encouraging me to think. It was a great pleasure to work with all of you.
## Contents

1 Introduction ............................................................... 1

2 Definitions .................................................................. 2
   2.1 Trees ...................................................................... 2
   2.2 Stars ...................................................................... 5

3 Unordered Tree Inclusion .............................................. 6
   3.1 Tree To Tree Inclusion Algorithm ............................... 7
   3.2 Tree To Tree Inclusion From Roots Algorithm ............. 8

4 Unordered Star inclusion .................................................. 10
   4.1 Star to Star Inclusion Algorithm .............................. 11
   4.2 Tree to Star Inclusion Algorithm .............................. 14

5 Unordered Tree Matching Algorithm ............................... 16
   5.1 Tree to Tree Matching Algorithm .............................. 17

6 Star Matching algorithms ................................................ 19
   6.1 Star to Star Matching Algorithm .............................. 20
   6.2 Tree To Star Matching Algorithm .............................. 21

7 Unordered Heaps Inclusion Algorithm ............................ 24
   7.1 Heap ordering ......................................................... 26
   7.2 Heap Inclusion Algorithm ......................................... 27

8 Experimental Results ...................................................... 29

9 Conclusions and Future Work ........................................ 32
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A typical graph representing the tree.</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>An example of Tree ordering</td>
<td>4</td>
</tr>
<tr>
<td>2.3</td>
<td>An example of Tree ordering from roots</td>
<td>4</td>
</tr>
<tr>
<td>2.4</td>
<td>A typical graph representing the Star.</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>Matching in a Bipartite Graph.</td>
<td>6</td>
</tr>
<tr>
<td>3.2</td>
<td>Example of tree Inclusion, Five pattern trees each of them is included in the target tree.</td>
<td>8</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of Star Inclusion, each of $P_1, P_2, P_3$ is included in $T$.</td>
<td>13</td>
</tr>
<tr>
<td>4.2</td>
<td>Example of Tree to Star Inclusion, each of $P_1, P_2, P_3$ is included in $T$</td>
<td>15</td>
</tr>
<tr>
<td>5.1</td>
<td>Trees Matching</td>
<td>18</td>
</tr>
<tr>
<td>6.1</td>
<td>Example of star to star matching</td>
<td>21</td>
</tr>
<tr>
<td>6.2</td>
<td>Example of tree to Star Matchings</td>
<td>23</td>
</tr>
<tr>
<td>7.1</td>
<td>A typical graph representing the heap.</td>
<td>24</td>
</tr>
<tr>
<td>7.2</td>
<td>Example of Heap Inclusion, each of $P_1, P_2, P_3$ can be included in $T$.</td>
<td>28</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The design of automatic verification methods for programs manipulating dynamic data structures is a challenging problem. The issue of verifying automatically such programs has received a lot of attention.

There are many approaches for addressing these problems for different kinds of programs and using different types of formalisms. Each of these approaches uses some special techniques and strategies for investigating special types of verification properties.

We consider programs that operate on a data structures with one next-pointer, such as singly linked lists and circular lists. In this work, we refer to such data structures as heaps.

We represent a heap as a graph, where vertices in the graph represent cells in the heap. The successor of a vertex in a graph represents the next cell that is pointed by current cell in the heap.

In general, the aim of this project is to design and implement algorithms to test inclusion between graphs that represent heaps. More precisely, we design and implement a number of algorithms to decide for two graphs say $G_1$ and $G_2$, whether $G_1 \subseteq G_2$ according to a number of different orderings.

Outline In Chapter 2 we give the general definition and the terminology used in the report. In Chapter 3, we describe tree to tree inclusion algorithms. In Chapter 4, we describe star to star inclusion algorithms. In Chapter 5, we describe tree to tree matching algorithms. In Chapter 6, we describes star to star matching algorithms. Chapter 7, we describe heap to heap inclusion algorithms. In Chapter 8, we give experimental results for all the algorithms. In Chapter 9, we give some conclusions and direction for future work.
Chapter 2

Definitions

2.1 Trees

A tree is a graph, where each node has a set of children and at most one parent. The root node is the top-most node in the tree, which has no parent. Each node in the tree can be reached from the root. In fact, there is unique path between the root and any of the other nodes in the tree.

Notations

We call the two trees that we want to test for inclusion by pattern tree and target tree, denoted by $P$ and $T$, respectively.

We consider for a given two trees $P$ and $T$ and a vertex $v \in T$, the set of nodes and the set of edges are denoted by $V(T)$ and $E(T)$ respectively.

The number of children of a node $v$ is denoted by $\text{outdeg}(v)$. The subtree of $T$ rooted at node $v$ is denoted by $T[v]$. The label of a node $v$ is denoted by $\text{label}(v)$. Further, $\text{root}(T)$ denotes the root of tree $T$.

The degree of a node is the number of children of that node. Nodes at the bottom-most level of the tree are called leaves. Since they are bottom-most, they do not have any children.

Types of Trees

In general, there are two type of trees, namely ordered tree and unordered tree. In this work, we concentrate on unordered trees.

Tree Representations

We represent a tree as a graph, where the vertices in the graph represent the nodes in the tree. The successors of a vertex in a graph represents the parent of a node that
represents that vertex. The set of children of a node in the tree are called predecessors of that node in the graph.

From the graph theoretical point of view, a tree is a connected acyclic graph, where each vertex has at most one successor. A rooted tree is such a graph with a vertex singled out as the root. In this case, any two vertices connected by an edge inherit a parent-child relationship. In a summary, a graph G is said to be a tree if the following properties are satisfied.

- Each node in G has at most one successor.
- G is connected and acyclic.

**Tree Ordering**  In general, consider two trees P and T and \(v \in T\), We check the ordering between P and T, performing sequences of the following operations

- **Leaf deletion**: Delete \(v\) such that \(v\) is a node with degree zero.

- **Contraction**: Delete \(v\) such that the following are satisfied:
  - \(v\) has no labels
  - \(v\) has one incoming edge with connects the parent of \(v\) (if any) to its children.

In this work, we consider two types of ordering between two trees

- We use \(P \sqsubseteq T\) to denote that \(P\) can be obtained from \(T\) through performing a sequence of the following operations:
  - Leaf deletion
  - Contraction

- We use \(P \sqsubseteq^r T\) to denote that \(P\) can be obtained from \(T\), from roots, through performing a sequence of the following operations:
Figure 2.2: An example of Tree ordering

Figure 2.3: An example of Tree ordering from roots

- Leaf deletion

- Contarction with additional constraint that $v$ has one incoming, and one outgoing edge.

**Example**$_1$ $P_1$, $P_2$ and $P_3$ can be obtained from the target tree shown in figure 2.2, performing the leaf deletion, but $P_4$ and $P_5$ can not be obtained from the target tree.

**Example**$_2$ $P_2$ can be obtained from $T$ from roots shown in figure 2.3, performing the leaf deletion, but $P_1$ can not be obtained from the target tree.
2.2 Stars

A *star* is a tree with the additional feature that there is an extra edge going from an arbitrary node in the tree back to the root of the tree. Another definition of a star is a set of trees having their roots connected through a simple cycle.

A graph $G$ is said to be a star if the following properties are satisfied:

- Each node in $G$ has at most one successor.
- $G$ is connected and possibly cyclic.

Figure 2.4: A typical graph representing the Star.
Chapter 3

Unordered Tree Inclusion

In this section, we describe two algorithms to solve tree inclusion. The first algorithm is described in [18], and concerns tree to tree inclusion. The second algorithm is an extension to the first algorithm. First, we define some notations that we will use in the algorithms, then we describe both algorithms.

**Bipartite Graph**  Given a graph $G = (V, E)$, $G$ is a bipartite graph, if $V$ can be divided into two disjoint sets $S_1$ and $S_2$, such that no two vertices within the same set are adjacent. In both algorithms, we use the routine of a bipartite graph to check trees inclusion.

**Matching in a Bipartite graph**  Given a bipartite graph $G = (V, E)$, a *matching* in $G$ is a subset of edges $M \subseteq E$, such that no two edges in $M$ are incident to the same vertex.

The following are two possible matching from the graph in the figure 3.2.\[\{(1, 2), (3, 4), (5, 6)\};\{(1, 6), (3, 2), (5, 4)\}\].

We use *Ford-Fulkerson Method* [2] to find matching in $G$.

![Figure 3.1: Matching in a Bipartite Graph.](image)
3.1 Tree To Tree Inclusion Algorithm

Given two trees $P$ and $T$, the tree inclusion problem described in this section consists of locating the smallest subtrees of $T$ that include $P$. This algorithm is based on subtree homeomorphism algorithm in [6], which involves solving a series of maximum bipartite matching problems.

Given $v \in P$ and $w \in T$, where $w$ has children $w_1, w_2, \ldots, w_t$, and $v$ has children $v_1, v_2, \ldots, v_p$. In order to determine whether $P[v]$ can be included in $T[w]$. It suffices to know if $P[x] \subseteq T[y]$, for all $x \in \{v_1, v_2, \ldots, v_p\}$ and $y \in \{w_1, w_2, \ldots, w_t\}$. More precisely, given two vertices $v \in P$ and $w \in T$, to decide whether $P[v] \subseteq T[w]$, we construct a bipartite graph $G = (\{w_1, w_2, w_3, \ldots, w_t\} \cup \{v_1, v_2, v_3, \ldots, v_p\}, E)$ with $(w_j,v_i) \in E$.

We say $P[v]$ can be included from $T[w]$ iff $G$ has at least one matching.

We write $S(w)$ to denote the included sub tree at node $w \in T$. We compute $S(w)$ as follows:

$$S(w) = \{v \in V(P)|P[v] \subseteq T[w]\}.$$  We say that $P$ can be included in $T$ if $\text{root}(P) \in S(w)$.

**Problem Definition** Let $P=(V,E)$ be a pattern tree, and $T=(V,E)$ be a target tree, the problem defined for this algorithm is to decide whether $P$ can be included in $T$. 
Figure 3.2: Example of tree Inclusion, Five pattern trees each of them is included in the target tree.

Algorithm 1: Tree To Tree Inclusion

Input: P and T

Output: $P \subseteq T$?

for all $w \in V(T)$ in the postorder do

$S(w) = \{v \in V(P) | \text{outdeg}(v) = 0 \text{ and } \text{label}(v) \subseteq \text{label}(w)\}$

if $\text{outdeg}(w) \neq 0$ then

Let $w_1, w_2, w_3, \ldots, w_t$ be children of $w$

$S(w) := S(w_1) \cup S(w_2) \cup S(w_3) \cup \ldots \cup S(w_t)$

for all non leaf $v \in V(T)$ with $\text{outdeg}(v) \leq \text{outdeg}(w)$ in the postorder do

if $v \notin S(w)$ then

Let $v_1, v_2, v_3, \ldots, v_p$ be children of $v$, we construct a bipartite graph

$G = (\{w_1, w_2, w_3, \ldots, w_t\} \cup \{v_1, v_2, v_3, \ldots, v_p\}, E)$ with

$(w_j, v_i) \in E$, if and only if $v_i \in S(w_j)$

$\forall 1 \leq i \leq p \text{ and } 1 \leq j \leq t$

if there is a matching in $G$ with $p$ edges then

$S(w) := S(w) \cup \{v\}$

if $\text{root}(P) \in S(w)$ then

return TRUE

end

end

end

return FALSE

end

3.2 Tree To Tree Inclusion From Roots Algorithm

In this section, we describe an algorithm to test the tree inclusion from roots. Consider two trees $P$ and $T$, this algorithm decides whether $P \subseteq_R T$ is true or not. In other word, let $v$ be a root node for the tree $P$, and $w$ be the root node for the tree $T$, this algorithm returns true if and only if $P[v]$ can be included in $T[w]$.

This algorithm differs from the first algorithm in such way that it keeps testing
the inclusion until it reaches roots of both trees, and then returns true in case these two trees are included. This algorithm follows the same steps as in the first algorithm, except that this algorithm returns true if and only if P can be included in T, from roots.

**Problem Definition** Given two trees $T_1$, $T_2$, decide whether $T_1$ can be included in $T_2$, from roots.

**Algorithm 2:** Tree To Tree Inclusion from roots  
**Input:** P and T  
**Output:** $P \subseteq_R T$?

```plaintext
for all $w \in V(T)$ in the postorder do 
  $S(w) = \{v \in V(P) | \text{outdeg}(v) = 0 \text{ and } \text{label}(v) \subseteq \text{label}(w)\}$
if $\text{outdeg}(w) \neq 0$ then 
  Let $w_1, w_2, w_3, \ldots, w_t$ be children of $w$
  $S(w) := S(w_1) \cup S(w_2) \cup S(w_3) \cup \ldots \cup S(w_t)$
for all non leaf $v \in V(T)$ with $\text{outdeg}(v) \leq \text{outdeg}(w)$ in the postorder do 
if $v \notin S(w)$ then 
  Let $v_1, v_2, v_3, \ldots, v_p$ be children of $v$, we construct a bi-partite graph 
  $G = (\{w_1, w_2, w_3, \ldots, w_t\} \cup \{v_1, v_2, v_3, \ldots, v_p\}, E)$ with
  $(w_j, v_i) \in E$, if and only if $v_i \in S(w_j)$.
  $\forall 1 \leq i \leq p$ and $1 \leq j \leq t$
if there is a matching in $G$ with $p$ edges then 
  $S(w) := S(w) \cup \{v\}$
if $v = \text{root}(P)$ and $w = \text{root}(T)$ then 
  return TRUE
return FALSE
end
```

Example of Tree Inclusion from roots, each of $P_1, P_3, P_4$ can not be included from roots in $T$, but $P_2, P_5$, can be included in $T$ from roots.
Chapter 4

Unordered Star inclusion

In this section, we describe two algorithms to deal with inclusion problems for stars. The first algorithm is to test whether a star is included in another star, and the second one is to test whether a tree is included in a star.

We first define all notations and terminologies, and then we describe the two algorithms.

Notations and terminologies The following notations and terminology will be used in this section.

We call the two stars that we want to test for inclusion by pattern star and target star, denoted by $P$ and $T$, respectively. Given a star $T$ and vertex $v \in T$, the set of nodes and the set of edges for $T$ are denoted by $V(T)$ and $E(T)$, respectively. The set of nodes in the cycle for $T$ is denoted by $C(T)$. We call the operation to get a successor of a node $v \in T$ by $T.succ(v)$. The subtree rooted at node $v$ is denoted by $T[v]$.

Preprocessing To test a star inclusion, we need to have the following computations and operations before we define algorithms.

- Given two vertices $v$ and $w \in T$, we write $T_2 = T.delete(v, w)$ to denote the operation that deletes an edge going from the vertex $v$ to the vertex $w$ in $T$. The tree $T_2$ is the result of deletion.

- We write $Alg_R(T_1, T_2)$ to denote the tree to tree inclusion algorithm from roots, described in previews section, where $T_1$ and $T_2$ are pattern and target trees. $Alg_R(T_1, T_2)$ returns true, if and only if $T_1$ can be included in $T_2$ from roots.

- We write $Alg(T_1, T_2)$ to denote the tree to tree inclusion algorithm, described in the previous section, where $T_1$ and $T_2$ are pattern and target trees. $Alg(T_1, T_2)$ returns true, if and only if $T_1$ can be included in $T_2$.
**Extract Trees from a Star**  To extract trees from a star, we get a list of trees that their roots connected in the cycle from that star. More precisely, given a star $S$, in order to extract a list of trees from the cycle in $S$. We get a list of trees having their roots connected in the cycle from $S$, with respecting to the order of all trees in the cycle.

We define an algorithm to extract trees from a star.

**Algorithm 3: Extract Trees From a Star**

**Input:** A star $T$

**Output:** A list of trees from the cycle in $T$

Let $S := C(T)$, $v :=$ random vertex from $S$ and $L := \text{length}(S)$

`list := \emptyset`

while $L > 0$ do

`list := list \cup \{T[v]\}`

$v := T\text{.succ}(v)$

$L := L - 1$

endWhile

end

**Rotation**  Given two lists of trees $L_1$ and $L_2$, to check that $L_1$ can be included in $L_2$, we need to test the inclusion of all possible rotations from $L_1$ with $L_2$. Therefore we define an algorithm to rotate a list of trees.

**Algorithm 4: Rotation**

**Input:** A list of trees to be rotated ($L$)

**Output:** Rotate $L$

Let $TEMP := L[1]$

for all $I \in \{1, \ldots, \text{length}(L)\}$ do

$L[I] := L[I+1]$

endfor

$L[\text{length}(L)] := L[TEMP]$

end

### 4.1 Star to Star Inclusion Algorithm

In this section, we describe an algorithm to test whether a star is included in another star. Consider two stars $S_1$ and $S_2$, in order to test the inclusion for $S_1$ in $S_2$, first we get two lists of trees from the cycle for $S_1$ and $S_2$, denoted $L_1$ and $L_2$ respectively, using the extract algorithm defined above. We say that $S_1$ is included in $S_2$ if and
only if $L_1$ can be included in $L_2$ using tree inclusion from roots algorithm, described in previews section. In order to test the inclusion $L_1$ from $L_2$, we need to check that for each tree $p \in L_1$ there is a tree $t \in L_2$, such that $p$ can be included in $t$ from roots. If we find that all trees in $L_1$ can be included in $L_2$, we say that $L_1$ can be included in $L_2$. Otherwise we rotate $L_1$ and we repeat the same steps until either we find that $L_1$ can be included in $L_1$ or there is no more possible rotations for $L_1$, in this case we say that $L_1$ is not included in $L_2$.

In this case, we should use tree inclusion algorithm every time we rotate a list of trees.

In order to improve efficiency, we construct a boolean two dimentional array, let say $M$ which contains the result of trees inclusion for $L_1$ and $L_2$. We only fill M once at the first step of the algorithm, and we use $M$ at each time we rotate a list, which saves a lot of time. Consequently, the algorithm runs much faster than with out using $M$.

In summary, let $L_1 = \{p_1, p_2, \ldots, p_a\}$ and $L_2 = \{t_1, t_2, \ldots, t_b\}$, we fill M as follow:

- $M[p_i, t_j] = \text{true}$, if and only if the tree $p_i$ can be included in the tree $t_j$ from roots, where $p_i \in L_1$ and $t_j \in L_2$.
- $M[p_i, t_j] = \text{false}$, if and only if the tree $p_i$ can not be included in the tree $t_j$ from roots, where $p_i \in L_1$ and $t_j \in L_2$.
- $M[p_i, t_j] = \text{NULL}$, if and only if $M[p_i, t_j]$ is not filled yet.

Let say $S_1 = \text{length of } L_1$ and $S_2 = \text{length of } L_2$, then to test that $L_1$ is included in $L_2$, we use two pointers let us say $P_1$ and $P_2$. $P_1$ points to the first element in $L_1$ and $P_2$ points to the first element in $L_2$, then we do the following:

- If $M[L_1[P_1], L_2[P_2]] = \text{true}$, we increment both $P_1$ and $P_2$.
- If $M[L_1[P_1], L_2[P_2]] = \text{false}$, we increment $P_2$.
- If $M[L_1[P_1], L_2[P_2]] = \text{NULL}$, we compute $M[p_i, t_j]$ using tree inclusion algorithm from roots, and increment $P_2$.
- If $S_1 = P_1, L_1$ can be included in $L_2$.
- If $S_2 = P_2$ then , we rotate $L_1$ using rotation algorithm described above, and we repeat the same steps until either we find that $L_1$ can be included in $L_2$, or there is no more possible rotations for $L_2$, in that case we say that $L_1$ can not be included in $L_2$. 

12
Problem definition  Given two stars P and S, the problem defined by this algorithm is to check whether P can be included in T.

Algorithm 5: Star to Star Inclusion
Input: P and T
Output: P ⊑ T?

Let $L_1 := Extract(P)$, $L_2 = Extract(T)$, M is a two dimensional boolean array, all elements in M is NULL at first.

for all $I \in \{1, \ldots, length(L_1)\}$ do
  $P_1 = 0, P_2 = 0$
  while $P_1 \leq length(L_1)$ and $P_2 \leq length(L_2)$ do
    if $M[L_1[P_1], L_2[P_2]] = \text{NULL}$ then
      $M[L_1[P_1], L_2[P_2]] := \text{Alg}_R(L_1[P_1], L_2[P_2])$
    else
      if $M[L_1[P_1], L_2[P_2]] = \text{TRUE}$ then
        $P_1 := P_1 + 1$
      end
    endif
    $P_2 := P_2 + 1$
  endwhile
  if $P_1 = \text{Length}(L_1)$ then
    return TRUE
  end
endfor
return FALSE
4.2 Tree to Star Inclusion Algorithm

In this section, we describe an algorithm to test whether a tree is included in a star. To test whether a tree is included in a star, we do the following.

- Given a star S, we get all vertices from the cycle in S, let say S’.
- For each $v \in S'$, we do the following:
  Let $w = T.succ(v)$, and $T_2 = T.delete(v, w)$, if P can be included in $T_2$, we say that P can be included in S. Otherwise, we repeat the same steps until either we find that P can be included in S, in that case we say that P can be included in T, or we find that there is no more possible edge deletions in T. In this case, we say that P can not be included in T.

**Problem definition**  Given a tree P and a star S, the problem defined by this algorithm is to check whether P can be included in S.

---

**Algorithm 6: Tree To Star Inclusion**

**Input:** A pattern tree P and target star T  
**Output:** $P \subseteq T$?

Let $S := C(T)$.

for all $v \in S$ do 
  $w := T.succ(v)$.
  Let $T_2 := T.delete(v, w)$.
  if $Alg(P, T_2) = TRUE$ then 
    return TRUE
  endfor
return FALSE
end
Figure 4.2: Example of Tree to Star Inclusion, each of $P_1, P_2, P_3$ is included in $T$
Chapter 5

Unordered Tree Matching Algorithm

In this section, we describe an algorithm to find all possible matching for two trees. First, we describe all notations and terminologies that we will use in the algorithm, then we describe the tree to tree matching algorithm with some examples.

Notations and terminologies

**Matching Concept:** Consider two Trees $T_1$ and $T_2$, all possible matching for $T_1$ from $T_2$ is a set of all possible trees that we can obtain from $T_2$, such that each of these tree is homomorphically equivalent to $T_1$, using the following operation:

- Leaf deletion
- Contraction
- Variable deletion

See Chapter 2 for the definition of above operations.

All possible matching for $T_1$ from $T_2$ is a set of injections from $T_1$ to each of these tree, and we denoted by $A_T(T_1,T_2)$

Given two trees $T_1$ and $T_2$, we use two notations to find all possible matching, we call these two notaions *Children Propagation* and *Matching Propagation*.

We define these two notations and how we compute, and use them to find all possible matching for trees in more details in the rest of this section.
**Children Propagation**  Given two trees $T_1$ and $T_2$. Let $w_1, w_2, w_3, \ldots, w_t$ be children of a $w \in T_2$. We assume that $r$ to be a two dimensional array that will contain matching between each vertex in $T_1$ and corresponding vertex in $T_2$. We define the children propagation for a given vertex $v \in T_1$ and vertex $w \in T_2$, as a set of matching between $v$ and children of $w$. More precisely, we compute children propagation as follow:

$$
\text{Children propagation} := r[v,w_1] \cup r[v,w_2] \cup \ldots \cup r[v,w_t].
$$

**Matching Propagation**  Given two trees $T_1$ and $T_2$ and $w \in T_1$, $v \in T_2$, let $w_1, w_2, w_3, \ldots, w_t$ be children of $w$, and $v_1, v_2, v_3, \ldots, v_t$ be a children of $v$. Suppose that $r$ is a two dimmensional array will contain all possible matching for $T_1$ from $T_2$. We construct a bipartite graph $G = (V,E)$, where $V := \{v_1, v_2, v_3, \ldots, v_t\} \cup \{w_1, w_2, w_3, \ldots, w_t\}$ with $(v_i, w_j) \in E$, if and only if $r[v_i, v_j] \neq \emptyset$, where $1 \leq i \leq p$ and $1 \leq j \leq t$. We define $M$ to be a two dimensional array that contains all possible matching for graph $G$. We define matching propagation for $v$ and $w$ as a set of matching between children of $w$ and $v$. In summary, For each $\text{match}((v_1, w_1), (v_2, w_2), \ldots, (v_p, w_t)) \in M$, we compute matching propagations as follow:

$$
\text{Matching propagation} := \text{Matching propagation} \cup r[v_1, w_1] \oplus r[v_2, w_2] \oplus \ldots \oplus r[v_p, w_t].
$$

Matching propagation := Matching propagation $\oplus (v, w)$, where $\oplus$ defined as follow:

Given two sets of sets $A$ and $B$, $A \oplus B = \{a \cup b | a \in A \text{ and } b \in B\}$.

### 5.1 Tree to Tree Matching Algorithm

In this section, we describe an algorithm to find all possible matching for two trees. We use both children and matching propagations that we described above, to compute all possible matchings for $T_1$ from $T_2$.

**Problem Definition**  Let $P = (V, E)$ be a pattern tree and $T = (V, E)$ be a target tree, the problem is to find all possible matching for $P$ from $T$. 
Algorithm 7: Tree to Tree Matching

Input:  
P and T

Output: all possible matching for P from T (A_T(P, T))

Let M be a two dimensional array will contain all possible matching.

for all  \( v \in V(P) \) in the postorder do
  for all  \( w \in V(T) \) in the postorder do
    if  \( \text{label}(v) \subseteq \text{label}(w) \) then
      \[ M[v, w] := M[v, w] \oplus \text{matching propagation}(v, w) \]
      \[ M[v, w] := M[v, w] \cup \text{Children propagation}(v, w) \]
    end

The following are all possible matching for the given pattern tree from the given target tree in figure 5.1, we find all possible matching that the given pattern tree can be matched to the given target tree. As we see in the example, there are 24 possible matchings for given pattern tree from the given target tree.

- \{((a, 1), (b, 2), (c, 3)), ((a, 1), (b, 4), (c, 7)), ((a, 1), (b, 5), (c, 7)), ((a, 1), (b, 6), (c, 7))\}
- \{((a, 2), (b, 4), (c, 7)), ((a, 2), (b, 5), (c, 7)), ((a, 2), (b, 6), (c, 7)), ((a, 2), (b, 4), (c, 7))\}
- \{((a, 2), (b, 1), (c, 3)), ((a, 3), (b, 5), (c, 7)), ((a, 3), (b, 6), (c, 7)), ((a, 3), (b, 4), (c, 7))\}
- \{((a, 4), (b, 6), (c, 5)), ((a, 4), (b, 2), (c, 7)), ((a, 4), (b, 1), (c, 7)), ((a, 4), (b, 6), (c, 7))\}
- \{((a, 6), (b, 4), (c, 5)), ((a, 5), (b, 1), (c, 7)), ((a, 5), (b, 2), (c, 7)), ((a, 5), (b, 3), (c, 5))\}
- \{((a, 6), (b, 4), (c, 7)), ((a, 6), (b, 3), (c, 7)), ((a, 6), (b, 1), (c, 7)), ((a, 6), (b, 2), (c, 7))\}
Chapter 6

Star Matching algorithms

In this section, given two stars $S_1$ and $S_2$, we describe two algorithms to find all possible stars from $S_2$ that is homomorphically equivalent to $S_1$, using the following operations.

- Leaf deletion
- Contraction
- Variable deletion

See Chapter 2 for the definition of above operations.

We call all possible injections from $S_1$ to each of these star by *all possible matching*, and we denote it by $A_S(S_1, S_2)$. The first algorithm is to find all possible matchings for a star from another star. The second algorithm is to find all possible matchings for a tree from a star.

First, we define all notations and terminologies that we use in this section, and then we describe the star from star matching algorithm. At the end of this section, we describe a tree from a star matching algorithm.

Notations and Terminology  The following notations and terminology will be used in both algorithms.

- Given a star $S$, we call the algorithm to extract trees from a star that was described in Chapter 4 by $\text{Extract}(S)$.
- Given a star $S$, we call the operation to get a set of all edges in the cycle from $S$ by $\text{getCycle}(S)$. 

19
We call the algorithm to find all possible matchings for two trees that described in preview Chapter by \textit{TreeToTree}(T_1, T_2), where \( T_1 \) and \( T_2 \) are pattern and target trees respectively.

Let \( G = (V, E) \), we call the operation to edge \( e \in E \) by \( T_2 = T.\text{delete}(e) \), where \( T_2 \) is a new tree from the result of deletion.

### 6.1 Star to Star Matching Algorithm

In this section, we define an algorithm to find all possible matchings between two stars. We define an algorithm to find all possible matchings for list of trees.

<table>
<thead>
<tr>
<th>Algorithm 8: List to list matching</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> ( L_1 ) and ( L_2 )</td>
</tr>
<tr>
<td><strong>Output:</strong> All possible matchings for ( L_1 ) from ( L_2 )</td>
</tr>
<tr>
<td>Let ( S_2 := \emptyset )</td>
</tr>
<tr>
<td>for all injection ( i ) from ( L_1 ) to ( L_2 ) do</td>
</tr>
<tr>
<td>( S_1 := \emptyset )</td>
</tr>
<tr>
<td>for all pair ( (p, t) \in i ) do</td>
</tr>
<tr>
<td>( S_1 := S_1 \oplus \text{treeToTree}(p, t) )</td>
</tr>
<tr>
<td>( S_2 := S_2 \cup S_1 )</td>
</tr>
<tr>
<td>endfor</td>
</tr>
<tr>
<td>return ( S_2 )</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

Given two stars, \( S_1, S_2 \), to find all possible matchings for \( S_1 \) from \( S_2 \), we do the following:

- We get a list of trees from both \( S_1 \) and \( S_2 \), let say \( L_1 \) and \( L_2 \), respectively, using \textit{Extract} algorithm.

- We get a set of all possible matchings for \( L_1 \) from \( L_2 \), using List to list matching algorithm described above.

- We get a set of all possible matchings for all possible rotations for \( L_1 \) from \( L_2 \).

**Problem Definition**

Given two stars, \( S_1 \) and \( S_2 \), the problem defined by this algorithm is to find all possible matchings for \( S_1 \) from \( S_2 \).
Algorithm 9: Star to Star Matching

Input: P and T

Output: All possible matchings for P from T(A_S(P, T))

Let S := ∅, L_1 = Extract(P), L_2 = Extract(T)
M := length(L_1), M will be the number of possible rotations for L_1
for all I=1 to M do
    S := S ∪ ListToList(L_1, L_2), where ListToList is the algorithm to find all possible matching for two list of trees, that is described above
    L_1 = ROTATE(L_1), where ROTATE is the algorithm to rotate a list of trees.
endfor
return S
end

Example  The following are all possible matchings for the given pattern star from the given target star shown in figures below. \{\{(a, 3), (b, 4)\}, \{(a, 4), (b, 3)\}\}.

6.2 Tree To Star Matching Algorithm

In this section, we define an algorithm to find all possible matchings for a tree form a star

Given a tree T and a star S, to find all possible matchings for P from S, we need
to find all possible trees from S that P can be included, using edge deletion operation described above. In order to find all possible trees from S, we get a set of edges making a cycle in S, using getCycle operation described above, let say C, we assume that M will be a set of all possible matching for T from S.

For each \( e \in C \) we do the following:

\( T_2 = S.delete(e) \), if P can be included in \( T_2 \), then \( M = M \cup TreeToTree(P, T_2) \)

**Problem Definition** Given a tree P and a star S, the problem defined by this algorithm is to find all possible matching for P from S.

```
Algorithm 10: Tree To Star Matching
Input: A pattern tree P and target star T
Output: All possible matchings for P from T
        Let \( S := getCycle(T) \), \( M := \emptyset \)
        for all \( e \in S \) do
            Let \( T_2 = T.delete(e) \)
            if \( TreeToTree(P, T_2) \neq \emptyset \) then
                \( M := M \cup TreeToTree(P, T_2) \)
            endif
        endfor
end
```

**Example** The following are all possible matchings for the given pattern tree from the given target star shown in figures above.

- \{ (a, 1), (b, 2), (c, 3) \} , \{ (a, 1), (b, 2), (c, 4) \} , \{ (a, 1), (b, 4), (c, 3) \}
- \{ (a, 2), (b, 1), (c, 3) \} , \{ (a, 4), (b, 1), (c, 3) \} , \{ (a, 2), (b, 4), (c, 3) \}
- \{ (a, 4), (b, 2), (c, 3) \}
Figure 6.2: Example of tree to Star Matchings
Chapter 7

Unordered Heaps Inclusion Algorithm

In this section, we describe an algorithm to test whether a heap is inclusion in another heap. We first define heap and how we represent a heap as a graph, then we define the ordering relations between cells in that heap. We also define all notations and terminologies that we use in heap inclusion algorithm.

**Heap** A heap consists of a set of *cells*, where each cell has one next-pointer. Examples of such heaps are singly liked lists and circular lists, possibly sharing their parts. A cell in the heap may contain a datum which is a natural number. A heap may use a finite set of *variables* representing *pointers* whose values are cells inside the heap.

**Heaps Representation**
We represent a heap as a graph, where each cell in the heap is represented by a vertex in a graph, in such way that each vertex in this graph has at most one successor.

In summary, a graph $G$ is said to be a heap if the following properties are satisfied:

- Each vertex in $G$ has at most one successor.
- $G$ possibly cyclic.

Figure 7.1: A typical graph representing the heap.
Heap ordering representation  We represent the ordering relations between cells in the heap by the following graphs.

- **Equality Graph:** We represent equality ordering relations (=) between vertices in the graph that represents a heap by undirected graph, we call this graph *Equality Graph*, in such way that edges in this graph represent the equality relations (if any) between all vertices in the graph represents the corresponding heap.

- **Inequality Graph:** We represent Inequality ordering relations (≠) (if any) between vertices in the graph that represents a heap by a directed graph, and we call this graph *Inequality Graph*, in such way that edges in this graph represent Inequality relations between all vertices in the graph represents the corresponding heap.

- **Structural Graph:** We represent the structure of the graph that represents a heap by directed graph, we call this graph *Structural Graph*, in such way that edges in this graph represent the structure of the corresponding heap.

**Notations and Terminologies**  The following notations and terminology will be used in the algorithm.

*Heap Ordering Relations*

We denote an Inequality graph in the graph represents a heap by \( E_r \) and we use color *red* to represent all edges in this graph. We denote an equality graph in the graph represents a heap by \( E_g \), and we use a color *green* to represent all edges in this graph. We denote the structure of the graph represents a heap by \( E_b \), and we use a color *blue* to represent all edges in this graph.

*Heap Components*

The graph that represents a heap is a set of *components*, each of these components can be either a tree or a star. Given a heap \( H \), we call the operation to find a set of components for \( H \) by \( \text{getComponent}(H) \).

*Star To Star Matching Algorithm*

We call the algorithm to test inclusion problems for two stars that was described in previews section by \( \text{StarToStar}(P, T) \), where \( P \) is a pattern star and \( T \) is a target star.
7.1 Heap ordering

In general, given two heaps $H_1$ and $H_2$, we check the ordering between $H_1$ and $H_2$, using a sequence of the followig operations:

- **Contraction**: we delete a node $v \in H_2$, such that the followig properties are satisfied:
  - $v$ has one incoming edge and possibly one outgoing edge
  - $v$ has no labels
  - $v$ has no ordring relation

  and then we connect a parent of $v$ (if any) to its children.

- **Edge deletion**: let say $H_2 = (V,E)$, we delete an edge $e \in E$.

- **Order deletion**: we delete the order relation of the node $v \in H_2$

- **Variable deletion**: we delete the label or variable of the node $v \in H_2$

We say that $H_1 \sqsubseteq H_2$, if we can obtain $H_1$ from $H_2$, performing sequences of the following operations:

- Edge deletion
- Contraction
- Cell deletion
- Order deletion
- Variable deletion
7.2 Heap Inclusion Algorithm

In this section, we describe an algorithm to test whether a heap is included in another heap. We first, describe how this algorithm works and then we define algorithm.

Given two heaps $H_1$ and $H_2$, a heap inclusion for $H_1$ in $H_2$ is to check that all graphs that represent all ordering relations form $H_1$ can be included in $H_2$.

More precisely, in order to test the inclusion $H_1$ in $H_2$, there must be a set of vertices in $H_2$, such that all vertices and edges from all graphs that represent all ordering relations in $H_1$ are included.

We use all possible matching for all components from $H_1$ and $H_2$, the solution is the matching that contains all graphs that represent all ordering relations in $H_1$.

In order to find all possible matching for $H_1$ from $H_2$, we construct a bipartite graph $G = (S_1 \cup S_2, E)$, where $S_1$ and $S_2$ are the set of components from $H_1$ and $H_2$ respectively, using $getComponents$ operation described above, with $(s_i, s_j) \in E$ if and only if the matching for $s_i$ from $s_j$ is not empty, where $s_i \in S_1$ and $s_j \in S_2$.

**Algorithm 11:** Heaps consistency  
**Input:** P, T and M, where M is the one of the possible matching for P from T  
**Output:** P is consist to T according to all $E_r$ and $E_g$  
Let $P.E_r$ is a set of all $E_r$ in P, and $P.E_g$ is a set of all $E_g$ in P  
$T.E_r$ is a set of all $E_r$ in T, and $T.E_g$ is a set of all $E_g$ in T  
**for each** pair $(v_1, v_2) \in P.E_r$ **do**  
Let $w_1 = M.v_1$, where $v_1$ and $w_1$ is a pair in M, and $M.v_1$ is a vertex in T that $v_1$ can match to it in given solution, $w_2 = M.v_2$  
**if** pair $(w_1, w_2) \notin T.E_r$ **then**  
return FALSE  
**endfor**  
**for each** pair $(v_1, v_2) \in P.E_g$ **do**  
Let $w_1 = M.v_1$, $w_2 = M.v_2$  
**if** pair $(w_1, w_2) \notin T.E_g$ and pair $(w_2, w_1) \notin T.E_g$ **then**  
return FALSE  
**endfor**  
**return** TRUE  
**end**

**Problem definition**
Let $P = (V_P, E_P)$ and $T = (V_T, E_T)$ be two heaps, where $E = (E_b \cup E_r \cup E_g)$. The
Figure 7.2: Example of Heap Inclusion, each of $P_1, P_2, P_3$ can be included in $T$.

The problem is to decide whether there exist an injection $\iota : V_P \to V_T$ such that $\iota$ contains all $E_b, E_g, E_r$ from $P$.

**Algorithm 12:** Heap to Heap Inclusion  
**Input:** $P$ and $T$  
**Output:** $P \subseteq T$

Let $S_1 := \text{getComponents}(P)$, $S_2 = \text{getComponents}(T)$, $S_3 = \emptyset$.

We construct a bipartite graph $G$ based on $S_1$ and $S_2$,

$G = (S_1 \cup S_2, E)$ with

$(s_i, s_j) \in E$, if and only if $\text{starToStar}(s_i, s_j) \neq \emptyset$, where $s_i \in S_1, s_j \in S_2$.

Let $M$ be an array that contains all possible matchings in $G$

for all match $\in M$ do

if $\text{consist}(P, T, \text{match}) = \text{TRUE}$, where $\text{consist}(P, T, \text{match})$ is the algorithm described above then

return $\text{TRUE}$

endfor

return $\text{FALSE}$

end
Chapter 8

Experimental Results

In this section, we have described the performance of our algorithms on a number of graphs with a large number of nodes.

The following tables are the results of running our algorithms on a number of graphs with number of nodes. In each table, we have run one of our algorithm several times on the same number of nodes, then we compute the best, worst and the average time that algorithm can take to test the corresponding number of nodes.

In each of the following table, the column Nodes shows the number of nodes to be tested, Column BestCase shows the best time that this algorithm takes to test corresponding number of nodes. The column WorstCase shows the worst time that this algorithm takes to test corresponding number of nodes. The column Average shows the average time that this algorithm takes to test corresponding number of nodes.

We can see from the results of matching algorithms showing below, that the large number of nodes has less effect than the structure of the graph, for testing the graph matching.

For example in table 8.7, we see that algorithm finds all possible matching for graph with 5 nodes in 42 millisecond, but with 10 nodes it tests in 16 milliseconds, this is becuase in this example,in Case 5 nodes, these is a maching in graph, but in Case 10 these is no matching at all.
## Table 8.1: Tree to Tree Inclusion performance in Milliseconds

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Best Case</th>
<th>Worst Case</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>87</td>
<td>260</td>
<td>114</td>
</tr>
<tr>
<td>200</td>
<td>349</td>
<td>400</td>
<td>379</td>
</tr>
<tr>
<td>300</td>
<td>812</td>
<td>855</td>
<td>831</td>
</tr>
<tr>
<td>400</td>
<td>1429</td>
<td>1497</td>
<td>1465</td>
</tr>
<tr>
<td>500</td>
<td>2170</td>
<td>2356</td>
<td>2246</td>
</tr>
<tr>
<td>600</td>
<td>3126</td>
<td>3393</td>
<td>3278</td>
</tr>
<tr>
<td>700</td>
<td>4380</td>
<td>4721</td>
<td>4563</td>
</tr>
<tr>
<td>800</td>
<td>5835</td>
<td>6212</td>
<td>6009</td>
</tr>
</tbody>
</table>

## Table 8.2: Tree to Tree from roots performance

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Best Case</th>
<th>Worst Case</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>89</td>
<td>262</td>
<td>113</td>
</tr>
<tr>
<td>200</td>
<td>354</td>
<td>409</td>
<td>374</td>
</tr>
<tr>
<td>300</td>
<td>813</td>
<td>884</td>
<td>841</td>
</tr>
<tr>
<td>400</td>
<td>1408</td>
<td>1629</td>
<td>1489</td>
</tr>
<tr>
<td>500</td>
<td>2203</td>
<td>2381</td>
<td>2313</td>
</tr>
<tr>
<td>600</td>
<td>3236</td>
<td>3554</td>
<td>3380</td>
</tr>
<tr>
<td>700</td>
<td>4426</td>
<td>4685</td>
<td>4564</td>
</tr>
<tr>
<td>800</td>
<td>5968</td>
<td>6298</td>
<td>6131</td>
</tr>
</tbody>
</table>

## Table 8.3: Star to Star Inclusion performance in Milliseconds

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Best Case</th>
<th>Worst Case</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>107</td>
<td>333</td>
<td>138</td>
</tr>
<tr>
<td>200</td>
<td>356</td>
<td>445</td>
<td>392</td>
</tr>
<tr>
<td>300</td>
<td>819</td>
<td>954</td>
<td>871</td>
</tr>
<tr>
<td>400</td>
<td>1417</td>
<td>1504</td>
<td>1464</td>
</tr>
<tr>
<td>500</td>
<td>2226</td>
<td>2399</td>
<td>2339</td>
</tr>
<tr>
<td>600</td>
<td>3224</td>
<td>3532</td>
<td>3326</td>
</tr>
<tr>
<td>700</td>
<td>4337</td>
<td>4592</td>
<td>4501</td>
</tr>
<tr>
<td>800</td>
<td>5641</td>
<td>6227</td>
<td>5994</td>
</tr>
<tr>
<td>Nodes.</td>
<td>Best Case</td>
<td>Worst Case</td>
<td>average</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
<td>------------</td>
<td>---------</td>
</tr>
<tr>
<td>100</td>
<td>294</td>
<td>725</td>
<td>483</td>
</tr>
<tr>
<td>200</td>
<td>1162</td>
<td>3949</td>
<td>2197</td>
</tr>
<tr>
<td>300</td>
<td>1661</td>
<td>9491</td>
<td>5655</td>
</tr>
<tr>
<td>400</td>
<td>4629</td>
<td>16677</td>
<td>11146</td>
</tr>
<tr>
<td>500</td>
<td>12341</td>
<td>30471</td>
<td>18936</td>
</tr>
<tr>
<td>600</td>
<td>10123</td>
<td>40548</td>
<td>24438</td>
</tr>
<tr>
<td>700</td>
<td>9842</td>
<td>48919</td>
<td>29146</td>
</tr>
<tr>
<td>800</td>
<td>24114</td>
<td>60298</td>
<td>46468</td>
</tr>
</tbody>
</table>

Table 8.4: Tree to Star Inclusion performance in Milliseconds

<table>
<thead>
<tr>
<th>Nodes.</th>
<th>Best Case</th>
<th>Worst Case</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>28</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>322</td>
<td>71</td>
</tr>
<tr>
<td>15</td>
<td>7</td>
<td>70</td>
<td>25</td>
</tr>
<tr>
<td>20</td>
<td>1173</td>
<td>156770</td>
<td>46483</td>
</tr>
<tr>
<td>25</td>
<td>258</td>
<td>52226</td>
<td>13742</td>
</tr>
</tbody>
</table>

Table 8.5: Tree to Tree matching performance in Milliseconds

<table>
<thead>
<tr>
<th>Nodes.</th>
<th>Best Case</th>
<th>Worst Case</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>155</td>
<td>41</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>150</td>
<td>71</td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>86596</td>
<td>17420</td>
</tr>
<tr>
<td>20</td>
<td>19</td>
<td>1879</td>
<td>558</td>
</tr>
<tr>
<td>25</td>
<td>27</td>
<td>17706</td>
<td>3681</td>
</tr>
</tbody>
</table>

Table 8.6: Tree to Star matching performance in Milliseconds

<table>
<thead>
<tr>
<th>Nodes.</th>
<th>Best Case</th>
<th>Worst Case</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>165</td>
<td>42</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>21</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>570</td>
<td>819</td>
<td>689</td>
</tr>
<tr>
<td>20</td>
<td>26</td>
<td>38</td>
<td>29</td>
</tr>
<tr>
<td>25</td>
<td>486</td>
<td>591</td>
<td>547</td>
</tr>
</tbody>
</table>

Table 8.7: Heap to Heap inclusion performance in Milliseconds
Chapter 9

Conclusions and Future Work

We have defined and implemented a number of algorithms to verify some safety properties on a program that manipulate dynamic data structure, that has one next pointer, such that traditional singly linked lists and circular lists. A program has a set of cells, each of these cell has one next pointer to point to next cell in the program.

We have represent a program as a graph such that each vertex in a graph represents a cell in the program, and the successor of any vertex in the graph represents the next cell that pointed by this cell. The problems that we have verified and described in our algorithms are graph ordering and graph matching.

First, we have designed and implemented two algorithms to verify the ordering between two trees, and then two algorithms for stars ordering. We also define two algorithms to find the matching problems for trees and stars, finaly we define one algorithm to verify the heap ordering.

We have run our algorithms on a number of graphs with a large number of nodes, using Computer with 1.7 GH CPU, and then we summarize the results from running our algorithms into some tables.

One direction for the future plan for our work, is to extend our algorithms on a program that manipulate more geneiric graphs. In other word, to verify the ordering and matching problems for the programs that manipulating more complex data structure rather than just singly linked lists, for instance doubly linked list and tree like structure. An intersting and more challenging subjects then for our work is to extend our algorithms and finding some methods to test the ordering and matching of graphs that have vertices with more than one successor.
Bibliography


