Dynamic connectivity algorithms for Feynman diagrams

by

Rubao Ji

(Under the direction of Robert W. Robinson)

Abstract

A Feynman diagram of order n can be viewed as a matching on $2n$ nodes (edges in the matching are undirected and called V-lines) along with a permutation of the $2n$ nodes (represented by directed edges called G-lines). Dynamic connectivity algorithms for Feynman diagrams are described in this thesis. The algorithms have two major components: diagram update and connectivity query. Two approaches are used to maintain and update Feynman diagrams. The key data structure for the first approach is a splay forest, which has an amortized expected cost of $O(\log n)$. The second approach applies a treap forest, which has an expected cost of $O(\log n)$. For query operations, three algorithms were designed and implemented having expected costs per operation of $O(n)$, $O(\log^2 n)$ and $O(\log n)$, respectively.

Both splay forests and treap forests were implemented with arrays, allowing the nodes to be switched to be found in constant time. Five experiments were carried out with our implementations. The results showed good agreement with the time complexity analysis, validating the algorithm design and implementation.

Index words: Feynman diagram, dynamic algorithm, connectivity, splay tree, treap, time complexity

Dynamic connectivity algorithms for Feynman diagrams

by

Rubao Ji

M.Sc., Ocean University of Qingdao, China 1994

A Thesis Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Master of Science

Athens, Georgia

2002

© 2002

Rubao Ji

All Rights Reserved

Dynamic connectivity algorithms for Feynman diagrams

by

Rubao Ji

Approved:

Major Professor: Robert W. Robinson

Committee: E. Rodney Canfield
Daniel M. Everett

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
December 2002

Acknowledgments

I would like to thank my advisor, Dr. Robert Robinson at Department of Computer Science, University of Georgia, for many ideas and suggestions, for great encouragement, and for giving me the opportunity to work with this thesis. His precious instruction always kept me on the right track and made this thesis possible.

I would also like to thank Drs. E. Rodney Canfield and Daniel M. Everett at Department of Computer Science, University of Georgia, for willing to serve as my committee members and offering their great help.

My special thanks go to Dr. Changsheng Chen at the School of Marine Science and Technology, University of Massachusetts (formerly a professor of Marine Science at The University of Georgia), for encouraging me to pursue the second degree in computer science, for financial support, and most importantly, for advising me with his profound attitude towards research.

Last but certainly not least, I wish to express my sincere thanks to my wife and my 2-year-old daughter for their invaluable love and support.

Table of Contents

Page

Acknowledgments iv

List of Figures vii

List of Tables viii

Chapter

1 Introduction 1

1.1 About Feynman diagrams 1

1.2 Fully dynamic connectivity algorithms 1

1.3 Terms and symbols 4

1.4 Outline of the thesis 5

2 Algorithm design and time complexity analysis 6

2.1 Update using splay forest data structure 8

2.2 Update using treap forest data structure 9

2.3 Query on connectivity of Feynman diagrams 10

3 Data structures and implementations 14

3.1 Array-based Splay Forest (ASF) update approach . 14

3.2 Array-based Treap Forest (ATF) update approach 19

3.3 Implementation of SCQ 24

3.4 Implementation of ACQ 24

3.5 Implementation of IDQ 25

4 Experimental results 29
4.1 Comparison of time complexities for ATF and ASF 29
4.2 Comparison of time complexities for update and query 29
4.3 The frequency of a disconnected diagram 33
4.4 Time per switching 34
4.5 Comparison of SCQ, ACQ and IDQ 34
5 Discussion and Future work 37
5.1 Advantages and disadvantages of the splay tree and treap approaches 37
5.2 ASF initialization 38
5.3 Switching from disconnected to connected diagrams 38
5.4 Dynamic irreducibility for Feynman diagrams . . . 38
Bibliography 41

List of Figures

1.1 An example of a Feynman diagram of order 3. 2

1.2 Illustration of the switch operation. 3

2.1 Illustration of the rotate operation in a treap 10

3.1 Splay tree after insertions. 15

3.2 Case2: Switch(a,b) in ASF, where nodes a and b are from two different G-cycles. 17

3.3 Case 3: Switch (a,b) in ASF, where nodes a and b are from the same G-cycle. 18

3.4 Illustration of each step of an insertion sequence and the resulting treap. 20

3.5 Case2: Switch(a,b) in ATF, where nodes a and b are from two different G-cycles. 22

3.6 Case 3: Switch (a,b) in ATF, where nodes a and b are from the same G-cycle. 23

3.7 Adjacency list representation of a G-cycle graph 26

4.1 Comparison of time costs for ASF and ATF. 30

4.2 Comparison of update and query times for ASF and ATF. 31

4.3 Percentage of update and query times for ASF and ATF. 32

4.4 Unsuccessful switchings as a function of diagram order. 33

5.1 A better way of initializing a splay tree. 39

List of Tables

1.1 The evolution of deterministic algorithms for the fully dynamic connectivity problem. 3

4.1 Comparison of time per switching for ASF and ATF. 35

4.2 Comparison of times for SCQ, ACQ and IDQ. 36

Chapter 1

Introduction

1.1 About Feynman diagrams

The Feynman diagram, introduced by physicist Richard P. Feynman in 1949, is widely used in physics to visualize and describe quantum electrodynamical interactions [10]. In a Feynman diagram each line represents the propagation of a free elementary particle and each node represents an interaction of elementary particles. A Feynman diagram with order of n can be viewed as a matching on $2n$ nodes (edges in the matching are undirected and called *V-lines*) along with a permutation of the $2n$ nodes (represented by directed edges called *G-lines*). Figure 1.1 shows an example of a Feynman diagram of order 3.

Feynman diagram expansions are used in quantum physics to express the energy of a system of particles, such as free electrons in a crystal [12]. It is essential to maintain diagram connectivity because most Feynman expansions employ only connected diagrams. In more specialized expansions, only irreducible diagrams (3-edge-connected with respect to G-lines) are considered. Although this study focuses only on connectivity problems in a dynamic Feynman diagram, it aims to serve as a starting point for more sophiscated algorithms dealing with irreducible diagrams.

1.2 Fully dynamic connectivity algorithms

In a fully dynamic graph problem, a graph G has a fixed vertex set V. The graph G may be updated by insertions and deletions of edges. These updates may be

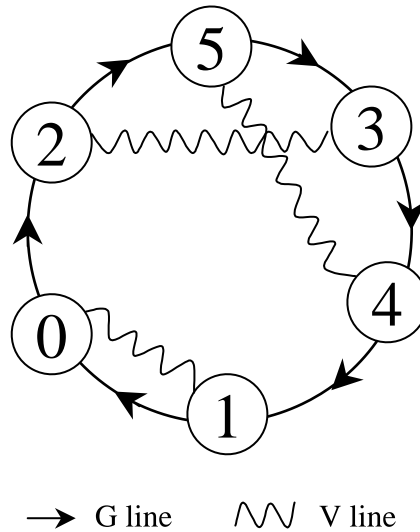

Figure 1.1: An example of a Feynman diagram of order 3.

interspersed with queries about properties of the graph. For the fully dynamic connectivity problem, the queries are connectivity queries, asking whether two given nodes are connected in graph G. Both updates and queries are presented on-line, meaning that we have to respond to an update or query without knowing anything about the future. A good fully dynamic algorithm is able to update the graph and compute the relevant properties (*e.g.* connectivity) of the resulting graph as quickly as possible. The main idea of a fully dynamic algorithm is to take advantage of previous computations to speed up the solution.

Strictly speaking, the connectivity problem in Feynman diagrams is not a “true” fully dynamic connectivity problem. In Feynman diagrams, the update is called a switching, a combination of two edge deletions and two edge insertions (see illustration in Figure 1.2). The query is always after the update (switching) to make sure that the diagram remains connected. Although any fully dynamic connectivity algorithm can be directly applied to our problem, that might involve some unnecessary

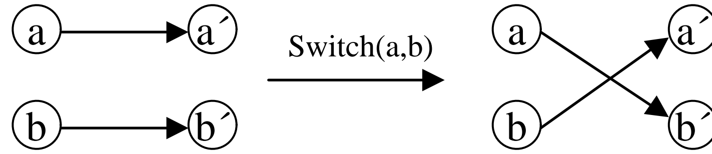

Figure 1.2: Illustration of the switch operation.

Table 1.1: The evolution of deterministic algorithms for the fully dynamic connectivity problem.

Year	Author	Approach	update | query
1985	Frederickson [3]	Topology tree	$O(\sqrt{m})\|O(1)$
1992	Epstein *et al.* [2]	Sparsification	$O(\sqrt{n})\|O(1)$
1997	Henzinger & King [6]	ET-tree	$O(\sqrt[3]{n}\log n)\|O(\log n/\log\log n)$
1998	Holm *et al.* [8]	ET-tree & B-tree	$O(\log^2 n)\|O(\log n/\log\log n)$

time and algorithm complexity. Based on unique characteristics of Feynman diagrams and the particular form of the sequence of updates and queries, it is possible to develop specialized simple algorithms with "good" expected time complexity.

We can get a better idea of how "good" a fully dynamic algorithm can be by reviewing existing algorithms. For deterministic algorithms, all the previous best solutions to the fully dynamic connectivity problem were also solutions to the minimum spanning tree problem. The evolution of deterministic algorithms for the fully dynamic connectivity problem is shown as table 1.1. The $O(\log^2 n)$ update time for Holm *et al.*'s 1998 result is an amortized time; the rest are worst-case times.

Randomization has been used to improve the bounds for the connectivity problems in fully dynamic graphs. Henzinger and King [5] showed that a spanning forest could be maintained in $O(\log^3 n)$ expected amortized time per update, and the connectivity queries are supported in time $O(\log n/\log\log n)$. The update time was later improved to $O(\log^2 n)$ (Henzinger and Thorup [7]). A near optimal algorithm was

provided by Thorup [15] with $O(\log n(\log\log n)^3)$ expected amortized update time and $O(\log n/\log\log\log n)$ query time.

In our algorithms, we designed two approaches using simple data structures for updates. The data structure used in the first approach is a splay forest, which has an amortized expected time of $O(\log n)$ per operation; the second approach applies a treap forest, which has an expected time of $O(\log n)$ per operation. For queries, we designed and implemented three algorithms having expected times of $O(n)$, $O(\log^2 n)$ and $O(\log n)$, respectively.

1.3 Terms and symbols

1. *V-line*: An undirected edge between two vertices, also called a boson line and depicted in figures by a wavy edge. The V-lines form a perfect matching on the vertices.

2. *G-line*: A directed edge, also called a fermion line and depicted as straight lines in figures.

   A G-line can be a loop (corresponding to a fixed point of the permutation) or join two vertices which are matched by a V-line.

   In a Feynman diagram, each vertex always has exactly one outgoing G-line and one incoming G-line.

3. *Order* of a Feynman diagram: the number of V-lines, denoted by n.

   For a Feynman diagram with order n, there are exactly $2n$ G-lines and n V-lines. By convention the vertices of a Feynman diagram of order n are numbered $\{0, 1, ..., 2n-1\}$, and the V-lines join vertices $2i$ and $2i+1$, for $i = 0, ..., n-1$.

4. *Connected* Feynman diagram: Any vertex is reachable from any other vertex by a mixed path of V-edges and G-edges. (It makes no difference here whether

or not the G-edges are required to be oriented in the same direction as the path.)

5. *Switch*: Switch(a,b) swaps the vertices to which a and b are incident by G-edges; See Figure 1.2 for an illustration

1.4 Outline of the thesis

In all there are 4 chapters in this thesis.

Chapter 2 describes the approaches for maintaining the connectivity of a Feynman diagram dynamically, including a high level description of update and query algorithms as well as time complexity analyses. Two data structures, splay forests and treap forests, are used for diagram update. Two query approaches, with linear and poly-logarithmic time, are also described. Based on our time complexity analysis, an expected amortized $O(\log n)$ time per switching can be achieved using splay forests, and an expected $O(\log n)$ time per switching using treap forests.

Chapter 3 describes the detailed data structures and algorithm implementations. Array-based splay forest (ASF) and array-based treap forest (ATF) data structures were implemented using C++. Three approaches to connectivity queries are also described in this chapter.

Chapter 4 describes the results of five experiments designed to: 1) compare the time complexities of the ATF and ASF approaches; 2) compare the time complexities of update and query operations; 3) examine the occurrence of disconnected diagrams; 4) compare times per switching for the ATF and ASF approach with different number of switchings; and 5) compare the three connectivity query algorithms. The results validate our algorithm time complexity analyses. Finally, the advantages and disavantages of the splay forest and treap forest approaches are discussed.

Chapter 2

Algorithm design and time complexity analysis

The crux of the algorithm lies in efficiently maintaining G-cycle information for the diagram and providing quick responses to connectivity queries. The design of our algorithm relies on the following features in order to achieve poly-logarithmic time complexity.

- There are a small number ($O(\log n)$) of G-cycles in a randomly generated Feynman diagram.
- There are known data structures for maintaining G-cycle information with amortized or expected $O(\log n)$ time
- The query can be done in expected logarithmic or poly-logarithmic time.

Lemma 2.1 For a randomly generated Feynman diagram of order n, the expected number of G-cycles is $O(\log n)$.

Proof: Let $Pr[\,]$ and $Ex[\,]$ be the probability and expectations, respectively, over the space in which all $(2n)!$ permutations are equally likely to be chosen as the set of G-edges for a Feynman diagram of order n. First, pick a vertex v and let C_l denote that the G-cycle containing vertex v has length l. We need to prove that $Pr[C_l] = 1/2n$. This can be seen by iteration as follows:

$Pr[C_1] = 1/2n;$

$Pr[C_2] = (2n-1)/2n \times 1/(2n-1) = 1/2n;$

$Pr[C_3] = (2n-1)/2n \times (2n-2)/(2n-1) \times 1/(2n-2) = 1/2n;$

:

$Pr[C_i] = (2n-1)/2n \times (2n-2)/2n \times \cdots \times (2n-i+1)/(2n-i+2) \times 1/(2n-i+1) = 1/2n.$

Now, Let X_i = the size of the G-cycle containing node i , then the total number of cycles S is $\sum_{i=1}^{2n} 1/X_i$.

Now

$$\begin{aligned} Ex[1/X_i] &= \sum_{i=1}^{2n} (1/i \times Pr[1/X_i = 1/i]) \\ &= \sum_{i=1}^{2n} (1/i \times 1/2n) \end{aligned}$$

Thus

$$\begin{aligned} Ex[S] &= Ex[\sum_{i=1}^{2n} 1/X_i] \\ &= \sum_{i=1}^{2n} Ex[1/X_i] \\ &= \sum_{i=1}^{2n} \sum_{i=1}^{2n} (1/i \times 1/2n) \\ &= \sum_{i=1}^{2n} (1/2n) \sum_{i=1}^{2n} (1/i) \\ &= \sum_{i=1}^{2n} (1/i). \end{aligned}$$

The sum $\sum_{i=1}^{n}(1/i)$ is called the n'th harmonic harmonic number and often denoted by $H(n)$. When $n \to \infty$, $H(n) \to \log n + \gamma$, where log denote the natural logarithm and γ is Euler's constant (0.57721...).

Thus,

$$Ex[S] = O(\log n).$$

2.1 Update using splay forest data structure

In this approach, we maintains a splay forest F for a Feynman diagram D. Each individual splay tree T in F represents a G-cycle in D, where the in-order traversal of T is the sequence of a G-cycle. The V-line information is represented implicitly according to the convention based on vertex numbering given in Section 1.3.

A splay tree is a binary search tree (BST) on which splaying operations are performed [14]. A splaying operation is one that causes a node in the BST to move up the tree and become the root. Splaying of a node is done by rotating the nodes of a BST, as described below.

Each switch(a,b) operation triggers a series of splay tree operations. Here we describe the basic splay tree operations. In Chapter 3 the switch(a,b) operation is detailed in terms of the basic splay tree operations.

1. *Splay(x)*: Repeat the following splaying steps until x becomes the root of the tree. Let x be the node being accessed, $p(x)$ the parent of node x and $g(x)$ the grandparent of node x.

    - Case 1 (zig). If $p(x)$ is the root, rotate the edge joining x with $p(x)$.
    - Case 2 (zig-zig). If $p(x)$ is not the root and x and $p(x)$ are both left or both right children, rotate the edge joining $p(x)$ with $g(x)$ and then rotate the edge joining x with $p(x)$.
    - Case 3 (zig-zag). If $p(x)$ is not the root and x is a left child and $p(x)$ a right child, or vice-versa, rotate the edge joining x with $p(x)$ and then rotate the edge joining x with the new $p(x)$ (*i.e.*, with the old $g(x)$.)

2. *Join($t1$, $t2$)* : Let y be the first element of $t2$ (in-order ordering); splay(y), then link $t1$'s root as left child of y;

3. *Split(x, t)* : Construct and return two trees, $t1$ and $t2$, where $t1$ contains all items in t less than or equal to x (with respect to the in-order ordering). *Split* is done by calling *splay(x)*, then cutting off the right subtree to form $t2$, and leaving the rest to form $t1$.

Lemma 2.2 (From [14]) Each splay operation has an amortized time of $O(\log n)$ over $\Omega(n)$ operations such that the tree size is always $O(n)$.

2.2 Update using treap forest data structure

This is similar to the splay forest approach, with treaps instead of splay trees.

A treap is a BST in which each node has both a key and a priority [13]. Nodes are ordered in in-order with respect to the keys and are heap-ordered with respect to their priorities. "In-order" means for any node x in the tree $y.key \leq x.key$ for all y in the left subtree of x and $x.key \leq y.key$ for y in the right subtree of x. "Heap order" means that each parent has a higher priority than its children). We can easily see that given a set of nodes with keys and priorities, the treap is unique (assuming priorities are all distinct).

Each switch(a, b) operation triggers a series of treap operations as follows:

1. *Insert(x)*: Attach x to T in an appropriate leaf position. At this point the keys of all nodes in the modified tree are in in-order. Rotate the tree to satisfy the priority requirement.

2. *Delete(x)*: Reverse the *Insert(x)* operation. Find x in the treap, set the priority of x to be the smallest, rotate x down until it reaches the leaf, clip away x.

3. *Rotate(x)*: See Figure 2.1 for rotating left and rotating right.

4. *Split (x,t)*: Insert a node y as the right child of x, set the priority of y to be "infinite". y becomes the root of tree t. The left subtree of y forms $t1$ and

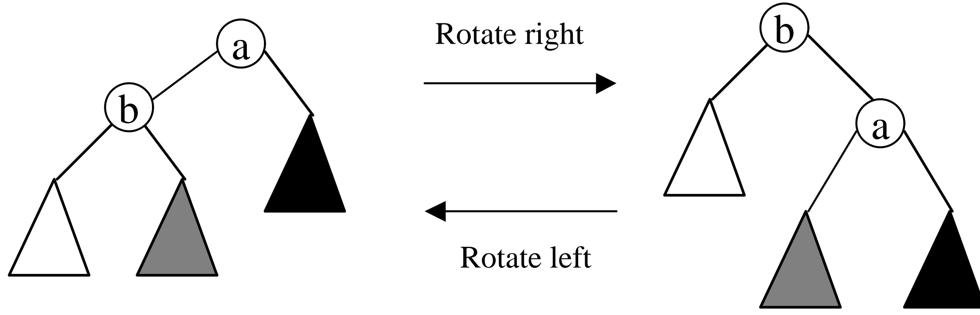

Figure 2.1: Illustration of the rotate operation in a treap

right subtree of y forms $t2$. $t1$ and $t2$ satisfy that for any node a in the tree $t1$ $a.key \leq b.key$ for all b in the tree $t2$.

5. *Join ($t1$,$t2$)*: Create a dummy root, link $t1$ as the left subtree and $t2$ as the right subtree to the root, then delete the dummy root as described in *Delete(x)*

Lemma 2.3 (From [13]) Each treap operation has an expected time of $O(\log n)$ averaged over all permutations of the nodes for the priority attribute.

2.3 Query on connectivity of Feynman diagrams

Three approaches are used to query the connectivity of Feynman diagram. They are called Simplified Connectivity Query (SCQ) , Advanced Connectivity Query (ACQ) and Integrated DFS (depth first search) Query (IDQ).

SCQ is a straightforward approach for querying the connectivity of a Feynman diagram. It arbitrarily chooses a G-cycle to start with by marking it as discovered, then checks for any other G-cycle connected to it via V-lines. If it discovers a new G- cycle, mark it as discovered. Otherwise, it continues checking other discovered G-cycles. The checking is done until either 1) all the discovered G-cycles have been

checked and the number of the discovered G-cycles is less than the total number of G-cycles or 2) The number of the discovered G-cycles is equal to the number of G-cycles. The diagram is not connected in the first case but connected in the second case.

SCQ can easily lead to a linear time complexity even though the number of G-cycle has an expected value of $O(\log n)$. The reason is that the time to discover a new G-cycle can be $O(n)$. To improve performance, ACQ is a better algorithm with an expected time complexity of $O(\log n)$. Instead of arbitrarily choosing a G-cycle to start with in SCQ, ACQ chooses a G-cycle with the smallest size, which allows us to have a better chance of finding a new G-cycle in a short time. After a new G-cycle is discovered, these two cycles are thought of as forming one new G cycle. The size of this new cycle is the sum of the sizes of the two joined cycles. Then ACQ chooses the G-cycle with smallest size, and starts over again until no new G-cycles are found. If the number of joins is fewer than 1 less than the initial number of G-cycles, the diagram is not connected, and otherwise it is connected. We provide a heuristic argument that this algorithm has an expected time complexity of $O(\log^2 n)$.

Lemma 2.4 ACQ algorithm has an expected cost of $O(\log^2 n)$ time to query the connectivity of Feynman diagram

Heuristic justification: A random Feynman diagram can be obtained by choosing a random permutation on the $2n$ vertices for the G-edges, and a random matching for the G-edges. let C be the number of components in the graph consisting of G-cycles and V-edges chosen so far (one for each *prior* round).

If $C = 1$, we are finished, as the diagram must be connected. If $C \geq 2$, let S be the number of vertices in the smallest component; then $S \leq n$.

Now let D_i be the probability that, in examing V-edges at vertices in the smallest component, a V-edge joining it to another component will be found in $\leq i$ attempts, and let S_i be the probability that at least i attempts will be needed. Thus $S_1 = 1$

and $S_{i+1} = 1 - D_i$. Then $Ex[Steps] = \sum_{i\geq 1} S_i < 1+1/2+1/4+... = 2$ assuming that the V-edge choices are independently random. This is not quite true, as previously examined edges affect the probabilities.

From Lemma 2.1, we know that the expected number of G-cycles for a randomly generated Feynman diagram is $O(\log n)$. Therefore, the maximum expected number of component joins is also $O(\log n)$(if the diagram is connected). After each join, it takes expected time $O(\log n)$ to find the component with smallest size (based on our unordered linked list implementation). Therefore, the total expected time to query for connectivity is $O(\log^2 n)$.

The calculation $D_1 = (2n - s)/(2n - 1) > 1/2$ and others based on it are approximations in general, as they assume that no V-edge is known (and hence, all perfect matchings are equally likely as the set of V-edges). However, these are applied in situations where some V-edges have in fact already been explored and thus known. Thus our analysis of expected time is heuristic rather than rigorous. It seems likely that the heuristic analysis is very close to correct in worst cases, and this is supported by the results of experiments as discussed in Chapter 4.

The third algorithm, IDQ, is more integerated with the update process than SCQ and ACQ. When we store and update the G-cycle information, we also store a V-edge which connects two G-cycles. This connecting set of V-edges is denoted by C. Since that the expected number of G-cycle is $m = O(\log n)$ (Lemma 2.1), then $|C| \geq m - 1$, and C contains $|C| - m + 1$ "extra" edges.

A switching which joins two G-cycles maintains connectivity, so we only need to update the names of G-cycles and delete possible extra edges caused by merging two G-cycles. If a switching splits a G-cycle into two, at most two connected components can be created. If one G-cycle, say A, is split into $A1$ and $A2$, any V-edge in C which connects to A, say e, is now either connected to $A1$ or $A2$. C is updated based on

the stored information about e (vertices which e joins). Knowing the vertex in an edge e allows us to find which G-cycle it is in, in time $O(\log n)$.

As the average degree of our DFS tree is under 2, we would expect only about 2 or so ends in the cycle A which need resolution into $A1$ or $A2$. However we can expect some bias toward larger cycles, and correlating with that toward cycles of higher degree in C, when we condition on a cycle splitting due to a switching (bigger cycles are more likely to split). Perhaps this pushes the expected time above $O(\log n)$ asymptotically, but probably not nearly to $O(\log^2 n)$, which is the estimate of time complexity of ACQ.

The next two lemmas summarize the results of this section. These results are heuristic, not rigorous.

Lemma 2.5 Using a splay forest data structure for Feynman diagram updates combined with ACQ for connectivity queries gives an amortized expected time per operation of $O(\log^2 n)$. Using a treap forest data structure for Feynman diagram updates combined with ACQ for connectivity queries gives an expected time per operation of $O(\log^2 n)$.

Lemma 2.6 Using a splay forest data structure for Feynman diagram updates combined with IDQ for connectivity queries gives an amortized expected time per operation of $O(\log n)$. Using treap a forest data structure for Feynman diagram updates combined with IDQ for connectivity queries gives an expected time per operation of $O(\log n)$.

Chapter 3

Data structures and implementations

3.1 Array-based Splay Forest (ASF) update approach

Feynman diagrams are created at random for testing. For a Feynman diagram of order n, an array named $Wout[\,]$ of size $2n$ is created, which is defined as follows: for each node i, $Wout[i]$ is the node to which i is adjacent by its outgoing G-line, which is a random permutation of $\{0, ..., 2n-1\}$. Another four arrays are also created. They are listed as follows:

1. $P[i]$: Parent of node i in the splay tree
2. $LC[i]$: Left child of node i in the splay tree
3. $RC[i]$: Right child of node i in the splay tree
4. $S[i]$: Size of subtree rooted at node i

To explain the algorithm more clearly, we use the Feynman diagram shown in Figure 1.1 as an example.

Step 1: Randomly generate a Feynman diagram

Node i:	0	1	2	3	4	5
$Wout[i]$:	2	0	5	4	1	3

Step 2: Insert nodes into a splay forest

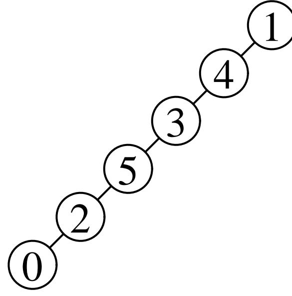

Figure 3.1: Splay tree after insertions.

The equence of insertions is based on $Wout[i]$. In our example, it is 0, 2, 5, 3, 4, 1. Each inserted node is treated as the root of a newly formed splay tree, and the previous node becomes the left child of the current root. Therefore, the in-order of splay tree follows the cyclic order in the G-cycle containing 0 (where started). The value of $P[i]$, $LC[i]$, $RC[i]$ and $S[i]$ are also updated corresponding to each insertion. The newly formed tree after insertions is shown in Figure 3.1, and the arrays $P[i]$, $LC[i]$,$RC[i]$, and $S[i]$ have following values.

$P[i]$:	2	1	5	4	1	3
$LC[i]$:	#	4	0	5	3	2
$RC[i]$:	#	#	#	#	#	#
$S[i]$:	1	6	2	4	5	3

Here, if a node v is the root of a splay tree, $p[v] = v$. If v is a left child of pv, $p[v]$ is stored as bit complementary of pv ($\sim pv$). Conversely, if v is a right child of pv, $p[v]$ is stored as pv. Using bit complementary can make the update and query operation easier and faster. # is a special value indicating the there is no left child of v or no right child of v. In the implementation a value greater than $2n$ is assigned to # to avoid any conflicts with node numbers.

Step 3: Making switches

Switch(a,b)

There are three different cases when switching nodes a and b.

- Case 1: $a = b$. Do nothing.
- Case 2: $a \neq b$, node a and b lie in two different G-cycles. Switch without querying connectivity.
- Case 3: $a \neq b$, node a and b in the same G-cycle. Switch, then query connectivity. If connected, switch successfully; otherwise, switch back.

Case 2 is relatively simple. As shown in Figure 3.2, *splay* node a in tree $T1$, node b in tree $T2$, *split* both a and b from their right subtrees, then connect a to b's right subtree and b to a's right subtree. Finally, find the leftmost node (w) of the tree rooted at b, splay w, connect a as left child of w.

In case 3, there are following two subcases to be considered based on the sequence of in-order traversal in the splay tree, as follows.

- case 3.1: Shown in Figure 3.3 (node a precedes b). After the *splay* at node a, *split* node a from its right subtree. We will see that nodes a and b have different roots. *Splay* node b, then connect node a to node b's right subtree.
- case 3.2: Shown in Figure 3.3 (node b precedes a). After the *splay* at node a, *split* node a from its right subtree. Nodes a and b still share the same root, which is node a. *Split* node a from both its right and left subtrees, *splay* node b and *split* node b from its right subtree, and then connect a to b's right subtree and b to a's right subtree.

The reason for maintaining an array $S[i]$ is that this information will be used later in the connectivity query algorithm, where a tree with the smallest size will

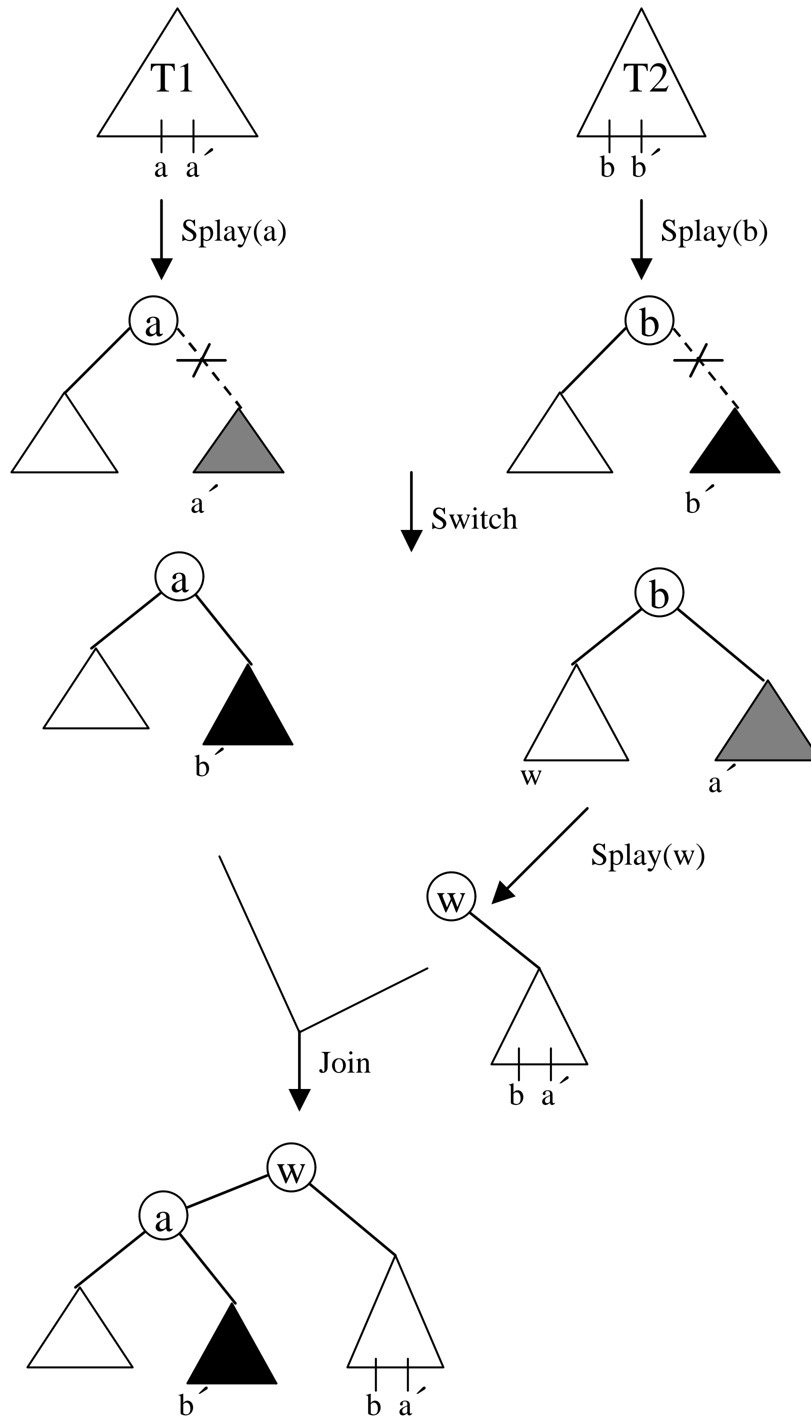

Figure 3.2: Case2: Switch(a,b) in ASF, where nodes a and b are from two different G-cycles.

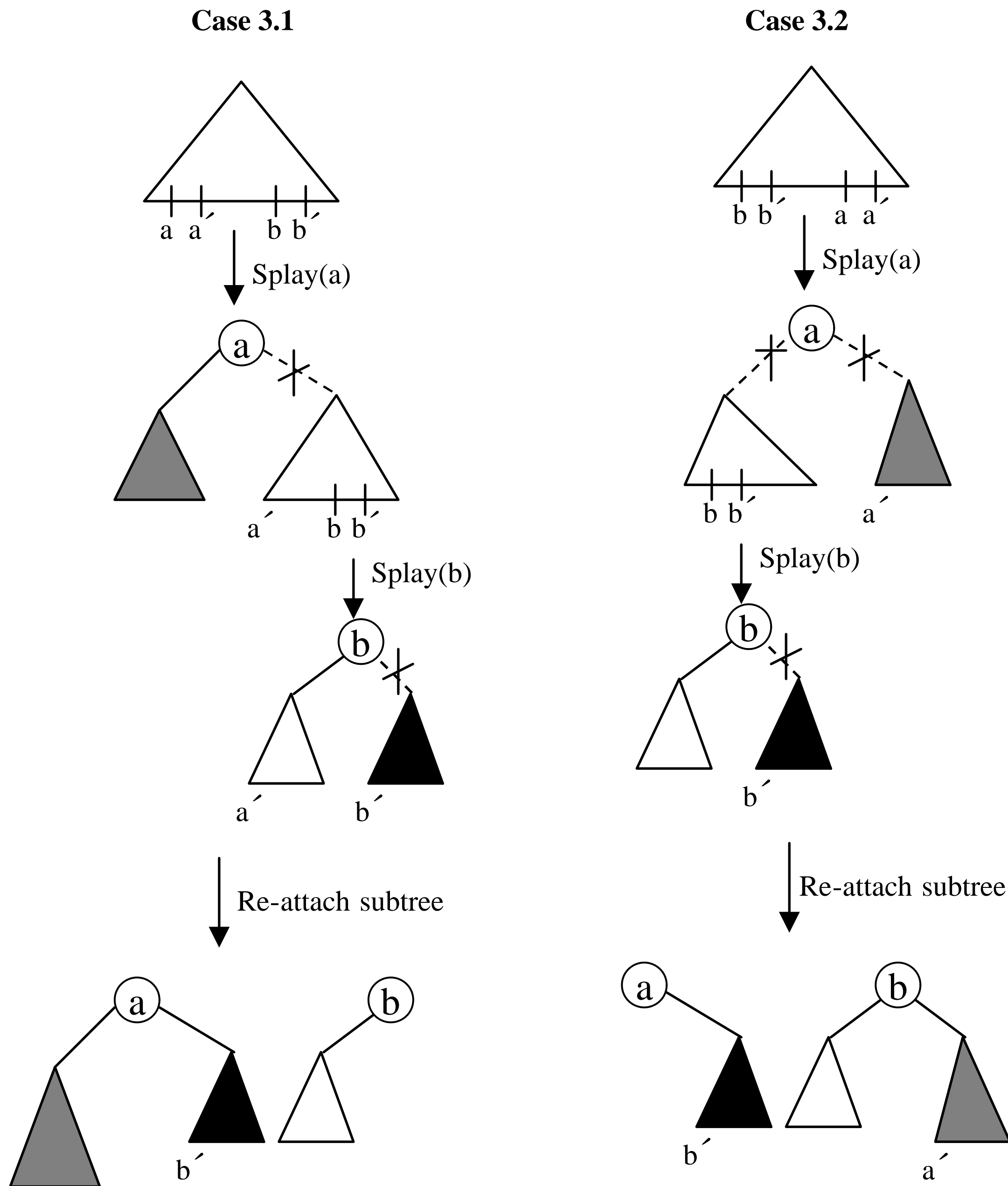

Figure 3.3: Case 3: Switch (a,b) in ASF, where nodes a and b are from the same G-cycle.

be selected to start the connectivity checking (detailed algorithms are described in section 2.3). As we can see, for each splay operation, $S[i]$ is updated for at most three nodes, which is $O(1)$ in time complexity.

3.2 Array-based Treap Forest (ATF) update approach

In this approach, all the arrays used in ASF maintained. In addition, one more array, $Prio[i]$, is created to represent the priority information for each node i. $Prio[i]$ is another random permutaiton of $\{0, ..., 2n-1\}$.

To explain the algorithm more clearly, we again use the Feynman diagram shown in Figure 1.1 as an example.

Step 1: Randomly gnerate a Feynman diagram

Node i:	0	1	2	3	4	5
$Wout[i]$:	2	0	5	4	1	3
$Prio[i]$:	3	2	4	0	5	1

Step 2: Insert nodes into a treap forest

As for the ASF approach, the sequence of insertions is based on $Wout[i]$. Starting from node 0 as the root of the tree, each node is inserted as a right child of the rightmost node of the treap. If the priority of this node is higher than its parent node, rotate this node with its parent node. The value of $P[i]$, $LC[i]$, $RC[i]$ and $S[i]$ are also updated corresponding to each insertion and rotation. Each step of the insertion and the final treap is shown in Figure 3.4. We can see that the sequence of an in-order traversal of this treap is same as the cyclic order of the G-cycle. The arrays $P[i]$, $LC[i]$,$RC[i]$ and $S[i]$ have following values.

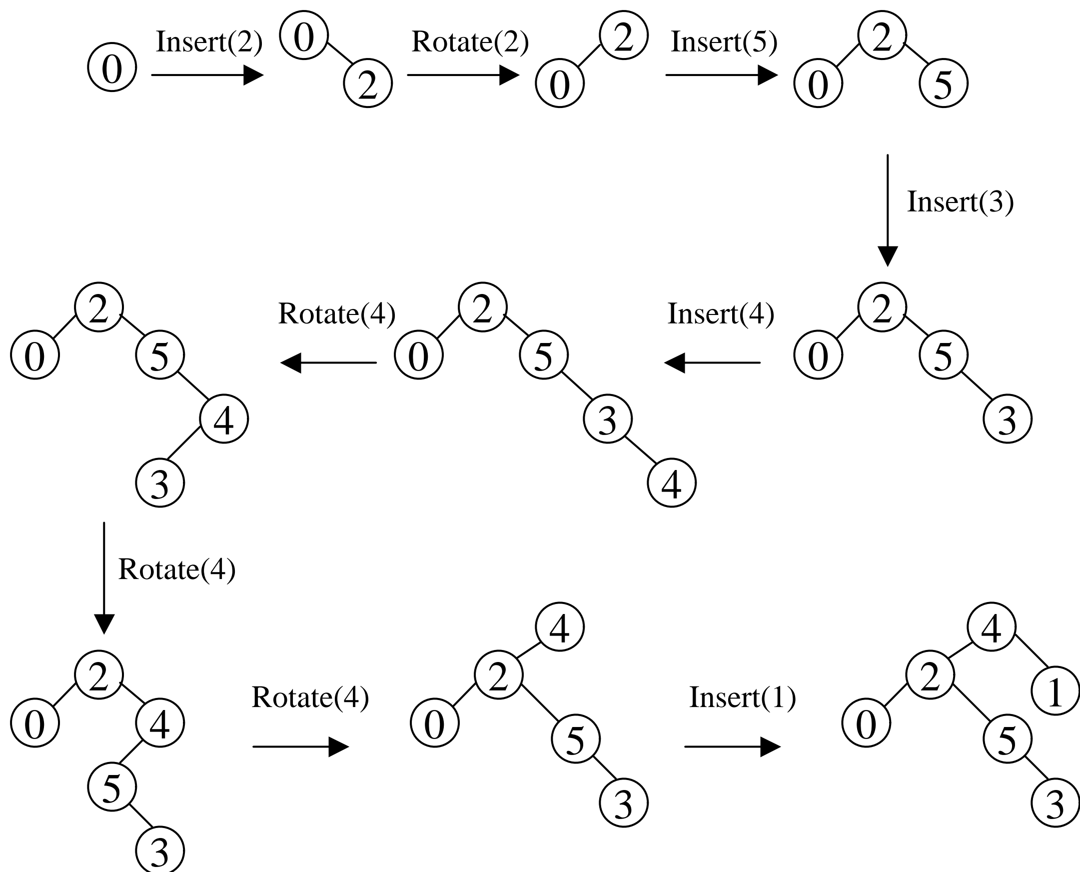

Figure 3.4: Illustration of each step of an insertion sequence and the resulting treap.

$P[i]$:	2	4	4	5	4	2
$LC[i]$:	#	#	0	#	2	#
$RC[i]$:	#	#	5	#	1	3
$S[i]$:	1	1	4	1	6	2

The notations used above is same as for the ASF approach.

Step 3: Making switches

Switch(a,b)

When switching nodes a and b the same three cases 1, 2, and 3 are considered as for the ASF approach.

In case 2, no connectivity query is necessary. As shown in Figure 3.5, this case involves a procedure to *split* and then *join* two treaps as described in chapter 2.

In case 3, there are following two subcases to be considered based on the sequence of in-order traversal in the treap, as follows.

- Case 3.1: Shown in Figure 3.3 (node a precedes b).

  Insert a special node w with highest priority into this treap (w is the right child of a). the new node w becomes the root of this treap after a series of rotations. Cut the edge between w and its left and right subtrees, creating two separate treaps $T1$ and $T2$. Repeat the same operations on $T2$ with node b; $T2$ is *split* and forms $T3$ and $T4$; *join* $T1$ and $T4$.

- Case 3.2: Shown in Figure 3.6 (node b precedes a). Insert a special node w with highest priority into treap (w is the right child of a). w becomes the root of this treap after a series of rotation. Two separate treaps $T1$ and $T2$ are created by cutting the edge between w and its left and right subtrees. Repeat the same operation on the $T1$ with node b. $T1$ is *splited* and forms $T3$ and $T4$; *join* $T3$ and $T2$.

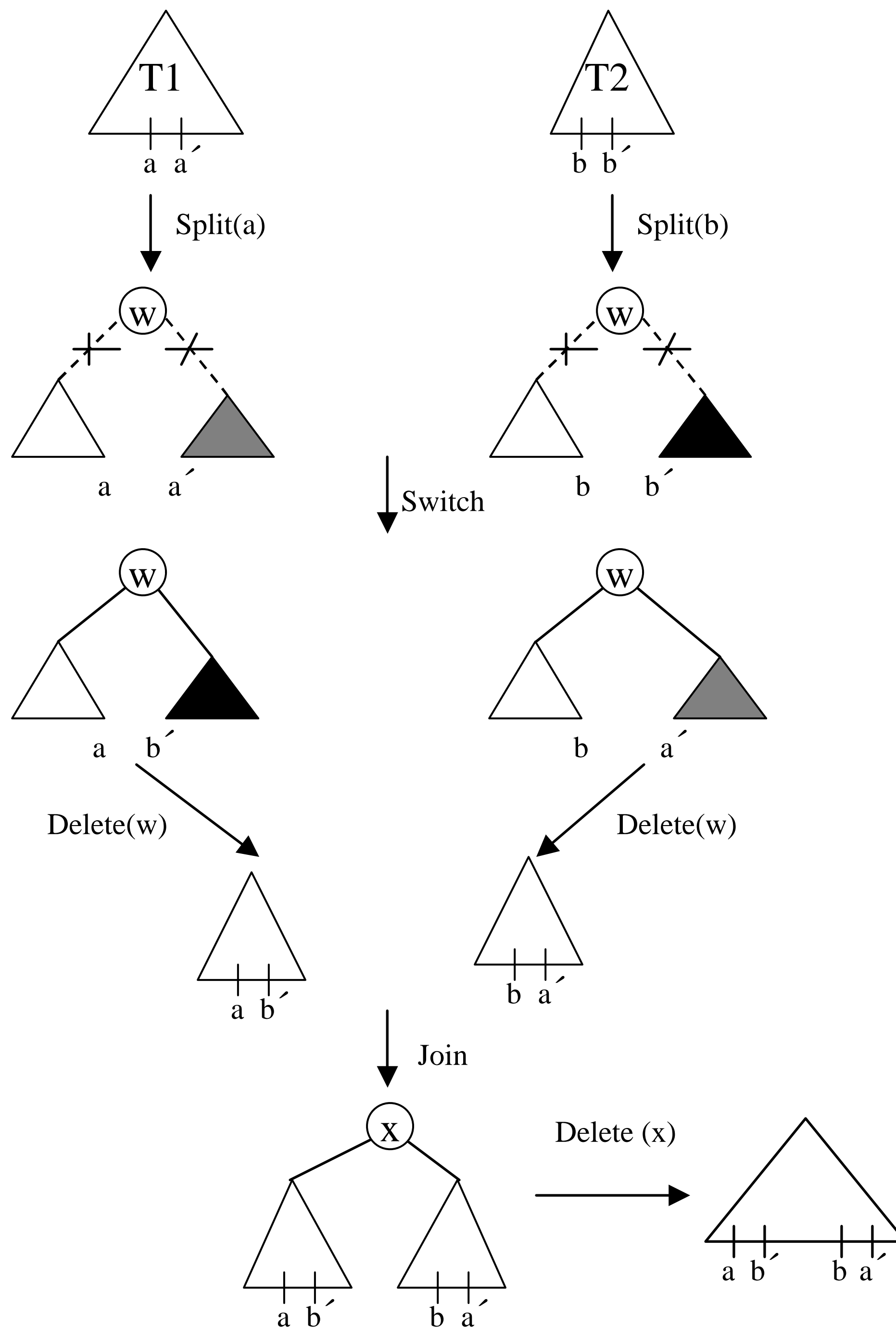

Figure 3.5: Case2: Switch(a,b) in ATF, where nodes a and b are from two different G-cycles.

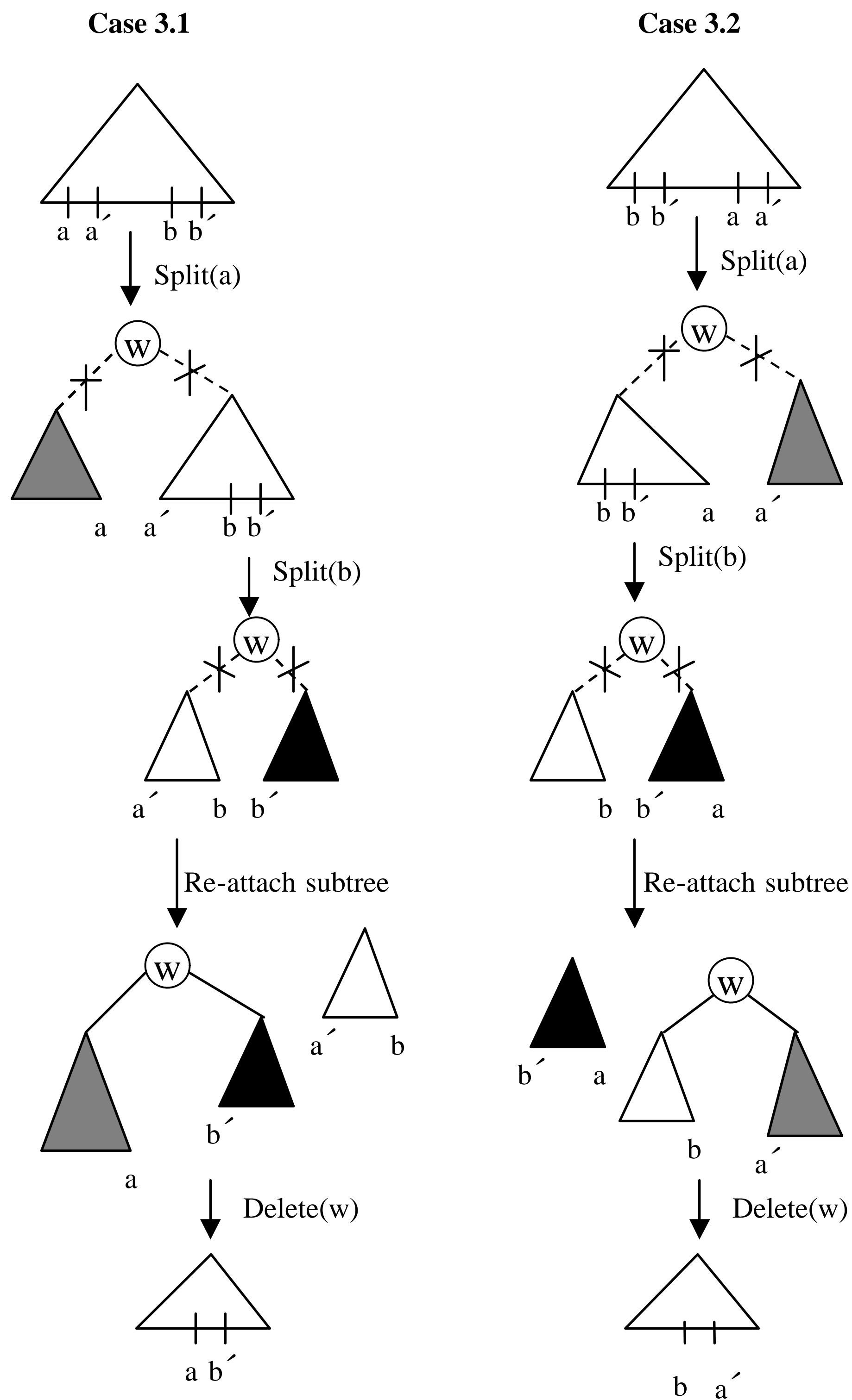

Figure 3.6: Case 3: Switch (a,b) in ATF, where nodes a and b are from the same G-cycle.

3.3 Implementation of SCQ

During the update of G-cycles, a list of roots for the different cycles is maintained. Since the expected number of G-cycles is $O(\log n)$, the expected time complexity for each operation on this list is also $O(\log n)$. SCQ uses the information stored in this list for connectivity checking. A stack is declared for operation on discovered G-cycles. The detailed algorithm is as follows:

```
bool SCQ()
     mark root of first G-cycle r1 as discovered;
     discover = 1;
     push r1 into a stack S;
     while (S not empty){
         topitem = S.top();
         call VConn(topitem)
         if a new G-cycle with root r is found connected
           discover ++ ;
           if(discover = number of G-cycles) return true
           mark r as discovered;
           S.push(r);
         else
           S.pop();
     }
     return false
```

3.4 Implementation of ACQ

As in SCQ, ACQ also uses the information stored in the list of roots for all G-cycles. In order to facilitate the operations for joining and searching the G-cycles, each item in the list is “spawned” and creates a single-item list. For example, if there are m items in the list of roots, then m lists are created with each item as a single-item list. In this way, we can easily add and remove items in each list, as well as join two different lists. The total number of items has an expected value of $O(\log n)$, and so is

the size of each spawned list. The detailed algorithm for ACQ is as follows (assume there are m G-cycles in a Feynman diagram)

```
bool ACQ
  while (total list join < m-1){
    Scan m lists and find list l with smallest size*
        traverse each root r in list l
        call VConn(r)
        if(another root (vr) in different
           list is found connected  via V-line)
             join list where vr is in to list where r is in
             make list where vr is in empty
        else
           return false
  }
  return true

* Here, size means the total number of forest nodes
```

3.5 Implementation of IDQ

In order to maintain set C as described in Section 2.3. an adjacency list is created to represent the G-cycle graph (Figure 3.7). Basically it is a linked list with two different types of nodes. The list of type A nodes contain a G-cycle list. Each type A node also points to a list of type B nodes which represent a adjacency list. Detailed structure is described as follows:

Type A node:

1. node: root vertex of a G-cycle
2. visited: 0 means not visited yet, 1 means visited
3. down: pointer to its adjacency list
4. next: pointer to next type A node

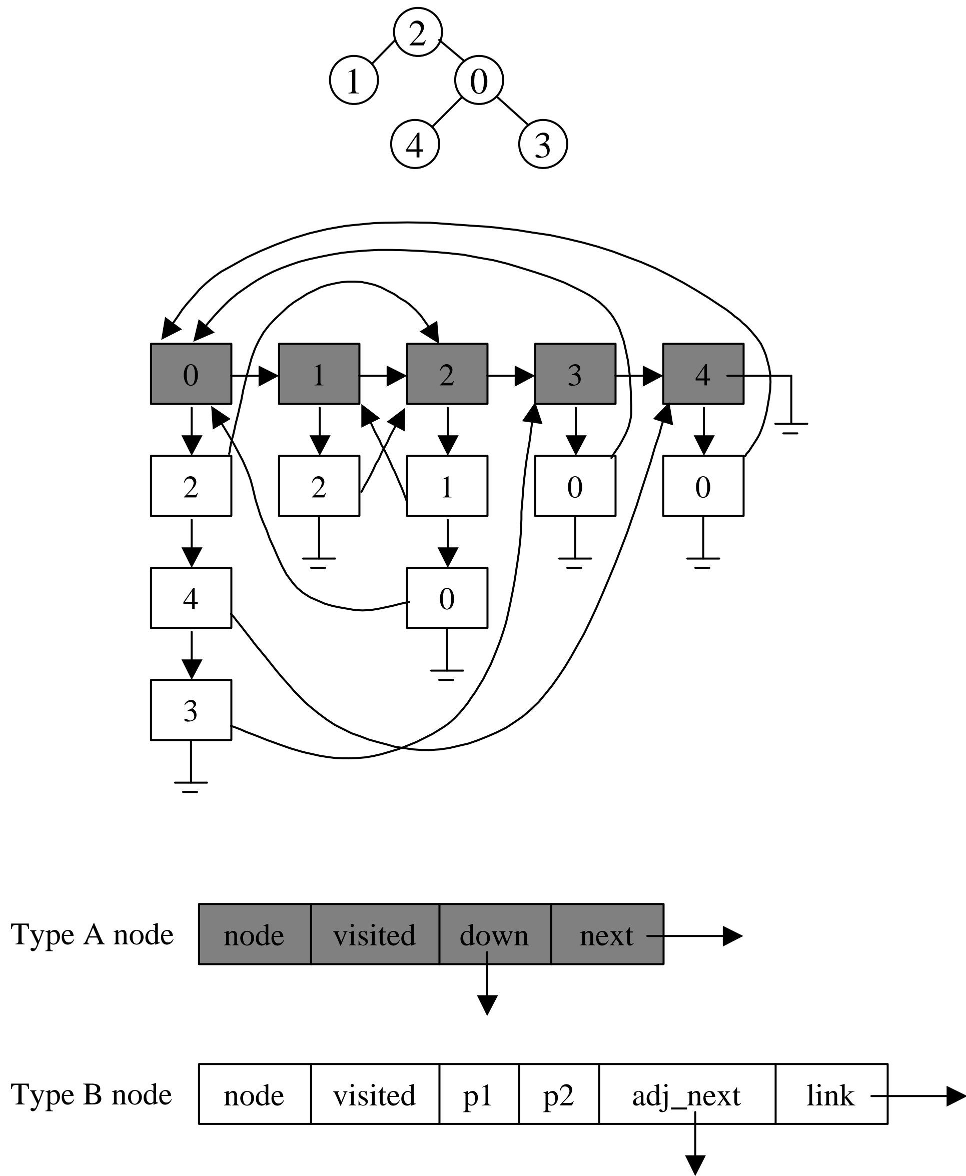

Figure 3.7: Adjacency list representation of a G-cycle graph

Type B node:

1. node: root vertex of G-cycle
2. visited: 0 means not visited yet, 1 means visited
3. $p1$: the start vertex of a V-edge connecting two G-cycles
4. $p2$: the end vertex of a V-edge connecting two G-cycles
5. adj_next: pointer to next type B node
6. link: pointer to the type A node with same root vertex number

When a switching is made, there are 3 cases to be considered.

- Case 1: $a = b$. Do nothing.
- Case 2: $a \neq b$, nodes a and b lie in two different G-cycles. Two type A nodes are merged into one. Change the adjacency lists accordingly.
- Case 3: $a \neq b$, nodes a and b lie in the same G-cycle. Split the corresponding type A node into two. Change the adjacency list accordingly.

Case 2 is rather simple because we know the diagram will remain connected. In Case 3, If a type A node, say a, is split into $a1$ and $a2$, the edge used to connect to node a needs now to connect to $a1$ or $a2$. The $p1$ and $p2$ information stored in type B nodes is used to make this decision. After updating of adjacency list, a DFS is made and the G-cycle graph will probably now disconnected (if still connected, no further steps necessary). Now we have two components in the G-cycle graph, say $D1$ and $D2$. The DFS explored one of these fully, so we know how many original vertices are in one of these (adding up the sizes of the G-cycles in it), say n_1 in $D1$. Thus there are $n_2 = 2 \times n - n_1$ in $D2$, where n is the order of the diagram. So if $n_1 \leq n_2$,

we search $D1$ for V-edges joining it to $D2$ (and vice versa if $n_1 > n_2$). The expected time to find such an edge is $O(1)$ assuming a random Feynman diagram. As soon as we find such an edge, add it to C and conclude that the diagram is connected. If the diagram is not connected, switch back. The time to know it is not connected will be $O(n)$, but on average this only occurs with probability of $O(1/n)$ [1]. So these cases only contribute $O(1)$ to the average time taken per switching.

Notice that in type B nodes, a pointer called "link" is used to avoid searching for the corresponding type A node. In this way, the DFS only takes time linear in the number of G-cycles.

In the SCQ, ACQ and IDQ algorithms, a subroutine VConn( ) has been called to check if there is a new G-cycle connected via a V-line. The basic idea in this subroutine is an in-order traversal of a binary tree. The detailed algorithm is:

```
int VConn(r)
    if(r is an odd number)
         Find the root of the G-cycle containing r-1
    else
         Find the root of the G-cycle containing r+1
    If the found G-cycle is newly discovered
       return the root of that G-cycle
    else
       VConn(r.leftchild)
       VConn(r.rightchild).
```

Chapter 4

Experimental results

Five experiments have been carried out to ensure the correctness of implementation and to compare the time complexities of different algorithms.

4.1 Comparison of time complexities for ATF and ASF

In this experiment, we applied ATF and ASF to the same diagrams. The orders of the diagrams increase from 10 to 100,000. In both approaches ACQ was used for connectivity checking. For each diagram, a total of 5,000 switches were made to insure that the results were statistically significant. In each switch(a,b) operation,a and b were randomly generated.

Figure 4.1 shows a poly logarithmic relationship between the order of the diagram and the switching times for both ATF and ASF. There is a consistent trend for the ASF implementation to take less time. This result indicates that when number of switches is large enough, the amortized expected $O(\log^2 n)$ time for ASF is less than the expected $O(\log^2 n)$ time for ATF. The difference looks to be statistically significant though not large enough to be a major consideration in choosing between the two.

4.2 Comparison of time complexities for update and query

It is interesting to see the percentage of time used to update a diagram and to query the connectivity of a diagram during switching. This information is critical

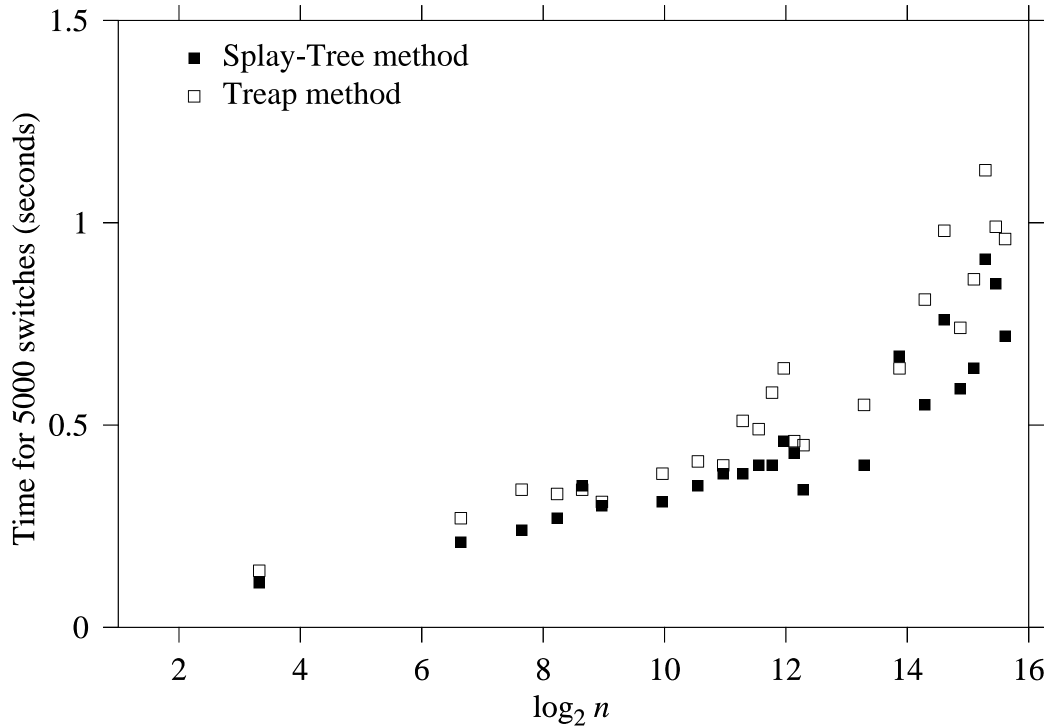

Figure 4.1: Comparison of time costs for ASF and ATF.

for improving our algorithms. For example, Figure 4.2 shows that for both ATF and ASF with ACQ, queries take longer than updates.

Again, these results agree very well with the time complexity analysis. They suggest that improving the query algorithm would affect time efficiency more significantly than improving the update algorithm. It can also be seen that the proportion of time taken by queries decreases as the diagram order increases (Figure 4.3). Two types of explanation for this trend can be hypothesized: 1) faster than expected increase in update times; or 2) slower than expected increase in query times.

It is not likely the first hypothesis will hold because for ASF the amortized time is $O(\log n)$. Increasing the diagram order will asymptotically decrease the update time per switching relative to the $O(\log^2 n)$ query time. For ATF, the increase of diagram

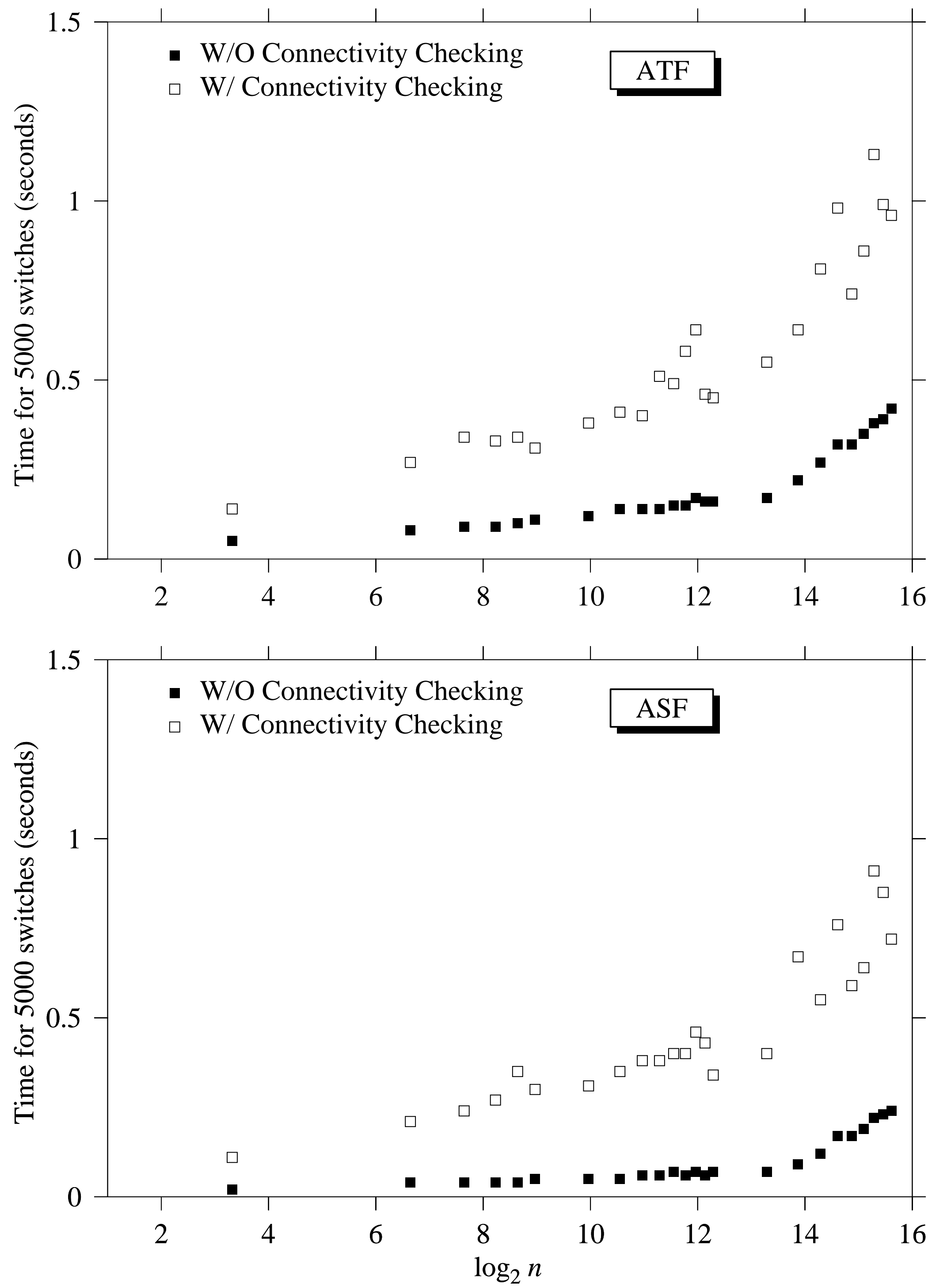

Figure 4.2: Comparison of update and query times for ASF and ATF.

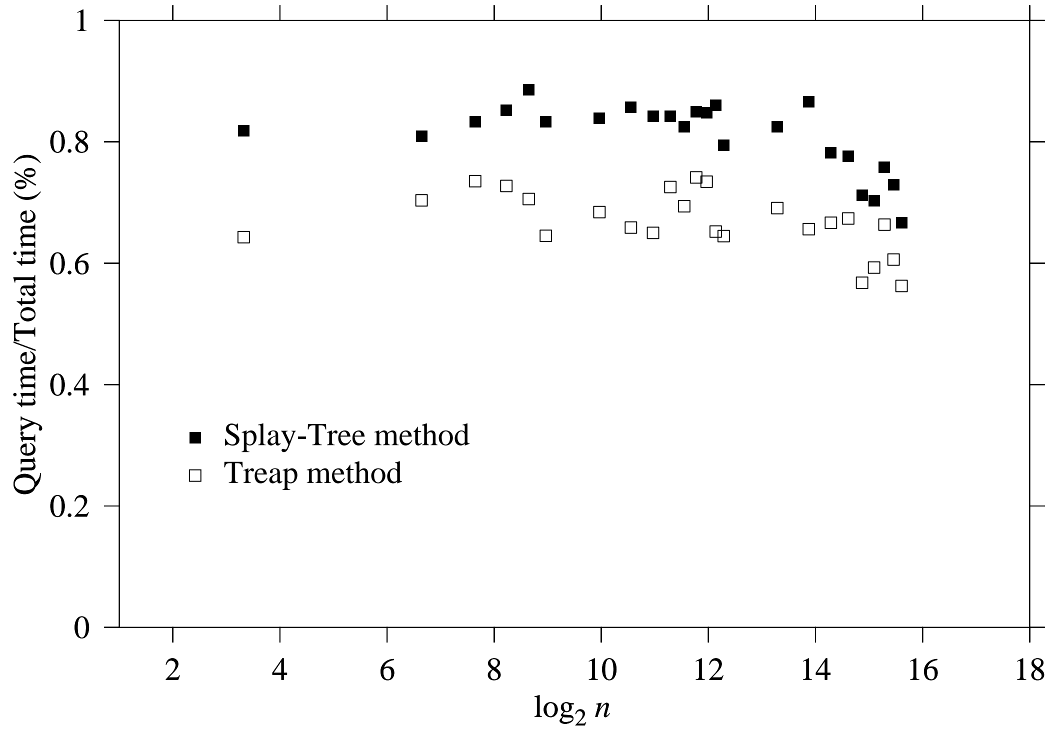

Figure 4.3: Percentage of update and query times for ASF and ATF.

order on the expected time per switching should be similar. See the detailed results in Table 4.1. The second hypothesis might hold if the probability of occurrence of a disconnected diagram decreases significantly. Therefore, the query algorithm would tend to return a positive result quickly. This is tested by our next experiment. However, as we can see in Figure 4.4, there is almost no occurrence of a disconnected diagram if the order is greater than 50.

To improve the connectivity algorithms, a more sophiscated data structure, such as a heap, could be applied to achieve $O(\log m)$ time for finding the root with smallest size, where m is the number of G-cycles. Then the total (amortized) expected time per operation would become $O((\log\log n)\log n)$ for ACQ.

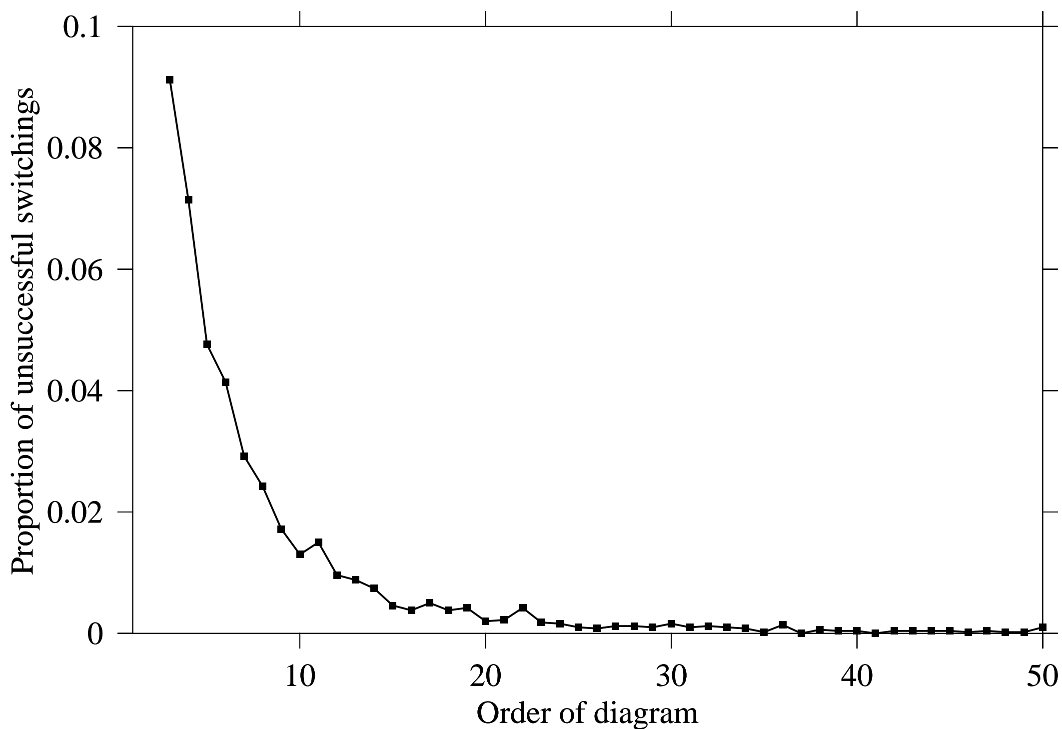

Figure 4.4: Unsuccessful switchings as a function of diagram order.

4.3 The frequency of a disconnected diagram

Figure 4.4 shows that with the increase of diagram size, the occurrence of a disconnected diagram during switching drops exponentially as a function of order. When the order of the diagram increases from 3 to 50, the proportion of unsuccessful switchings (due to the occurrence of a disconnected diagram) decreases from 0.09 to almost 0, indicating that if the diagram order is greater than 50, the diagram is almost always connected no matter how the switches are made.

4.4 Time per switching

A splay tree is a self-adjusting binary tree with an amortized time of $O(\log n)$ per operation. In our implementation, the initialization of the splay tree is done by inserting new nodes as the root of the tree, which means the newly generated tree is linear, a worst case scenario. When the switching starts, it is expected that the first few switches take longer per switching time than the later ones. If more switches are made, then the average switching time should decrease. On the other hand, a treap is initialized differently. After each new node is inserted, the treap rotates based on the priority assigned to the node. Thus we expect to see that there is no significant difference in time per switching between the early and the later switchings for ATF. Table 4.1 shows this pattern clearly. In our experiment, the order of the diagram was 5,000. The number of switches varied from 1 to 9000. It is seen that ASF approach has a much longer time per switching than ATF during the first 1000 switches. However, when more than 2000 switches were made, ASF approach caught up and showed a shorter per switching time, indicating that the splay forest had adjusted itself and had a better performance than the treap forest.

It is important to notice that the order of diagram could be important for the turning point (where ASF become faster than ATF). That is because ASF may take longer to adjust itself in a larger diagram.

4.5 Comparison of SCQ, ACQ and IDQ

Table 4.2 indicates that both IDQ and ACQ show a significant improvement over SCQ. The experiment was carried out using ASF. The order of the tested diagrams ranged from 10 up to 20,000. In SCQ, the switching time is linear in the order of diagram or even worse when n is greater than 10,000. The difference between IDQ and ACQ is smaller. IDQ performs a little better than ACQ. The improvement

Table 4.1: Comparison of time per switching for ASF and ATF.

Switches	Time per switching (milliseconds)	
	ASF	ATF
1	10.00	0.00
2	20.00	0.00
3	16.67	0.00
4	15.00	0.00
5	12.00	0.00
6	10.00	0.00
7	10.00	0.00
8	10.00	0.00
9	8.89	1.11
10	9.00	1.00
20	5.00	0.00
30	4.00	0.33
40	3.00	0.25
50	2.40	0.40
60	2.00	0.17
70	1.71	0.29
80	1.62	0.25
90	1.44	0.22
100	1.30	0.20
200	0.70	0.25
300	0.53	0.23
400	0.43	0.23
500	0.38	0.22
600	0.33	0.23
700	0.30	0.24
800	0.28	0.24
900	0.24	0.24
1000	0.25	0.24
2000	0.18	0.22
3000	0.17	0.20
4000	0.15	0.20
5000	0.14	0.20
6000	0.14	0.20
7000	0.14	0.19
8000	0.13	0.19
9000	0.14	0.19

Table 4.2: Comparison of times for SCQ, ACQ and IDQ.

Order	Time (*seconds*)		
	IDQ	ACQ	SCQ
10	0.08	0.11	0.10
100	0.13	0.21	0.37
200	0.14	0.24	0.81
300	0.15	0.27	1.32
400	0.16	0.35	1.60
500	0.16	0.30	1.88
1000	0.17	0.31	3.31
1500	0.19	0.35	6.28
2000	0.20	0.38	8.21
2500	0.20	0.38	9.39
3000	0.20	0.40	13.47
3500	0.18	0.40	9.35
4000	0.23	0.46	21.58
4500	0.22	0.43	21.54
5000	0.19	0.34	14.84
10000	0.22	0.40	33.36
15000	0.33	0.67	174.07
20000	0.30	0.55	341.06

is not as marked as the difference between the $O(\log n)$ and $O(\log^2 n)$ amortized times from our time complexity analysis (Chapter 2, Section 2.3). There are several possible reasons for this, all of which may be simultaneously involved. First, $\log n$ is very slowly growing as a function of n, so the true asymptotic behavior of IDQ and ACQ may not be reflected in the data. Second, the $O(\log n)$ analysis for IDQ is heuristic, and as explained in Section 2.3 may be overoptimistic. Third, the $O(\log^2 n)$ analysis for ACQ is only an upper bond argument and maybe overpessimistic.

Chapter 5

Discussion and Future work

5.1 Advantages and disadvantages of the splay tree and treap approaches

Some balanced binary search trees, such as red-black trees, offer $O(\log n)$ worst-case time complexity per operation. We chose splay trees because 1) splay trees are conceptually simple and easy to implement; 2) splay trees need less space, since no balance information is stored; 3) splay trees offer $O(\log n)$ amortized time per operation, which is good for a long sequence of operations on Feynman diagrams. The drawbacks of splay trees are that individual operations within a sequence can be expensive, and more local adjustments are required [14].

The treap approach uses randomization to balance the binary search tree. It seems to have advantages over the splay tree. For example, it can achieve expected case bounds that are comparable to amortized bounds in a splay tree without any assumption about the inputs. In our case, the generation of a diagram and the switching operations are both random. We expected improved performance with the treap approach. However, a treap needs to store priority information and keep checking the priority, which may degrade the performance. That probably is the reason that treaps did not show better performance in our experiment, as shown in Figure 4.1. Actually, the time for ATF is even a little higher compared to ASF when the order of the Feynman diagram is greater.

5.2 ASF initialization

The ASF approach shows a longer time per switching for the earlier switches as we saw in Table 4.1. Future investigation might show how to mitigate this effect by initializing the splay tree in a better way. For example, in Figure 5.1, node 4 is inserted as a root of a seperate tree. It is attached to node 3 if either the height of the tree containing node 4 is same as height of the tree containing node 3 or no more nodes are to be inserted. In this way, we can always keep the tree approximately balanced when new nodes are inserted. This should reduce the time for the splay tree to adjust itself and improve the per switching time for the earlier switches.

5.3 Switching from disconnected to connected diagrams

In our switching operations, if a disconnection occurred, the diagram needed to be switched back immediately to maintain connectivity. An alternative worth exploring would be to modify the algorithms to allow for disconnected diagrams, and to keep switching until the diagram becomes connected again.

5.4 Dynamic irreducibility for Feynman diagrams

A Feynman diagram is *irreducible* if and only if it is 3-edge-connected with respect to G-lines. It is hoped that the work of this thesis will provide a basis upon which to develop algorithms for the more difficult problem of dynamic irreducibility for Feynman diagrams. For a general graph, Galil and Italiano [4] developed an algorithm with time complexity $O((n+m)\alpha(m,n))$ for the incremental dynamic 3-edge-connectivity problem, where m is the number of queries, n is the number of vertices, and $\alpha(.,.)$ is the inverse of Ackermann's function. La Poutre [11] obtained similar results for this problem. Using a sparsification technique [2], fully dynamic maintenance of 3-edge connected components can be done in $O(n^{2/3}))$ time per

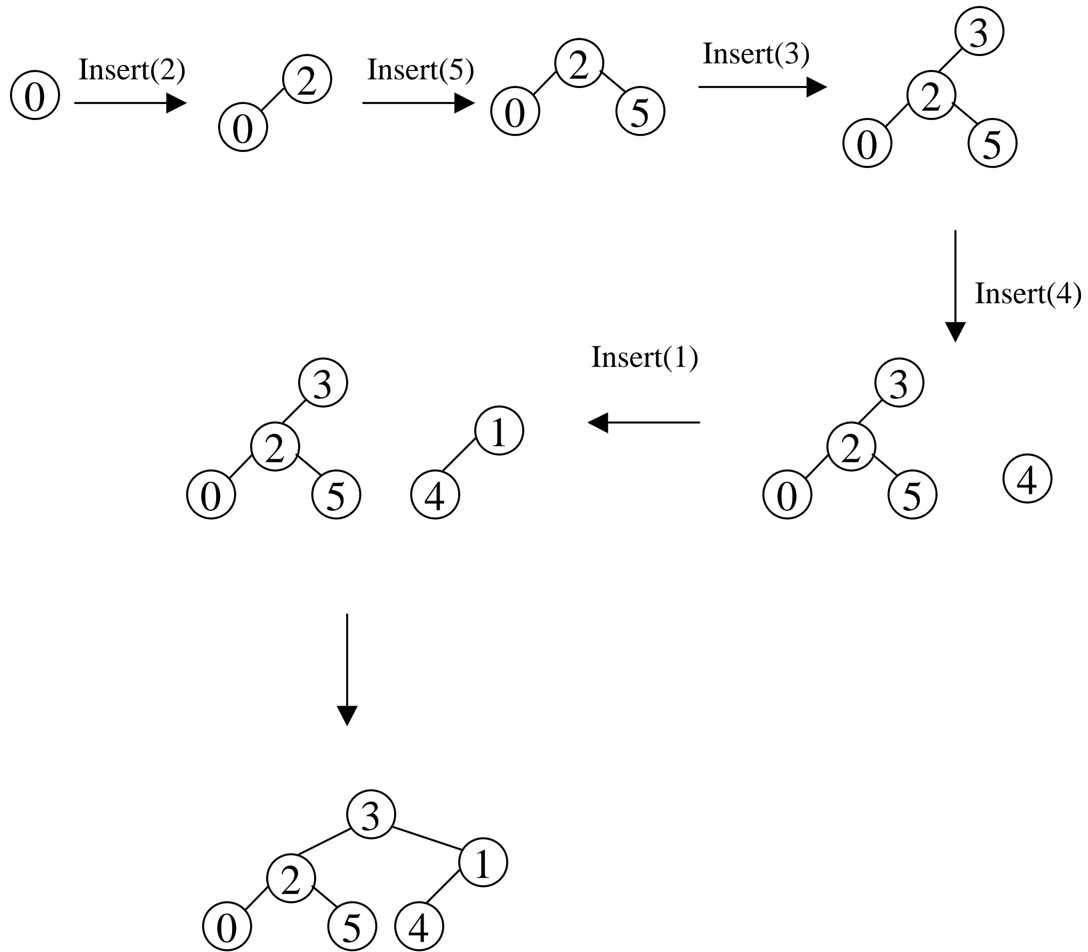

Figure 5.1: A better way of initializing a splay tree.

update and per query. Liang *et al.* [9] devoloped an NC algorithm, which takes $O(\log n \log \frac{m}{n} + \log n \log\log n / \alpha(3n, n))$ time using $O(n\alpha(3n, n)/\log n)$ processors per update and $O(1)$ time with a single processor per query. For Feynman diagrams, a better time complexity should be achieveable due to their special structure and the restricted nature of the query sequences. This problem is a challenging one for future research.

Bibliography

[1] Cvitanovic, P., Lautrup, B., and Pearson, R. B. 1978. Number and weights of Feynman diagrams. *Phys. Rev. D 18*, 1939-1949.

[2] Eppstein, D., Galil, Z., Italiano, G. F., and Nissenzweig, A. 1992. Sparsification - a technique for speeding up dynamic graph algorithms. In *Proc. 33rd Symp. Foundations of Computer Science*. Computer Society Press, pp. 60-69.

[3] Frederickson, G. N. 1985. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput. 14*, 781-798.

[4] Galil, Z., and Italiano, G. F. 1993. Maintaining the 3-edge-connected components of a graph on-line. *SIAM J. Comput. 22*, 11-28.

[5] Henzinger, M. R., and King, V. 1995. Randomized dynamic graph algorithms with logarithmic time per operation. In *Proc. 27th Symp. on Theory of Computing*. pp. 519-527.

[6] Henzinger, M. R., and King, V. 1997. Maintaining minimum spanning trees in dynamic graphs. *International Colloquium on Automata, Languages, and Programming*. pp. 594-604.

[7] Henzinger, M. R., and Thorup, M. 1997. Sampling to provide or to bound: With applications to fully dynamic graph algorithms. *Random Structures and Algorithms 11*, 369-379.

[8] Holm, J., De Lightenberg, K., and Thorup, M. 1998. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge and biconnectivity. In *Proc. 30th Symp. on Theory of Computing.* pp. 79-89.

[9] Liang, W., Brent, R. P., and Shen, H. 2001. Fully Dynamic Maintenance of K-Connectivity in Parallel. *IEEE Transactions on Parallel and Distributed Systems, 12*, 846-864.

[10] Nakanishi, N. 1971. Graph Theory and Feynman Integrals. Gordon and Breach Science Publishers, New York, NY.

[11] Poutre, H. L. 2000. Maintenance of 2- and 3-edge-connected components of Graphs II. *SIAM J. Comput. 29*, 1521-1549.

[12] Robinson, R. W. 1999. Counting irreducible Feynman diagrams, *CATS seminar*. Department of Computer Science, University of Georgia.

[13] Seidel, R. G., and Aragon, C. R. 1996. Randomized search trees. *Algorithmica 16*, 464-497.

[14] Sleator, D. M., and Tarjan, R. E. 1985. Self-adjusting binary search trees. *J. ACM 12*, 652-686.

[15] Thorup, M. 2000. Near-optimal fully-dynamic graph connectivity. In *Proc. 32nd ACM Symp. Theory Computing*, pp. 343-350.