12-2010

Dynamic distributed programming and applications to swap edge problem

Feven Z. Andemeskel
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/thesesdissertations

Part of the Computer and Systems Architecture Commons, Digital Communications and Networking Commons, and the OS and Networks Commons

Repository Citation
Andemeskel, Feven Z., "Dynamic distributed programming and applications to swap edge problem" (2010). UNLV Theses, Dissertations, Professional Papers, and Capstones. 677.
https://digitalscholarship.unlv.edu/thesesdissertations/677

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Theses, Dissertations, Professional Papers, and Capstones by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
DYNAMIC DISTRIBUTED PROGRAMMING AND APPLICATIONS TO
ALL BEST SWAP EDGES PROBLEM

by

Feven Z. Andemeskel

Bachelor of Science in Computer Science
University of Asmara, Eritrea
July 2006

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
December 2010
THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Feven Z. Andemeskel

entitled

Dynamic Distributed Programming and Applications to Swap Edge Problem

be accepted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
School of Computer Science

Ajoy K. Datta, Committee Chair
Lawrence L. Larmore, Committee Member
Yoohwan Kim, Committee Member
Emma E. Regentova, Graduate Faculty Representative

Ronald Smith, Ph. D., Vice President for Research and Graduate Studies
and Dean of the Graduate College

December 2010
ABSTRACT

Dynamic Distributed Programming and Applications to All Best Swap Edges Problem

by
Feven Z. Andemeskel

Dr. Ajoy K Datta, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

Link failure is a common reason for disruption in communication networks. If communication between processes of a weighted distributed network is maintained by a spanning tree $T$, and if one edge $e$ of $T$ fails, communication can be restored by finding a new spanning tree, $T'$. If the network is 2-edge connected, $T'$ can always be constructed by replacing $e$ by a single edge, $e'$, of the network. We refer to $e'$ as a swap edge of $e$.

The best swap edge problem is to find the best choice of $e'$, that is, that $e$ which causes the new spanning tree $T'$ to have the least cost, where cost is measured in a way that is determined by the application. Two examples of such measures are total weight of $T'$ and diameter of $T'$.

The all best swap edges problem is the problem of determining, in advance of any failure, the best swap edge for every edge in $T$. The justification for this problem is that we wish to be ready, when a failure occurs, to quickly activate a replacement for the failed edge.

In this thesis, we give algorithms for the all best swap edges problem for six different cost measures. We first present an algorithm which can be adapted to all six measures, and which takes $O(d^2)$ time, where $d$ is
the diameter of $T$. This algorithm is essentially a form of distributed
dynamic programming, since we compute the answers to sub problems
at each node of $T$.

We then present a novel paradigm for speeding up distributed
computations under certain conditions. We apply this paradigm to find
$O(d)$-time distributed algorithms for the all best swap edge problem for all
but one of our cost measures.

Formal algorithms and their correctness proofs will be given.
ACKNOWLEDGEMENTS

I would like to start off by thanking my advisor and mentor, Dr. Ajoy K Datta, whose encouragement, guidance, and support have been invaluable from the initial to the final level of this thesis work. I would like to express my greatest gratitude to him for his patience, motivation, enthusiasm, and sincerity throughout my graduate program.

My sincere thanks also goes to Dr. Lawrence L. Larmore for his remarkable guidance and help in the technical work of my thesis. I would like to thank members of my committee, Dr. Yoohwan Kim and Dr. Emma Regentova.

I would like to thank the School of Computer Science for their financial support through a Graduate Assistantship.

I would also like to thank my loving family, my parents Zemuy Andemeskel and Alganesh Ande, and my husband Billen Habte for standing by me.

Last but not least, I thank God who has made all this possible.
TABLE OF CONTENTS

ABSTRACT ............................................................................................... iii

ACKNOWLEDGEMENTS ...................................................................... iii

LIST OF TABLES ............................................................................. vii

LIST OF FIGURES .......................................................................... viii

CHAPTER 1 INTRODUCTION ............................................................. 1
  1.1 Our Contributions ................................................................. 1
  1.2 Outline of the Thesis ......................................................... 2
  1.3 Preliminaries ........................................................................... 3

CHAPTER 2 DISTRIBUTED SYSTEM AND NETWORKS .............. 5
  2.1 Distributed Systems ............................................................. 5
  2.2 Spanning Trees ................................................................. 6
    2.2.1. Minimum Spanning Tree .............................................. 6
    2.2.2. Shortest Paths Tree ................................................... 7
    2.2.3. Minimum Diameter Tree .......................................... 8
  2.3 SWAPPING ALGORITHMS .................................................... 9
    2.3.1. MST Node Replacement Problem .............................. 10
    2.3.2. Minimum Diameter Spanning Tree Swap Edge Problem . 11
  2.4 All Best Swap Edges Problem .............................................. 12

CHAPTER 3 QUADRATIC TIME SWAP EDGE ALGORITHM .......... 17
  3.1 The Algorithm BSE ............................................................. 17
  3.2 General Overview of BSE ....................................................... 17
  3.3 $BSE_{dist}$ and $BSE_{incr}$ ...................................................... 25
  3.4 $BSE_{wght}$ .............................................................................. 27
  3.5 $BSE_{max}$ .............................................................................. 28
  3.6 $BSE_{diam}$ .............................................................................. 32
    3.6.1. Preprocessing Phase of $BSE_{diam}$ ............................. 36
    3.6.2. Optimization Phase of $BSE_{diam}$ ............................. 38
    3.6.3. Computation of $ecc_{T-x} (y')$................................. 39
  3.7 $BSE_{sum}$ .............................................................................. 42
  3.8 Implementation and Complexity of BSE ............................... 44
    3.8.1. Messages ...................................................................... 44
VITA .................................................................................................................. 115
LIST OF TABLES

Table 2.4-1  $F(T,r,e,e')$ for various $F$ ................................................. 16
Table 3.2-1  Optimization Phase of BSE ............................................... 20
Table 3.6-1  Computation of $l_{sol}(y, x)$ in BSE$_{diam}$ .......................... 39
Table 3.8-1  Variables in Messages of BSE .......................................... 46
Table 3.8-2  $l_{sol}(y, x)$ ....................................................................... 54
Table 4.1-1  Values of $\mincost(x)$ and $\best(x)$ ..................................... 61
Table 4.2-1  Quadratic Time Algorithm for the Min-Max Problem ....... 64
Table 4.3-1  Critical Levels .................................................................. 64
Table 4.3-2  Linear Time and Space Algorithm ...................................... 68
Table 4.3-3  Input, output, and some intermediate values ...................... 73
Table 5.2-1  LINEAR ........................................................................ 77
Table 5.4-1  Ranking Phase ................................................................. 79
Table 5.6-1  Computation of $\text{subtree}_{\mincost}(y, l)$ in LINEAR$_{\text{dist}}$ .... 81
Table 6.2-1  Critical Level Phase .......................................................... 90
Table 6.3-1  Computation of $\text{up package}(y, l)$ for LINEAR$_{\max}$ ......... 94
Table 6.6-1  Special Level Phase for $S_1$ ............................................... 101
Table 6.6-2  Special Level Phase for Processes Not in $S_1$ ..................... 102
Table 6.7-1  Optimization Phase of LINEAR$_{\text{diam}}$ ................................. 109
Table 6.7-2  LINEAR$_{\text{diam}}$ ................................................................. 110
# LIST OF FIGURES

| Figure 2.4-1 | Swap Edges | 14 |
| Figure 2.4-2 | Cross Edges | 16 |
| Figure 3.2-1 | Ancestry of processes | 19 |
| Figure 3.2-2 | Cross edges are shown as dashed | 22 |
| Figure 3.2-3 | Computation of `down_package` | 22 |
| Figure 3.2-4 | Children of x compute `down_package` | 22 |
| Figure 3.2-5 | Broadcast wave continues | 22 |
| Figure 3.2-6 | Broadcast wave is completed | 23 |
| Figure 3.2-7 | Convergecast continues | 23 |
| Figure 3.2-8 | Convergecast continues | 23 |
| Figure 3.2-9 | `down_package` and `l_sol` | 23 |
| Figure 3.2-10 | Convergecast wave is completed | 24 |
| Figure 3.2-11 | `solution(x)` is computed | 24 |
| Figure 3.3-1 | `l_sol (u, x)` | 26 |
| Figure 3.5-1 | `F_{max}` | 29 |
| Figure 3.5-2 | An example where `η(x) = 4` | 30 |
| Figure 3.5-3 | An example where `μ(y, x) = 4` | 31 |
| Figure 3.6-1 | `F_{diam}` | 34 |
| Figure 3.6-2 | Swap edge with respect to `F_{diam}` | 34 |
| Figure 3.6-3 | `avoid(x)` | 36 |
| Figure 3.6-4 | If i = j, then `ecc_{T-x}(y') = depth(y') + h_k` | 40 |
| Figure 3.6-5 | `α`, `β`, `γ`, and `δ_k` | 41 |
| Figure 3.7-1 | `F_{sum}` | 42 |
| Figure 4.1-1 | Doubly Weighted Chain | 59 |
| Figure 4.1-2 | `cost(x, y)` | 60 |
| Figure 4.1-3 | `best(x)` and `cost(x, y)` | 60 |
| Figure 4.1-4 | Array W and cost | 61 |
| Figure 4.1-5 | Doubly Weighted Tree | 62 |
| Figure 4.3-1 | `cost` matrix | 65 |
| Figure 4.3-2 | Comparison of the algorithms | 71 |
| Figure 4.3-3 | `mincost(x)` and `best(x)` | 74 |
| Figure 5.4-1 | Levels of processes and ranks of cross edges | 78 |
| Figure 6.2-1 | Example of `T_x`, where `tail (x)` is defined | 87 |
| Figure 6.2-2 | Example of `T_x`, where `tail (x)` is undefined | 87 |
| Figure 6.2-3 | Illustration of Lemma 6.2 | 90 |
| Figure 6.3-1 | `swap_edge_cost`, `min_up_cost` and `min_fan_cost` | 93 |
| Figure 6.5-1 | `restricted`, `local` and `full eccentricity` | 100 |
| Figure 6.6-1 | Computation of Special Levels | 103 |
CHAPTER 1

INTRODUCTION

This thesis considers the concept of *Swap Edges*. This concept has been around for quite some time and is becoming increasingly popular. Link failures leading to disconnection of the backbone tree in a network are quite common. This becomes a serious issue especially in networks where construction of the backbone tree is expensive. Edge swapping provides a relatively less expensive way of maintaining communication through the backbone in the event of such failures. We will first consider the *all best swap edges problem* [5], then give a less expensive algorithm for the same problem. We will consider six measures, $F_{\text{dist}}$, $F_{\text{incr}}$, $F_{\text{wght}}$, $F_{\text{max}}$, $F_{\text{sum}}$, and $F_{\text{diam}}$. Detailed explanation of these measures will be given in later sections.

1.1 Our Contributions

In this thesis, we give an algorithm for the *all best swap edges* problem which takes $O(h^2)$ time, where $h$ is the unweighted height of $T$, i.e., the greatest hop-distance from $r$ to any leaf of $T$, and uses $O(\delta_x)$ space for each process $x$, where $\delta_x$ is the degree of $x$. This algorithm can be used for any one of the six measures mentioned above as an input parameter.

We then give faster algorithms for all but one of the six measures, namely all except $F_{\text{sum}}$. Each of these algorithms takes $O(h)$ time, and
still uses only $O(\delta)$ space per process.

1.2 Outline of the Thesis

We will start off this paper by giving a brief introduction to Distributed Systems and Spanning Trees. We will mention some of the common Spanning trees in the second section of CHAPTER 2. In the third section of that chapter we will introduce the concept of Swap Edges. We will give some examples of Swap edge algorithms that have been developed.

In CHAPTER 3, we present our version of an algorithm presented in [8] which we call BSE, which solves the all best swap edge problem for each of the above measures, differing only in detail for the different measures. BSE requires space for $O(\delta_x)$ variables to be stored at each process $x$, where $\delta_x$ is the degree of $x$. The time complexity of BSE is $O(h^2)$, where $h$ is the number of layers of $T$, namely the largest hop-distance from $r$ to a leaf of $T$. In separate sections of CHAPTER 3, we describe the details of each of the six versions of BSE, and we summarize those details Section 3.8.

In CHAPTER 4, we introduce a new technique, called the critical level paradigm, and in Chapter 5, we present faster algorithms for the all swap edges problem for some of the measures, i.e., all except $F_{\text{sum}}$, using the critical level paradigm. In each case, the space complexity of our algorithm is $O(\delta_x)$ for each $x$, and the time complexity is $O(h)$. 

2
1.3 Preliminaries

**Output.** A solution to the best swap edge problem, given any measure $F$ and any tree edge $e$, is an ordered pair $(F(T, e, e'), e')$, where $e'$ is a swap edge for $e$. We order these pairs lexically, so that $e'$ is a tie-breaker in the case that there are equally good swap edges for $e$. A solution to the all best swap edges problem then consists of such a solution $(F(T, e, e'), e')$ for every tree edge $e$. (By an abuse of notation, we may also refer to just $F(T, e, e')$ as the solution.)

We can encode $e'$ in any convenient manner. For example, if $e' = \{z, z'\}$, and if processes have unique IDs, we could encode $e'$ as $(\text{id } (z), \text{id } (z'))$. We could also encode $e'$ as $(\text{index } (z), \text{index } (z'))$, where index $(z)$ is the ordered pair $(\text{pre}_\text{index } (z), \text{post}_\text{index}(z))$ of integers defined in Section 3.2, whose definition depends only on the topology of $T$ as an ordered tree and the position of $z$ in that tree; $\text{pre}_\text{index } (z)$ is the index of $z$ in the preorder visitation of $T$, while $\text{post}_\text{index } (z)$ is the index of $z$ in the “mirror preorder” visitation obtained by reversing left and right. We suggest that the latter encoding is better; if that pair is stored at each end of $e$, the indices aid in navigation through $T$, enabling efficient communication with the processes at the ends of $e'$, as we explain in Section 3.2.

**Model of Computation.** We use the message passing model of computation. A process $x$ can send messages to any neighbor $y$, and can also receive messages from $y$, i.e., there are two channels, one in each
direction, between any pair of neighbors. No message is lost, and any message sent reaches its destination within one time unit. The FIFO rule holds for each channel.

We assume that if $x$ receives a message from any neighbor, it reads that message instantly. We also assume that if $x$ is enabled to change its variables or send a message to a neighbor, it will do so instantly.

We define the size of a message to be the number of items (IDs, numbers, or weights) it contains. We define the space complexity of each process $x$ to be the maximum number of items that $x$ holds at any one time. In the algorithms we present in this thesis, all messages will have size $O(1)$, and we will show that the space complexity of any process $x$ is $O(\delta_x)$, where $\delta_x$ is the degree, i.e., number of neighbors, of $x$. Our algorithms will also have the property that no channel holds more than one message at any given time.
CHAPTER 2

DISTRIBUTED SYSTEM AND NETWORKS

In this chapter, we will give a broad idea of Distributed Systems and Spanning Trees. In the first section, we will give a brief description of distributed systems. In the second section, we will discuss some of the very common spanning trees widely used in distributed systems.

2.1 Distributed Systems

A distributed system is a collection of individual computing devices that can communicate with each other. It encompasses a wide range of computer systems, ranging from a VLSI chip, to a tightly-coupled shared memory multiprocessor, to a local-area cluster of workstation, to the Internet [1]. The motivation for using a distributed system may include inherently distributed computations, resource sharing, access to geographically remote data and resources, enhanced reliability, increased performance/cost ratio, and scalability. Each computer has a memory-processing unit, and the computers are connected by a communication network. These processors need to communicate with each other in order to achieve some level of coordination to complete a task. There are two types of communication among these processors; Message Passing and Shared Memory. Shared memory systems are those in which there is a shared address space throughout the system. Communication among processors takes place via shared data variables and control
variables. In Message passing systems, the processors communicate by sending and receiving messages through the links in the network.

2.2 Spanning Trees

A spanning tree for a network is a subgraph of the graph representing the network that is a tree, and contains all the processors of the network. They are used whenever one wants to find a simple, cheap, yet efficient way to connect a set of processors. Spanning trees are very common because they provide a lot of advantages. They create a sparse sub graph that reflects a lot about the original graph. They play an important role in designing efficient routing algorithms. They have also come very handy in solving very popular problems, such as the Steiner tree problem and, the traveling salesperson problem.

2.2.1 Minimum Spanning Tree

A minimum spanning tree (MST) of a weighted graph G is a spanning tree of G whose edges sum to minimum weight. In other words, a minimum spanning tree is a tree formed from a subset of the edges in a given undirected graph, with two properties: (1) it spans the graph, i.e., it includes every vertex in the graph, and (2) it is a minimum, i.e., the total weight of all the edges is as low as possible [10]. Some common properties of the tree include possible multiplicity (there may be more than one MST), uniqueness (if each edge has a distinct weight, then there will only be one unique minimum spanning tree), minimum-cost sub
graph (if the weights are non-negative), cycle property (for any cycle C in
the graph, if the weight of an edge e of C is larger than the weights of
other edges of C, then this edge cannot belong to an MST), cut property
(for any cut C in the graph, if the weight of an edge e of C is smaller than
the weights of other edges of C, then this edge belongs to all MSTs of the
graph.), and minimum-cost edge (if the edge of a graph with the minimum
cost e is unique, then this edge is included in any MST).

The first algorithm for finding a minimum spanning tree was
developed by Czech scientist Otakar Boruvka in 1926. There are now
two algorithms commonly used, Prim's algorithm and Kruskal's
algorithm [10].

MSTs have a wide range of applications, such as Cable TV, Circuit
design, Islands connection, Clustering gene expression data, and
approximations like the traveling salesperson problem.

2.2.2. Shortest Paths Tree

A shortest path tree, in graph theory, is a sub graph of a given
(possibly weighted) graph constructed so that the distance between a
selected root node and all other nodes is minimal. A known problem
with using shortest path tree in network design is cost, reliability, and
bandwidth required at the node. There are two known algorithms for
finding this tree, Dijkstra’s algorithm and Bellman–Ford Algorithm.

The shortest-paths tree problem comes up in practice and arises as a
sub problem in many network optimization algorithms. The shortest
path tree is widely used in IP multicast and in some of the application-level multicast routing algorithms.

2.2.3. Minimum Diameter Tree

The minimum diameter spanning tree (MDST) of G is a spanning tree of minimum diameter among all possible spanning trees. Some of the known algorithms for finding MDST are based on the fact that any shortest-paths tree rooted at a center of an MST is a MDST. Thus this problem can be reduced to finding the absolute center of a graph and constructing a tree rooted at that center.

Many computer communication networks require nodes to broadcast information to other nodes for network control purposes, which is done efficiently by sending messages over a spanning tree of the network. Now optimizing the worst-case message propagation delays over a spanning tree is naturally achieved by reducing the diameter to a minimum, especially in high-speed networks, where the message delay is essentially equal to the propagation delay. The use of a control structure spanning the entire network is a fundamental issue in distributed systems and interconnection networks. Since all distributed total algorithms have a time complexity $O(D)$, where $D$ is the network diameter, having a spanning tree of minimum diameter makes it possible to design a wide variety of time efficient distributed algorithms.
2.3 Swapping Algorithms

Survivability of a communication network denotes the ability of the network to remain operational even if individual network components (such as a link or even a node) fail. In the past few years, several survivability problems have been studied extensively [16]. Sparse Networks are becoming very popular with the arrival of fiber optics providing a large bandwidth. However Sparse Networks are vulnerable to failures. Trees are widely used as the backbone for communication in most networks. However, we have to look out because a single link failure might disconnect the backbone if that failing link happens to be a tree edge. Two different approaches can be followed to solve the problem of a link failure: either rebuilding a new tree from scratch, or using a single non-tree edge (called a swap edge) to replace the failing link and reconnect the network, thus obtaining the so-called swap tree.

In the first case, we are guaranteed to have the most efficient tree for the network, but it is very expensive both in terms of setup costs and of time complexity for computing a new tree. The new constructed tree may also be completely different from the initial one, and therefore, the updating of a large amount of nodes may be necessary. Furthermore, constructing a tree for every possible link failure in the network is very inefficient especially if failing link is supposed to be quickly restored.

In cases where link failures are temporary and can be easily restored, swapping the failing tree edge with another non-tree edge becomes
preferable. This saves us a lot of computation, and makes it also easier
to switch back to the old link as soon as it is restored. In the future
sections, we will see swap algorithms for some of the common trees used
in networks.

Swapping algorithms have been studied in two aspects. One is the
AER (All Edge Replacement) algorithm and the second is ANR (All Node
Replacement) algorithm. In the first case, a swap edge is computed for
every tree edge. The second case deals with pre-computing a new tree
should a node fail. This paper focuses on the AER problem. In the next
few sections, we will see some common swapping algorithms that have
been developed.

2.3.1. MST Node Replacement Problem

Both the ANR and AER problems have been extensively studied in
case of MSTs. In the AER problem, it is easy to see that the failing edge
has to be replaced by a minimum weight non-tree edge forming with the
failing edge a fundamental cycle in G (i.e., a cycle containing just a single
non-tree edge). It was originally addressed by Tarjan [16], under the
guise of the sensitivity analysis of an MST. Later Dixon et al. [5]
proposed an optimal deterministic algorithm and a randomized linear
time algorithm, while Booth and Westbrook [2] devised a linear time
algorithm for the special case in which the graph G is planar. An
improved solution was later developed by Nardelli, Proietti, and
Widmayer [13].
For the ANR problem, Tsin first presented an algorithm to update an MST after a single node deletion [17]. A subsequent parallel solution to ANR is obtained by combining the parallel algorithms presented by Johnson and Metaxas [11]. A more efficient parallel technique has been designed by Das and Loui [4]. The more complex problem of updating a MST with multiple node and edge deletions was also considered by Cheng, Cimet, and Kumar [3].

A more efficient algorithm later appeared which solved the ANR problem where the total amount of data items communicated during the computation (the data complexity) is $O(n^2)$. This was a distributed algorithm with a broadcast and convergecast phases [7].

2.3.2. Minimum Diameter Spanning Tree Swap Edge Problem

Computing all best swaps of a MDST was one of the first swap problems that were studied. In [15], an algorithm for this problem is given which requires $O(n\sqrt{m})$ time and $O(m)$ space, where the given underlying 2-edge-connected communication network $G = (V,E)$ has $n = |V|$ nodes and $m = |E|$ edges. For each of the $n-1$ different tree edges, their algorithm uses somewhat augmented topology trees to select $O(\sqrt{m})$ best swap candidates, then evaluates the quality of each of the $O(\sqrt{m})$ candidate swap edges in $O(1)$ amortized time, and selects the best among them. In order to obtain the $O(1)$ amortized time for computing the diameter of the swap tree associated with a given swap edge, information from a preprocessing phase is used, and then combined with an
inductive computation that uses path compression.

Later In [9], the problem was solved with an algorithm that computes all best swap edges of $T$ in $O(n^*)$ messages of size $O(1)$ each, and $O(D)$ time. If the failing edge $e = (p(x), x)$, each node in $T_x$ considers its own local swap edges for $e$, then in total all swap edges for $e$ are considered in a minimum finding process. This has three phases. In a first preprocessing phase, a root of the MDST is chosen, and various pieces of information are computed for each node. Then, in a top-down phase, each node computes and forwards some “enabling information” for each node in its own subtree. This information is collected and merged in a third bottom-up phase, during which each node obtains its best local swap edge for each edge on its path to the root.

### 2.4 All Best Swap Edges Problem

In this thesis, we consider the *all best swap edges* problem [14]. We are given a 2-edge connected positively weighted network $X$ of processes, together with a spanning tree $T$ of $X$, rooted at a process $r$. We will assume that $T$ is an ordered tree, i.e., the children of any given process have a given left-to-right order (although the choice of that order is arbitrary). Let $w(x, y)$ denote the weight of an edge $\{x, y\}$ of $X$. If $x \neq r$ is a process, we denote the parent of $x$, in the tree $T$, by $p(x)$, the set of children of $x$ by $\text{Chldrn}(x)$, and the subtree of $T$ rooted at $x$ by $T_x$. We also write $W_T(x, y)$ for the weighted length of the path in $T$ between processes
We refer to an edge of $T$ as a \textit{tree edge}, and any other edge of $X$ as a \textit{cross edge}. Suppose all communication between processes is routed through $T$. If one tree edge $e$ fails, we can write $e = \{x, p(x)\}$ for some process $x$, which we call the \textit{point of failure}. Since $X$ is 2-edge connected, communication can be restored by replacing $e$ by some cross edge $e'$ where the ends of $e'$ lie in different components of $T - e$. We call such an edge $e'$ a swap edge of $e$, or a swap edge of $x$, and we define $\text{SwapEdges}(e) = \text{SwapEdges}(x)$ to be the set of all swap edges of $e$. Of all possible swap edges of $e$, we would like to choose the best, where “best” is defined in a manner determined by the application. The \textit{all best swap edges problem} is to identify the best swap edge for every tree edge, so that in case of any edge failure, the best swap edge can be activated quickly.

In Figure 2.4-1(a) we show a network with a spanning tree $T$ and four cross edges. The tree edges are solid, while the cross edges, $\{u, u'\}, \{v, v'\}, \{w, w'\}$ and $\{z, z'\}$, are dashed. In (b) and (c), we show all swap edges of two different choices of failed tree edge, namely $\{x, p(x)\}$ and $\{y, p(y)\}$. The swap edges of $x$ are $\{u, u'\}, \{v, v'\},$ and $\{w, w'\}$, shown in (b). The swap edges of $y$ are $\{v, v'\}, \{w, w'\},$ and $\{z, z'\}$, shown in (c).
In [8][9], several different criteria for determining “best” are considered. In each case, the best swap edge for $e$ is that swap edge $e'$ for which some measure $F(T, r, e, e')$ is minimized. We consider six such measures in this thesis. In each case, let $T' = T - e + e'$, the spanning tree of $X$ which results from deleting $e$ and adding $e'$, and $x$ is the point of failure, i.e., $e = \{x, p(x)\}$.

- $F_{\text{dist}}(T, r, e, e') = W_{T'}(x, r)$, the distance from the root to the point of failure in $T'$.
- $F_{\text{incr}}(T, r, e, e') = \max \{W_T(u, r) - W_{T'}(u, r) : u \in T_X\}$, the maximum increase of distance from the root to any process when $T$ is replaced
by $T'$. In Section 3.3, we show that minimizing $F_{incr}$ is equivalent to minimizing $F_{dist}$.

- $F_{wght}(T, r, e, e') = w(e')$, the weight of the swap edge. If $T$ is a minimum spanning tree of the network $X$, then $T' = T - e + e'$ is a minimum spanning tree of $X - e$.

- $F_{max}(T, r, e, e') = \max \{W_T(u, r) : u \in T_x\}$, the maximum distance, in $T'$, from the root to any process in $T_x$, which is the component of $T - e$ that contains the point of failure. (The distance from the root to any process in $T-x = T - e - T_x$, the other component, remains unchanged.)

- $F_{sum}(T, r, e, e') = \sum_{u \in T_x} W_T(u, r)$, the sum of the distances, in $T'$, from the root to all processes in $T_x$.

- $F_{diam}(T, r, e, e') = \max \{W_T(u, v) : u \in T_x \text{ and } v \not\in T_x\}$. Minimizing this function minimizes the diameter of $T'$.

If $T$ is a spanning tree of minimum diameter for the network $X$, then $T' = T - e + e'$ may not be a spanning tree of $X - e$ of minimum diameter, as the example given in Figure 3.6-1 shows.

In Figure 2.4-2, we illustrate an example where an edge $\{x, r\}$ has four swap edges, $e'_1, e'_2, e'_3, \text{ and } e'_4$. In Table 2.4-1, we give the values of $F(T, r, e, e')$ for the six choices of $F$, where we assume that all edges have weight 1. Note that in the case of $F_{dist}$ or $F_{incr}$, $e'_1$ is the best swap edge for $e$, in the case of $F_{max}$ or $F_{diam}$, $e'_2$ is best, and in the case of $F_{sum}$, $e'_3$ is
best. Since all edges have the same weight, all swap edges are equally good in the case of $F_{\text{wght}}$.

![Cross Edges](image)

Figure 2.4-2: Cross Edges
We show $T$ in (a). The edge $e$ has four cross edges. In (b), (c), and (d), we show the resulting tree $T' = T - e + e'$ for three choices of $e'$. We do not show the case $e' = e'_4$

<table>
<thead>
<tr>
<th>$F$</th>
<th>$F_{\text{dist}}$</th>
<th>$F_{\text{incr}}$</th>
<th>$F_{\text{wght}}$</th>
<th>$F_{\text{max}}$</th>
<th>$F_{\text{sum}}$</th>
<th>$F_{\text{diam}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F(T, r, e, e'_1)$</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>34</td>
<td>7</td>
</tr>
<tr>
<td>$F(T, r, e, e'_2)$</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>31</td>
<td>5</td>
</tr>
<tr>
<td>$F(T, r, e, e'_3)$</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>$F(T, r, e, e'_4)$</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>8</td>
<td>42</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 2.4-1: $F(T, r, e, e')$ for the network in Figure 2.4-2 for various $F$. 16
CHAPTER 3
QUADRATIC TIME SWAP EDGE ALGORITHM

3.1 The Algorithm BSE

In [6], Flocchini et al. give an algorithm for solving the all best swap edge problem using $F_{\text{dist}}$ as the measure. In [8], Flocchini et al. give a general algorithm which we call BSE, for the all best swap edges problem, and then give specific versions of the technique to solve the problem BSE problem using each of the measures $F_{\text{incr}}$, $F_{\text{max}}$, and $F_{\text{sum}}$. In [9], Gfeller et al. give an algorithm for the all best swap edge problem, using the measure $F_{\text{diam}}$. Their algorithm is also a version of BSE.

We will write $BSE_{\text{dist}}$, $BSE_{\text{incr}}$, $BSE_{\text{wght}}$, $BSE_{\text{max}}$, $BSE_{\text{sum}}$, and $BSE_{\text{diam}}$ to denote the versions of BSE which minimize the measures $F_{\text{dist}}$, $F_{\text{incr}}$, $F_{\text{wght}}$, $F_{\text{max}}$, $F_{\text{sum}}$ and $F_{\text{diam}}$ respectively.

The space complexity of BSE is $O(\delta \cdot n)$ for each process $x$, provided we measure space not in bits, but in number of values stored, where each value is a weight, a pointer to a neighbor of $x$, or an integer which does not exceed $n$. The time complexity of BSE is $O(h^2)$, since it proceeds in waves, one for each level $l$, where $1 \leq l \leq h$. The level of a process $x$ is defined to be the hop-distance from $x$ to $r$. Wave $l$ computes the best swap edge for all processes at level $l$, and each wave takes $O(h)$ time.

3.2 General Overview of BSE

BSE consists of two phases, the preprocessing phase and the
optimization phase. The preprocessing phase computes variables that will be needed by processes during the optimization phase. The set of variables that are computed during preprocessing depends on which of the six measures we are to minimize, but those variables always include \textit{size}(x) and \textit{index} (x). We compute \textit{size}(x), the cardinality of \( T_x \), the subtree of \( T \) rooted at \( x \), for all \( x \) in one convergecast wave, starting at the leaves of \( T \).

We define a left-to-right ordering on the children of each process of \( T \). Then, define \( \text{index} (x) = (\text{pre-index} (x), \text{post-index} (x)) \), the index of \( x \), where \( \text{pre-index} (x) \) is the order of \( x \) in the preorder visitation of \( T \), and where \( \text{post-index} (x) \) is the order of \( x \) in the reverse postorder visitation of \( T \). (Reverse postorder visitation \( T \) is the same as preorder visitation after reversing the roles of left and right.)

Indices are used to determine whether a given process is a descendant of another. We define a partial order, “\( \leq \)" on ordered pairs of integers; we say \((a, b) \leq (c, d)\) if and only if \( a \leq c \) and \( b \leq d \). Then \( x \) is an ancestor of \( y \), i.e., \( y \in T_x \), if and only if \( \text{index} (x) \leq \text{index} (y) \). Thus, if \( e' = \{y, y'\} \) is a cross edge and \( y \in T_x \), then \( e' \in \text{SwapEdges}(x) \) if and only if \( \text{index} (x) \leq \text{index} (y') \). In Figure 3.2-1, we show an example of \( T \) where each process is labeled with its index.

Indices also enable delivery of a message packet along the shortest path in \( T \). Suppose a process \( x \) needs to send a packet to another process \( y \), and \( x \) knows the value of \( \text{index} (y) \). If \( \text{index} (x) \leq \text{index} (y) \), then
$x$ is an ancestor of $y$, and $x$ sends the packet to whichever of its children is also an ancestor of $y$. Otherwise, $x$ sends the packet to its parent.

Once size has been computed, the values of $pre\_index$ and $post\_index$ are computed in a single top down wave. Initially, $pre\_index (r) = 1$ and $post\_index (r) = 1$. Each process chooses an ordering of its children, which we call left-to-right order. If the children of $x$ are $y_1, y_2, \ldots, y_m$, then

\begin{align*}
pre\_index (y_i) &= pre\_index (x) + 1 + \sum_{1 \leq j < i} size(y_j) \\
post\_index (y_i) &= post\_index (x) + 1 + \sum_{1 < j \leq m} size(y_j)
\end{align*}

![Figure 3.2-1 : Ancestry of processes. Processes are labeled with their indices. A process $x$ is an ancestor of $y$ if and only if index $(x) \leq index (y)$.](image)

19
In every case, there are at most \( O(\delta_n) \) values to be computed for each process \( x \), and the time complexity of the preprocessing phase is \( O(h) \).

During the optimization phase, the iteration for each process \( x \neq r \), which we call Iteration \( (x) \), is represented by Lines 2–10 of Table 3.2-1, and computes the best swap edge for \( x \). Iteration \( (x) \) consists of a broadcast wave starting at \( x \), represented by Lines 2–5 of Table 3.2-1, followed by a convergcast wave which ends at \( x \), represented by Lines 6–10. In the broadcast wave, each process \( y \) of \( T_x \) creates a down package, using the down package of its parent (unless \( y = x \)), and also using variables computed during preprocessing. The contents of the down package depend on which of the six measures is being minimized, but it always includes \( \text{index}(x) \), since comparison of the index of \( x \) with the index of the farther end of a cross edge determines whether that cross edge is a swap edge of \( x \).

---

1: for all \( x \neq r \) in top down order do \{Iteration \( (x) \)\}
2: Compute \( \text{down\_package}(x, x) \).
3: for all \( y \in T_x - x \) in top down order do
4: \hspace{1em} Compute \( \text{down\_package}(y, x) \), using \( \text{down\_package}(p(y), x) \).
5: \hspace{1em} end for
6: for all \( y \in T_x \) in bottom up order do
7: \hspace{1em} Compute \( l_{\text{sol}}(y, x) \), using \( \text{down\_package}(y, x) \).
8: \hspace{1em} subtree\_mincost \( (y, x) \leftarrow \min \{l_{\text{sol}}(y, x), \min \{\text{subtree\_mincost}(z, x) : z \in \text{Chldrn}(y)\}\} \).
9: \hspace{1em} end for
10: solution \( (x) \leftarrow \text{subtree\_mincost}(x, x) \).
11: end for

Table 3.2-1 : Optimization Phase of BSE
During the convergecast wave of Iteration($x$), each process $y$ of $T_x$ computes $l_{sol} (y, x)$, the minimum cost for any swap edge of $x$ which is incident to $y$. During this computation, $y$ makes use of its down package, as well as variables computed during preprocessing. Then, $y$ computes $subtree_{mincost} (y, x)$, the minimum cost for any swap edge of $x$ which is incident to any process of $T_y$, by comparing $l_{sol} (y, x)$ with $subtree_{mincost} (z, x)$ for all $z \in Chldrn(y)$. Finally, the minimum cost for any swap edge of $x$ is $solution(x) = subtree_{mincost} (x, x)$.

The executions of these iterations cannot overlap, i.e., no process can be participating in more than one of them at a given time. If $x_1$ and $x_2$ are independent, meaning that $T_{x_1}$ and $T_{x_2}$ are disjoint, the computation of the best swap edges for $x_1$ and $x_2$ can be executed concurrently. On the other hand, if $y \in Chldrn(x)$, then Iteration($y$) cannot begin until $y$ is finished with its participation in Iteration($x$).

At the end of Iteration($x$), all variables computed by all $y \in T_x$, other than $solution(x)$ itself, are deleted, to make space for the variables of subsequent iterations. We review this in detail in Section 3.8.

Figure 3.2-2 through Figure 3.2-11, below, illustrate an example of Iteration($x$). Figure 3.2-2 shows a network, with a rooted spanning tree and several cross edges. Figure 3.2-3 shows the beginning of the iteration, after Line 2 of the code given in Table 3.2-1 has executed. The circle around the process $x$ indicates that it has computed $down_{package}(x, x)$. 
Figure 3.2-2: Cross edges are shown as dashed. Point of failure is x.

Figure 3.2-3: Iteration begins with computation of \textit{down\_package}(x, x).

Figure 3.2-4: Children of x compute \textit{down\_package}

Figure 3.2-5: Broadcast wave continues. Variables of \textit{down\_package} are retained until needed.
In Figure 3.2-4 through Figure 3.2-6, the broadcast wave spreads to the leaves of $T_x$. The small circle around each process $y$ indicates that $down\_package(y, x)$ has been computed.

Figure 3.2-6: Broadcast wave is completed. $l\_sol$ is computed for some leaves.

Figure 3.2-7: Convergecast continues.

Figure 3.2-8: Convergecast continues

Figure 3.2-9: $down\_package$ and $l\_sol$ are deleted when not needed.
In Figure 3.2-6 through Figure 3.2-10, the broadcast wave of the iteration is illustrated. The double circle around any process $y$ indicates that $y$ has computed $l\_sol(y, x)$ and $subtree\_mincost(y, x)$.

After a process $y$ no longer needs those values, $down\_package(y, x)$ and $subtree\_mincost(y, x)$ are deleted. After $subtree\_mincost(x, x)$ is computed, $solution(x)$ is computed, as indicated by the box around $x$. No other variable of $Iteration(x)$ is retained by any process of $T_x$, and thus its space is free to be used in the next iteration.

BSE takes $O(h)$ time to execute each $Iteration(x)$. Iterations for all
processes at a given level can take place concurrently. Since there are \( h \) such levels, the overall time complexity of BSE is \( O(h^2) \).

### 3.3 \( \text{BSE}_{\text{dist}} \) and \( \text{BSE}_{\text{incr}} \)

In addition to \( \text{size}(x) \) and \( \text{index}(x) \), the preprocessing phase of \( \text{BSE}_{\text{dist}} \) computes \( \text{depth}(x) = W_T(x, r) \) for all \( x \).

In the broadcast portion of \( \text{Iteration}(x) \), \( \text{down\_package}(x) \) consists of the variables \( \text{index}(x) \) and \( W_T(y, x) \). Line 4 of Table 3.2-1 is then executed by \( y \) simply copying the value of \( \text{index}(x) \) from \( p(y) \), and by computing \( W_T(y, x) = w(y, p(y)) + W_T(p(y), x) \).

Line 8 of Table 3.2-1 is then executed by \( y \) by first computing the length of the shortest path from \( y \) to \( r \) which uses a swap edge of \( x \) incident to \( y \), and then comparing this value to \( \text{subtree\_mincost}(z, x) \) for all \( z \in \text{Chldrn}(y) \):

1. Compute \( l_{\text{sol}}(y, x) = \min \{ W_T(y, x) + w(y, y') + \text{depth}(y') : \{y, y'\} \in \text{SwapEdges}(x)\} \)
2. Compute \( \text{subtree\_mincost}(y, x) = \min \{ l_{\text{sol}}(y, x), \min \{ \text{subtree\_mincost}(z, x) : z \in \text{Chldrn}(y) \}\} \)

Finally, \( \text{solution}(x) = \text{subtree\_mincost}(x, x) \).

Figure 3.3-1 illustrates computation of \( l_{\text{sol}}(y, x) \) and \( \text{solution}(x) \) for \( F_{\text{dist}} \). We assume all edge weights are 1. In (a), \( y = u \), \( W_T(u, x) = 2 \), \( \text{depth}(u') = 3 \), and \( l_{\text{sol}}(u, x) = 6 \). In (b), \( y = v \), \( W_T(v, x) = 1 \), \( \text{depth}(v') = 2 \), and \( l_{\text{sol}}(v, x) = 4 \). Other possible swap edges are not shown; they would
give larger values of \( l_{\text{sol}}(y, x) \). Then \( \text{solution}(x) = 4 \), the smallest value of \( l_{\text{sol}}(y, x) \).

Figure 3.3-1: \( l_{\text{sol}}(u, x) \)
All edge weights are 1. Then \( l_{\text{sol}}(u, x) = 6 \), and \( \text{solution}(x) = l_{\text{sol}}(v, x) = 4 \).

We do not need to separately describe an algorithm which minimizes \( F_{\text{incr}} \), since the best swap edge for \( F_{\text{dist}} \) is also the best swap edge for \( F_{\text{incr}} \), as stated in Lemma 3.1.

**Lemma 3.1** For any tree edge \( e = \{x, p(x)\} \) and any swap edge \( e' \) of \( e \),
\[
F_{\text{incr}}(T, r, e, e') = F_{\text{dist}}(T, r, e, e') - \text{depth}(x).
\]

**Proof:** Since \( x \in T_x, F_{\text{incr}}(T, r, e, e') \geq W_T(x, r) - W_T(x, r) = F_{\text{dist}}(T, r, e, e') - \text{depth}(x) \).
To prove the converse, let $e' = (z, z')$, where $z \in T_x$ and pick $y \in T_x$ such that $F_{\text{incr}} (T, r, e, e') = W_T(y, r) - W_T(x, r)$. Then

$$F_{\text{incr}} (T, r, e, e') = W_T(y, r) - W_T(x, r)$$

$$= W_T(y, r) - W_T(x, r) - W_T(y, x)$$

$$\leq W_T(x, r) + W_T(y, x) - W_T(x, r) - W_T(y, x)$$

$$= W_T(x, r) - \text{depth}(x)$$

$$= F_{\text{dist}}(T, r, e, e') - \text{depth}(x)$$

and we are done.

### 3.4 BSE$_{\text{wght}}$

BSE$_{\text{wght}}$ is the simplest of our six versions of BSE. $F(T, r, e, e') = w(e')$, and thus all BSE$_{\text{wght}}$ needs to do is find the swap edge of $e$ of smallest weight.

The preprocessing phase of BSE$_{\text{wght}}$ computes only $\text{size}(x)$ and $\text{index}(x)$ for all $x$, and $\text{down\_package}(y, x)$ contains only the variable $\text{index}(x)$.

Line 4 of Table 3.2-1 is then executed by $y$ simply copying the value of $\text{index}(x)$ from $p(y)$.

Line 8 of Table 3.2-1 is then executed by $y$ by first computing smallest weight of any swap edge of $x$ incident to $y$, i.e., $\min \{w(y, y') : (y, y') \in \text{SwapEdges}(x)\}$, and then comparing this value to $\text{subtree\_mincost}(z, x)$ for all $z \in \text{Chldrn}(y)$:

1. Compute $l_{\text{sol}}(y, x) = \min \{w(y, y') : (y, y') \in \text{SwapEdges}(x)\}$
2. Compute \( \text{subtree}_\text{mincost}(y, x) = \min \{ l_\text{sol}(y, x), \min \{ \text{subtree}_\text{mincost}(z, x) : z \in \text{Chldrn}(y) \} \} \)

Finally, \( \text{solution}(x) = \text{subtree}_\text{mincost}(x, x) \).

If \( T \) is a minimum spanning tree of \( X \) and \( e' \) is the best swap edge for \( e \), then \( T' = T - e + e' \) is a minimum spanning tree of \( X - e \). This follows from the fact that \( e' \) is a swap edge of \( e \) if and only if the ends of \( e' \) lie in two different components of \( T - e \), and the well-known result that, if an edge of the minimum spanning tree of a weighted graph is deleted, and if the graph remains connected, then a new minimum spanning tree is formed by adding the edge of minimum weight that does not create a cycle.

### 3.5 BSE\(_{\text{max}}\)

For any weighted network \( Y \) and any process \( x \) of \( Y \), we define \( \text{ecc}_Y(x) = \max_{u \in Y} \text{Wy}(x, u) \), the eccentricity of \( x \) in \( Y \). Recall that \( F_{\text{max}}(T, r, e, e') = \text{depth}(y') + w(e') + \text{ecc}_T(y) \), where \( e = \{x, p(x)\} \) and \( e' = \{y, y'\} \) is a swap edge of \( e \), and \( y \in T_x \). We illustrate \( F_{\text{max}}(T, r, e, e') \) in Figure 3.5-1.
Figure 3.5-1: $F_{\text{max}}$
We show $T$ in (a), together with some cross edges. In (b) we show $F_{\text{max}} (T, r, e, e_1')$ and in (c) we show $F_{\text{max}} (T, r, e, e_2')$, where $e_1' = \{u, u'\}$ and $e_2' = \{v, v'\}$.

Besides $\text{size}(x)$ and $\text{index}(x)$, the preprocessing phase of BSE_{\text{max}} computes

1. $\text{depth}(x)$.
2. $\text{height}(x) = \text{ecc}_{T_x}(x)$, the largest weight of any path from $x$ to a leaf of $T_x$.
3. If $x \neq r$, $\eta(x) = \max \{W_T(p(x), u) : u \in T_{p(x)} - T_x\}$, the largest weight of any path in $T_{p(x)} - T_x$ from $p(x)$ to a leaf of $T_{p(x)}$. We can also write $\eta(x) = \text{ecc}_{T_{p(x)} - T_x}(p(x))$. We illustrate an example of $\eta(x)$ in Figure 3.5-2.

All values of $\text{depth}$ are computed in a broadcast wave, and all values
of height are computed in a convergecast wave. Once height has been computed for all processes, all values of \( \eta \) can be computed simultaneously in \( O(1) \) time, since \( \eta(x) = \max \{ w(y, p(x)) + \text{height}(y) : y \in \text{Chldrn}(p(x)) - x \} \).

For notational convenience, we write

- \( \text{path}_T(x, y) = \) the path in \( T \) from \( x \) to \( y \). We will write \( \text{path}(x, y) \) if \( T \) is understood.
- \( \text{longest}_T(x) = \) the longest path in \( T \) starting at \( x \). Thus, \( W_T(\text{longest}_T) = \text{ecc}_T(x) \),
- \( \text{down}_T(x) = \text{longest}_T_{T_x}(x) \) the longest path from \( x \) to a leaf of \( T_x \). Thus, \( W_T(\text{down}_T(x)) = \text{height}(x) \).

Figure 3.5-2: An example where \( \eta(x) = 4 \). We assume that all edge weights are 1.
For any process $x$ and any $y \in T_x$, define $\mu(y, x) = \max \{W_T(y, u) : u \in T_x - T_y\}$. Then $\mu(x, x) = 0$, and, for $y \neq x$, we can write $\mu(y, x) = \text{ecc}_Y(x)$, where $Y$ is the network $T_x - T_y + \{y, p(y)\}$. Intuitively, $\mu(y, x)$ is the length of the longest path in $T_x$ which starts at $y$ and avoids all children of $y$. We illustrate an example of $\mu(y, x)$ in Figure 3.5-3.

For any process $x$ and any $y \in T_x$, define $\phi(y, x) = \text{ecc}_{T_x}(y) = \max\{\text{height}(y), \mu(y, x)\}$, since the longest path in $T_x$ which starts at $y$ must either go down to a leaf of $T_y$ or up through $p(y)$.

Figure 3.5-3: An example where $\mu(y, x) = 4$. We assume that all edge weights are 1.
For each \( x \) and \( y \in T_x \), \( \text{down}_\text{package}(y, x) \) consists of the variables \( \text{index}(x) \) and \( \mu(y, x) \). In Line 2 of Table 3.2-1 for \( F_{\text{max}} \), we already have \( \text{index}(x) \) from the preprocessing phase. We let \( \mu(x, x) = 0 \). To execute Line 4, \( y \) simply copies \( \text{index}(x) \) from its parent, and computes \( \mu(y, x) = w(y, p(y)) + \max \{ \eta(y), \mu(p(y), x) \} \).

Computation of \( l_{\text{sol}}(y, x) \) depends on the fact that \( \phi(y, x) \) is the maximum of \( \text{height}(y) \) and \( \mu(y, x) \). In Line 8 of Table 3.2-1, \( y \) computes \( \phi(y, x) = \text{ecc}_{T_x}(y) = \max \{ \text{height}(y), \mu(y, x) \} \), and then \( l_{\text{sol}}(y, x) = \min \{ \text{depth}(y') + w(y, y') + \phi(y, x) : \{ y, y' \} \in \text{SwapEdges}(x) \} \).

3.6 \( \text{BSE}_{\text{diam}} \)

The original goal of \( \text{BSE}_{\text{diam}} \) is to find, for each tree edge \( e \), the swap edge \( e' \) which minimizes the diameter of \( T' = T - e + e' \). But \( F_{\text{diam}} \) is defined to maximize the length of any path from a point in \( T_x \) to a point in \( T_{\sim x} \), rather than the diameter of \( T' \). However, minimizing \( F_{\text{diam}} \) minimizes the diameter of \( T' \), as we state in Lemma 3.2 below.

**Lemma 3.2** If \( e \) is a tree edge of \( T \), and if \( e' \in \text{SwapEdges}(e) \) is chosen to minimize \( F_{\text{diam}}(T, r, e, e') \), then \( e' \) is also a choice of swap edge of \( e \) which minimizes the diameter of \( T' = T - e + e' \).

**Proof:** Write \( e = \{ x, p(x) \} \). Let \( A \) and \( B \) be the diameters of \( T_x \) and \( T_{\sim x} \), respectively, and let \( C' = F_{\text{diam}}(T, r, e, e') \). Then \( \text{diam}(T') \), the diameter of \( T' \), is equal to \( \max \{ A, B, C' \} \). Pick \( e'' \in \text{SwapEdges}(e) \) to minimize the
diameter of $T'' = T - e + e''$, and let $C'' = F_{diam}(T, r, e, e'')$. By definition, $diam(T'') \leq diam(T)$. Since $C' \leq C''$, by definition of $e'$, we also have $diam(T'') = \max \{A, B, C''\} \geq diam(T')$. Thus, $e'$ is also an optimal choice of swap edge to minimize the diameter of the resultant tree.

We say that a process $c$ of $Y$ is a center of $Y$ if $ecc_Y(c) \leq ecc_Y(x)$ for any process $x$ of $Y$. If $Y$ is a tree, then the center (or centers) of $Y$ can be computed by a distributed algorithm in $O(diam(Y))$ time using $O(\delta_x)$ space per process $x$, where space is defined in terms of number of items, rather than bits [12]. We will assume that $r$ is the center of $T$; if we are given a rooted tree where the root is not the center, we first apply the algorithm [12] to redefine the root to be the center.

If $e = \{x, p(x)\}$ is a tree edge and $e' = \{z, z'\}$ is a swap edge of $e$, let $T'' = T - e + e'$. Then we define $F_{diam}(T, r, e, e') = ecc_{T_x}(z) + w(z, z') + ecc_{T-x}(z')$.

If $T$ is a spanning tree of minimum diameter for the network $X$, then $T' = T - e + e'$ may not be a spanning tree of $X - e$ of minimum diameter, as the example given in Figure 3.6-1 shows.
Network X with four processes and five edges is shown in (a). Let all edge weights be 1. The minimum diameter spanning tree T shown in (a) has diameter 2. In (b), let $e = \{x, r\}$. The swap edge for e which minimizes $F_{\text{diam}}$ is $e' = \{x, y\}$, and the resulting tree $T' = T - e + e'$ has diameter 3. However, the minimum diameter spanning tree of $X - e$ has diameter 2, as shown in (c).

In Figure 3.6-2(a), we show $T, e = \{x, p(x)\}$, and one cross edge, $e' = \{z,
\( z' \). In Figure 3.6-2(b), we show \( T' = T - e + e' \). The heavy edges show the path whose length is \( F_{\text{diam}}(T, r, e, e') \), consisting of the longest path in \( T_x \) starting at \( z \), the longest path in \( T_{\sim x} \) starting at \( z' \), and the swap edge.

We now give some additional definitions which are needed to describe \( BSE_{\text{diam}} \).

- A weighted tree graph always has either one or two centers. We will assume that \( r \) is one of those centers. Let \( \text{Chldrn}(r) = \{c_1, \ldots, c_m\} \), where \( m = \delta_r \). Let \( S \) be the network obtained from \( T \) by deleting \( r \) and all edges of \( T \) incident to \( r \). Then \( S \) is the disjoint union of \( m \) trees, \( S_1, \ldots, S_m \), where \( S_i \) is rooted at \( c_i \).

- For \( 1 \leq i \leq m \), we define \( h_i = w(r, c_i) + \text{height}(c_i) \), the largest weight of any path from \( r \) to a leaf of \( S_i \). Without loss of generality, the values of \( h_i \) are monotone decreasing, i.e., \( h_i \geq h_{i+1} \) for \( 1 \leq i < m \). Thus, in particular, \( h_1 = h = \text{height}(r) \).

- For any \( 1 \leq i \leq m \) and any \( x \in S_i \) we define \( \text{avoid}(x) \) to be the largest weight of any path from \( r \) to a leaf of \( S_i \) which avoids, i.e., does not contain, \( x \). If no such path exists, we let \( \text{avoid}(x) = 0 \).
Let all edge weights be 1. Then \( h_1 = 7, h_2 = 6, \) and \( h_3 = 2. \) The magenta path is the longest path from \( r \) through \( c_1 \) that avoids \( x, \) and thus \( \text{avoid}(x) = 4. \)

We can compute \( \text{avoid}(x) \) for all \( x \) in \( O(h) \) time, in a broadcast wave. If \( x = c_i \) for some \( i, \) then \( \text{avoid}(x) = 0. \) Otherwise, \( \text{avoid}(x) = \max \{\text{avoid}(p(x), \eta(p(x) + \text{depth}(p(x))\}. \)

### 3.6.1. Preprocessing Phase of \( \text{BSE}_{\text{diam}} \)

The preprocessing phase of \( \text{BSE}_{\text{diam}} \) computes the following variables for each process \( x. \)

1. \( \text{size}(x). \)
2. \( \text{index}(x). \)
3. height(x).

4. depth(x).

5. $\eta(x) = \max \{W_T(p(x), u) : u \in T_{p(x)} - T_x\}$, as defined in Section 3.5.

6. branch(x), provided $x \neq r$, which is defined to be that value of $i$ such that $x \in S_i$.

7. $h_1$, $h_2$, and $h_3$. If $c_3$ does not exist, i.e., $\delta_r = 2$, we let $h_3 = 0$.

8. Recall the definitions of $\mu$ and $\phi$ given in Section 3.5.

   (a) $local_\mu(x) = \mu(x, c_i)$ where $x \in S_i$. This is the length of the longest path in $S_i$ starting from $x$ which avoids $Chldrn(x)$.

   (b) $local_\phi(x) = \phi(x, c_i) = ecc_{S_i}(x)$ where $x \in S_i$. This is the length of the longest path in $S_i$ starting from $x$, and thus equal to $\max \{local_\mu(x), height(x)\}$.

9. avoid(x) for $x \neq r$.

The values of size and index are computed in one convergecast wave followed by one broadcast wave. The values of height are computed in a convergecast wave, and the values of depth in a broadcast wave.

After the values of height have been computed, $r$ assigns indices to its children, using the rule that $height(c_{i+1}) \leq height(c_i)$, and then assigns $branch(c_i) = i$. The values of branch(x) for all other $x \neq r$ are then assigned to all processes in a broadcast wave, since $branch(x) = branch(p(x))$.

The values of $h_i$ are computed by $r$. The largest three of those values, namely $h_i$ for $i = 1, 2, 3$, are broadcast to all processes.
Once $\text{height}(x)$ has been computed for all $x$, all values of $\eta$ can be computed in $O(1)$ time. The values $\text{branch}(c_i) = i$ are assigned by $r$ to its children, and $r$ also computes $h_i$ for $i \leq 3$. The value of $\text{branch}(c_i)$ is simply broadcast to all processes in $S_i$, and the values $h_i$ for $i \leq 3$ are simply broadcast to all processes.

Once $\text{height}(x)$ has been computed for all $x$, all values of $\text{local}_\mu$ can be computed in a broadcast wave, using the appropriate version of a formula given in Section 3.5, namely $\text{local}_\mu(x) = \omega(x, p(x)) + \max\{\eta(x), \text{local}_\mu(p(x))\}$.

Once $\text{local}_\mu(x)$ has been computed for all $x$, $\text{local}_\phi(x) = \max\{\text{local}_\mu(x), \text{height}(x)\}$ can be computed for all $x$ in $O(1)$ time altogether.

The values of $\text{avoid}(x)$ for $x \neq r$ are computed in a broadcast wave. Let $\text{avoid}(c_i) = 0$. For all other $x$, compute $\text{avoid}(x) = \max\{\text{avoid}(p(x), \eta(x) + \text{depth}(p(x))\}$.

### 3.6.2 Optimization Phase of BSE$_{\text{diam}}$

For all $x \neq r$ and all $y \in T_x$, $\text{down\_package}(y, x)$ consists of $\text{index}(x)$ and $\mu(y, x)$.

If $y \neq x$, then $y$ computes $\text{index}(x)$ from its parent. The variable $\mu(y, x)$ is computed by $y$ in the same manner as given in Section 3.5, and $\phi(y, x) = \max\{\mu(y, x), \text{height}(y)\}$.

Execution of Line 8 of Table 3.2-1 for BSE$_{\text{diam}}$ is far more complex than for BSE for any of the other measures. For that reason, we give the code for that execution in algorithmic form in Table 3.6-1 below.
1: for all \( y' \) such that \( \{ y', y \} \in \text{SwapEdges}(x) \) do  
2: \( i \leftarrow \text{branch}(y) \)  
3: \( j \leftarrow \text{branch}(y') \)  
4: \( k \leftarrow \) the smallest positive integer which is neither \( i \) nor \( j \)  
5: if \( i = j \) then  
6: \( \text{ecc}_{T-x}(y') \leftarrow \text{depth}(y') + h_k \)  
7: else  
8: \( \text{ecc}_{T-x}(y') \leftarrow \max \{ \text{local}_x \phi (y'), \text{depth}(y') + \text{avoid}(x), \text{depth}(y') + h_k \} \)  
9: end if  
10: \( \text{cost}(y, y', x) \leftarrow \phi (y, x) + w(y, y') + \text{ecc}_{T-x}(y') \)  
11: end for  
12: \( l_{sol}(y, x) \leftarrow \min \{ \text{cost}(y, y', x) : \{ y, y' \} \in \text{SwapEdges}(x) \} \)

Table 3.6-1: Computation of \( l_{sol}(y, x) \) in BSE_{diam} 

3.6.3. Computation of \( \text{ecc}_{T-x}(y') \) 

We now explain the computation of \( \text{ecc}_{T-x}(y') \), the eccentricity of \( y' \) in the subgraph \( T_x \). Let \( i = \text{branch}(y) \) and \( j = \text{branch}(y') \), and let \( k \) be the smallest positive integer which is neither \( i \) nor \( j \). We consider the two cases: \( i = j \) and \( i \neq j \).

If \( i = j \), then the longest path in \( T_x \) from \( y' \) runs from \( y' \) to \( r \), then from \( r \) to the farthest leaf of \( S_k \), as shown in Figure 3.6-4. If \( i \neq j \), let \( \alpha \) be the longest path in \( S_j \) from \( y' \), i.e., the path whose length is \( \text{ecc}_{S_j}(y') \), let \( \beta \) be the path from \( y' \) to \( r \), let \( \gamma \) be the longest path from \( r \) to a leaf of \( S_i \) which avoids \( x \), and \( \delta_k \) the longest path from \( r \) to a leaf of \( S_k \). (If \( k = 3 \) and \( S_3 = \emptyset \), we take \( \delta_k \) to be the trivial path at \( r \).) The paths \( \alpha, \beta, \gamma, \) and \( \delta_k \) are illustrated in Figure 3.6-5(a). The path whose length is \( \text{ecc}_{T-x}(y') \) is \( \alpha \), \( \beta + \delta_k \) or \( \beta + \gamma \) whichever is longer. The three possibilities are illustrated in Figure 3.6-5(b)–(d).
Figure 3.6-4: If $i = j$, then $\text{ecc}_{T-x}(y') = \text{depth}(y') + h_k$. 
If \( i \neq j \), \( ecc_{r-x}(y') \) is the maximum length of any path in \( T_{-x} = T - T_x \) from \( y' \). In (a), we show \( \alpha \), the longest path in \( S_j \) from \( y' \); \( \beta \), the path from \( y' \) to \( r \); \( \gamma \) the longest path from \( r \) to a process in \( S_i \) which avoids \( x \), and \( \delta_k \), the longest path from \( r \) to a leaf of \( S_k \). The maximum length path in \( T_{-x} = T - T_x \) from \( y' \) is one of three possibilities, shown in (b)–(d) with heavy lines. In (b), we show \( \alpha \), in (c) we show \( \beta + \delta_k \), and in (d) we show \( \beta + \gamma \).
3.7 BSE\textsubscript{sum}

For any weighted network \( Y \) and any process \( x \) in \( Y \), we define \( \text{path}_sum_Y \ (x) = \sum_{u \in Y} W_Y (x, u) \), the \textit{path sum} of \( x \) in \( Y \), the sum of the shortest weights of paths from \( x \) to all processes of \( Y \). \( F_{\text{sum}}(T, r, e, e') = \text{path}_sum_{Tx \cup v(r)} \), where \( v \) is a \textit{virtual edge} (not an edge of the original network) from \( y \) to \( r \) of length \( W_T(r, y) \). We illustrate \( T_x \cup v \) in Figure 3.7-1.

![Figure 3.7-1: F\textsubscript{sum}](image)

The network \( T_x \cup v \), where \( v \) is a virtual edge of length \( depth(y') + w(y, y') \) from \( y \) to \( r \), where \( y \in T_x \) and \( e' = \{y, y'\} \) is a swap edge of \( x \). \( F_{\text{sum}}(T, r, e, e') \) is the sum of the lengths of the red lines.
For convenience, we introduce shorter notation for certain instances of $\text{path\_sum}_Y(x)$:

- For any process $x$, let $\text{sum}(x) = \text{path\_sum}_{Tx}(x) = \sum_{u \in Tx} W_T(u, x)$.
- For any process $x \neq r$, let $\theta(x) = \text{path\_sum}_{T \setminus Tx}(p(x)) = \sum_{u \in T \setminus Tx} W_T(u, y)$.
- For any processes $x$ and $y \in Tx$, let $\nu(y, x) = \text{path\_sum}_{Tx \setminus Ty}(y) = \sum_{u \in Tx \setminus Ty} W_T(u, x)$.
- For any processes $x$ and $y \in Tx$, let $\psi(y, x) = \text{path\_sum}_{Tx}(y) = \sum_{u \in Tx} W_T(u, y)$.

Note that $\psi(y, x) = \nu(y, x) + \text{sum}(y)$ for $y \in Tx$.

The implementation of $\text{BSE}_\text{sum}$ depends on the following observation.

**Lemma 3.3** : If $y \in Tx$ and $e' = \{y, y'\}$ is a swap edge of $e = \{x, p(x)\}$, then

$$F_{\text{sum}}(T, r, e, e') = \text{size}(x) \cdot (\text{depth}(y') + w(y, y')) + \psi(y, x)$$

**Proof**: Let $T' = T - e + e'$. Then

$$F_{\text{sum}}(T', r, e, e') = \sum_{u \in Tx} W_{T'}(r, u)$$

$$= \sum_{u \in Tx} (W_T(r, y) - W_T(y, u))$$

$$= \text{size}(x) \cdot W_T(r, y) + \sum_{u \in Tx} W_T(y, u)$$

$$= \text{size}(x) \cdot (\text{depth}(y') + w(y, y')) + \psi(y, x)$$

During the preprocessing phase of $\text{BSE}_\text{sum}$, we compute $\text{size}(x)$, $\text{index}(x)$, and $\text{sum}(x)$ for all $x$. The values of $\text{sum}(x)$ are computed in a
convergecast wave. If $x$ is a leaf, then $\text{sum}(x) = 0$. Otherwise, $\text{sum}(x) = \sum_{y \in \text{Chldrn}(x)} (\text{size}(y) \cdot w(x, y) + \text{sum}(y))$.

For each $x$ and $y \in T_x$, $\text{down\_package}(y, x)$ consists of the variables $\text{size}(x)$, $\text{index}(x)$, and $\nu(y, x)$.

Note that

$$\nu(y, x) = w(y, p(y)) \cdot (\text{size}(x) - \text{size}(y)) + \theta(y) + \nu(p(y), x)$$

$$\psi(y, x) = \text{sum}(y) + \nu(y, x)$$

In Line 8 of Table 3.2-1, $y$ computes

$$l_{sol}(y, x) = \min\{ \text{size}(x) \cdot (\text{depth}(y') + w(y, y')) + \text{path\_sum}_{T_x}(y) : \{y, y'\} \in \text{SwapEdges}(x) \}$$

3.8 Implementation and Complexity of BSE

We now detail the implementation of BSE, in such a way as to ensure the complexity results outlined in Section 3.1. The computation of BSE is primarily organized into either broadcast (topdown from $r$) or convergecast (bottom-up from the leaves of $T$) waves. Each process $x$ knows its neighbors, $N(x)$, and the weight $w(x, y)$ of the edge to each $y \in N(x)$. Furthermore, $x$ knows its parent in $T$, $p(x)$, and its children in $T$, $\text{Chldrn}(x)$, and thus $x$ knows $\text{Cross\_N}(x)$, the set of all neighbors $x'$ of $x$ such that $\{x, x'\}$ is a cross edge. We also assume an ordering on $\text{Chldrn}(x)$, although the choice of that ordering is arbitrary.

3.8.1 Messages

BSE is implemented using eight species of messages. Six of those
eight messages are packets of values, which can vary depending on which of the six functions is used as the measure. Two of the messages, PRE_DONE and OPT_DONE, carry no values.

1. PRE_DOWN_I is the message sent by each process to its children during the first broadcast wave of the preprocessing phase.

2. PRE_UP is the message sent by each process, except $r$, to its parent during the convergecast wave of the preprocessing phase.

3. PRE_DOWN_II is the message sent by each process to its children during the second broadcast wave of the preprocessing phase.

4. CROSS($x$) is the message sent by a process $x$ to each of its cross neighbors.

5. PRE_DOWN is the message sent by each process, except $r$, to its parent to indicate that it is done with the preprocessing phase.

6. OPT_DOWN($x$) is the message sent by each process in $T_x$ to its children during the broadcast wave Iteration($x$) of the optimization phase.

7. OPT_UP($x$) is the message sent by each process in $T_x$, other than $x$, to its parent, during the convergecast wave of Iteration($x$) of the optimization phase. At the end of this wave, $x$ computes its best swap edge.

8. OPT_DONE($x$) is the message sent by $x$ to each $y \in Chldrn(x)$ to inform $y$ that Iteration($x$) is done, and to start Iteration($y$).

3.8.2. Variables Computed during each Wave

In Table 3.8-1, we show which variables of each process are computed
during each wave of BSE.

<table>
<thead>
<tr>
<th>Message / Wave</th>
<th>BSE\textsubscript{dist}</th>
<th>BSE\textsubscript{incr}</th>
<th>BSE\textsubscript{wght}</th>
<th>BSE\textsubscript{max}</th>
<th>BSE\textsubscript{diam}</th>
<th>BSE\textsubscript{sum}</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRE\textsubscript{DOWN_I}</td>
<td>depth</td>
<td>depth</td>
<td>(none)</td>
<td>depth</td>
<td>depth</td>
<td>depth</td>
</tr>
<tr>
<td>First Preprocessing Broadcast Wave</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRE\textsubscript{UP}</td>
<td>size</td>
<td>height size</td>
<td>size</td>
<td>height size</td>
<td>height size</td>
<td>height size sum</td>
</tr>
<tr>
<td>Preprocessing Convergecast Wave</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRE\textsubscript{DOWN_I}</td>
<td>index</td>
<td>index</td>
<td>index</td>
<td>index η</td>
<td>index η branch h\textsubscript{1}, h\textsubscript{2}, h\textsubscript{3} local_µ avoid local_φ</td>
<td>index θ</td>
</tr>
<tr>
<td>I Second Preprocessing Broadcast Wave</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CROSS(x)</td>
<td>index (x) depth(x)</td>
<td>index (x) depth(x)</td>
<td>index (x) depth(x)</td>
<td>index (x) depth(x) local_µ(x) local_φ(x) branch(x)</td>
<td>index (x) depth(x)</td>
<td></td>
</tr>
<tr>
<td>OPT\textsubscript{DOWN(x)}</td>
<td>index (x) W\textsubscript{T} (y,x)</td>
<td>index (x) W\textsubscript{T} (y,x)</td>
<td>index (x) μ(y,x) φ(y,x)</td>
<td>index (x) μ(y,x) φ(y,x) subtree_mincost (y,x) subtree_mincost (y,x)</td>
<td>size(x) index (x) v(y,x) ψ(y,x)</td>
<td></td>
</tr>
<tr>
<td>Broadcast Wave of Iteration(x) y ∈ T\textsubscript{x}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPT\textsubscript{UP(x)}</td>
<td>l_sol (y,x) subtree_mincost (y,x)</td>
<td>l_sol (y,x) subtree_mincost (y,x)</td>
<td>l_sol (y,x) subtree_mincost (y,x)</td>
<td>l_sol (y,x) subtree_mincost (y,x)</td>
<td>l_sol (y,x) subtree_mincost (y,x)</td>
<td>l_sol (y,x) subtree_mincost (y,x)</td>
</tr>
<tr>
<td>Convergecast Wave of Iteration(x) y ∈ T\textsubscript{x}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.8-1 : Variables in Messages of BSE
3.8.3. **Message Protocol**

We now show the protocol which guides the timing of the waves of BSE. Each process is either in the *preprocessing mode* or the *optimization mode*. We assume that the algorithm is initiated by \( r \), and that all processes are initially in the preprocessing mode. In the case of \( BSE_{diam} \), we also assume that \( r \) is the center of \( T \), as explained in Section 3.6.

Below, we list which messages each process must receive before sending a given message. Of course, a process cannot send any message until after it has computed the variables that it needs to include in that message; however, we have assumed that that computation is done instantly.

1. **Preprocessing Phase.** Processes retain all values computed or read during the preprocessing phase.
   
   a) (First Broadcast Wave.)
      
      i. \( r \) sends `PRE_DOWN_I` to its children.
      
      ii. For \( x \neq r \), when \( x \) receives `PRE_DOWN_I` from its parent, \( x \) sends `PRE_DOWN_I` to its children.

   b) (Convergecast Wave.)
      
      i. For \( x \neq r \), when \( x \) has received `PRE_DOWN_I` from its parent and `PRE_UP` from all its children, \( x \) sends `PRE_UP` to its parent.

   c) (Second Broadcast Wave.)
      
      i. When \( r \) receives `PRE_UP` from all its children, \( r \) sends
PRE_DOWN_II to all its children.

ii. For \( x \neq r \), when \( x \) receives PRE_DOWN_II from its parent, \( x \) sends PRE_DOWN_II to all its children and PRE_DONE to its parent.

d) (Cross.) For all \( x \), when \( x \) has received PRE_DOWN_II from its parent (if any) and PRE_DONE from all its children, \( x \) sends CROSS\( (x) \) to each of its cross neighbors. After \( x \) has sent CROSS to, and also received CROSS\( (x') \) from, each \( x' \in \text{Cross}_N(x) \), \( x \) enters the optimization mode.

2. Optimization Phase.

a) (Broadcast Wave.)

i. When \( r \) is in the optimization mode, \( r \) initiates Iteration\( (r) \) by sending OPT_DOWN\( (r) \) to all its children.

ii. For \( x \neq r \), when \( x \) has received OPT_DONE\( (p(x)) \) from its parent and is in the optimization mode, \( x \) initiates Iteration\( (x) \) by sending OPT_DOWN\( (x) \) to all its children.

iii. For \( y \neq r \), if \( y \) is in the optimization mode and \( y \) has received OPT_DOWN\( (x) \) from its parent, then \( y \) sends OPT_DOWN\( (x) \) to all its children.

b. (Convergecast Wave.)

i. For \( y \neq x \), when \( y \) has received OPT_DOWN\( (x) \) from its parent and OPT_UP\( (x) \) from all its children, \( y \) sends OPT_UP\( (x) \) to its parent and deletes all variables it has computed during
Iteration\(x\).

ii. When \(x\) receives OPT_UP\(x\) from all its children, \(x\) sends OPT_DONE \(x\) to all its children, then computes solution\(x\) and deletes all other variables it has computed during Iteration\(x\).

3.8.4. Computation of Variables

In Section 3.8.3, we did not mention the calculations a process must make before sending a message. We now explain those calculations in detail.

- Computation of \textit{depth}. Initially, \(\text{depth}(r) \leftarrow 0\). The message PRE_DOWN_I sent by a process \(x\) to its children contains the value of \(\text{depth}(x)\). When \(x\) receives PRE_DOWN_I from its parent, it computes \(\text{depth}(x) \leftarrow w(x, p(x)) + \text{depth}(p(x))\).

- Computation of \textit{height}, in all cases except BSE\textsubscript{dist} and BSE\textsubscript{wght}. The message PRE_UP sent by a process \(x\) to its parent contains \(\text{height}(x)\). If \(x\) is a leaf of \(T\), then \(\text{height}(x) \leftarrow 0\). Otherwise, when \(x\) receives PRE_UP from all its children, \(x\) computes

\[
\text{height}(x) \leftarrow \max \{w(x, y) + \text{height}(y) : y \in \text{Chldrn}(x)\}.
\]

- Computation of \textit{size}. The message PRE_UP sent by a process \(x\) to its parent contains \(\text{size}(x)\). If \(x\) is a leaf of \(T\), then \(\text{size}(x) \leftarrow 1\). Otherwise, when \(x\) receives PRE_UP from all its children, \(x\) computes

\[
\text{size}(x) \leftarrow 1 + \sum_{y \in \text{Chldrn}(x)} \text{size}(y).
\]

- Computation of \textit{index}. The message PRE_DOWN_{II} sent by a process \(x\) to each child \(y\) contains \(\text{index}(y)\), while \(r\) computes \(\text{index}(r) \leftarrow (1, \ldots, \ldots)\).
When a process $x$ knows the value of $\text{index}(x)$, then $x$ computes $\text{index}(y)$ for all $y \in \text{Chldrn}(x)$. If $x$ is not a leaf, let $\text{Chldrn}(x) = \{y_1, \ldots, y_m\}$. Then $x$ computes

$$\text{pre\_index}(y_i) = \text{pre\_index}(x) + 1 + \sum_{i \leq j < i} \text{size}(y_j)$$

$$\text{post\_index}(y_i) = \text{post\_index}(x) + 1 + \sum_{i < j \leq m} \text{size}(y_j)$$

and $\text{index}(y_i) = (\text{pre\_index}(y_i), \text{post\_index}(y_i))$.

- Computation of $\text{sum}$, in the case $\text{BSE}_{\text{sum}}$. The message $\text{PRE\_UP}$ sent by a process $x$ to its parent contains $\text{sum}(x)$. If $x$ is a leaf of $T$, then $\text{sum}(x) \leftarrow 0$. Otherwise, when $x$ receives $\text{PRE\_UP}$ from all its children, $x$ computes $\text{sum}(x) \leftarrow \sum_{y \in \text{Chldrn}(x)} (\text{size}(y) \cdot w(y, x) + \text{sum}(y))$.

- Computation of $\eta$, in the cases $\text{BSE}_{\text{max}}$, $\text{BSE}_{\text{diam}}$, and $\text{BSE}_{\text{sum}}$. The message $\text{PRE\_DOWN\_II}$ sent by a process $x$ to each child $y$ contains $\eta(y)$. At the beginning of the second preprocessing broadcast wave, $\eta(r) \leftarrow 0$. Each $x$ computes $\eta(y) \leftarrow \max_{y \in \text{Chldrn}(x) \setminus \{y\}} (w(x, z) + \text{height}(z))$ for each $y \in \text{Chldrn}(x)$. If $\text{Chldrn}(x) = \{y\}$, then $\eta(y) \leftarrow 0$.

- Computation of $\text{branch}$, $h_1$, $h_2$, $h_3$, $\text{local\_\mu}$, $\text{avoid}$, and $\text{local\_\phi}$ in the case of $\text{BSE}_{\text{diam}}$. Recall that $\text{branch}(r)$, $\text{local\_\mu}(r)$, $\text{avoid}(r)$, and $\text{local\_\phi}(r)$ are undefined, while $h_1$, $h_2$ and $h_3$ are constants; these are computed by $r$ and then sent to all other processes in the second
preprocessing down wave.

Let $\text{Chldrn}(r) = \{c_1, c_2, \ldots, c_m\}$ and $h_i = w(r, c_i) + \text{height}(c_i)$, indexed such that $h_i \geq h_{i+1}$ for all $1 \leq i < m$. If $m = 2$, we let $h_3 = 0$. When $r$ receives PRE_UP from all its children, it sends $h_1, h_2$ and $h_3$ to all its children in the message PRE_DOWN_II. Also, for each $i$, $r$ computes $\text{branch}(c_i) \leftarrow i$, $\text{local}_\mu(c_i) \leftarrow 0$, $\text{avoid}(c_i) \leftarrow 0$, and $\text{local}_\phi(c_i) \leftarrow \text{height}(c_i)$, and sends those values to $c_i$ in PRE_DOWN_II.

For $x \neq r$, when $x$ receives PRE_DOWN_II from $p(x)$, it has the values of $\text{branch}(x)$, $h_1$, $h_2$, $h_3$, $\text{local}_\mu(x)$, $\text{avoid}(x)$, and $\text{local}_\phi(x)$. For each $y \in \text{Chldrn}(x)$, $x$ sends the values $h_1$, $h_2$, $h_3$ to $y$ in the message PRE_DOWN_II, as well as the following values which $x$ computes:

1. $\text{branch}(y) \leftarrow \text{branch}(x)$.

2. $\text{local}_\mu(y) \leftarrow w(x, y) + \max \begin{cases} \text{local}_\mu(x), \\ \eta(y) \end{cases}$

3. $\text{avoid}(y) \leftarrow \max \begin{cases} \text{avoid}(x), \\ \text{depth}(x) + \eta(y) \end{cases}$

4. $\text{local}_\phi(y) \leftarrow \max \begin{cases} \text{local}_\mu(y), \\ \text{height}(y) \end{cases}$

- Computation of $\theta$, in the case $\text{BSE}_{\text{sum}}$. The message PRE_DOWN_II sent by a process $x$ to each child $y$ contains $\theta(y)$. At the beginning of
the second preprocessing broadcast wave, $\theta(r) \leftarrow 0$. Each $x$
computes $\theta(y) \leftarrow \sum_{y \in \text{Children}(x) \setminus \{y\}} (w(x, z) + \text{sum}(z))$ for each $y \in \text{Children}(x)$. If $\text{Children}(x) = \{y\}$, then $\theta(y) \leftarrow 0$.

- The message $\text{CROSS}(x)$ from a process $x$ to $x' \in \text{Cross}_N(x)$ contains information that $x'$ needs during the optimization phase. $\text{CROSS}(x)$ contains $\text{index}\ (x)$, and, in all cases except $\text{BSE}_{\text{wght}}$, it contains $\text{depth}(x)$. In the case of $\text{BSE}_{\text{diam}}$, it also contains $\text{branch}(x)$ and $\text{local}_\phi\ (x)$.

- The message $\text{OPT_DOWN}(x)$ from any process $y \in T_x$ to any $z \in \text{Children}(x)$ contains the value $\text{index}\ (x)$. In the case $\text{BSE}_{\text{sum}}$, the message also contains $\text{size}(x)$.

- Computation of $W_T(y, x)$, for $\text{BSE}_{\text{dist}}$ and $\text{BSE}_{\text{incr}}$. Each process $x$
computes $W_T(x, x) \leftarrow 0$. The message $\text{OPT_DOWN}(x)$ from any
process $y \in T_x$ to any $z \in \text{Children}(y)$ contains the value $W_T(y, x)$.
When $z \in \text{Children}(x)$ receives the message $\text{OPT_DOWN}(x)$ from $y$, then
$z$ computes $W_T(z, x) \leftarrow w(z, y) + W_T(y, x)$.

- Computation of $\mu(y, x)$ and $\phi(y, x)$, for $\text{BSE}_{\text{max}}$ and $\text{BSE}_{\text{diam}}$. Each
process $x$ computes $\mu(x, x) \leftarrow 0$. The message $\text{OPT_DOWN}(x)$ from any
process $y \in T_x$ to any $z \in \text{Children}(x)$ contains the value $\mu(y, x)$, and
$y$ computes $\phi(y, x) \leftarrow \max \{\mu(y, x), \text{height}(y)\}$.

- After $z \in \text{Children}(y)$ receives the message $\text{OPT_DOWN}(x)$ from $y$, then
$z$ computes $\mu(z, x) \leftarrow \max \{w(z, y) + \mu(y, x), \eta(z)\}$.
• Computation of \( \nu(y, x) \), for BSE\(_{\text{sum}}\). Each process \( x \) computes \( \nu(x, x) \leftarrow 0 \). The message OPT\_DOWN\( (x) \) from any process \( y \in T_x \) to any \( z \in \text{Child}(x) \) contains the value \( \nu(y, x) \). When \( z \in \text{Child}(y) \) receives the message OPT\_DOWN\( (x) \) from \( y \), then \( z \) computes \( \nu(z, x) \leftarrow w(z, y) \cdot (\text{size}(x) - \text{size}(z)) + \theta(z) + \nu(y, x) \).

• Computation of \( l_{\text{sol}}(y, x) \). For \( y \in T_x \), define \( \text{Swap}_N(y, x) = \{ y' \in \text{Cross}_N(y) : y' \notin T_x \} \). Recall that we can determine whether \( y' \in T_x \) by comparing \( \text{index}(y') \) and \( \text{index}(x) \), both of which are known to \( y \) after \( y \) receives CROSS from \( y' \); and either \( y = x \), or \( y \) has received OPT\_DOWN\( (x) \) from \( p(y) \). If \( \text{Swap}_N(y, x) = \emptyset \), \( y \) assigns \( l_{\text{sol}}(y, x) \) the default value 1. Otherwise, \( y \) computes \( l_{\text{sol}}(y, x) \), an ordered pair, in each case as given in Table 3.8-2.
Table 3.8-2: $l_{sol}(y, x)$

<table>
<thead>
<tr>
<th>Case</th>
<th>$l_{sol}(y, x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSE$_{dist}$</td>
<td>$\min {(W_T(y, x) + w(y, y') + depth(y'), {y, y'}) : y' \in Swap_N(y, x)}$</td>
</tr>
<tr>
<td>BSE$_{incr}$</td>
<td>$\min {(W_T(y, x) + w(y, y') + depth(y') - depth(x), {y, y'}) : y' \in Swap_N(y, x)}$</td>
</tr>
<tr>
<td>BSE$_{wght}$</td>
<td>$\min {(w(y, y'), {y, y'}) : y' \in Swap_N(y, x)}$</td>
</tr>
<tr>
<td>BSE$_{max}$</td>
<td>$\min {(\varphi(y, x) + w(y, y') + depth(y'), {y, y'}) : y' \in Swap_N(y, x)}$</td>
</tr>
<tr>
<td>BSE$_{diam}$</td>
<td>$\min {(\varphi(y, x) + w(y, y') + ecc_{T-x}(y'), {y, y'}) : y' \in Swap_N(y, x)}$ where $ecc_{T-x}(y')$ is as explained in Section 3.6</td>
</tr>
<tr>
<td>BSE$_{sum}$</td>
<td>$\min {(\psi(y, x) + size(x) \cdot (w(y, y') + depth(y')), {y, y'}) : y' \in Swap_N(y, x)}$</td>
</tr>
</tbody>
</table>

Value of $l_{sol}(y, x)$ computed by $y$ during $Iteration(x)$ of the optimization broadcast wave. Note that $l_{sol}(y, x)$ is an ordered pair, where the second member is a swap edge of $x$. In the case of BSE$_{diam}$, let $i = \text{branch}(y)$, $j = \text{branch}(y')$, and $k$ the smallest positive integer not equal to $i$ or $j$. If $i = j$, $ecc_{T-x}(y') = depth(y') + h_k$. If $i \neq j$, $ecc_{T-x}(y') = \max \{\text{local}_\mu(y'), depth(y') + h_k, depth(y') + avoid(x)\}$

- Computation of $subtree_{mincost}(y, x)$. After $y$ receives the message OPT_UP($x$) from all its children, $y$ computes $subtree_{mincost}(y, x)$
  $\leftarrow \min\{l_{sol}(y, x), \min_{z \in \text{Child}_\mu(y)} \text{subtree}_{mincost}(z, x)\}$

- Computation of $solution(x)$. After a process $x$ has received the message OPT_UP($x$) from all its children, $x$ computes $solution(x)$
  $\leftarrow subtree_{mincost}(x, x)$

### 3.8.5 Complexity

In this section, we prove that BSE satisfies the desired complexity bounds. Let $n$ be the number of processes of the network, $m$ the number of edges, and $\delta_x$ the degree of any given process $x$. Let $h$ be the hop-
height of $T$, and let $n^*$ be the number of edges of the transitive closure of $T$, i.e., the number of pairs $(y, x)$ such that $y \in T_x$. (Note that $n^* = O(nh)$.)

We measure space by number of items rather than number of bits. An item can be an integer, a distance (sum of weights of edges), or an ID of a process.

**Lemma 3.4**

(A) The time complexity of BSE is $O(h^2)$.

(B) The size of each message is $O(1)$.

(C) The space complexity of a process $x$ is $O(\delta_x)$.

(D) The number of messages in each channel at any given time does not exceed 1.

(E) The total number of messages sent during the execution of BSE is $O(m + n^*)$.

**Proof:** (A): Each wave moves at least one level up or down $T$ in each time unit, and hence finishes within $h$ time units, and there are $O(h)$ waves.

(B): The variables carried in a broadcast or convergecast wave are listed in Table 3.8-1. The message CROSS contains $O(1)$ variables, and the other messages carry no variables.

(C): A process $x$ needs $O(1)$ space to store the variables received from one member of Cross_N ($x$), and Cross_N ($x$) has cardinality at most $\delta_x$. The space needed by $x$ to store the computations of the preprocessing phase is $O(1)$. During each iteration of the optimization phase, $x$ stores
$O(1)$ temporary variables, but erases all of them at the end of the iteration. The only information that $x$ retains from the optimization phase is $\text{solution}(x)$, which takes $O(1)$ space.

(D): We consider an edge to have two channels, one in each direction. A cross edge channel carries only one message altogether. No wave can be started by a process until the previous wave has passed that process; this rule is enforced by the messages PRE_DONE and OPT_DONE($x$).

(E): The number of CROSS messages is $2m$. Each process, other than $r$ receives exactly one message of type PRE_DOWN_I, PRE_DOWN_II, and OPT_DONE, while each process, other than $r$, sends exactly one message of type PRE_UP and PRE_DONE. The number of message sent during the optimization broadcast waves totals $n^*$, as does the number of message sent during the optimization convergecast waves. The total number of messages is thus $2m + 2n^* + 5(n - 1)$.

### 3.9 Complexity Tradeoffs for BSE

There are tradeoffs between space and time complexities of BSE. For example, Gfeller et al. implement BSE$_{\text{diam}}$ in $O(h)$ time units, where the space complexity of each process $x$ is $O(h + \delta_x)$, and still have $O(m + n^*)$ messages, of size $O(1)$ each. Alternatively, by allowing messages of $O(h + \delta_x)$, the number of messages can be reduced to $O(n + m)$.

In CHAPTER 4, we introduce a new technique, which we call the critical level paradigm. This technique involves precomputation of $l_{\text{sol}}(y$,
for all $x$ and all $y \in T_x$, followed by identification of critical levels for each $y$. The values of $l_{sol}(y, x)$ are then deleted to save space; pipelining permits all calculations to be done without exceeding the $O(\delta_x)$ space capacity of each process $x$. In CHAPTER 5, we use this new paradigm to solve the all best swap edges problem in $O(h)$ time with $O(m + n^*)$ messages of size $O(1)$ each, and space complexity $O(\delta_x)$ for each $x$, such that no channel holds more than one message at a time. The solutions given in that section cover the measures $F_{dist}$, $F_{incr}$, $F_{wght}$, $F_{max}$, and $F_{diam}$, but not $F_{sum}$.
CHAPTER 4
THE CRITICAL LEVEL PARADIGM

In CHAPTERS 5 and 6, we present linear time algorithms for all measures given in Section 3.1, except \( F_{\text{sum}} \). In each case, we overcome the need for alternating broadcast and convergecast waves by making use of the concept of critical levels, which we introduce in Section 4.3 below. The heart of the critical level paradigm is that critical levels are pre-computed during the critical level phase of the algorithm, and that, during the optimization phase, a process uses its critical level to choose which of two candidate values to retain, without necessarily being able to evaluate both of them.

Critical levels are used in several different ways in the various linear time algorithms, sometimes in different ways within the same algorithm. Our linear time algorithms for \( F_{\text{dist}} \) and \( F_{\text{wght}} \) each use critical levels in just one way. However, our linear time algorithm for \( F_{\text{max}} \) uses critical levels in two different ways, and our linear time algorithm for \( F_{\text{diam}} \) uses critical levels in three different ways.

4.1 The Min-Max Problem

In general, the critical level paradigm is used when the goal is to find the minimum of maxima. We first consider a very simple application. Suppose we have a tree \( T \) of processes, rooted at \( r \), where each process \( x \) has a weight, \( F(x) \), and there is a non-negatively weighted edge, with
weight $w(x, y)$, between $x$ and $y$ if $x$ and $y$ are neighbors. We call this a *doubly weighted rooted tree*. We let $W(x, y)$ be the (weighted) length of the path in $T$ from $x$ to $y$.

We write $x \leq y$ if $x$ is an ancestor of $y$, $x < y$ if $x$ is a proper ancestor of $y$. If $x \leq y$, define $cost(x, y) = \max \{W(x, y), F(y)\}$. The output of the min-max problem is the value of $\min cost(x) = \min \{cost(x, y): x \leq y\}$.

**Required Output.** The required output for the minmax problem is for each process $x$ to compute $\min cost(x) = \min \{cost(x, y): y \geq x\}$. We define $best(x)$ to be equal to that $y \geq x$ for which $cost(x, y) = \min cost(x)$. It is a fairly straightforward to augment any algorithm that computes $\min cost(x)$, using well-known data structure techniques, so that it also computes $best(x)$. To simplify our exposition, we will not detail these augmentations.

We now consider two instances of the min-max problem. In Section 4.1.1, we consider an example where $T$ is a chain, while in Section 4.1.2, we consider a more general case of a doubly weighted rooted tree.

**4.1.1. Chain Example**

We first consider the special case that $T$ is a chain. Figure 4.1-1 shows an example.

![Figure 4.1-1: Doubly Weighted Chain](Image)

Figure 4.1-1: Doubly Weighted Chain
Figure 4.1-2: \( \text{cost}(x, y) \)
\( W(b, e) = 36, W(d, f) = 14, F(e) = 33, \) and \( F(f) = 16. \) Thus \( \text{cost}(b, e) = 36 \) and \( \text{cost}(d, f) = 16. \)

Figure 4.1-3: \( \text{best}(x) \) and \( \text{cost}(x, y). \)
Arrows indicate the choices of \( y = \text{best}(x) \) for each \( x. \) The arrows above the line indicate cases where \( \text{cost}(x, y) = W(x, y), \) while the arrows below the line indicate cases where \( \text{cost}(x, y) = F(y). \)

Figure 4.1-3 shows an arrow from \( x \) to \( \text{best}(x) \) for each \( x \) in the example shown in Figure 4.1-1. The values of \( \text{mincost}(x) \) and \( \text{best}(x) \) are shown in Table 4.1-1 below.

<table>
<thead>
<tr>
<th>( x )</th>
<th>( r )</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
<th>( e )</th>
<th>( f )</th>
<th>( g )</th>
<th>( h )</th>
<th>( i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{mincost}(x) )</td>
<td>39</td>
<td>38</td>
<td>36</td>
<td>26</td>
<td>16</td>
<td>11</td>
<td>10</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>( \text{best}(x) )</td>
<td>( b )</td>
<td>( e )</td>
<td>( e )</td>
<td>( g )</td>
<td>( g )</td>
<td>( g )</td>
<td>( g )</td>
<td>( i )</td>
<td>( i )</td>
<td>( i )</td>
</tr>
</tbody>
</table>
Table 4.1-1 : Values of \( \mincost(x) \) and \( \text{best}(x) \)

If \( T \) is a chain, we can reduce the min-max problem to the problem of finding all row minima of a triangular matrix. In Figure 4.1-4(a), we show the array of values of \( W(x, y) \) for all \( x \leq y \), while in Figure 4.1-4(b), we show the array of values of \( \text{cost}(x, y) \) for all \( x \leq y \), for our chain example. In both arrays, \( x \) is the row index and \( y \) is the column index. Then \( \mincost(x) \) is the minimum entry in row \( x \) the \( \text{cost} \) matrix, while \( \text{best}(x) \) is the index of the column in which that entry is found.

![Array W and cost matrices for chain example](image)

**Figure 4.1-4**: Array \( W \) is shown in (a), and \( \text{cost} \) in (b), for our chain example

### 4.1.2. General Tree Example

We now consider an instance of the min-max problem where \( T \) is not a chain, illustrated in Figure 4.1-5. The values of \( F \) are enclosed in the circles representing the vertices, and the edge weights are the labels on the edges. Each vertex is given a name, a letter in the range \( a...w \).
Figure 4.1-5: Doubly Weighted Tree.
The values of $F(x)$ are written inside the circles representing the processes, and the edge between processes $x$ and $y$ is labeled with the value $w(x, y)$.

We list just a few results for that example:

$$\text{best}(r) = j$$

$$\text{mincost}(r) = \max \{W(r, j), F(j)\}$$

$$= \max \{10, 5\} = 10$$

$$\text{best}(g) = i$$

$$\text{mincost}(g) = \max \{W(g, i), F(i)\}$$

$$= \max \{11, 22\} = 22$$

A more extensive summary of the results will be given in Table 4.2-1 below.

4.2 Quadratic Time Algorithm

We can easily solve the min-max problem with a distributed algorithm whose time complexity is $O(h^2)$, and whose space complexity per process is $O(1)$ per process. For any $x$, during the first wave of Iteration($x$) of the
algorithm, each process \( y \geq x \) calculates \( W(x, y) \) and \( \text{cost}(x, y) \), and then sends these values to \( \text{Chldrn}(y) \). During the second wave, each \( y \geq x \) calculates an intermediate value of \( \text{mincost}(x) \), which is the minimum choice of \( \text{cost}(x, z) \) for all \( z \geq y \). If \( y > x \), then \( y \) sends the intermediate value up to \( p(y) \), while if \( y = x \), the intermediate value is the final value of \( \text{mincost}(x) \). All intermediate values calculated during this wave, other than \( \text{mincost}(x) \) itself, are deleted to make room for the intermediate values of subsequent waves.

We now give the code for the quadratic time algorithm in algorithmic form.

Define \( \text{subtree}_\text{mincost}(x, y) = \min_{z \geq y} \text{mincost}(x, z) \), the best candidate for \( \text{mincost}(x) \) among the processes \( z \geq y \). During the broadcast wave of Iteration\((x)\), the values of \( W(x, y) \) and \( \text{cost}(x, y) \) are computed for all \( y \) in increasing order. During the convergecast wave of Iteration\((x)\), the values of \( \text{subtree}_\text{mincost}(x, y) \) are computed for all \( y \), in decreasing order. Finally, when \( y = x \) in the convergecast wave, \( \text{mincost}(x) \) is known.

1: \textbf{for} all \( x \) in top down order \textbf{do} (Iteration \((x)\))
2: \hspace{1cm} \( W(x, x) \leftarrow 0 \)
3: \hspace{1cm} \( \text{cost}(x, x) \leftarrow F(x) \)
4: \hspace{1cm} \textbf{for} all \( y \) such that \( y > x \) in top down order \textbf{do} (Broadcast Wave)
5: \hspace{2cm} \( W(x, y) \leftarrow W(x, p(y)) + w(p(y), y) \)
6: \hspace{2cm} \( \text{cost}(x, y) \leftarrow \max W(x, y), F(y) \)
7: \hspace{1cm} \textbf{end for}
8: \hspace{1cm} \textbf{for} all \( y \) such that \( y \geq x \) in bottom up order \textbf{do} (Convergecast Wave)
9: \[ \text{subtree_mincost}(x, y) \leftarrow \min \left\{ \begin{array}{l}
\text{cost}(x, y) \\
\min \{ \text{subtree_mincost}(x, z) : z \in \text{Chldrn}(y) \} 
\end{array} \right\} \]

10: \textbf{end for}

11: \[ \text{mincost}(x) \leftarrow \text{subtree_mincost}(x, x) \]

12: \textbf{end for}

Table 4.2-1: Quadratic Time Algorithm for the Min-Max Problem

In Table 4.2-1, we omit the message passing details. As in our implementation of BSE, \( \text{Iteration}(x) \) does not begin until \( \text{Iteration}(p(x)) \) is done, and the convergecast wave of each iteration does not begin until the broadcast wave is done. Figure 4.3-2(a) shows the pattern of these waves where \( h = 5 \).

4.3 Critical Levels and the Linear Time Algorithm

We now give a distributed algorithm for the min-max problem whose time complexity is linear, \( \text{i.e.}, O(h) \), although the space complexity is still \( O(1) \) per process. In order to accomplish this speed up, we reorganize the order of computation, and introduce the concept of a \textit{critical_level}.

Define \( \text{critical_level}(y) = \min \{ \text{level}(x) : \text{cost}(x, y) = F(y) \} \). Table 4.3-1 gives the critical level of each process in the chain example.

<table>
<thead>
<tr>
<th>( y )</th>
<th>( r )</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
<th>( e )</th>
<th>( f )</th>
<th>( g )</th>
<th>( h )</th>
<th>( i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{critical_level}(j)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 4.3-1: Critical Levels
It is relatively easy to visualize the meaning of critical levels, when cost is given in matrix form, as in Figure 4.3-1

<table>
<thead>
<tr>
<th>x \ y</th>
<th>r</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>42</td>
<td>43</td>
<td>39</td>
<td>57</td>
<td>67</td>
<td>75</td>
<td>76</td>
<td>81</td>
<td>84</td>
<td>86</td>
</tr>
<tr>
<td>a</td>
<td>43</td>
<td>38</td>
<td>57</td>
<td>45</td>
<td>53</td>
<td>60</td>
<td>59</td>
<td>62</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>38</td>
<td>57</td>
<td>37</td>
<td>36</td>
<td>60</td>
<td>42</td>
<td>45</td>
<td>47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>57</td>
<td>37</td>
<td>33</td>
<td>60</td>
<td>26</td>
<td>32</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>37</td>
<td>33</td>
<td>60</td>
<td>16</td>
<td>32</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>33</td>
<td>60</td>
<td>16</td>
<td>32</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>60</td>
<td>16</td>
<td>32</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>16</td>
<td>32</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>32</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.3-1: cost matrix.
The cost matrix shown in Figure 5.4(b). For each \( y, x = \text{critical_level}(y) \) is the smallest \( x \) such that \( \text{cost}(x, y) = F(y) \), as indicated by a box around one entry in each column.

The significance of critical levels is that they allow us to speed up the distributed algorithm for the min-max problem by an order of magnitude. The critical level paradigm depends on a few simple results, given below.

**Lemma 4.1**: If \( x_1 < x_2 \leq y \) and \( \text{cost}(x_1, y) = F(y) \), then \( \text{cost}(x_2, y) = F(y) \).

**Proof**: Suppose \( \text{cost}(x_1, y) = F(y) \); then \( F(y) - W(x_1, y) \geq 0 \). Thus \( F(y) - W(x_2, y) = F(y) - W(x_1, y) + W(x_1, x_2) \geq W(x_1, x_2) \geq 0 \).

**Corollary 4.2**: If \( \text{critical_level}(y) \leq x \leq y \), then \( \text{cost}(x, y) = F(y) \).

We give the code for the linear time algorithm for the min-max
problem in algorithmic form in Table 4.3-2. The algorithm consists of
two phases. During the critical level phase, $W(x, y)$ is computed for all $x \leq y$, and $\text{critical}_{-}\text{level}(y)$ is computed for all $y$. However, the values of $W$ are deleted as soon as they are no longer needed in order to save space.

For each $x$, the entries of $W(x, y)$ are computed in increasing order of $y$, i.e., left to right in Figure 4.1-4(a). The rows are chosen in decreasing order, i.e., bottom to top. If $W(x, y) \leq F(y)$, the value of $\text{critical}_{-}\text{level}(y)$ is set to $\text{level}(x)$.

For each $y$, the value of $\text{critical}_{-}\text{level}(y)$ can be set several times, but the last value is the correct one. The final values of $\text{critical}_{-}\text{level}(y)$ for our example are shown in Table 4.3-2 below.

The iterations are pipelined as soon as Iteration($p(x)$) has passed process $x$, Iteration($x$) can begin. The waves of the iterations do not have to be synchronous, but they must not collide; process $y$ sends a message to $p(y)$ when it is done with Iteration($p(x)$), permitting $p(y)$ to send its message for Iteration($x$). Thus, all iterations of the phase can be completed within $2h$ time units, where $h$ is the height of $T$.

The optimization phase of the linear time algorithm consists of a convergecast wave, Iteration($x$), for each $x$. The order of computation is the opposite of that of the critical level phase. The rows are done in bottom-up (decreasing $x$) order, and each row is done in top down (increasing $y$) order, i.e., left to right in the matrix shown in Figure 4.1-4 in the chain case). During Iteration($x$), a process $y$ computes two values,
subtree_minF (x, y) and subtree_minW (x, y). The heart of the critical level paradigm is the fact that each \( z \geq y \) contributes either to the computation of subtree_minF (x, y) or subtree_minW (x, y), but not both; it decides which one by examining its critical level, and that it can make this choice without necessarily knowing both candidate solutions. More specifically:

\[
\begin{align*}
\text{subtree_minF (x, y)} & = \min \{ F (z): y \leq z \text{ and } \text{critical level}(z) \leq \text{level}(x) \} \\
\text{subtree_minW (x, y)} & = \min \{ W (y, z): y \leq z \text{ and } \text{critical level}(z) > \text{level}(x) \}
\end{align*}
\]

Note that subtree_minW (x, y) = W (y, z) instead of W (x, z), which might be the actual value of mincost(x). This is because \( y \) lacks the information to compute W (x, z). When \( W (y, z) \) is sent to \( p(y) \), then, if \( p(y) \) decides to keep that value, it adds \( W (p(y), y) \) to that value. If it turns out that \( z = \text{best}(x) \) and \( \text{cost}(x, z) = W (x, z) \), then subtree_minW (x, x) will equal W (x, z), which is the correct choice of mincost(x).

The waves of the optimization phase are pipelined in the same manner as those of the critical level phase, and thus that phase takes no more than \( 2h \) time units. Figure 4.3-2 consists of simplified sketches comparing the wave structures of the quadratic time and the linear time algorithms, in the case that \( T \) is a chain.
1: { begin first phase of the linear algorithm }
2: for all x in top down order do
3: compute level(x)
4: end for
5: for all x in bottom up order do
6: e ← level(x)
7: for all y in T, in top down order do
8: delete W(y, p(x)) { if it exists, to save space }
9: compute W(y, x)
10: if F(y) ≥ W(y, x) then
11: critical level(y) ← e { overwrites any prior value of critical level(y) }
12: end if
13: end for
14: end for
15: { begin second phase of the linear algorithm }
16: for all e in increasing order do
17: for all y such that level(y) ≥ e in bottom-up order do
18: delete subtree_minF(w, e − 1) and subtree_minW(y, e − 1) { if they exist, to save space }
19: if level(y) ≤ e then
20: subtree_minF(y, e) ← min
21: subtree_minW(y, e) ← min
22: subtree_minF(y, e) ← min
23: subtree_minW(y, e) ← 0
24: end if
25: if level(y) = l then
26: mincost(Ty) ← min
27: end if
28: end for
29: end for
30: end for

Table 4.3-2: Linear Time and Space Algorithm for the Min-Max Problem

- Line 2 computes level(x) in a straightforward top-down wave:
  
  \( level(r) = depth(r) = 0, \) and \( level(x) = 1 + level(p(x)) \) for \( x ≠ r. \)

- Lines 5–14 give the code for the main loop of the first phase,
which computes all values of \textit{critical\_level}. The values of \( W(y, x) \) are computed for this purpose, but are then deleted to save space. If they were all retained, the space complexity of the algorithm would be \( O(h) \) per process, where \( h \) is the height of \( T \). Any value of \( W(y, x) \) which is part of the final solution will be recomputed during the second phase.

- In Line 9, \( W(x, x) \leftarrow 0 \), and otherwise \( W(y, x) \leftarrow W(y, p(y)) + W(p(y), x) \).
- The heart of the linear time and space algorithm is the fact that \( F(y) \geq W(y, x) \) if and only if \( \text{critical\_level}(y) \leq \text{level}(x) \). The first phase calculates all values of \textit{critical\_level} with \( O(h^2) \) calculations, organized into \( O(h) \) waves which take \( O(h) \) time each. Using pipelining, all these waves are completed in \( O(h) \) time. By erasing values computed by each wave, other than the values of \textit{critical\_level} itself, we save space, and maintain space complexity of \( O(1) \) for each process.
- The value of \textit{critical\_level}(y) can be reset during any number of iterations of the main loop of the first phase. The correct value will be the value assigned during the last iteration for which \( F(y) \geq W(y, x) \).
- For each \( y \), the values of \textit{subtree\_min}\( F(y, e) \) and \textit{subtree\_min}\( W(y, e) \) are computed during every iteration for which \( e \leq \text{level}(y) \). In Line 19, we delete the values computed during the previous
iteration, to save space.

• In Line 27, we assign the final value for each \( y \), when \( e = \text{level}(y) \).

At this point, we offset the value of \( \text{subtree\_minW}(y, e) \) by subtracting \( \text{depth}(y) \).

The optimization phase of the linear time and space algorithm does not begin until the critical level phase is done. Within each phase, the waves (Iterations) are pipelined so that, even though there are \( h + 1 \) waves which take \( O(h) \) time each, After each process \( y = r \) executes its action for Iteration(\( x \)) of the critical level phase, \( y \) sends a message telling \( p(y) \) that it is ready to participate in Iteration(\( p(x) \)). Thus, the waves of those two iterations do not collide. Since every message must be delivered within one time unit, all iterations of the critical level phase are completed within \( 2h \) time units. We can similarly ensure that the waves of the optimization phase also do not collide, and that that phase is completed within \( 4h \) time units.

Figure 4.3-2(b) shows the pattern of these waves where \( h = 5 \).
Figure 4.3-2: Comparison of the algorithms.
Comparison of quadratic time algorithm (a) and the linear time
algorithm (b). The algorithms have the same number and length of
waves, but the linear time algorithm uses pipelining in a way that
cannot be done by the quadratic time algorithm without overlapping.

Correctness of the linear time algorithm for the min-max problem
follows from Corollary 4.2 and from Lemma 4.3 below.

**Lemma 4.3** Suppose \((a_1, \ldots, a_m)\) and \((b_1, \ldots, b_m)\) are sequences of elements
of an ordered set. Let \(c_i = \max \{a_i, b_i\}\) for all \(1 \leq i \leq n\). Let \(A = \{1 \leq i \leq m : a_i \geq b_i\}\), and \(B = \{1 \leq i \leq m : a_i < b_i\}\). Let \(M_A = \min \{a_i : i \in A\}\) and \(M_B = \min \{b_i : i \in B\}\). If \(A = \emptyset\), let \(M_A = \infty\) by default, while if \(B = \emptyset\), let \(M_B = \infty\). Then
\[
\min_{1 \leq i \leq m} c_i = \min \{M_A, M_B\}.
\]

**Proof:** If \(A = \emptyset\), then \(c_i = b_i\) for all \(i\), \(M_A = \infty\), and \(M_B = M\), and thus we are done. If \(B = \emptyset\), we are done by a similar argument.

Otherwise, pick \(1 \leq i, j, k \leq m\) such that

- \(i \in A\) and \(a_i = M_A\),
- \(j \in B\) and \(b_j = M_B\),
• $c_k = M$.

Then $M = c_k \leq c_i = a_i = M_A$, and $M = c_k \leq c_j = b_j = M_B$. Thus, $M \leq \min\{M_A, M_B\}$.

To prove the converse, suppose that $M < \min\{M_A, M_B\}$. If $k \in A$, then $M = c_k = a_k \geq a_i = M_A$, contradiction. On the other hand, if $k \in B$, then $M = c_k = b_k \geq b_j = M_B$, contradiction. This completes the proof of Lemma 4.3.

**Lemma 4.4** The linear time algorithm for the min-max problem is correct.

**Proof:** We first note that $\text{critical\_level}(y)$ is defined for each process $y$, since $W(y, y) = 0 \leq F(y)$.

We next show that the value of $\text{critical\_level}(y)$, which is stored by the process $y$, is correct after completion of the critical level phase. Let $e$ be the true value of $\text{critical\_level}(y)$, and let $x$ be the ancestor of $y$ whose level is $l$. By definition, $F(y) \geq W(x, y)$, and thus $\text{critical\_level}(y) \leftarrow l$ during Iteration($x$) of the critical level phase. Also by definition, $F(y) < W(x', y)$ for all $x' < x$, and thus $\text{critical\_level}(y)$ will not be reset during any subsequent iteration.

By Corollary 4.2, $\text{cost}(x, y) = \begin{cases} F(y) & \text{if } x \geq \text{critical\_level}(y) \\ W(x, y) & \text{otherwise} \end{cases}$

We now apply Lemma 4.3. We can conclude that the linear time algorithm computes the correct value of $\text{mincost}(i)$.

In Table 4.3-3, we give the input, output, and some intermediate
values calculated by the linear time and space algorithm for the instance shown in Figure 4.1-1. We define $\text{choice}(x) \in \{F, W\}$. If $\text{mincost}(x) = F(\text{best}(x))$, then $\text{choice}(x) = F$. Otherwise, $\text{choice}(x) = W$.

Table 4.3-3: Input, output, and some intermediate values for the example instance shown in Figure 5.6.
In Figure 4.3-3, we show the same tree, where an arrow points from \( x \) to \( \text{best}(x) \) for each \( x \). The label on the arrow is \( \text{mincost}(x) \).

Figure 4.3-3: \( \text{mincost}(x) \) and \( \text{best}(x) \)
Arrows are from \( x \) to \( \text{best}(x) \), and the label on that arrow is \( \text{mincost}(x) \).
CHAPTER 5
LINEAR TIME ALGORITHMS

Gfeller et al. [9] give an $O(h)$-time algorithm for the all best swap edge problem, in the case of $F_{\text{diam}}$. However, their algorithm uses $O(h + \delta x)$ space for a process $x$.

In CHAPTERS 5 and 6, we present $O(h)$-time algorithms for the all best swap edges problem, for five of the six measures defined in Section 3.1.

We call these algorithms $\text{LINEAR}_{\text{dist}}$, $\text{LINEAR}_{\text{incr}}$, $\text{LINEAR}_{\text{wght}}$, $\text{LINEAR}_{\text{max}}$, and $\text{LINEAR}_{\text{diam}}$, respectively, and all can be considered to be versions of a general algorithm, which we call $\text{LINEAR}$. The space complexity of each of these five algorithms is $O(\delta x)$, i.e., each process $x$ requires only enough space to store $O(\delta x)$ variables (where each variable is an integer or a weight) at any given time. In each case, we achieve the speed-up by one or more applications of the critical level paradigm introduced in CHAPTER 4.

- Each of the five algorithms uses the critical level paradigm to compute $\text{rank}(y, y')$ for every cross edge $\{y, y'\}$ of $T$. This is the only use of the paradigm by $\text{LINEAR}_{\text{wght}}$, $\text{LINEAR}_{\text{dist}}$, and $\text{LINEAR}_{\text{incr}}$.
- $\text{LINEAR}_{\text{max}}$ and $\text{LINEAR}_{\text{diam}}$ use the critical level paradigm to compute $\text{critical\_level}(x)$ for each process $x$. We explain that computation in this Section 6.2.
- $\text{LINEAR}_{\text{diam}}$ uses the critical level paradigm to compute
special_level(x), which is another version of critical level. We explain that computation in Section 6.6.

At this point, the reader may ask what, exactly, the critical level paradigm is; and what, in particular, qualifies a function to be called a critical level?

We do not give a complete theoretical treatment of critical levels in this thesis. However, in general, a critical level function is a function that can be computed top down, which enables another function, whose computation would otherwise require independent top down followed by bottom up waves for all processes, to be computed in a single bottom up wave for each process, thus allowing the waves to be pipelined. All three of the functions used in this thesis, namely rank, critical level, and special level, fit this definition.

Each of the five versions of LINEAR consists of at least three phases. The first phase of each algorithm is preprocessing, and the last is optimization. Each of the algorithms also includes one phase for each of the one, two, or three critical level computations.

We will reuse as much notation from Section 3.1 as possible. In each of our versions of LINEAR, the preprocessing phase computes many of the same variables computed in the corresponding version of BSE.

5.1 LINEAR\textsubscript{dist} and LINEAR\textsubscript{incr}

We do not need to give separate code for LINEAR\textsubscript{incr}, since the all best
swap edges problem for $F_{\text{incr}}$ reduces to the problem for $F_{\text{dist}}$ in a trivial way, as stated in Lemma 3.1.

5.2 Overview of LINEAR

We give the general code for LINEAR in Table 5.2-1 below. Each of our four remaining linear time algorithms is a special case of LINEAR.

<table>
<thead>
<tr>
<th>1: Preprocessing phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>2: Ranking phase</td>
</tr>
<tr>
<td>3: (Possibly other critical level phases)</td>
</tr>
<tr>
<td>4: for $1 \leq l \leq d$ do</td>
</tr>
<tr>
<td>5: for all $y$ such that $\text{level}(y) \geq l$ in bottom up order do (Wave $l$)</td>
</tr>
<tr>
<td>6: Compute $\text{up_package}(y, l)$.</td>
</tr>
<tr>
<td>7: if $\text{level}(y) = l$ then</td>
</tr>
<tr>
<td>8: Compute $\text{swap_edge_cost}(y)$.</td>
</tr>
<tr>
<td>9: end if</td>
</tr>
<tr>
<td>10: end for</td>
</tr>
<tr>
<td>11: end for</td>
</tr>
</tbody>
</table>

Table 5.2-1 : LINEAR

Each of the phases uses $O(1)$ space per process, except for the ranking phase, which uses $O(\delta x)$ space for each process $x$. The overall space complexity of LINEAR is thus $O(\delta x)$ for each $x$.

5.3 The Preprocessing Phase

In the preprocessing phase (Line 1 of Table 5.2-1), each process $x$ computes and retains a list of variables, many of which are the same as for BSE. The exact list depends on which version of LINEAR, but the list
always includes \( \text{index}(x) \). In the subsection devoted to each individual algorithm, we list the variables computed by that phase.

5.4 The Ranking Phase

In the second phase (Line 2 of Table 5.2-1), the rank of every cross edge is computed. The rank of a cross edge \( \{x, x'\} \) is defined to be the level of the nearest common ancestor, in \( T \), of \( x \) and \( x' \), and is stored by both \( x \) and \( x' \).

Figure 5.4-1 shows an example network, where tree edges are in bold and cross edges are dashed. The level of each process is indicated, and the rank of each cross edge is indicated in color.

![Figure 5.4-1: Levels of processes (black) and ranks of cross edges (red). The purpose of computing ranks is to allow us to more easily identify the swap edges of a given process, as stated by Remark 5.1 below.](image)
**Remark 5.1** If \( x \neq r \) is a process and \( e' = \{z, z'\} \) is a cross-edge, where \( z \in T_x \), then \( e' \) is a swap edge for \( x \) if and only if \( \text{rank}(z, z') < \text{level}(x) \).

For each \( 0 \leq l \leq d \), a top-down wave, which we call Wave \( l \), contains the index of the \( \text{ancestor_index}(y) = \text{index}(x) \) if \( x \) is the ancestor of \( y \) at level \( l \). That wave assigns the value \( l \) to the rank of any cross edge \( e' = \{y, y'\} \) which are swap edges of \( x \). At the next wave, the value of \( \text{rank}(y, y') \) could be reassigned, but the last value of \( \text{rank}(y, y') \) assigned will be the true value.

All values computed during the ranking phase are deleted as soon as they are no longer needed; only the ranks of the edges are retained. The rank of each cross edge will be computed and stored twice, once for each end of that edge. The values computed by the two ends will be the same.

1: for \( 0 \leq l \leq d \) in increasing order do \( \{\text{Wave } l\} \)
2:  for all \( y \) such that \( \text{level}(y) \geq l \) in top-down order do
3:   if \( \text{level}(y) = l \) then
4:     \( \text{ancestor_index}(y, l) \leftarrow \text{index}(y) \)
5:   else
6:     \( \text{ancestor_index}(y, l) \leftarrow \text{ancestor index}(p(y),l) \)
7:   end if
8:  for all cross edges \( \{y, y'\} \) do
9:    if \( \text{index}(y') \geq \text{ancestor index}(y, l) \) then
10:   \( \text{rank}(y, y') \leftarrow l \)
11: end if
12: end for
13: end for

Table 5.4-1 : Ranking Phase

**Remark 5.2** If \( \text{rank}(x, x') = l \), then, for all \( l' \leq l \), the computed value of \( \text{rank}(x, x') \) will be set to \( l' \) during Wave \( l' \), and thus the final computed
value of \( \text{rank}(x, x') \) will be \( l \).

5.5 Optimization Phase

The code for the optimization phase is given in Lines 4–11 of Table 5.2-1. The list of variables in \( \text{up\_package}(y, l) \) depends on the version of LINEAR. In each case, each process \( y \) is able to compute \( \text{up\_package}(y, l) \) by using the variables stored at \( y \) during the earlier phases, as well as the variables of \( \text{up\_package}(z, l) \) for all \( z \in \text{Chldrn}(y) \).

5.6 LINEAR\text{dist}

The preprocessing phase of LINEAR\text{dist} computes \( \text{size}(x) \), \( \text{index}(x) \), \( \text{level}(x) \), and \( \text{depth}(x) \) for all \( x \). These variables are computed in the same manner as given in Section 3.1. More specifically, \( \text{depth} \) and \( \text{level} \) are computed in a top down wave, \( \text{size} \) by a subsequent bottom up wave, and then \( \text{index} \) by another top down wave.

For any \( l \) and any process \( y \) such that \( \text{level}(y) \geq l \), \( \text{up\_package}(y, l) \) consists of only one variable, namely \( \text{subtree\_mincost}(y, l) \). Let \( x \) be the ancestor of \( y \) at level \( l \), and let \( e = \{x, p(x)\} \). Then \( \text{subtree\_mincost}(y, l) \) is defined to be the minimum, over all \( e' \in \text{SwapEdges}(e) \) such that \( e' \) has one end in \( T_y \), of the length of the path in \( T' = T - e + e' \) from \( y \) to \( r \). Code for the computation of \( \text{subtree\_mincost}(y, l) \) is given in Table 6.3.

\begin{verbatim}
1: Swap N (y, l) ← {y’ : \{y’,y\} is a cross edge and rank(y, y’) > l}
2: for all y’ such that y’ ∈ Swap N (y, l) do
3:     cost(y, y’) ← w(y, y’) + depth(y’)
4: end for
\end{verbatim}
\[ \text{subtree_mincost}(y, l) \leftarrow \min \left\{ \min \{ w(z, z') + \text{subtree_mincost}(z, l) : z \in \text{Chldrn}(y) \} : z' \in \text{Swap}(y, l) \right\} \]

Table 5.6-1: Computation of \text{subtree_mincost}(y, l) in \text{LINEAR}_{\text{dist}}

The final step of Wave \( l \) is to compute swap edge \( \text{cost}(y) \) to be \( \text{subtree_mincost}(y, l) \) for all \( y \) such that \( \text{level}(y) = l \).

5.7 \text{LINEAR}_{\text{wght}}

\text{LINEAR}_{\text{wght}} is by far the simplest version of \text{LINEAR} we consider. The preprocessing phase computes only \( \text{size}(x) \) and \( \text{index}(x) \) for each \( x \), and \( \text{up\_package}(y, l) \) consists of only one variable, namely \( \text{subtree_mincost}(y, l) = \min \{ w(z, z') : z \in T_y \text{ and } z' \in \text{Swap}(z, l) \} \), which is computed by

\[ \text{subtree_mincost}(y, l) \leftarrow \min \left\{ \min \{ w(y, y') : y' \in \text{Swap}(y, l) \} \right\} \]

\]\[ \text{subtree_mincost}(y, l) \leftarrow \min \left\{ \min \{ w(z, z') + \text{subtree_mincost}(z, l) : z \in \text{Chldrn}(y) \} : z' \in \text{Swap}(y, l) \right\} \]

81
CHAPTER 6

LINEAR$\text{max}$ and LINEAR$\text{diam}$

In this section, we describe LINEAR$\text{max}$ and LINEAR$\text{diam}$, which have a great deal of common computation.

Suppose $\text{level}(x) = l$, $e = \{x, p(x)\}$, $y \in T_x$, and $e' = \{y, y'\} \in \text{SwapEdges}(e)$. Recall that

$$F_{\text{max}}(T, r, e, e') = \text{ecc}_T(x) + \text{w}(y, y') + \text{depth}(y')$$

and

$$F_{\text{diam}}(T, r, e, e') = \text{ecc}_T(x) + \text{w}(y, y') + \text{ecc}_{T-x}(y')$$

Recall that $T-x$ is the subgraph of $T$ obtained by deleting the vertices of $T_x$ as well as the edge $e$.

In both LINEAR$\text{max}$ and LINEAR$\text{diam}$, we would like to compute $F_{\text{max}}(T, r, e, e')$ or $F_{\text{diam}}(T, r, e, e')$, respectively, when Wave$(l)$ of the optimization phase reaches $y$. In BSE, this is no problem, because the broadcast portion of Wave$(l)$ has brought $\text{down\_package}(y, x)$, which contains the data that $y$ needs to compute the function. However, for LINEAR, there is no down package. At the time $y$ wants to compute the value of the function, it does not even know the identity of $x$ (although it knows $l$).

Our first problem, common to both algorithms, is to determine whether a given cross edge is a member of $\text{SwapEdges}(x)$. Just as in CHAPTER 5, we execute the ranking phase, whose code is given in Table 5.4-1 before the optimization phase. That phase assigns a rank to every cross edge such that $\{y, y'\} \in \text{SwapEdges}(l)$ if and only if $\text{rank}(y, y') < l$. 82
Our second problem, also common to both LINEAR\textsubscript{max} and LINEAR\textsubscript{diam}, is to compute $ecc_{T_x}(y)$. Recall, from Section 3.1, that

$$
ecc_{T_x}(y) = \max\left\{ \text{height}(x) \right. \left\mid \mu(y, x) \right\}$$

where $\mu(y, x)$ is the maximum length of any path in $T_x$ from $y$ to any point of $T_x - T_y$, as defined in Section 3.5 and illustrated in Figure 3.5-3.

Because LINEAR uses only constant space per edge, we cannot store enough information for $y$ to know $\mu(y, x)$ for all choices of $x$. We solve this problem using the paradigm described in CHAPTER 4, executing a \textit{critical level} phase before the optimization phase; this phase erases all its computation except for one number, called the \textit{critical level}, at each process. Using that value, $y$ decides whether $ecc_{T_x}(y) = \text{height}(y)$. If so, there is no problem, since all values of $\text{height}$ are computed during the preprocessing step. Otherwise, $y$ cannot compute $ecc_{T_x}$ directly, but rather, sends enough information up the tree to enables $x$ to compute that value, if needed, when the wave reaches $x$.

Our third, and most difficult, problem is encountered only for LINEAR\textsubscript{diam}, and that is to compute $ecc_{T^-x}(y')$. (The last term of the formula for $F_{\text{max}}$ is $\text{depth}(y')$, which is computed during preprocessing.) Once again, we are able use the critical level paradigm to define the \textit{special level} (which is also a critical level, using other criteria) for each
process so that it is possible for \( y \) compute enough information, and pass that information up the tree, for \( x \) to be able to compute \( \text{ecc}_{T-x}(y') \) if \( e' \) is the best swap edge for \( e \). We give the details in Section 6.4

6.1 LINEAR\(_{\text{max}}\)

The preprocessing phase of LINEAR\(_{\text{max}}\) computes the following variables for each process \( x \).

1. \( \text{size}(x), \text{index}(x), \text{height}(x), \text{depth}(x), \text{level}(x), \text{and} \ \eta(x) \), which have the same definitions as given in Section 3.1.

2. \( \text{best\_child}(x) \), the best child of \( x \), defined to be that \( y \in \text{Chldrn} \) such that \( w(x, y) + \text{height}(y) > w(x, z) + \text{height}(z) \) for any other child \( z \) of \( x \). Note that, since we use a strict inequality in this definition, a process can have at most one best child. If \( \text{Chldrn}(x) = \emptyset \), or if there is more than one choice of \( y \) for which \( w(x, y) + \text{height}(y) \) is maximum, \( \text{best\_child}(x) \) is undefined.

3. We define \( \text{Normal\_Chldrn}(x) \) to be the set of all normal children of \( x \), namely all children which are not the best child of \( x \).

4. \( \text{secondary\_down\_path}(x) \) is defined to be the longest path in \( T_x \) that starts at \( x \) does not pass through \( \text{best\_child}(x) \). If \( \text{Normal\_Chldrn}(x) = \emptyset \), we define \( \text{secondary\_down\_path}(x) \) to be the trivial path at \( x \). Let \( \text{secondary\_height}(x) = W_T(\text{secondary\_down\_path}(x)) \).

Note that all of the above variables can be computed with definitely
many broadcast and convergecast waves in $O(h)$ total time.

6.2 Computing $\text{ecc}_{Tx}(y)$

We first introduce some additional notation.

- $\text{Spine}(x) = \{y \in T_x : \text{ecc}_{Tx}(y) = \text{depth}(y)\}$.
- $\text{Spine}(l) = \{\text{Spine}(x) : \text{level}(x) = l\}$.
- $\text{Up}(x) = T_x - \text{Spine}(x)$.
- $\text{Up}(l) = \{\text{Up}(x) : \text{level}(x) = l\}$.

**Lemma 6.1** For any process $x$

(a) If $y \in \text{Spine}(x)$ and $y = x$, then $p(y) \in \text{Spine}(x)$ and $y = \text{best child}(p(y))$.

(b) $\text{Spine}(x)$ is a chain.

**Proof:** We first prove (a). Suppose $p(y) \in \text{Spine}(x)$. Let $\sigma$ be the longest path in $T_x$ from $p(y)$, i.e., $W(\sigma) = \text{ecc}_{Tx}(p(y))$. Since $W(\sigma) > \text{depth}(p(y))$, we know that $y' \notin \sigma$. Let $\tau = (y, p(y)) + \sigma$. Then

$$\text{depth}(y) < \text{depth}(p(y))$$

$$< W(\sigma)$$

$$< W(\tau)$$

$$\leq \text{ecc}_{Tx}(y)$$

and thus $y' \notin \text{Spine}(x)$, contradiction.

Now, suppose that $y$ is not the best child of $p(y)$. There exists $z \in \text{Chldrn}(p(x))$, where $z = y$ and $\text{depth}(z) + w(z, p(y)) \geq \text{depth}(y) + w(y, p(y))$. Let $\sigma$ be the longest path from $z$ to a leaf of $T_z$, and let $\tau = (y, p(y)) + (p(y), z) + \sigma$. Then
\[ \text{depth}(y) < \text{depth}(y) + \text{w}(y, \text{p}(y)) \]
\[ \leq \text{depth}(z) + \text{w}(z, \text{p}(y)) \]
\[ < W(\tau) \]
\[ \leq \text{ecc}_{T_x}(y) \]

and thus \( y' \notin \text{Spine}(x) \), contradiction.

Part (b) follows immediately from (a).

Continuing our list of terms, we let

- \( \text{base}(x) = \) the bottom member of \( \text{Spine}(x) \), \( i.e., \) that process in \( \text{Spine}(x) \) of greatest level, which we call the \textit{base process} of \( x \).
- \( \text{Base}(l) = \{ \text{base}(x) : \text{level}(x) = l \} \)
- \( \text{tail}(x) = \text{best\_child}(\text{base}(x)) \), the \textit{tail process} of \( x \), which may or may not be defined.
- \( \text{Tail}(l) = \{ \text{tail}(x) : \text{level}(x) = l \} \)
- \( \text{Fan}(x) = T_{\text{tail}(x)} \). If \( \text{tail}(x) \) is undefined, we let \( \text{Fan}(x) = \emptyset \).
- \( \text{Fan}(l) = \bigcup \{ \text{Fan}(x) : \text{level}(x) = l \} \).

In Figure 6.2-1 and Figure 6.2-2, we illustrate some of these definitions.
Figure 6.2-1: Example of $T_x$, where $\text{tail}(x)$ is defined. Processes of $\text{Spine}(x)$ are solid black, processes of $\text{Fan}(x)$ are cyan, and other processes of $\text{Up}(x)$ are gold. The red path in (b) has length $\text{ecc}_{T_x}(y)$, and the cyan path in (c) has length $\text{ecc}_{T_x}(z)$.

Figure 6.2-2: Example of $T_x$, where $\text{tail}(x)$ is undefined. Processes of $\text{Spine}(x)$ are solid black, and processes of $\text{Up}(x)$ are gold. The red path in (b) has length $\text{ecc}_{T_x}(y)$, and the red path in (c) has length $\text{ecc}_{T_x}(z)$.
We now characterize $\text{longest\_path}_{Tx}(y)$, the path in $Tx$ from $y$ whose weight is $\text{ecc}_{Tx}$. Recall that $\text{down\_path}(z) = \text{longest\_path}_{Tz}(z)$ for any process $z$, i.e., $W(\text{down\_path}(z)) = \text{depth}(z)$.

**Lemma 6.2** Let $y \in T_x$. Let $u$ be the process of minimum level on $\text{longest\_path}_{Tx}(y)$. Then

(a) $u \in \text{Spine}(x)$.

(b) If $y \in \text{Fan}(x)$, then $\text{longest\_path}_{Tx}(y) = \text{path}(y, u) + \text{secondary\_down\_path}(u)$.

(c) If $y' \in \text{Fan}(l)$, then $\text{longest\_path}_{Tx}(y) = \text{path}(y, u) + \text{down\_path}(u)$.

**Proof:** Let $s$ be the other end of $\text{longest\_path}_{Tx}(y)$. We first prove (a) by contradiction. Suppose $u' \in \text{Spine}(x)$. Then $\mu(u, x) > \text{depth}(u)$, which implies that $\text{up\_path}(u, x)$ is longer than $\text{path}(u, s)$. Thus, $\text{path}(y, u) + \text{up\_path}(u, x)$ is longer than $\text{longest\_path}_{Tx}(y)$, contradiction.

We now prove (b). By the definition of $u$, $p(u)$ does not lie on $\text{longest\_path}_{Tx}(y)$, since $\text{best\_child}(u)$ lies on $\text{path}(y, u)$, $\text{path}(u, s) = \text{secondary\_down\_path}(u)$, and we are done.

We now prove (c). If $y \in \text{Spine}(l)$, then $u = y$ and $\text{longest\_path}_{Tx}(y) = \text{down\_path}(y)$, and we are done. Otherwise, let $v$ be the first member of $\text{Spine}(x)$ in $\text{path}(y, u)$. Pick $z \in \text{Child}(v) \cap \text{path}(y, v)$. Then $z \neq \text{best\_child}(v)$ since $y' \notin \text{Fan}(x)$. Thus, $\text{down\_path}(v) = \text{longest\_path}_{Tx}(v)$ does not contain $z$, and hence $\text{longest\_path}_{Tx}(y) = \text{path}(y, v) + \text{down\_path}(v)$, and $u = v$, and we are done.
The examples shown in Figure 6.2-1(b) and Figure 6.2-2(b) and Figure 6.2-2(c) illustrate Part (b) of Lemma 6.2, while the example shown in Figure 6.2-1(c) illustrates Part (c) of the lemma.

In LINEAR\textsubscript{max}, a process \( y \) must know whether it is a member of \( Up(l) \) or \( Spine(l) \). It must also know whether it is in \( Base(l) \), and whether it is in \( Tail(l) \). These questions can all be answered by \( y \) in constant time, provided \( \text{critical}_\text{level}(y) = \min \{l : y \in Spine(l)\} \), the critical level of \( y \), has been computed. We calculate the critical levels using the same technique that we used in CHAPTER 4.

The critical value of a process \( y \) enables \( y \) to determine whether it lies in \( Up(l) \) for any given \( l \), as we show in the following lemma.

\textbf{Lemma 6.3}

(a) If \( l' < l \), then \( Up(l) \subseteq Up(l') \).

(b) \( y \in Up(l) \) if and only if \( \text{critical}_\text{level}(x) \leq l \leq \text{level}(x) \).

\textbf{Proof:} To prove (a), pick \( y \in Up(l) \). Let \( x \) be the ancestor of \( y \) at level \( l \), and let \( x' \) be the ancestor of \( x \) at level \( l' \) (which is also an ancestor of \( y \)). Then

\[
\text{ecc}_{Tx}(y) \geq \text{ecc}_{Tx}(y) > \text{depth}(y)
\]

and hence \( y \in Up(l') \) by definition. Part (b) follows immediately.
In the tree shown, the weights of the edges are proportional to vertical distance in the figure. $\text{Spine}(l)$ is the union of chains headed by all processes at level $l$. Processes of $\text{Spine}(l)$ are solid black. Processes of $\text{Fan}(l)$ are filled in cyan; other processes of $\text{Up}(l)$ are filled in gold. Processes in $\text{Base}(l)$ and $\text{Tail}(l)$ are circled in black. $\text{Spine}(2)$, $\text{Up}(2)$, and $\text{Fan}(2)$ are shown in (a). $\text{Spine}(3)$, $\text{Up}(3)$, and $\text{Fan}(3)$ are shown in (b). Note that $\text{Up}(3) \subseteq \text{Up}(2)$. The values of $\text{critical}_\text{level}$ are shown in (c). Note that $x \in \text{Spine}(l)$ if and only if $\text{critical}_\text{level}(x) \leq l \leq \text{level}(x)$.

In Table 6.2-1, we give the code for the critical level phase.

```
1: for 0 \leq l \leq d in decreasing order do [Wave $l$]
2:     for all $x$ such that $\text{level}(x) = l$ concurrently do
3:         for all $y \in T_x - x$ in top down order do
4:             $p \leftarrow p(y)$
5:             $\mu(y, x) \leftarrow \max \{\mu(p, x) + w(y, p) \eta(y)\}$
6:             if $\mu(y, x) \leq \text{depth}(y)$ then
7:                 $\text{critical}_\text{level}(y) \leftarrow l$
8:         end if
9:     end for
10: end for
```

Table 6.2-1 : Critical Level Phase
The waves are pipelined, so that the total time required for the critical level phase is only $O(h)$.

### 6.3 Optimization Phase of $\text{LINEAR}_{\text{max}}$

For any $y$ such that $\text{level}(y) \geq l$, $y$ can compute the following.

- $y \in \text{Up}(l)$ if and only if $\text{critical level}(y) \geq l$.
- $y \in \text{Spine}(l)$ if and only if $\text{critical level}(y) < l$.
- $y \in \text{Base}(l)$ if and only if $y \in \text{Spine}(l)$, and either $\text{best\_child}(y) \in \text{Up}(l)$, or $\text{best\_child}(y)$ is undefined.
- $y \in \text{Tail}(l)$ if and only if $p(y) \in \text{Base}(l)$ and $y = \text{best\_child}(p(y))$.

The optimization phase is a dynamic programming algorithm, where $\text{up\_package}(y, l)$ is the solution to certain sub problems associated with the process $y$ during Wave $l$ of the phase. To understand the steps of the optimization phase, we describe the sub problems that must be solved by $y$ during Wave $l$.

We first define $\text{local\_cost}(y, l) = \min \{w(y, y') + \text{depth}(y') : y' \in \text{Swap\ N}\ (y, l)\}$.

1. For $y \in \text{Up}(l)$, $\text{up\_package}(y, l)$ contains
   
   (a) $\text{min\_up\_cost}(y, l) = \min \{\text{local\_cost}(z, l) + W(y, z) : z \in T_y\}$.

2. If $y \in \text{Spine}(y)$, then $\text{up\_package}(y)$ contains
   
   (a) $\text{min\_normal\_cost}(y, l) = \min \{\text{local\_cost}(z, l) + W(y, z) : z \in T_y$ and $z \neq T_{\text{best\_child}(y)}\}.$

   (b) $\text{min\_fan\_cost}(y, l) = \min \{\text{local\_cost}(z, l) + W(y, z) : z \in T_y \cap$
(c) subtree_mincost(y, l) = min \{local_cost(z, l) + ecc_{Ty}(z): z \in Ty. \}

At the conclusion of Wave l, we compute swap_edge_cost(x) = subtree_mincost(x, l) for all x such that level(x) = l.

Figure 6.3-1 illustrates some of these functions. In the case shown, level(x) = l = 2. In Figure 6.3-1 (a), we show a path, in red, whose length is min_up_cost(y, 2), and a path, in cyan, whose length is min_up_cost(z, 2), where y, z \in Up(2). In 7.5(b), y \in Spine(2). We show a path, in red, whose length is min_normal_cost(y, 2), and a path, in cyan, whose length is min_fan_cost(y, 2). In 7.5(c), swap edge cost(x) = subtree_mincost(x, 2) is the length of the shorter of the two paths (one red, the other cyan) shown.
Figure 6.3-1: \textit{swap\_edge\_cost, min\_up\_cost and min\_fan\_cost}
Functions in \textit{up\_package} of descendants of \(x\), where level \((x) = l = 2\). In (a), we show \(\text{min\_up\_cost}(y, 2)\) and \(\text{min\_up\_cost}(z, 2)\), where \(y, z \in \text{Up}(2)\).
In (b), we show \(\text{min\_normal\_cost}(y, 2)\) and \(\text{min\_fan\_cost}(y, 2)\), where \(y \in \text{Spine}(2)\). In (c), we show two paths whose lengths are candidates for \(\text{swap\_edge\_cost}(x) = \text{subtree\_mincost}(x, 2)\); the smaller of those two lengths will be the result.

Finally, in Table 6.3-1, we give the code that is executed at Line 6 of Table 6.1 in the case of \(\text{LINEAR}_{\max}\).

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>(\text{local_cost}(y, l) = \min{w(y, y') + \text{depth}(y'): y' \in \text{Swap_N}(y, l)})</td>
</tr>
<tr>
<td>2:</td>
<td>\textbf{if} (y \in \text{Up}(l)) \textbf{then}</td>
</tr>
<tr>
<td>3:</td>
<td>(\text{min_up_cost}(y, l) \leftarrow \min{\text{local_cost}(y, l), \min{\text{min_up_cost}(z, l): z \in \text{Chldrn}(y)}})</td>
</tr>
<tr>
<td>4:</td>
<td>\textbf{else} (y \in \text{Spine}(l))</td>
</tr>
<tr>
<td>5:</td>
<td>(\text{min_normal_cost}(y, l) \leftarrow \min{\text{local_cost}(y, l), \min{\text{min_up_cost}(z, l) + w(y, z): z \in \text{Normal_Chldrn}(y)}})</td>
</tr>
</tbody>
</table>
6: if best_child(y) is defined then
    7:     z ← best child(y)
8: if z ∈ Spine(l) then
9:     min_fan_cost(y, l) ← min_fan_cost(z, l) + w(z, y)
10: else
11:     min_fan_cost(y, l) ← min_up_cost(z, l) + w(z, y)
12: end if
13: subtree_mincost(y, l) ← min
    {min_normal_cost(y, l) + height(y)
     min_fan_cost(y, l) + secondary_height(y)
     subtree_mincost(z, l)}
14: else \{y = base(x), and tail(x) undefined\}
15:     min_fan_cost(y, l) ← ∞
16:     subtree_mincost(y, l) ← min_normal_cost(y, l) + height(y)
17: end if
18: end if

---

Table 6.3-1: Computation of up package(y, l) for LINEAR$_{max}$

6.3.1. Detailed Explanation of Table 6.3-1

The best way to understand the code of Table 6.3-1 is to think of it as computation of one sub problem of a dynamic programming algorithm. Let $x$ be the ancestor of $y$ at level $l$, and $e = \{x, p(x)\}$. The sub problem is to compute all information needed to determine whether some $e' = \{z, z'\} \in$ SwapEdges($x$) for $z \in T_y$ is the best swap edge for $x$, and if so, the value of $F_{max}(T, r, e, e')$.

Recall that $F_{max}(T, r, e, e') = W($longest_path$_T(z)) + w(z, z') + depth(z')$, where $e' = \{z, z'\}$. If $y \in Up(l)$, then the only information that up_package($y, l$) needs to contain is min_up_cost($y, l$), the minimum value of $W($path($z, y$)) + w($z, z'$) + depth($z'$) over all $z \in T_y$ such that $\{z, z'\} \in$ SwapEdges($x$), i.e., rank($z, z'$) > $l$; local_cost($y, l$) is a temporary value used in the computation of min_up_cost($y, l$).
If \( y \in \text{Spine}(l) \) and if \( \text{Fan}(l) \cap T_x = \emptyset \), then \( \text{up\_package}(y, l) \) also needs only one variable, namely \( \text{subtree\_mincost}(y, l) \), which is the minimum value of \( \text{ecc}_{T_y}(z) + w(z, z') + \text{depth}(z') \) over all \( \{z, z'\} \in \text{SwapEdges}(x) \) such that \( z \in T_y \).

If \( y \in \text{Spine}(l) \) and \( \text{Fan}(l) \cap T_x = \emptyset \), then \( \text{up\_package}(y, l) \) needs two variables, \( \text{subtree\_mincost}(y, l) \), as described above, and \( \text{min\_fan\_cost}(y, l) \), which is the minimum value of \( W(\text{path}(z, y)) + w(z, z') + \text{depth}(z') \) over all \( \{z, z'\} \in \text{SwapEdges}(x) \) such that \( z \in T_y \cap \text{Fan}(l) \).

At most one of those two values will be needed to compute \( \text{swap\_edge\_cost}(x) \). If \( \text{subtree\_mincost}(y, l) \geq \text{min\_fan\_cost}(y, l) + \mu(y, x) \), then \( \text{min\_fan\_cost}(y, l) \) could be discarded; otherwise \( \text{subtree\_mincost}(y, l) \) could be discarded. But since \( y \) does not know the value of \( \mu(y, x) \), it cannot discard either.

### 6.4 Overview of LINEARdiam

Recall that \( F_{diam}(T, r, e, e') = \text{ecc}_{T_X}(y) + w(y, y') + \text{ecc}_{T^{-x}}(y') \), where \( e = \{x, p(x)\}, y \in T_x, \) and \( \{y, y'\} \in \text{SwapEdges}(x) \). LINEARdiam has all the complexity of LINEARmax, since it must handle the impossibility of calculating \( \text{ecc}_{T_X}(y) \) during the optimization phase; it also has additional complexity due to the impossibility of calculating \( \text{ecc}_{T^{-x}}(y') \) during the optimization phase.

We handle the latter problem in the same was the former, namely by running a phase which calculates another version of critical level, which
we call special level. If \( x \in C_2 \) and \( y' \in S_1 \), or if \( x \in C_1 \) and \( y' \notin S_1 \), we use both the special level of \( y' \) and the critical level of \( y \) (as computed in Table 6.2-1) to decide which one of following four formulas for \( \text{ecc}_{T_x}(y) + w(y, y') + \text{ecc}_{T^{-}\!x}(y') \) has the largest value:

1. \( \text{depth}(y) + w(y, y') + \text{restr}_\text{ecc}(y') \)
2. \( \mu(y, x) + w(y, y') + \text{restr}_\text{ecc}(y') \)
3. \( \text{depth}(y) + w(y, y') + \text{secondary}_\text{ecc}(y') \)
4. \( \mu(y, x) + w(y, y') + \text{secondary}_\text{ecc}(y') \)

where \( \text{restr}_\text{ecc}(y') \) and \( \text{secondary}_\text{ecc}(y') \), defined below, are computed during the preprocessing phase.

Otherwise, we only need to use \( \text{critical}_\text{level}(y) \) to choose among the two formulas

1. \( \text{depth}(y) + w(y, y') + \text{ecc}_{T_x}(y') \)
2. \( \mu(y, x) + w(y, y') + \text{ecc}_{T_x}(y') \)

That decision can be made at the time that \( \text{up}_\text{package}(y, l) \) is computed, for \( l = \text{level}(x) \), despite the fact that only one of the two or four choices can actually be computed at the time.

### 6.5 The Preprocessing Phase of \( \text{LINEAR}_{\text{diam}} \)

The preprocessing phase of \( \text{LINEAR}_{\text{diam}} \) computes all the same variables as the preprocessing phase of \( \text{LINEAR}_{\text{max}} \), together with the variables in the list that follows. This list is quite long, and the purpose
of some of these variables is obscure. We will do our best to explain them later in the section.

1. $\text{branch}(x)$, provided $x \neq r$, which is defined to be that value of $i$ such that $x \in S_i$. After $\text{height}$ is computed for all processes, the root labels its children $c_1, c_2, \ldots$ such that $h_i \geq h_j$ if $i > j$, where we define $h_i = w(c_i, r) + \text{height}(c)$. The value $i$ is then broadcast to all processes in $S_i$.

2. $h_1, h_2, \text{ and } h_3$. The root knows the values of $h_i$ for all $i$, but only the values of $h_i$ for $i \leq 3$ are broadcast to all processes.

3. We use the function $\text{best_child}$ to define a chain of processes $C_i \subseteq S_i$. $C_i$ contains $c_i$: otherwise, $x \in C_i$ if and only if $p(x) \in C_i$ and $x = \text{best_child}(p(x))$.

If $x \in S_i$, we compute $\text{chain_level}(x)$ to be the level of the closest ancestor of $x$ which is in $C_i$. More formally, let $\text{chain level}(c) = 1$; for all other $x \in S_i$, let $p = p(x)$, and let

$$
\text{chain_level}(x) = \begin{cases} 
\text{level}(x) & \text{if } x = \text{best_child}(p) \text{ and } \text{chain_level}(p) = \text{level}(p) \\
\text{chain_level}(p) & \text{otherwise}
\end{cases}
$$

4. $\text{local}_\mu(x) = \mu(x, c_i)$, provided $x \neq r$, where $x \in S_i$. Recall the definition of $\mu$ given in Section 3.5.

The values of $\text{local}_\mu(x)$ are computed in a broadcast wave, using the definition
local_µ(x) = \begin{cases} 
0 & \text{if } x = c_i \\
w(x, p(x)) + \max \{\text{local}_\mu(p(x)), \eta(x)\} & \text{otherwise}
\end{cases}

5. local_φ(x) = ecc_{Si}(x) \text{ for all } i = 1, 2, \text{ provided } x \in S_i, \text{ the local eccentricity of } x. \text{ Local eccentricities are computed for all } x \text{ concurrently in } O(1) \text{ time as follows.}

\text{local}_φ(x) = \max \begin{cases} 
\text{local}_\mu(x) \\
\text{depth}(x)
\end{cases}

6. avoid(x) for all \( x \in C_1 + C_2 \), as defined in Section 3.6. If \( x \in C_i \), then avoid(x) is the length of the longest path from r to a leaf of \( S_i \) which avoids \( x \). We can compute avoid(x) for all \( x \in S_i \) in a broadcast wave, as follows:

\text{avoid}(x) = \begin{cases} 
0 & \text{if } x = c_i \\
\max \{\text{avoid}(p(x)), \eta(x) + \text{depth}(p(x))\} & \text{otherwise}
\end{cases}

7. ecc_T(x) = \begin{cases} 
\text{depth}(x) + h_2 & \text{if } x \in S_1 \\
\text{depth}(x) + h_1 & \text{otherwise}
\end{cases}

The full eccentricity of \( x \).

8. secondary_ecc(x) = \begin{cases} 
\text{depth}(x) + h_3 & \text{if } x \in S_1 \\
\text{depth}(x) + h_2 & \text{otherwise}
\end{cases}

The secondary eccentricity of \( x \). Intuitively, the secondary eccentricity is the length of the longest path from \( x \), through \( r \), to a leaf of \( T \) which avoids the largest subtree that does not contain \( x \).
9. \( \text{restr}_e(x) = \begin{cases} \text{ecc}_T - S_2(x) & \text{if } x \in S_1 \\ \text{ecc}_T - S_1(x) & \text{otherwise} \end{cases} \)

The restricted eccentricity of \( x \). We compute the restricted eccentricity of all \( x \) as follows.

\[
\text{restr}_e(x) = \begin{cases} \max \left\{ \text{local}_\varphi(x) \right\} & \text{if } x \in S_1 + S_2 \\ \text{secondary}_e(x) & \text{otherwise} \end{cases}
\]

Intuitively, the restricted eccentricity is the length of the longest path from \( x \) to a process of \( T \) which avoids the largest subtree that does not contain \( x \). (Unlike for the definition of \( \text{secondary}_e(x) \), that path need not contain \( r \).)

Figure 6.5-1 below illustrates the definitions of \( \text{local}_\varphi(x) \), \( \text{ecc}_T (x) \), and \( \text{restr}_e(x) \).
100

Figure 6.5-1: restricted, local and full eccentricity. If \( x \in S_1 \), the local eccentricity of \( x \) is the length of the longest path from \( x \) in \( S_1 \), shown in brown. The full eccentricity of \( x \) is the length of the longest path from \( x \) to a point in \( S_2 \), shown in blue. The restricted eccentricity of \( x \) is either the length of the longest path from \( x \) to a point in \( S_3 \), shown in magenta, or \( \text{local}_{\varphi}(x) \), whichever is greater.

6.6 Special Levels

We define \( \text{special}_{\text{level}}(x) \) for all \( x \neq r \), the special level of \( x \), actually another kind of critical level in the sense defined in Section 5. We use special levels to decide among the optional values of \( \text{ecc}_{T-x}(y') \) during the optimization phase of \( \text{LINEAR}_{\text{diam}} \).

For any \( x \in C_1 \), we define

\[
A(x) = \{ y' \in S_1 : \text{restr}_{\text{ecc}}(y') \geq \text{depth}(y') + \text{avoid}(x) \}
\]

\[
B(x) = \{ y' \in S_1 : \text{restr}_{\text{ecc}}(y') < \text{depth}(y') + \text{avoid}(x) \}
\]
For any $x \in C_2$, we define

\[ A(x) = y' \in S_1 : \text{restr\_ecc}(y') \geq \text{depth}(y') + \text{avoid}(x) \]

\[ B(x) = y' \in S_1 : \text{restr\_ecc}(y') < \text{depth}(y') + \text{avoid}(x) \]

Finally, let

\[ A(l) = \bigcup \{ A(x) : \text{level}(x) = l \} \]

\[ B(l) = \bigcup \{ B(x) : \text{level}(x) = l \} \]

special level($y$) = $\min \{ l : y \in B(l) \}$

If $y \in A(l)$ for all $l$, we define special_level($y$) = $\infty$.

Special levels are computed by a phase that is analogous to the computation of critical levels. Computation of special_level($y$) for $y \in S_1$ is slightly different than for other processes, so we write two separate algorithms for the phase.

---

1: initialize special_level($y$) $\leftarrow \infty$ for all $y \in S_1$
2: for all $x \in C_2$ in bottom-up order do
3: \hspace{1em} $l = \text{level}(x)$
4: \hspace{1em} for all $z \in C_2$ which are ancestors of $x$ in bottom-up order do
5: \hspace{2em} copy avoid($x$) to $z$
6: \hspace{1em} end for
7: \hspace{1em} copy avoid($x$) to $r$
8: \hspace{1em} for all $y \in S_1$ in top down order do
9: \hspace{2em} copy avoid($x$) to $y$
10: \hspace{2em} if restr\_ecc($y$) $< \text{depth}(y) + \text{avoid}(x)$ then
11: \hspace{3em} special_level($y$) $\leftarrow l$
12: \hspace{2em} end if
13: \hspace{1em} end for
14: end for

---

Table 6.6-1 : Special Level Phase for $S_1$
The code for computing \textit{special\_level}(x) for \(x \in S_1\) is given in Table 6.6-1. The phase consists of pipelined waves, one for each process \(x\) of \(C_2\). The wave starts at \(x\), passes through \(r\), and then is broadcast down to all processes of \(S_1\). The variables of each wave (other than the values of \textit{special\_level}(y)) are erased after the wave passes.

The value of \textit{special\_level}(y) could be set, and then reset by successive waves. The last value is the one that is correct. If Line 11 is never executed for a specific \(y\), then \textit{special\_level}(y)\(=\infty\) when the phase is done.

The code for computing \textit{special\_level}(x) for \(x' \in S_1\), given in Table 6.6-2, is very similar.

\begin{verbatim}
1: initialize \textit{special\_level}(y) \leftarrow \infty\) for all \(y \notin S_1\)
2: for all \(x \in C_1\) in bottom-up order do
3: \(l = \text{level}(x)\)
4: for all \(z \in C_1\) which are ancestors of \(x\) in bottom-up order do
5: \hspace{1em} copy \textit{avoid}(x) to \(z\)
6: end for
7: for all \(y \notin S_1\) in top down order do
8: \hspace{1em} copy \textit{avoid}(x) to \(y\)
9: \hspace{1em} if \textit{restr\_ecc}(y) > \text{depth}(y) + \text{avoid}(x) then
10: \hspace{2em} \textit{special\_level}(y) \leftarrow l
11: end if
12: end for
13: end for
\end{verbatim}

Table 6.6-2 : Special Level Phase for Processes Not in \(S_1\)

In Figure 6.6-1 below, we illustrate steps of the computation of \textit{special\_level}(x).
Figure 6.6-1: Computation of Special Levels.
(a) shows the depth of all processes, as well as subtrees $S_1$, $S_2$, and $S_3$.
(b) shows $ecc_r(x)$ in blue and $restr_ecc(x)$ in red for all $x$. 
Figure 6.6-1: (Continued): (c) shows $\text{avoid}(x)$ for all $x \in C_1 + C_2$ in black, and $\text{chain\_level}(x)$ for all $x \in S_1 + S_2$ in red. Processes of $C_1 + C_2$ are black, and other processes of $S_1 + S_2$ are gold. (d) shows processes of the set $A(3)$ in magenta, and processes of the set $B(3)$ in green.
Figure 6.6-1: (Continued): (e) shows processes of the set $A(4) = A(5)$ in magenta, and processes of the set $B(4) = B(5)$ in green. (f) shows processes of the set $A(6)$ in magenta, and processes of the set $B(6)$ in green.
Figure 6.6-1: (Continued): For any $l \geq 7$, $A(l) = \{c_1\}$, and all other processes are in $B(l)$, as shown in (g). (h) shows special level($x$) for all $x$, in magenta.

6.7 Partition of $Swap_N(y, f)$

For any process $y$ and any $l \leq level(y)$, the set $Swap_N(y, l)$ is partitioned, by $y$, into three sets, $C(y, l), E(y, l)$, and $F(y, l)$. These sets are defined so that, for $x$ the ancestor of $y$ at level $l$:
The partition is implemented by $y$ as follows. For any $y' \in \text{Swap}_N (y, l)$:

- If $y \in S_1$ and $\text{chain level}(y) \geq l$ then
  - $y' \in C(y, l)$ if $y' \in S_1$.
  - $y' \in D(y, l)$ if $y' \not\in S_1$ and $\text{special level}(y') > l$.
  - $y' \in E(y, l)$ if $y' \not\in S_1$ and $\text{special level}(y') \leq l$.

- If $y \in S_2$ and $\text{chain level}(y) \geq l$ then
  - $y' \in C(y, l)$ if $y' \not\in S_1$.
  - $y' \in D(y, l)$ if $y' \in S_1$ and $\text{special level}(y') > l$.
  - $y' \in E(y, l)$ if $y' \in S_1$ and $\text{special level}(y') \leq l$.

- If $y \in S_1 + S_2$ and $\text{chain level}(S) < l$, or if $y' \in S_1 + S_2$, then $y' \in C(y, l)$.

Using that partition, we now give code for the optimization phase of $\text{LINEAR}_\text{diam}$ in Table 6.7-1. We make use of intermediate variables whose names are the same as previously defined variables, concatenated with $C$, $D$, or $E$.

We give the complete code of $\text{LINEAR}_\text{diam}$ in Table 6.7-1.
1: \textbf{for} \ l \leq l \leq d \ \textbf{do} \\
2: \quad \textbf{for} \ all \ y \ such \ that \ level(y) \geq l \ in \ bottom \ up \ order \ \textbf{do} \\
3: \quad \quad local\_costC(y, l) = \min \{w(y, y') + ecc_{\Delta}(y'): y' \in C(y, l)\} \\
4: \quad \quad local\_costD(y, l) = \min \{w(y, y') + restr\_ecc(y'): y' \in D(y, l)\} \\
5: \quad \quad local\_costE(y, l) = \min \{w(y, y') + depth(y'): y' \in D(y, l)\} \\
6: \quad \textbf{if} \ y \in Up(l) \ \textbf{then} \\
7: \quad \quad \quad \quad \quad min\_up\_costC(y, l) \leftarrow \min \{\min\{min\_up\_costC(z, l): z \in Chldrn(y)\}\} \\
8: \quad \quad \quad \quad \quad min\_up\_costD(y, l) \leftarrow \min \{\min\{min\_up\_costD(z, l): z \in Chldrn(y)\}\} \\
9: \quad \quad \quad \quad \quad min\_up\_costE(y, l) \leftarrow \min \{\min\{min\_up\_costE(z, l): z \in Chldrn(y)\}\} \\
10: \quad \textbf{else} \{y \in Spine(l)\} \\
11: \quad \quad \quad \quad \quad min\_normal\_costC(y, l) \leftarrow \min \{\min\{min\_up\_costC(z, l): z \in Normal\_Chldrn(y)\}\} \\
12: \quad \quad \quad \quad \quad min\_normal\_costD(y, l) \leftarrow \min \{\min\{min\_up\_costD(z, l): z \in Normal\_Chldrn(y)\}\} \\
13: \quad \quad \quad \quad \quad min\_normal\_costE(y, l) \leftarrow \min \{\min\{min\_up\_costE(z, l): z \in Normal\_Chldrn(y)\}\} \\
14: \quad \textbf{if} \ best\_child(y) \ is \ defined \ \textbf{then} \\
15: \quad \quad z \leftarrow best\_child(y) \\
16: \quad \textbf{if} \ z \in Spine(l) \ \textbf{then} \\
17: \quad \quad \quad \quad \quad min\_fan\_costC(y, l) \leftarrow min\_fan\_costC(z, l) + w(z, y) \\
18: \quad \quad \quad \quad \quad min\_fan\_costD(y, l) \leftarrow min\_fan\_costD(z, l) + w(z, y) \\
19: \quad \quad \quad \quad \quad min\_fan\_costE(y, l) \leftarrow min\_fan\_costE(z, l) + w(z, y) \\
20: \quad \textbf{else} \\
21: \quad \quad \quad \quad \quad min\_fan\_costC(y, l) \leftarrow min\_up\_costC(z, l) + w(z, y) \\
22: \quad \quad \quad \quad \quad min\_fan\_costD(y, l) \leftarrow min\_up\_costD(z, l) + w(z, y) \\
23: \quad \quad \quad \quad \quad min\_fan\_costE(y, l) \leftarrow min\_up\_costE(z, l) + w(z, y) \\
24: \quad \end if
25: \[ \text{subtree}_\text{mincost}C(y, l) \leftarrow \min \begin{cases} \text{min}_\text{normal}\_\text{cost}_C(y, l) + \text{height}(y) \\ \text{min}_\text{fan}\_\text{cost}_C(y, l) + \text{secondary}\_\text{height}(y) \\ \text{subtree}_\text{mincost}(z, l) \end{cases} \]

26: \[ \text{subtree}_\text{mincost}D(y, l) \leftarrow \min \begin{cases} \text{min}_\text{normal}\_\text{cost}_D(y, l) + \text{height}(y) \\ \text{min}_\text{fan}\_\text{cost}_D(y, l) + \text{secondary}\_\text{height}(y) \\ \text{subtree}_\text{mincost}(z, l) \end{cases} \]

27: \[ \text{subtree}_\text{mincost}E(y, l) \leftarrow \min \begin{cases} \text{min}_\text{normal}\_\text{cost}_E(y, l) + \text{height}(y) \\ \text{min}_\text{fan}\_\text{cost}_E(y, l) + \text{secondary}\_\text{height}(y) \\ \text{subtree}_\text{mincost}(z, l) \end{cases} \]

28: \[ \text{else} \]
29: \[ \text{min}_\text{fan}\_\text{cost}_C(y, l) \leftarrow \infty \]
30: \[ \text{min}_\text{fan}\_\text{cost}_D(y, l) \leftarrow \infty \]
31: \[ \text{min}_\text{fan}\_\text{cost}_E(y, l) \leftarrow \infty \]
32: \[ \text{subtree}_\text{mincost}C(y, l) \leftarrow \text{min}_\text{normal}\_\text{cost}_C(y, l) + \text{height}(y) \]
33: \[ \text{subtree}_\text{mincost}D(y, l) \leftarrow \text{min}_\text{normal}\_\text{cost}_D(y, l) + \text{height}(y) \]
34: \[ \text{subtree}_\text{mincost}E(y, l) \leftarrow \text{min}_\text{normal}\_\text{cost}_E(y, l) + \text{height}(y) \]
35: \[ \text{end if} \]
36: \[ \text{end if} \]
37: \[ \text{if level}(y) = l \text{ then} \]
38: \[ \text{swap}\_\text{edge}\_\text{cost}(y) \leftarrow \min \begin{cases} \text{subtree}_\text{mincost}C(y, l) \\ \text{subtree}_\text{mincost}D(y, l) \\ \text{subtree}_\text{mincost}E(y, l) \end{cases} \]
39: \[ \text{end if} \]
40: \[ \text{end for} \]
41: \[ \text{end for} \]

Table 6.7-1: Optimization Phase of LINEAR\textsubscript{diam}

6.7.1. Explanation of Table 6.7-1

Lines 3–39 of Table 6.7-1 are basically an expansion of Table 6.3-1 to take into account the multiple possible formulas for $ecct_x(y')$ in LINEAR\textsubscript{diam}. A line of Table 6.3-1 corresponds to up to three lines of Table 6.7-1.
Line 1 of Table 6.3-1 corresponds to Lines 3–5 of Table 6.7-1
Line 3 of Table 6.3-1 corresponds to Lines 7–9 of Table 6.7-1.
Line 5 of Table 6.3-1 corresponds to Lines 11–13 of Table 6.7-1
Line 9 of Table 6.3-1 corresponds to Lines 17–19 of Table 6.7-1
Line 11 of Table 6.3-1 corresponds to Lines 21–23 of Table 6.7-1
Line 13 of Table 6.3-1 corresponds to Lines 25–27 of Table 6.7-1
Line 15 of Table 6.3-1 corresponds to Lines 29–31 of Table 6.7-1
Line 16 of Table 6.3-1 corresponds to Lines 32–34 of Table 6.7-1

6.7.2. Summary of LINEARD_{diam}

Finally, we summarize the algorithm LINEARD_{diam} in Table 6.7-2 below.
The time complexity of each phase, and hence of LINEARD_{diam}, is $O(h)$, and
no more than $O(\delta_j)$ variables are stored in any process $x$ at any one time.

---

1: Preprocessing Phase. {Section 6.5}
2: Ranking Phase. {Table 5.4-1}
3: Critical Level Phase. {Table 6.2-1}
4: Special Level Phase. {Table 6.6-1 and Table 6.6-2}
5: Optimization Phase. {Table 6.7-1}

Table 6.7-2: LINEARD_{diam}
CHAPTER 7

CONCLUSION

This Thesis concentrates on 2-edge connected and weighted distributed networks that maintain communication by a spanning tree $T$. The main purpose is the restoration of such a tree should any of the tree edges fail. This is resolved by finding a swap edge $e' \not\in T$, that gives the least cost, to replace the failing edge $e$. This is done in advance of any failure allowing us to be ready and we refer to it as the all best swap edges problem.

We started off by giving algorithms for the all best swap edges problem for six different cost measures. First, we presented an algorithm which can be adapted to six cost measures, and which takes $O(d^2)$ time, where $d$ is the diameter of $T$. We then presented a novel paradigm for speeding up distributed computations under certain conditions. We have applied this paradigm to find $O(d)$-time distributed algorithms for the all best swap edge problem for all the cost measures except $F_{\text{sum}}$.

As a future research work, we will try to design a linear time algorithm for $F_{\text{sum}}$. We can also investigate possible implementation of our protocols with the self-stabilization property.
BIBLIOGRAPHY


and Distributed Computing 68(7), pp. 976-983.


VITA

Graduate College
University of Nevada, Las Vegas

Feven Z. Andemeskel

Degrees:
   Bachelor of Science in Computer Science, 2006
   University of Asmara, Eritrea

Thesis Title: Dynamic Distributed Programming and Applications to
   Swap Edge Problem

Thesis Examination Committee:
   Chair Person, Dr. Ajoy K. Datta, Ph.D.
   Committee Member, Dr. Lawrence L. Larmore, Ph.D.
   Committee Member, Dr. Yoo-hwan Kim, Ph.D
   Committee Member, Dr. Emma Regentova, Ph.D.