12-2010

Dynamic distributed programming and applications to swap edge problem

Feven Z. Andemeskel
University of Nevada, Las Vegas

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

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

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

LIST OF TABLES

LIST OF FIGURES

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_{dist}, F_{incr}, F_{wght}, F_{max}, F_{sum}, and F_{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 δ_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_{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 δ_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_{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 $O(h)$.

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 *(id (z), id (z')).* We could also encode *e'* as *(index (z), index (z'))*, where *index (z)* is the ordered pair *(pre_index (z), post_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; *pre_index (z)* is the index of *z* in the preorder visitation of *T*, while *post_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 δ_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, Djikstra’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 extensely [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 *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 *x* and *y*.

We refer to an edge of T as a *tree edge*, and any other edge of X as a *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 *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 *SwapEdges(e)* = *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 *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).

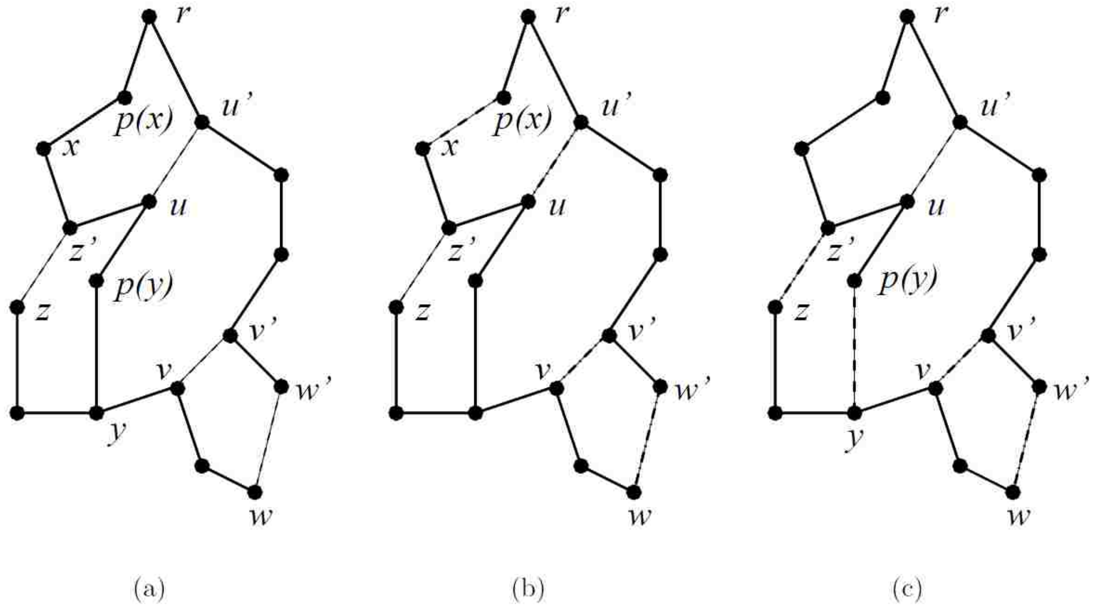


Figure 2.4-1 : Swap Edges
(a): Tree edges are solid, cross edges are dashed. b): Failure at *x*. *{u, u'}*, *{v, v'}*, and *{w,w'}* are the swap edges of *x*. c): Failure at y. *{v, v'}*, *{w,w'}*, and *{z, z'}* are the swap edges of *y*.

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., $e = \{x, p(x)\}$.

- $F_{dist}(T, r, e, e') = W_{T'}(x, r)$, the distance from the root to the point of failure in *T'*.
- $F_{incr}(T, r, e, e') = max \{W_{T'}(u, r) - W_T(u, r) : u \in T_x\}$, 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 $T_{\sim x} = T - e - T_x$, the other component, remains unchanged.)
- $F_{sum}(T, r, e, e') = \sum_{u \in Tx} W_{T'}(u, r)$, the sum of the distances, in *T'*, from the root to all processes in T_x.
- $F_{diam}(T, r, e, e') = max \{W_{T'}(u, v) : u \in T_x \text{ and } v \notin T_x\}$. 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, *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_{wght}.

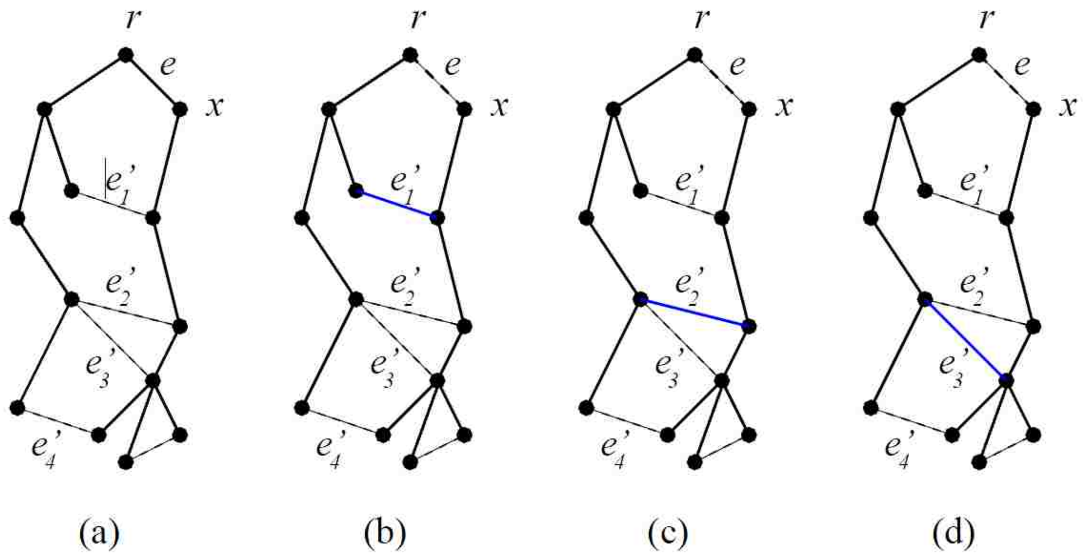


Figure 2.4-2 : Cross Edges
We show T in (a). The edge e has four cross edges. In (b), (c), and (d), we show the resulting tree $T' = T - e + e'$ for three choices of e'. We do not show the case $e' = e'_4$

F	F_{dist}	F_{incr}	F_{wght}	F_{max}	F_{sum}	F_{diam}
$F(T, r, e, e'_1)$	4	3	1	6	34	7
$F(T, r, e, e'_2)$	5	4	1	4	31	5
$F(T, r, e, e'_3)$	6	5	1	6	30	6
$F(T, r, e, e'_4)$	8	7	1	8	42	8

Table 2.4-1 : F(T,r,e,e') for the network in Figure 2.4-2 for various F.

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_{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 $F_{incr,}$ F_{max}, and F_{sum}. In [9], Gfeller et al. give an algorithm for the all best swap edge problem, using the measure F_{diam}. Their algorithm is also a version of BSE.

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

The space complexity of BSE is $O(\delta_x)$ 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 *size(x)* and *index (x).* We compute *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 *index (x) = (pre_index (x), post_index (x)),* the *index of x*, where *pre_index (x)* is the order of x in the preorder visitation of T, and where *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, "≤" 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 *index (x) ≤ index (y).* Thus, if $e' = \{y, y'\}$ is a cross edge and $y \in T_x$, then $e' \in$ *SwapEdges(x)* if and only if *index (x) ≤ 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 *index (y).* If *index (x) ≤ 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 $y_1, y_2, . . . , y_m$, then

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

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

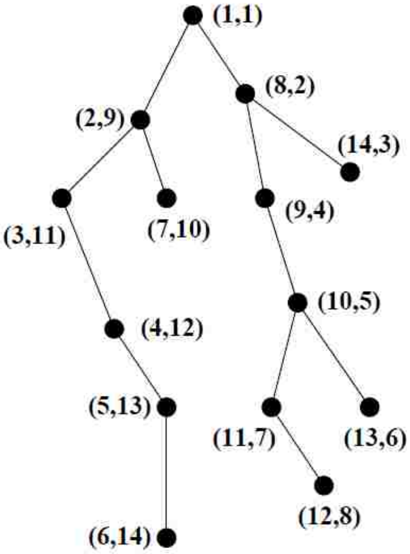


Figure 3.2-1 : Ancestry of processes.
Processes are labeled with their indices. A process x is an ancestor of y if and only if index (x) ≤ index (y).

In every case, there are at most *O*(δ_x) 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 convergecast 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 *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 down_package(x, x).
 3:         for all $y \in T_x - x$ in top down order do
 4:                 Compute down_package(y, x), using down_package(p(y), x).
 5:         end for
 6:         for all $y \in T_x$ in bottom up order do
 7:             Compute l_sol (y, x), using down_package(y, x).
 8:                subtree_mincost (y, x) $\longleftarrow$ min {l_sol (y, x), min {subtree_mincost(z, x) : z
                                                  $\in$ Chldrn(y)}}.
 9:         end for
10:       solution(x) $\longleftarrow$ 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 s*ubtree_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_{x1} and T_{x2} 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).*

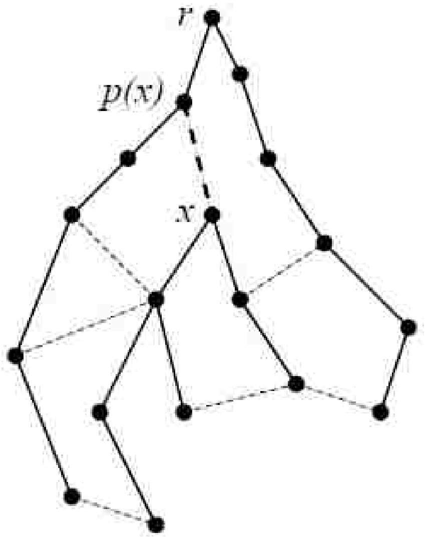


Figure 3.2-2 : Cross edges are shown as dashed.
Point of failure is x.

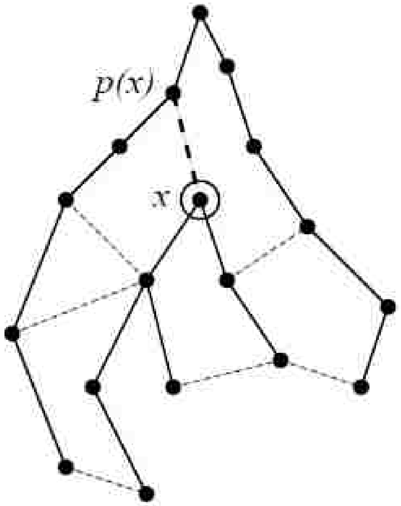


Figure 3.2-3 : Iteration begins with computation of *down_package(x, x).*

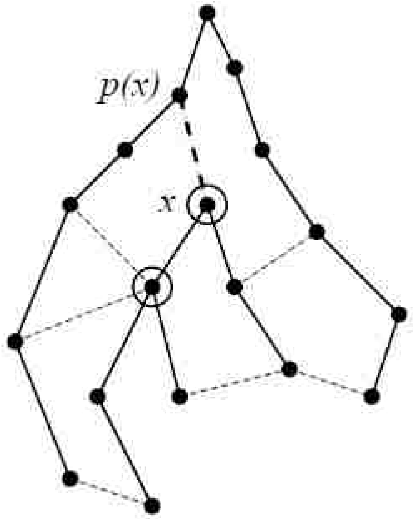


Figure 3.2-4 : Children of x compute down_package

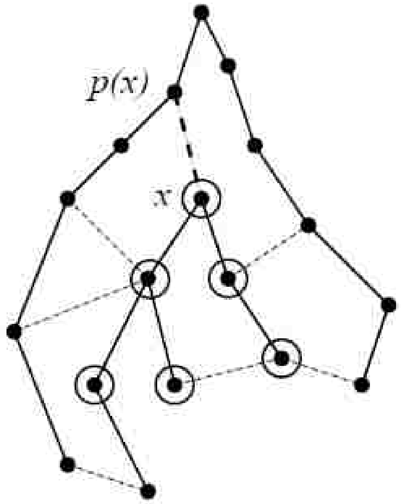


Figure 3.2-5 : Broadcast wave continues.
Variables of down_package are retained until needed.

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.

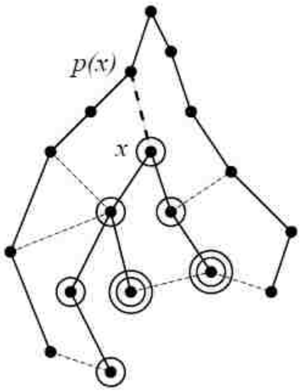


Figure 3.2-6 : Broadcast wave is completed.
l_sol is computed for some leaves.

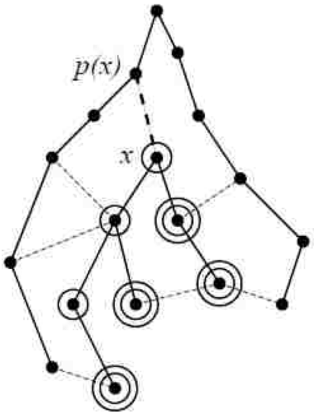


Figure 3.2-7 : Convergecast continues.

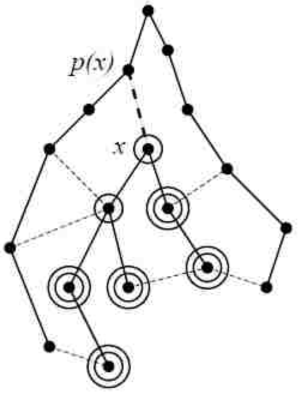


Figure 3.2-8 : Convergecast continues

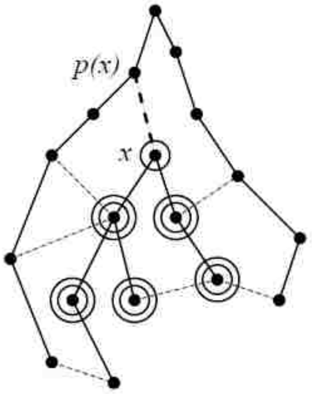


Figure 3.2-9 : down package and l_sol are deleted when not needed.

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).*

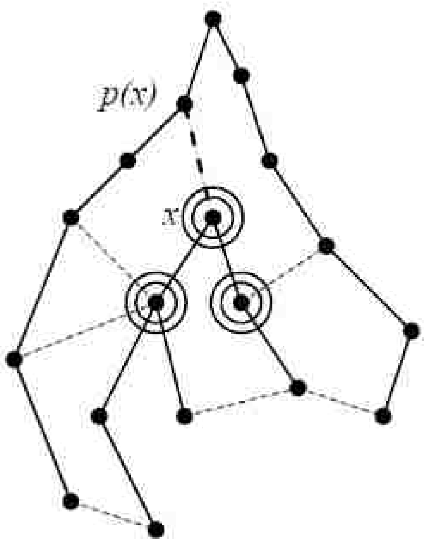


Figure 3.2-10 : Convergecast wave is completed.

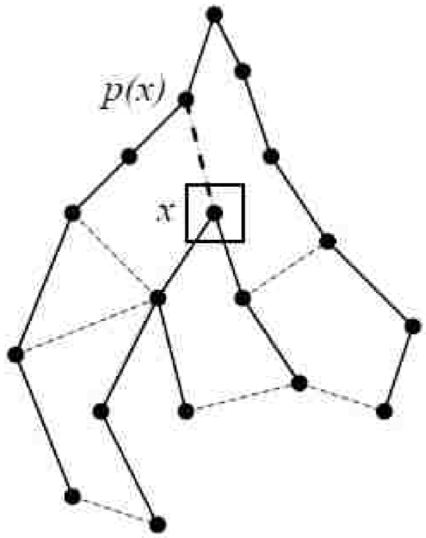


Figure 3.2-11 : solution(x) is computed.
All values of down package and *l_sol* have been deleted.

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 BSE$_{dist}$ and BSE$_{incr}$

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

In the broadcast portion of Iteration(x), *down_package(x)* consists of the variables *index (x)* and $W_T(y, x)$. Line 4 of Table 3.2-1 is then executed by y simply copying the value of *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 *subtree_mincost (z, x)* for all $z \in$ *Chldrn(y):*

1. Compute $l\_sol(y, x) = min\ \{W_T(y, x) + w(y, y') + depth(y') : \{y, y'\} \in SwapEdges(x)\}$

2. Compute $subtree\_mincost(y, x) = min\ \{\ l\_sol(y, x),\ min\ \{subtree\_mincost(z, x) : z \in Chldrn(y)\}\}$

Finally, $solution(x) = subtree\_mincost(x, x)$.

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

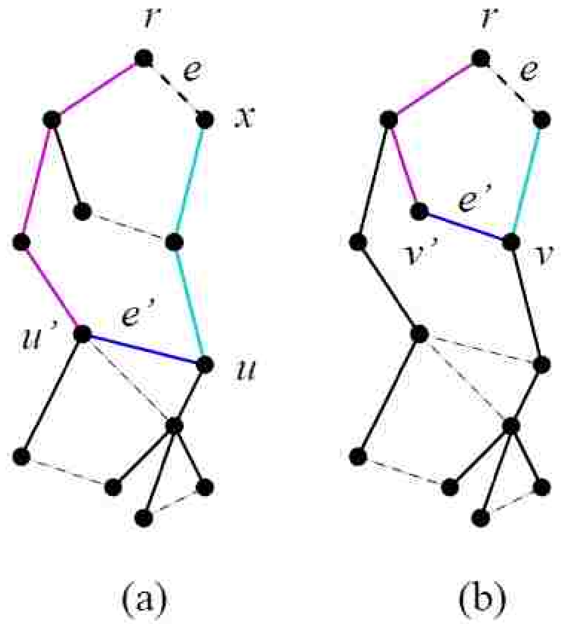


Figure 3.3-1 : *l_sol (u, x)*
All edge weights are 1. Then *l_sol (u, x)* = 6, and *solution(x)* = *l_sol (v,x)* = *4.*

We do not need to separately describe an algorithm which minimizes F_{incr}, since the best swap edge for F_{dist} is also the best swap edge for F_{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_{incr} *(T, r, e, e') =* F_{dist} *(T, r, e, e') − depth(x).*

Proof: Since $x \in T_x$, $F_{incr}(T, r, e, e') \geq W_{T'}(x, r) - W_T(x, r) = F_{dist}(T, r, e, e')$ − *depth(x).*

To prove the converse, let $e' = \{z, z'\}$, where $z \in T_x$,and pick $y \in T_x$ such that $F_{incr}\ (T, r, e, e') = W_{T'}(y, r) - W_T\ (y, r)$.Then

$$
\begin{aligned}
F_{incr}(T, r, e, e') &= W_{T'}(y, r) - W_T\ (y, r) \\
&= W_{T'}(y, r) - W_T\ (x, r) - W_T\ (y, x) \\
&\leq W_{T'}(x, r) + W_{T'}(y, x) - W_T\ (x, r) - W_T\ (y, x) \\
&= W_{T'}(x, r) + W_T\ (y, x) - W_T\ (x, r) - W_T\ (y, x) \\
&= W_{T'}(x, r) - \mathit{depth}(x) \\
&= F_{dist}(T, r, e, e') - \mathit{depth}(x)
\end{aligned}
$$

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 *size(x)* and *index (x)* for all *x*, and *down_package(y, x)* contains only the variable *index (x)*.

Line 4 of Table 3.2-1 is then executed by *y* simply copying the value of *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.*, $\min \{w(y, y') : \{y, y'\} \in \mathit{SwapEdges}(x)\}$, and then comparing this value to *subtree_mincost (z, x) for all* $z \in$ *Chldrn(y):*

1. Compute $\mathit{l\_sol}\ (y, x) = \min \{w(y, y') : \{y, y'\} \in \mathit{SwapEdges}(x)\}$

2. Compute *subtree_mincost (y, x)* = *min { l_sol (y, x), min {subtree_mincost (z, x) : z* $\in$ *Chldrn(y)}}*

Finally, *solution(x) = subtree_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$_{max}$

For any weighted network *Y* and any process *x* of *Y*, we define $ecc_Y(x) = max_{u \in Y} W_Y(x, u)$, the *eccentricity of x in Y*. Recall that $F_{max}(T, r, e, e') = depth(y') + w(e') + ecc_{T_x}(y)$, where $e = \{x, p(x)\}$ and $e' = \{y, y'\}$ is a swap edge of *e*, and $y \in T_x$. We illustrate $F_{max}(T, r, e, e')$ in Figure 3.5-1.

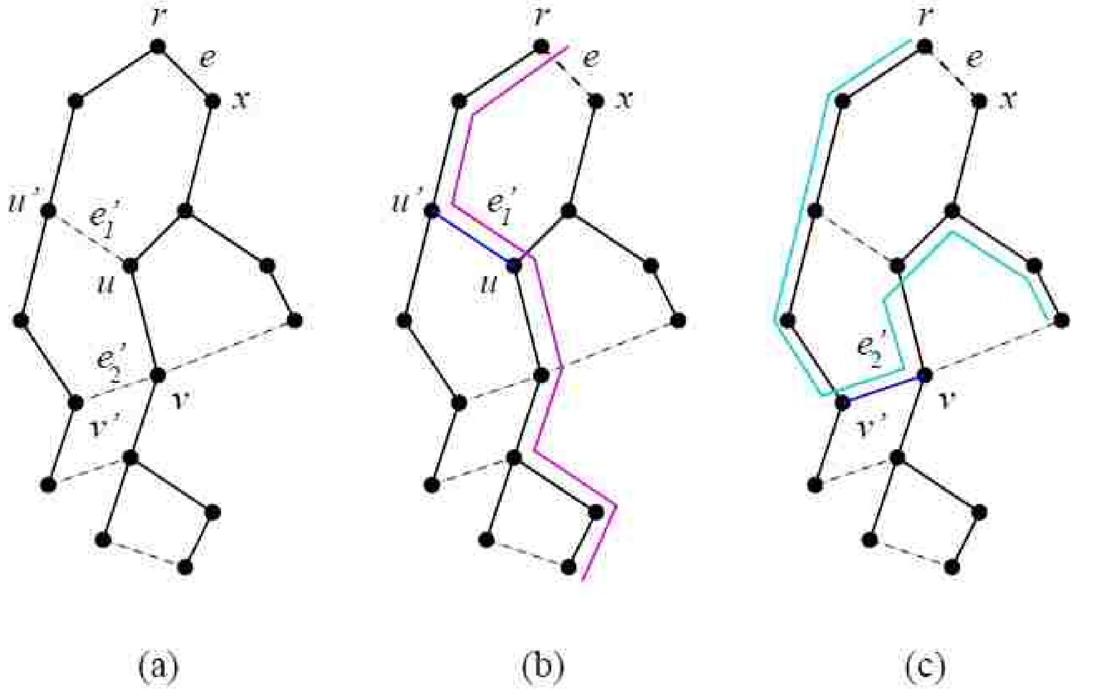


Figure 3.5-1 : F_{max}
We show T in (a), together with some cross edges. In (b) we show F_{max} (T, r, e, e_1') and in (c) we show F_{max} (T, r, e, e_2'), where $e_1' = \{u, u'\}$ and $e_2' = \{v, v'\}$.

Besides *size(x)* and *index (x)*, the preprocessing phase of BSE$_{\max}$ computes

1. *depth(x).*

2. *height(x)* = $ecc_{Tx}(x)$, 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 $\eta(x) = ecc_{Tp(x) - Tx}(p(x))$. We illustrate an example of $\eta(x)$ in Figure 3.5-2.

All values of *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 η can be computed simultaneously in O(1) time, since $\eta(x) = max\ \{w(y, p(x)) + height(y) : y \in Chldrn(p(x)) - x\}$.

For notational convenience, we write

- $path_T\ (x, y)$ = the path in *T* from *x to y*. We will write *path(x, y)* if *T* is understood.
- $longest\_path_T\ (x)$ = the longest path in *T* starting at *x*. Thus, $W_T\ (longest\_path_T) = ecc_T(x)$,
- $down\_path(x) = longest\_pathT_x\ (x)$ the longest path from *x* to a leaf of T_x. Thus, $W_T(down\_path(x))=height(x)$.

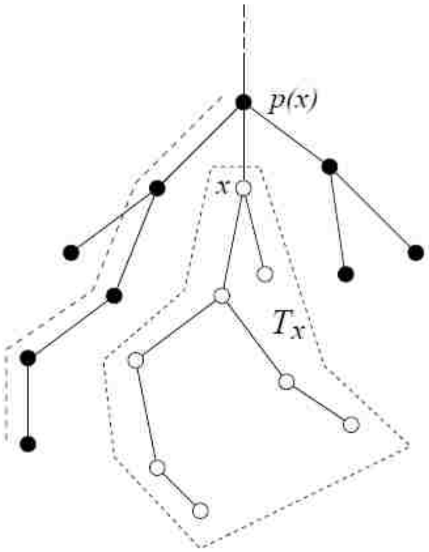


Figure 3.5-2 : An example where η(x) = 4.
We assume that all edge weights are 1.

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 $\mu(y, x) = ecc_Y (x)$, 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 $\phi (y, x) = ecc_{Tx} (y), = max \{height(y), \mu(y, x)\}$, since the longest path in T_x which starts at y must either go down to a leaf of T_y or up through $p(y)$.

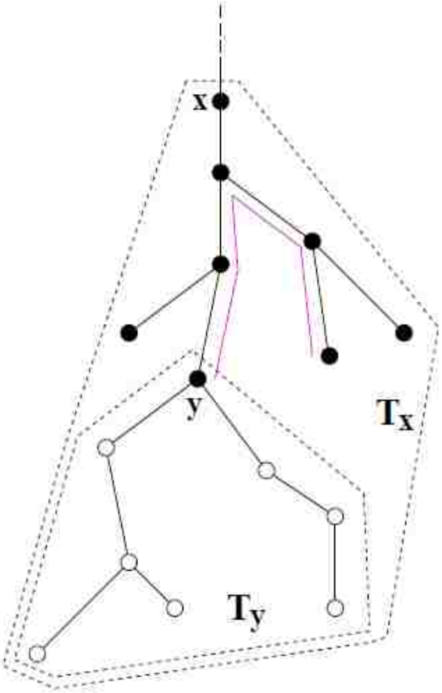


Figure 3.5-3 : An example where μ(y, x) = 4.
We assume that all edge weights are 1.

For each x and $y \in T_x$, *down_package(y, x)* consists of the variables *index (x)* and $\mu(y, x)$. In Line 2 of Table 3.2-1 for F_{max}, we already have *index (x)* from the preprocessing phase. We let $\mu(x, x) = 0$. To execute Line 4, y simply copies *index (x)* from its parent, and computes $\mu(y, x) = w(y, p(y)) + max\ \{\eta(y), \mu(p(y), x)\}$.

Computation of *l_sol (y, x)* depends on the fact that $\phi(y, x)$ is the maximum of *height(y)* and $\mu(y, x)$. In Line 8 of Table 3.2-1, y computes $\phi(y, x) = ecc_{Tx}(y) = max\ \{height(y), \mu(y, x)\}$, and then $l\_sol\ (y, x) = min\ \{depth(y') + w(y, y') + \phi(y, x) : \{y, y'\} \in SwapEdges(x)\}$.

3.6 BSE$_{\text{diam}}$

The original goal of 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* $e' \in SwapEdges(e)$ *is chosen to minimize* $F_{diam}(T, r, e, e')$, *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 $C' = F_{diam}(T, r, e, e')$.Then $diam(T')$, the diameter of T', is equal to max $\{A,B,C'\}$. Pick $e'' \in SwapEdges(e)$ 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 $F_{diam}(T, r, e, e') = ecc_{Tx}(z) + w(z, z') + ecc_{T\sim x}(z')$.

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.

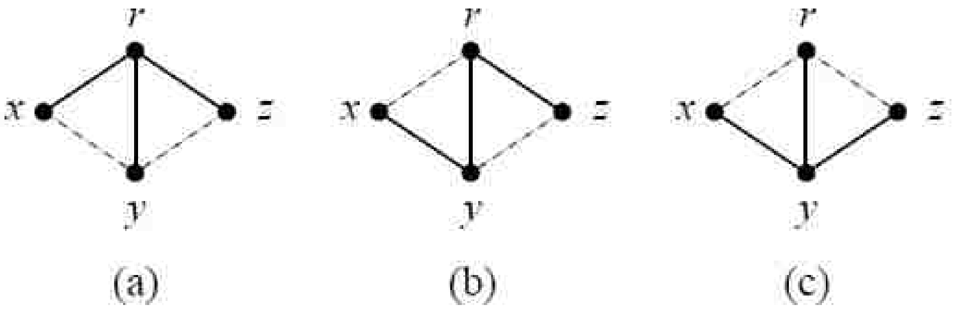


Figure 3.6-1 : F$_{diam}$
Network X with four processes and five edges is shown in (a). Let all edge weights be 1. The minimum diameter spanning tree T shown in (a) has diameter 2. In (b), let *e = {x, r}*. The swap edge for e which minimizes F$_{diam}$ is e' = {x, y}, and the resulting tree *T' = T − e + e'* has diameter 3. However, the minimum diameter spanning tree of *X − e* has diameter 2, as shown in (c).

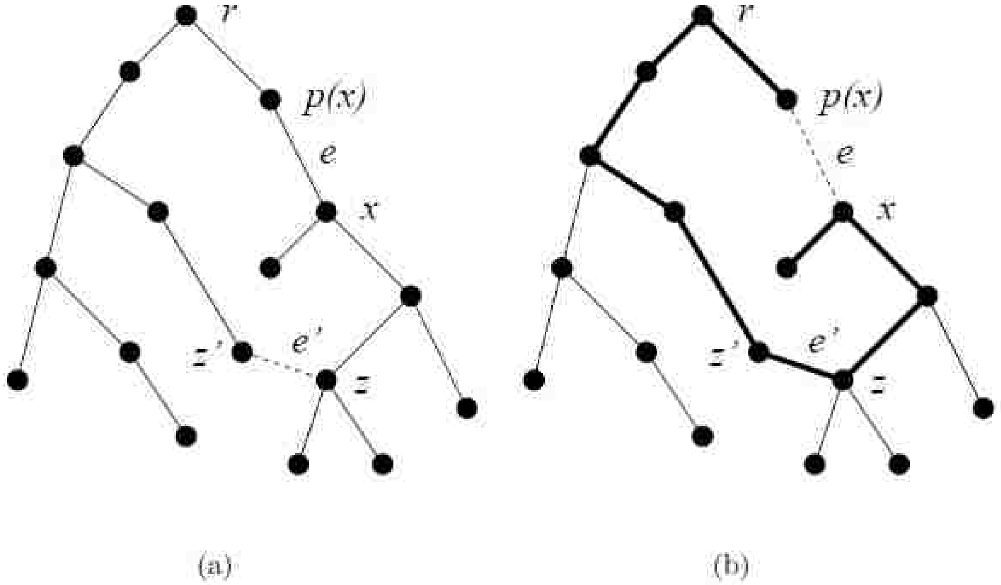


Figure 3.6-2 : Swap edge with respect to F$_{diam}$
T is shown in (a). The failure point is *x*, and the swap edge is *e'*. In (b), the path whose length is *F$_{diam}$(T, e, e')* is indicated by heavy lines.

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 $F_{diam}(T, r, e, e')$, 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_{diam}.

- A weighted tree graph always has either one or two centers. We will assume that r is one of those centers. Let $Chldrn(r) = \{c_1, \ldots c_m\}$, 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 $h_i = w(r, c_i) + height(c_i)$, 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, $h_1 = h = height(r)$.
- For any $1 \leq i \leq m$ and any $x \in S_i$, we define $avoid(x)$ 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 $avoid(x) = 0$.

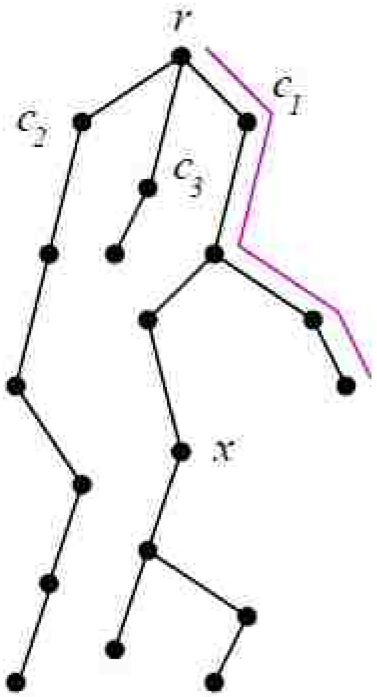


Figure 3.6-3 : avoid(x).
Let all edge weights be 1. Then h_1 = 7, h_2 = 6, and h_3 = 2. The magenta path is the longest path from r through c_1 that avoids x, and thus *avoid(x)* = 4.

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

3.6.1. Preprocessing Phase of BSE_{diam}

The preprocessing phase of BSE_{diam} computes the following variables for each process x.

1. *size(x).*
2. *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 μ and ϕ 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_*μ*(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 *height(x)* has been computed for all *x*, all values of η can be computed in O(1) time. The values *branch*(c_i) = *i* are assigned by *r* to its children, and *r* also computes h_i for $i \leq 3$. The value of *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 *height(x)* has been computed for all *x*, all values of *local*_μ can be computed in a broadcast wave, using the appropriate version of a formula given in Section 3.5, namely *local*_$\mu(x) = w(x, p(x)) + max\ \{\eta(x), local_\mu(p(x))\}$.

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

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

3.6.2. Optimization Phase of BSE$_{diam}$

For all $x \neq r$ and all $y \in T_x$, *down_package(y, x)* consists of *index (x)* and $\mu(y, x)$.

If $y \neq x$, then *y* computes *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 $\phi(y, x) = max\ \{\mu(y, x), height(y)\}$.

Execution of Line 8 of Table 3.2-1 for BSE$_{diam}$ 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.

```
for all $y'$ such that $\{y', y\} \in$ SwapEdges($x$) do
        i $\longleftarrow$ branch($y$)
        j $\longleftarrow$ branch($y'$)
        k $\longleftarrow$ the smallest positive integer which is neither $i$ nor $j$
        if $i = j$ then
                $ecc_{T\sim x}(y') \longleftarrow depth(y') + h_k$
      else
                $ecc_{T\sim x}(y') \longleftarrow max\ \{local\_\phi\ (y'), depth(y') + avoid(x), depth(y') + h_k\}$
      end if
      $cost(y, y', x) \longleftarrow \phi\ (y, x) + w(y, y') + ecc_{T\sim x}(y')$
end for
$l\_sol\ (y, x) \longleftarrow min\ \{cost(y, y', x) : \{y, y'\} \in SwapEdges(x)\}$
```

Table 3.6-1 : Computation of $l\_sol\ (y, x)$ in $\mathrm{BSE}_{\mathrm{diam}}$

3.6.3. Computation *of ecc* $_{T\sim x}$ *(y')*

We now explain the computation of $ecc_{T\sim x}(y')$, the eccentricity of y' in the subgraph $T_{\sim x}$. Let $i = branch(y)$ and $j = branch(y')$, 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 $T_{\sim x}$ 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 α be the longest path in S_j from y', i.e., the path whose length is $ecc_{S_j}(y')$, let β be the path from y' to r, let γ be the longest path from r to a leaf of S_i which avoids x, and δ_k the longest path from r to a leaf of S_k. (If $k = 3$ and $S_3 = \emptyset$ we take δ_k to be the trivial path at r.) The paths α, β, γ, and δ_k are illustrated in Figure 3.6-5(a). The path whose length is $ecc\ T_{\sim x}(y')$ *is* α, $\beta+\delta_k$ or $\beta+\gamma$ whichever is longer. The three possibilities are illustrated in Figure 3.6-5(b)–(d).

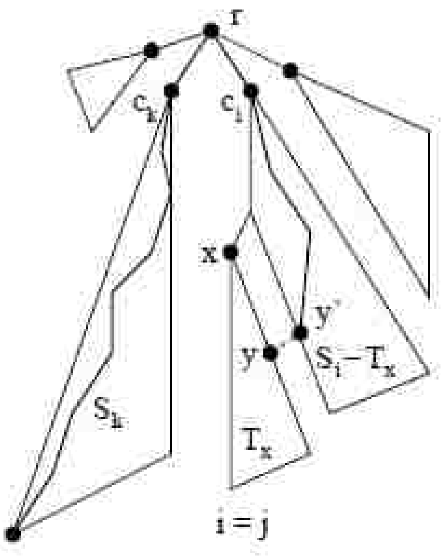
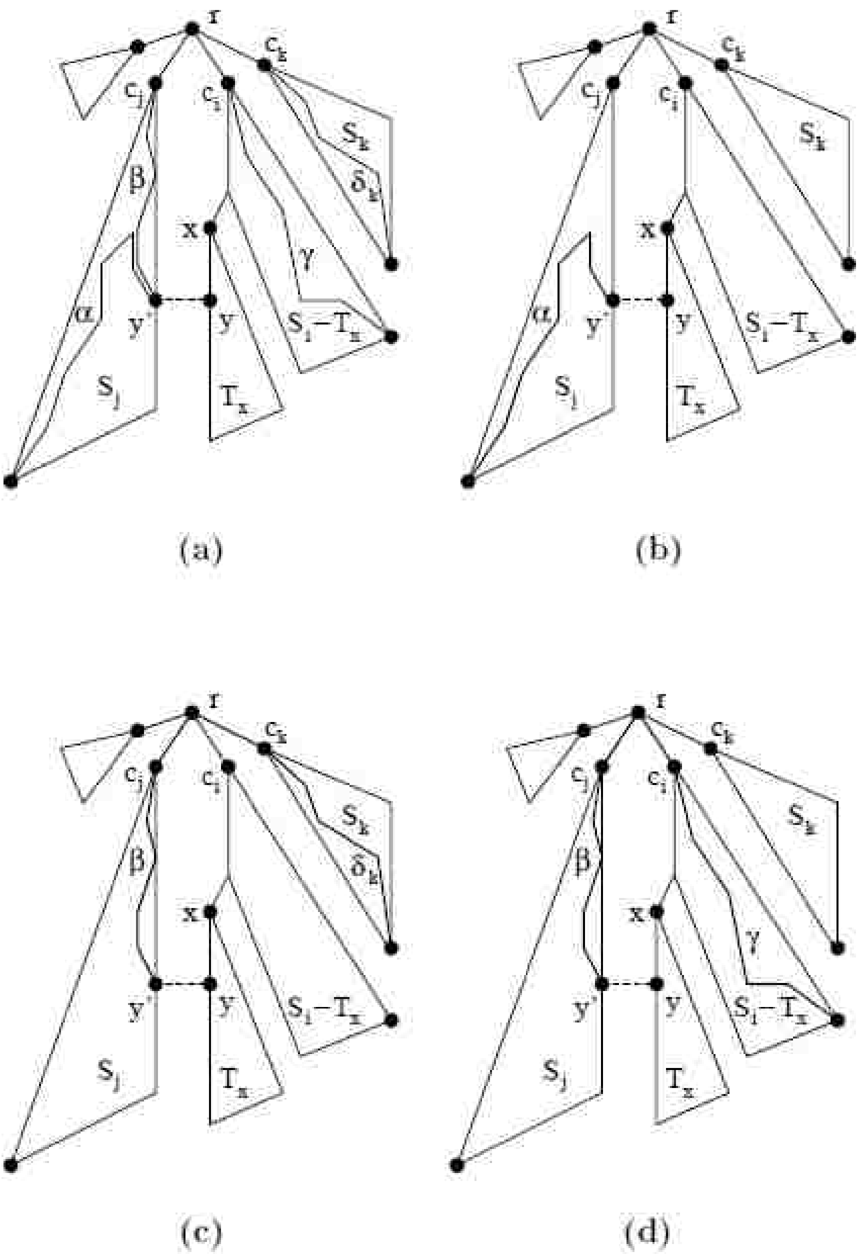


Figure 3.6-4 : If i = j, then $ecc_{T\sim x}(y') = depth(y') + h_k$.



(a) (b)

(c) (d)

Figure 3.6-5 : α , β , γ , and δ_k

If $i \neq j$, $ecc_{T\text{~}x}(y')$ is the maximum length of any path in $T_{\text{~}x} = T - T_x$ from *y'*. In (a), we show α , the longest path in S_j from *y'*; β , the path from *y' to r*; γ the longest path from *r* to a process in S_i which avoids *x*, and δ_k , the longest path from r to a leaf of S_k. The maximum length path in $T_{\text{~}x} = T - T_x$ from *y'* is one of three possibilities, shown in (b)–(d) with heavy lines. In (b), we show α , in (c) we show $\beta + \delta_k$, and in (d) we show $\beta + \gamma$.

3.7 BSE$_{sum}$

For any weighted network *Y* and any process *x in Y,* we define *path_sum$_Y$ (x)* =$\sum_{u \in Y}$ *W$_Y$ (x, u),* the *path sum of x in Y,* the sum of the shortest weights of paths from *x* to all processes of *Y*. *F$_{sum}$(T, r, e, e')* = *path_sum$_{T_x \cup v}$(r),* where *v* is a *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.

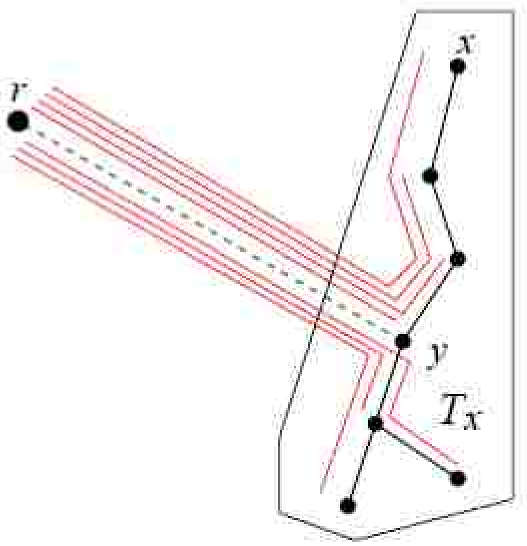


Figure 3.7-1 : F$_{sum}$
The network *T$_x \cup$ v*, where *v* is a virtual edge of length *depth(y')+w(y, y')* from *y* to r, where *y* $\in$ *T$_x$* and *e' = {y, y'}* is a swap edge of *x*. *F$_{sum}$(T, r, e, e')* is the sum of the lengths of the red lines.

For convenience, we introduce shorter notation for certain instances of $path\_sum_Y\ (x)$:

- For any process x, let $sum(x) = path\_sum_{Tx}(x) = \sum_{u \in Tx} W_T\ (u, x)$.
- For any process $x \neq r$, let $\theta\,(x) = path\_sum_{Tp(x)-Tx}\ (p(x)) = \sum_{u \in Tp(x)-Tx} W_T\ (u, y)$.
- For any processes x and $y \in T_x$, let $\nu\,(y, x) = path\_sum_{Tx-Ty}\ (y) = \sum_{u \in Tx-Ty} W_T\,(u, x)$.
- For any processes x and $y \in T_{x,}$ let $\psi\,(y, x) = path\_sum_{Tx}(y) = \sum_{u \in Tx} W_T\,(u, y)$.

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

The implementation of $\mathrm{BSE_{sum}}$ depends on the following observation.

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

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

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

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

$$= \sum_{u \in T_x} (W_{T'}\ (r,y)\ \text{-}\ W_T\,(y,u))$$

$$= size(x)\ \cdot W_{T'}(r,\ y) + \sum_{u \in T_x} W_T\,(y,u)$$

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

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

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

Note that

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

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

In Line 8 of Table 3.2-1, y computes

$l\_sol(y, x) = min\{ size(x) \cdot(depth(y') + w(y, y')) + path\_sum_{Tx}(y) : \{y, y'\} \in SwapEdges(x)$.

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, $Chldrn(x)$, and thus x knows $Cross\_N(x)$, the set of all neighbors x' *of* x such that $\{x, x'\}$ is a cross edge. We also assume an ordering on $Chldrn(x)$, 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.

Message / Wave	BSE$_{dist}$	BSE$_{incr}$	BSE$_{wght}$	BSE$_{max}$	BSE$_{diam}$	BSE$_{sum}$
PRE_DOWN_I First Preprocessing Broadcast Wave	*depth*	*depth*	*(none)*	*depth*	*depth*	*depth*
PRE_UP Preprocessing Convergecast Wave	*size*	*height* *size*	*size*	*height* *size*	*height* *size*	*height* *size* *sum*
PRE_DOWN_II Second Preprocess-ing Broadcast Wave	*index*	*index*	*index*	*index* η	*index* η *branch* h_1, h_2, h_3 *local_*μ *avoid* *local_* φ	*index* θ

CROSS(x)	*index (x)*
	depth(x)
		index (x)
		depth(x)
			index (x)	*index (x)*
				depth(x)
					index (x)
					depth(x)
					local_*φ(x)*
					branch(x)
						index (x)
						depth(x)

OPT_DOWN(x) Broadcast Wave of Iteration(x) $y \in T_x$	*index (x)* W_T *(y,x)*	*index (x)* W_T *(y,x)*	*index (x)*	*index (x)* $\mu(y,x)$ $\varphi(y,x)$	*index (x)* $\mu(y,x)$ $\varphi(y,x)$	*size(x)* *index (x)* $\nu(y,x)$ $\psi(y,x)$
OPT_UP(x) Convergecast Wave of Iteration(x) $y \in T_x$	*l_sol (y,x)* *subtree_mincost (y,x)*	*l_sol (y,x)* *subtree_mincost (y,x)*	*l_sol (y,x)* *subtree_mincost (y,x)*	*l_sol (y,x)* *subtree_mincost (y,x)*	*l_sol (y,x)* *subtree_mincost (y,x)*	*l_sol (y,x)* *subtree_mincost (y,x)*

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 ≠ 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'* ∊ *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* ≠ *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 ≠ 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* ≠ *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 *depth*. Initially, $depth(r) \longleftarrow 0$. The message PRE_DOWN_I sent by a process x to its children contains the value of $depth(x)$. When x receives PRE_DOWN_I from its parent, it computes $depth(x) \longleftarrow w(x, p(x)) + depth(p(x))$.
- Computation of *height*, in all cases except BSE_{dist} and BSE_{wght}. The message PRE_UP sent by a process x to its parent contains $height(x)$. If x is a leaf of T, then $height(x) \longleftarrow 0$. Otherwise, when x receives PRE_UP from all its children, x computes $height(x) \longleftarrow max \{w(x, y) + height(y) : y \in Chldrn(x)\}$.
- Computation of *size*. The message PRE_UP sent by a process x to its parent contains $size(x)$. If x is a leaf of T, then $size(x) \longleftarrow 1$. Otherwise, when x receives PRE_UP from all its children, x computes $size(x) \longleftarrow 1 + \sum_{y \in Chldrn(x)} size(y)$.
- Computation of *index*. The message PRE_DOWN_II sent by a process x to each child y contains $index\ (y)$, while r computes $index\ (r) \longleftarrow (1,$

1). When a process *x* knows the value of *index (x)*, then *x* computes *index (y)* for all $y \in Chldrn(x)$. If *x* is not a leaf, let $Chldrn(x) = \{y_1, \ldots, y_m\}$. Then *x* computes

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

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

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

- Computation of *sum*, in the case BSE_{sum}. The message PRE_UP sent by a process *x* to its parent contains *sum(x).* If *x* is a leaf of *T*, then $sum(x) \longleftarrow 0$. Otherwise, when *x* receives PRE_UP from all its children, *x* computes $sum(x) \longleftarrow \sum_{y \in \text{Chldrn(x)}} (size(y) \cdot w(y, x) + sum(y))$.
- Computation of η, in the cases BSE_{max}, BSE_{diam}, and BSE_{sum}. The message PRE_DOWN_II sent by a process *x* to each child *y* contains $\eta\,(y)$. At the beginning of the second preprocessing broadcast wave, $\eta\,(r) \longleftarrow 0$. Each x computes $\eta\,(y) \longleftarrow max_{\ y \in \text{Chldrn(x)-\{y\}}}\{w(x, z) + height(z)\}$ for each $y \in Chldrn(x)$. *If* $Chldrn(x) = \{y\}$, *then* $\eta\,(y) \longleftarrow 0$.
- Computation of *branch,* h_1, h_2, h_3, *local_*μ*, avoid, and local_*ϕ in the case of $\text{BSE}_{\text{diam.}}$ Recall that *branch(r), local_*μ*(r), avoid(r),* and *local_*ϕ *(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 $Chldrn(r) = \{c_1, c_2,. . . , c_m\}$ and $h_i = w(r, c_i) + height(c_i)$, indexed such that $hi \geq hi{+}1$ 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 $branch(c_i) \longleftarrow i$, $local\_\mu(c_i) \longleftarrow 0$, $avoid(c_i) \longleftarrow 0$, *and* $local\_\phi(c_i) \longleftarrow height(ci)$, 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 $branch(x)$, h_1, h_2, h_3, $local\_\mu(x)$, $avoid(x)$, *and* $local\_\phi(x)$. For each $y \in Chldrn(x)$, 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. $branch(y) \longleftarrow branch(x).$

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

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

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

- Computation of θ, in the case BSE_{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, $\theta(r) \longleftarrow 0$. Each x computes $\theta(y) \longleftarrow \sum_{y \in \text{Chldrn(x)-(y)}} (w(x, z) + sum(z))$ *for each* $y \in Chldrn(x)$. If $Chldrn(x) = \{y\}$, *then* $\theta(y) \longleftarrow 0$.

- The message CROSS(x) from a process x *to* $x' \in Cross\_N(x)$ contains information that x' needs during the optimization phase. CROSS(x) contains *index* (x), and, in all cases except BSE_{wght}, it contains *depth*(x). In the case of BSE_{diam}, it also contains *branch*(x) and $local\_\phi(x)$.
- The message OPT_DOWN(x) from any process $y \in T_x$ to any $z \in Chldrn(x)$ contains the value *index* (x). In the case BSE_{sum}, the message also contains *size*(x).
- Computation of $W_T(y, x)$, for BSE_{dist} and $\text{BSE}_{\text{incr.}}$ Each process x computes $W_T(x, x) \longleftarrow 0$. The message OPT_DOWN($x$) from any process $y \in T_x$ to any $z \in Chldrn(y)$ contains the value $W_T(y, x)$. When $z \in Chldrn(x)$ receives the message OPT_DOWN(x) from y, then z computes $W_T(z,x) \longleftarrow w(z, y) + W_T(y, x)$.
- Computation of $\mu(y, x)$ *and* $\phi(y, x)$, for BSE_{max} and BSE_{diam}. Each process x computes $\mu(x, x) \longleftarrow 0$. The message OPT_DOWN($x$) from any process $y \in T_x$ to any $z \in Chldrn(x)$ contains the value $\mu(y, x)$, and y computes $\phi(y, x) \longleftarrow max\ \{\mu(y, x), height(y)\}$.
- After $z \in Chldrn(y)$ receives the message OPT_DOWN(x) from y, then z computes $\mu(z, x) \longleftarrow max\ \{w(z, y) + \mu(y, x),\ \eta(z)\}$.

- Computation of $\nu(y, x)$, for BSE$_{\text{sum}}$. Each process x computes $\nu(x, x) \longleftarrow 0$. The message OPT_DOWN$(x)$ from any process $y \in T_x$ to any $z \in Chldrn(x)$ contains the value $\nu(y, x)$. When $z \in Chldrn(y)$ receives the message OPT_DOWN(x) from y, then z computes $\nu(z, x) \longleftarrow w(z, y) \cdot (size(x) - size(z)) + \theta(z) + \nu(y, x)$
- Computation of $l\_sol(y, x)$. For $y \in T_x$, define $Swap\_N(y, x) = \{y' \in Cross\_N(y) : y' \notin Tx\}$. Recall that we can determine whether $y' \in T_x$ by comparing $index(y')$ and $index(x)$, 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 $Swap\_N(y, x) = \emptyset$, y assigns $l\_sol(y, x)$ the default value 1. Otherwise, y computes $l\_sol(y, x)$, an ordered pair, in each case as given in Table 3.8-2.

Case	*l_sol(y, x)*
$\mathrm{BSE}_{\mathrm{dist}}$	$min\ \{(W_T\ (y, x) + w(y, y') + depth(y'), \{y, y'\}) : y' \in Swap\ N\ (y, x)\}$
$\mathrm{BSE}_{\mathrm{incr}}$	$min\ \{(W_T\ (y, x) + w(y, y') + depth(y') - depth(x), \{y, y'\}) : y' \in Swap\_N\ (y, x)\}$
$\mathrm{BSE}_{\mathrm{wght}}$	$min\ \{(w(y, y'), \{y, y'\}) : y' \in Swap\_N\ (y, x)\}$
$\mathrm{BSE}_{\mathrm{max}}$	$min\ \{(\varphi(y, x) + w(y, y') + depth(y'), \{y, y'\}) : y' \in Swap\_N\ (y, x)\}$
$\mathrm{BSE}_{\mathrm{diam}}$	$min\ \{(\varphi(y, x) + w(y, y') + eccT_{\sim x}\ (y'), \{y, y'\}) : y' \in Swap\ N\ (y, x)\}$ where $ecc_{T\sim x}\ (y')$ is as explained in Section 3.6
$\mathrm{BSE}_{\mathrm{sum}}$	$min\ \{(\psi(y, x) + size(x) \cdot (w(y, y') + depth(y')), \{y, y'\}) : y' \in Swap\_N\ (y, x)\}$

Table 3.8-2 : *l_sol (y, x)*
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 $\mathrm{BSE}_{\mathrm{diam}}$, let $i = branch(y)$, $j = branch(y')$, and k the smallest positive integer not equal to i or j. If $i = j$, $ecc_{T\sim x}(y') = depth(y') + h_k$. If $i \neq j$, $ecc_{T\sim x}(y') = max\ \{local\_\mu(y'), depth(y') + h_k, depth(y') + avoid(x)\}$

- Computation of *subtree_mincost (y, x).* After *y* receives the message OPT_UP*(x)* from all its children,*y* computes *subtree_mincost (y, x)* $\longleftarrow min\{l\_sol(y, x), min\{\ min_{\ z \in Chldrn(y)}\ subtree\_mincost\ (z, x)$
- Computation of *solution(x).* After a process *x* has received the message OPT_UP*(x)* from all its children, *x* computes *solution(x)* $\longleftarrow subtree\_mincost\ (x, x)$

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 δ_x the degree of any given process *x*. Let *h* be the hopheight 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 δ_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 *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 $\mathrm{BSE}_{\mathrm{diam}}$ 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_sol(y,* *x)* 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_{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_{dist} and F_{wght} each use critical levels in just one way. However, our linear time algorithm for F_{max} uses critical levels in two different ways, and our linear time algorithm for F_{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 $mincost(x) = min \{cost(x, y): x \leq y\}$.

Required Output. The required output for the minmax problem is for each process x to compute $mincost(x) = min \{cost(x, y): y \geq x\}$. We define $best(x)$ to be equal to that $y \geq x$ for which $cost(x, y) = mincost(x)$. It is a fairly straightforward to augment any algorithm that computes mincost(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.

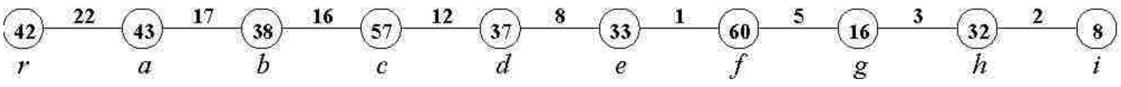


Figure 4.1-1 : Doubly Weighted Chain

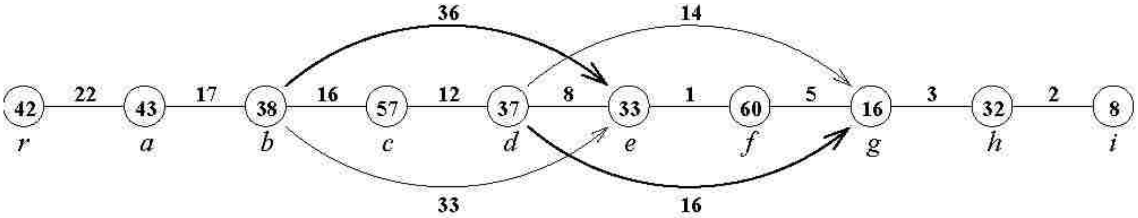


Figure 4.1-2 : *cost(x, y)*
W (b, e) = 36, W (d, f) = 14, F (e) = 33, and F (f) = 16. Thus *cost(b, e) = 36* and *cost(d, f) = 16.*

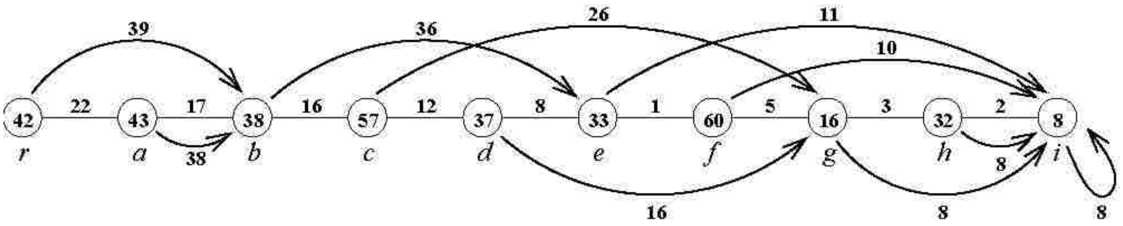


Figure 4.1-3 : *best(x)* and *cost(x, y).*
Arrows indicate the choices of *y = best(x)* for each *x*. The arrows above the line indicate cases where *cost(x, y)= W (x, y)*, while the arrows below the line indicate cases where *cost(x, y)= F (y).*

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

x	*r*	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*
mincost (x)	39	38	36	26	16	11	10	8	8	8
best (x)	*b*	*e*	*e*	*g*	*g*	*g*	*g*	*i*	*i*	*i*

Table 4.1-1 : Values of *mincost(x) and 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 *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 *cost* matrix, while *best(x)* is the index of the column in which that entry is found.

$x\backslash y$	r	a	b	c	d	e	f	g	h	i
r	0	22	39	35	67	75	76	81	84	86
a		0	17	33	45	53	54	59	62	64
b			0	16	28	36	37	42	45	47
c				0	12	20	21	26	29	31
d					0	8	9	14	17	19
e						0	1	6	9	11
f							0	5	8	10
g								0	3	5
h									0	2
i										0

(a)

$x\backslash y$	r	a	b	c	d	e	f	g	h	i
r	42	43	39	57	67	75	76	81	84	86
a		43	38	57	45	53	60	59	62	64
b			38	57	37	36	60	42	45	47
c				57	37	33	60	26	32	31
d					37	33	60	16	32	19
e						33	60	16	32	11
f							60	16	32	10
g								16	32	8
h									32	8
i										8

(b)

Figure 4.1-4 : Array W is shown in (a), and 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.*

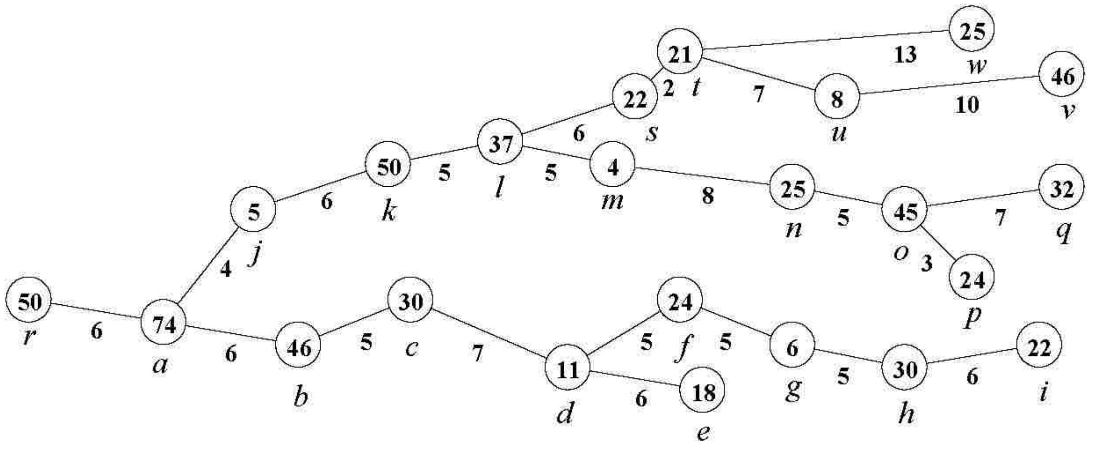


Figure 4.1-5 : Doubly Weighted Tree.
The values of *F (x)* are written inside the circles representing the processes, and the edge between processes *x and y* is labeled with the value *w(x, y).*

We list just a few results for that example:

best(r) = *j*

mincost(r) = max *{W (r, j),F (j)}*

= max {10, 5} = 10

best(g) = *i*

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 $cost(x, y)$, and then sends these values to $Chldrn(y)$. During the second wave, each $y \geq x$ calculates an intermediate value of $mincost(x)$, which is the minimum choice of $cost(x, z)$ 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 $mincost(x)$. All intermediate values calculated during this wave, other than $mincost(x)$ 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 $subtree\_mincost(x, y) = min_{z \geq y}\ mincost(x, z)$, the best candidate for $mincost(x)$ among the processes $z \geq y$. During the broadcast wave of Iteration(x), the values of $W(x, y)$ and $cost(x, y)$ are computed for all y in increasing order. During the convergecast wave of Iteration(x), the values of $subtree\_mincost(x, y)$ are computed for all y, in decreasing order. Finally, when $y = x$ in the convergecast wave, $mincost(x)$ is known.

```
for all $x$ in top down order do {Iteration ($x$)}
    $W(x, x) \leftarrow 0$
    $cost(x, x) \leftarrow F(x)$
    for all $y$ such that $y>x$ in top down order do {Broadcast Wave}
        $W(x, y) \leftarrow W(x, p(y)) + w(p(y),y)$
        $cost(x, y) \leftarrow max\ W(x, y),F(y)$
    end for
    for all $y$ such that $y \geq x$ in bottom up order do {Convergecast Wave}
```

```
            $subtree\_mincost(x, y) \leftarrow min \begin{cases} cost(x, y) \\ min \{subtree\_mincost(x, z): z \in Chldrn(y)\} \end{cases}$
    end for
    $mincost(x) \leftarrow subtree\_mincost(x, x)$
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, Iteration(*x*) does not begin until 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

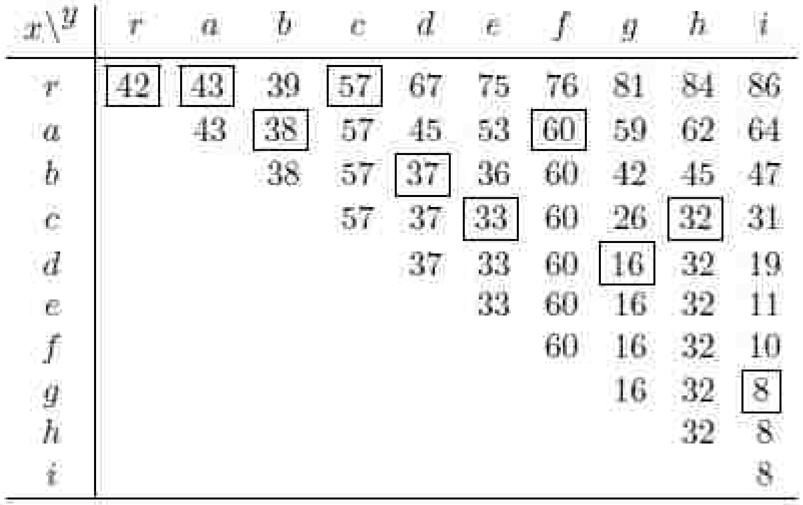
We now give a distributed algorithm for the min-max problem whose time complexity is linear, *i.e., O(h),* although the space complexity is still *O(1)* per process. In order to accomplish this speed up, we reorganize the order of computation, and introduce the concept of a *critical_level.*

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

y	*r*	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*
critical_level(j)	0	0	1	0	2	3	1	4	3	7

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



$x\backslash y$	r	a	b	c	d	e	f	g	h	i
r	$\boxed{42}$	$\boxed{43}$	39	$\boxed{57}$	67	75	76	81	84	86
a		43	$\boxed{38}$	57	45	53	$\boxed{60}$	59	62	64
b			38	57	$\boxed{37}$	36	60	42	45	47
c				57	37	$\boxed{33}$	60	26	$\boxed{32}$	31
d					37	33	60	$\boxed{16}$	32	19
e						33	60	16	32	11
f							60	16	32	10
g								16	32	$\boxed{8}$
h									32	8
i										8

Figure 4.3-1 : cost matrix.
The cost matrix shown in Figure 5.4(b). For each *y*, *x* = *critical_level(y)* is the smallest *x* such that *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* $x_1 < x_2 \le y$ *and* $cost(x_1,y)= F(y)$, *then* $cost(x_2,y)= F(y)$.

Proof: *Suppose* $cost(x_1,y)= F(y)$; *then* $F(y) - W(x_1,y) \ge 0$. *Thus* $F(y) - W(x_2,y)= F(y) - W(x_1,y)+ W(x_1,x_2) \ge W(x_1,x_2) \ge 0$.

Corollary 4.2 : If *critical_level(y)* $\le x \le y$, then $cost(x, y)= F(y)$.

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* ≤ *y*, and *critical_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)* ≤ *F (y),* the value of *critical_level(y)* is set to *level(x).*

For each *y*, the value of *critical_level(y)* can be set several times, but the last value is the correct one. The final values of *critical_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:

$$\mathit{subtree\_minF}\ (x, y) \quad = \quad \min \{F\ (z)\text{: } y \leq z \text{ and critical level}(z) \leq \mathit{level}(x)\}$$

$$\mathit{subtree\_minW}\ (x, y) \quad = \quad \min \{W\ (y, z)\text{: } y \leq z \text{ and critical level}(z) > \mathit{level}(x)\}$$

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 = *best(x)* and *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:     $e \leftarrow level(x)$
 7:     for all $y$ in $T_x$ in top down order do
 8:         delete $W(y, p(x))$ { if it exists, to save space }
 9:         compute $W(y, x)$
10:         if $F(y) \geq W(y, x)$ then
11:             critical $level(y) \leftarrow e$ { 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 $level(y) \geq e$ in bottom-up order do
18:         delete $\mathit{subtree\_minF}(w, e - 1)$ and $\mathit{subtree\_minW}(y, e - 1)$ { if they exist,
              to save space }
19:         if $level(y) \leq e$ then
20:             $\mathit{subtree\_minF}(y, e) \leftarrow \min \begin{cases} F(y) \\ \min_{z \in Chldrn(y)} \{\mathit{subtree\_minF}(z, e)\} \end{cases}$
21:             $\mathit{subtree\_minW}(y, e) \leftarrow \min_{z \in Chldrn(y)} \{W(z, y) + \mathit{subtree\_minW}(z, e)\}$
22:         else
23:             $\mathit{subtree\_minF}(y, e) \leftarrow \min_{z \in Chldrn(y)} \{\mathit{subtree\_minF}(z, e)\}$
24:             $\mathit{subtree\_minW}(y, e) \leftarrow 0$
25:         end if
26:         if $level(y) = l$ then
27:             $mincost(Ty) \leftarrow \min \begin{cases} \mathit{subtree\_minF}(y, e) \\ \mathit{subtree\_minW}(y, e) \end{cases}$
28:         end if
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 $x \neq r$.
- Lines 5–14 give the code for the main loop of the first phase,

which computes all values of *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)* ← *0,* and otherwise *W (y, x)* ← *W (y, p(y))* + *W (p(y),x).*
- The heart of the linear time and space algorithm is the fact that *F (y)* ≥ *W (y, x)* if and only if *critical level(y)* ≤ *level(x).* The first phase calculates all values of *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 *critical_level* itself, we save space, and maintain space complexity of *O(1)* for each process.
- The value of *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)* ≥ *W (y, x).*
- For each *y,* the values of *subtree_minF (y, e)* and *subtree_minW (y, e)* are computed during every iteration for which *e* ≤ *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* = *level(y).* At this point, we offset the value of *subtree_minW (y, e)* by subtracting *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.*

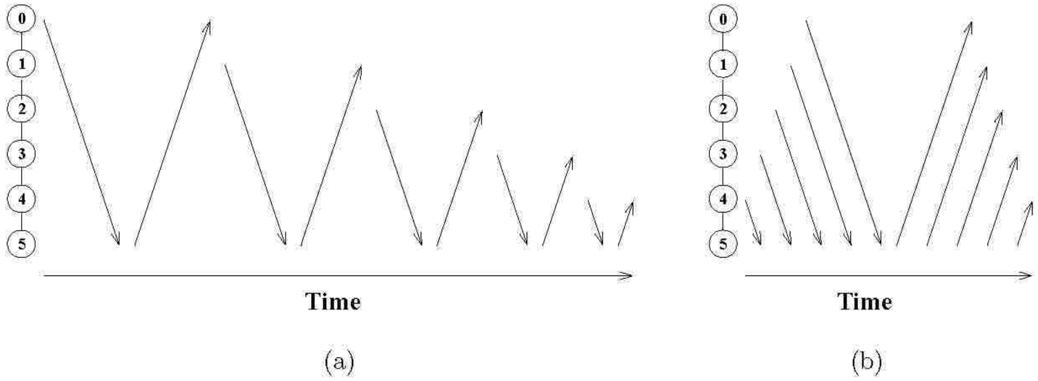


Figure 4.3-2 : Comparison of the algorithms.
Comparison of quadratic time algorithm (a) and the linear time algorithm (b).The algorithms have the same number and length of waves, but the linear time algorithm uses pipelining in a way that cannot be done by the quadratic time algorithm without overlapping.

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* $(a_1,...a_m)$ *and* $(b_1,...b_m)$ *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* $_{\min 1 \leq i \leq m}$ ci = $\min \{M_A,M_B\}$.

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 ∈ A, then $M = c_k = a_k \geq a_i = M_A$, contradiction. On the other hand, if k ∈ 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 *critical_level(y)* is defined for each process *y*, since $W(y, y) = 0 \leq F(y)$.

We next show that the value of *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 *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 *critical_level(y)* ← *l* during Iteration(*x*) of the critical level phase. Also by definition, $F(y) < W(x', y)$ for all $x' < x$, and thus *critical_level(y)* will not be reset during any subsequent iteration.

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

We now apply Lemma 4.3. We can conclude that the linear time algorithm computes the correct value of *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 *choice(x)* $\in$*{F, W }*. *If mincost(x)*= *F (best(x))*, then *choice(x)*= *F*. Otherwise, *choice(x)*= *W*.

x	*r*	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*	*n*	*o*	*p*	*q*	*s*	*t*	*u*	*v*	*w*
p(x)	*r*	*r*	*a*	*b*	*c*	*d*	*d*	*f*	*g*	*h*	*a*	*j*	*k*	*l*	*m*	*n*	*o*	*o*	*l*	*s*	*t*	*u*	*t*
W(x,p(x))	0	6	6	5	7	6	5	5	5	6	4	6	5	5	8	5	3	7	5	2	7	10	13
F(x)	50	74	46	30	11	18	24	6	30	22	5	50	37	4	25	45	24	32	22	21	8	46	25
level(x)	0	1	2	3	4	5	5	6	7	8	2	3	4	5	6	7	8	8	5	6	7	8	7
crt_lev(x)	0	0	0	0	3	2	5	5	2	4	1	0	0	5	2	0	4	3	1	2	6	1	4
best(x)	*j*	*j*	*d*	*d*	*g*	*e*	*g*	*g*	*i*	*i*	*j*	*m*	*m*	*m*	*p*	*p*	*p*	*q*	*u*	*u*	*u*	*v*	*w*
choice(x)	*W*	*F*	*W*	*F*	*W*	*F*	*F*	*F*	*F*	*F*	*F*	*W*	*W*	*F*	*F*	*F*	*F*	*F*	*W*	*F*	*F*	*F*	*F*
mincost(x)	10	5	12	11	10	18	6	6	22	22	5	10	5	4	24	24	24	32	9	8	8	46	25

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 *best(x)* for each *x*. The label on the arrow is *mincost(x).*

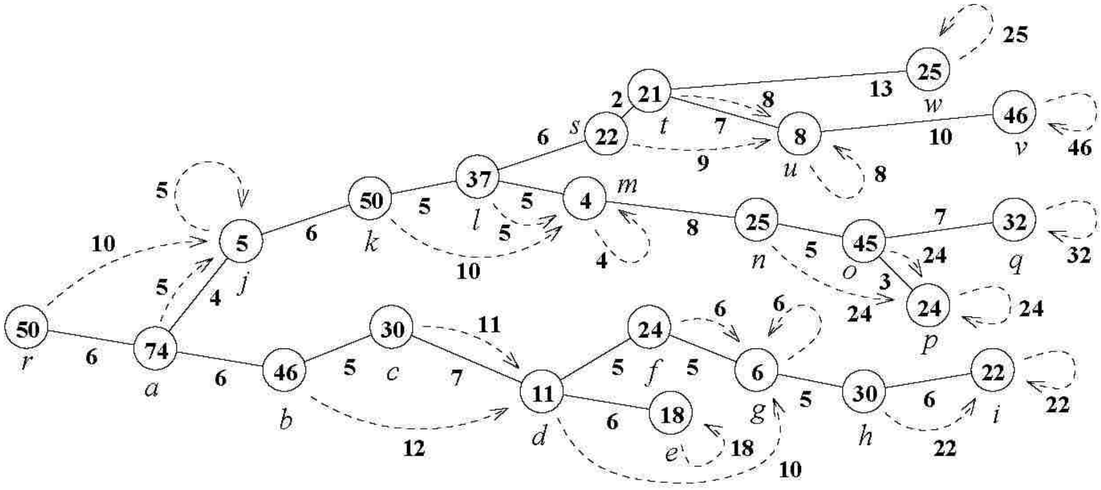


Figure 4.3-3 : *mincost(x) and best(x)*
Arrows are from *x* to *best(x)*, and the label on that arrow is *mincost(x).*

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_{diam}. However, their algorithm uses *O(h + δx)* 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 $LINEAR_{dist}$, $LINEAR_{incr}$, $LINEAR_{wght}$, $LINEAR_{max}$, and $LINEAR_{diam}$, respectively, and all can be considered to be versions of a general algorithm, which we call LINEAR. The space complexity of each of these five algorithms is *O(δx), i.e.,* each process *x* requires only enough space to store *O(δx)* 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 *rank(y, y')* for every cross edge *{y, y'}* of T. This is the only use of the paradigm by $LINEAR_{wght}$, $LINEAR_{dist}$, and $LINEAR_{incr}$.
- $LINEAR_{max}$ and $LINEAR_{diam}$ use the critical level paradigm to compute *critical_level(x)* for each process *x*. We explain that computation in this Section 6.2.
- $LINEAR_{diam}$ 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$_{dist}$ and LINEAR$_{incr}$

We do not need to give separate code for LINEAR$_{incr}$, since the all best swap edges problem for F_{incr} reduces to the problem for F_{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.

```
 1: Preprocessing phase
 2: Ranking phase
 3: (Possibly other critical level phases)
 4: for $1 \leq l \leq d$ do
 5:     for all $y$ such that $level(y) \geq l$ in bottom up order do {Wave $l$}
 6:         Compute $up\_package(y, l)$.
 7:         if $level(y) = l$ then
 8:             Compute $swap\_edge\_cost(y)$.
 9:         end if
10:     end for
11: end for
```

Table 5.2-1 : LINEAR

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

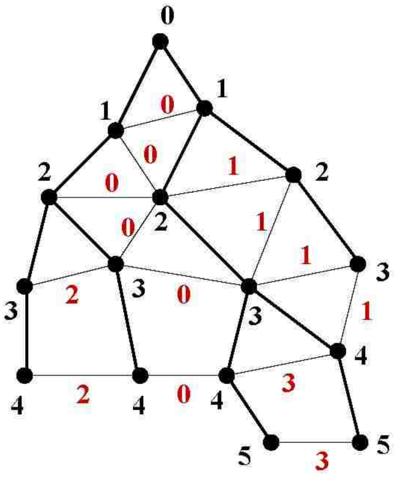


Figure 5.4-1 : Levels of processes (black) and ranks of cross edges (red).

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.

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 *rank(z, z’) < level(x).*

For each $0 \leq l \leq d$, a top-down wave, which we call Wave l, contains the index of the *ancestor_index(y)= 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 *rank(y, y’)* could be reassigned, but the last value of *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.

```
for $0 \leq l \leq d$ in increasing order do {Wave $l$}
    for all $y$ such that $level(y) \geq l$ in top-down order do
        if $level(y) = l$ then
            $ancestor\_index(y, l) \leftarrow index(y)$
        else
            $ancestor\_index(y, l) \leftarrow ancestor\ index(p(y), l)$
        end if
        for all cross edges $\{y, y'\}$ do
            if $index(y') \geq ancestor\ index(y, l)$ then
                $rank(y, y') \leftarrow l$
            end if
        end for
    end for
end for
```

Table 5.4-1 : Ranking Phase

Remark 5.2 If *rank(x, x’)= l*, then, for all $l' \leq l$, the computed value of *rank(x, x’)* will be set to l' during Wave l', and thus the final computed value of *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 *up_package(y, l)* depends on the version of LINEAR. In each case, each process *y* is able to compute *up_package(y, l)* by using the variables stored at *y* during the earlier phases, as well as the variables of *up_package(z, l)* for all $z \in Chldrn(y)$.

5.6 $\text{LINEAR}_{\text{dist}}$

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

For any *l* and any process *y* such that $level(y) \geq l$, *up_package(y, l)* consists of only one variable, namely *subtree_mincost(y, l)*. Let *x* be the ancestor of *y* at level *l*, and let $e = \{x, p(x)\}$. Then *subtree_mincost(y, l)* is defined to be the minimum, over all $e' \in SwapEdges(e)$ 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 *subtree_mincost(y, l)* is given in Table 6.3.

```
Swap N (y, l) ← {y' : {y',y} is a cross edge and rank(y, y') >l}
for all y' such that y' ∈ Swap N (y, l) do
        cost(y, y') ← w(y, y')+ depth(y')
end for
```

$$5\text{: } \textit{subtree mincost}(y, l) \leftarrow \min \begin{cases} \min \{cost(y, y'): y' \in \textit{Swap N}\ (y, l)\} \\ \min \{w(z, y) + \textit{subtree mincost}(z, l): z \in Chldrn(y)\} \end{cases}$$

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

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

5.7 LINEAR$_{wght}$

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

$$\textit{subtree\_mincost}(y, l) \leftarrow \min \left\{ \begin{array}{l} \min \{w(y, y'): y' \in \textit{Swap N}\ (y, l)\} \\ \min \{\textit{subtree\_mincost}(z, l): z \in Chldrn(y)\} \end{array} \right\}$$

CHAPTER 6

LINEAR$_{max}$ and LINEAR$_{diam}$

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

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

F$_{max}$*(T, r, e, e')= ecc$_{Tx}$ (y)+ w(y, y')+ depth(y')*

and

F$_{diam}$(T, r, e, e')= ecc$_{Tx}$ (y)+ w(y, y')+ ecc$_{T\sim x}$ (y')

Recall that T$_{\sim x}$ is the subgraph of *T* obtained by deleting the vertices of *T$_x$* as well as the edge *e.*

In both LINEAR$_{max}$ and LINEAR$_{diam}$, we would like to compute *F$_{max}$(T, r, e, e') or F$_{diam}$(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 *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 *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'}$\in$ SwapEdges(l)* if and only if *rank(y, y') <l.*

Our second problem, also common to both LINEAR$_{max}$ and LINEAR$_{diam}$, is to compute *ecc*$_{Tx}$ *(y).* Recall, from Section 3.1, that

$$ecc_{Tx}(y) = max \begin{cases} height(x) \\ \mu(y, x) \end{cases}$$

where *μ(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 *μ(y, x)* for all choices of *x*. We solve this problem using the paradigm described in CHAPTER 4, executing a *critical_level* phase before the optimization phase; this phase erases all its computation except for one number, called the *critical_level*, at each process. Using that value, *y* decides whether *ecc*$_{Tx}$ *(y)= height(y).* If so, there is no problem, since all values of *height* are computed during the preprocessing step. Otherwise, *y* cannot compute ecc$_{Tx}$ 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$_{diam}$, and that is to compute ecc$_{T\sim x}$ (y'). (The last term of the formula for F$_{max}$ is *depth(y'),* which is computed during preprocessing.) Once again, we are able use the critical level paradigm to define the *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 $ecc_{T\sim x}$ *(y')* if *e'* is the best swap edge for *e*. We give the details in Section 6.4

6.1 LINEAR$_{\max}$

The preprocessing phase of LINEAR$_{\max}$ computes the following variables for each process *x*.

1. *size(x), index(x), height(x), depth(x), level(x), and η(x),* which have the same definitions as given in Section 3.1.
2. *best_child(x), the best child* of *x,* defined to be that $y \in Chldrn$ such that *w(x, y)+ height(y)* > *w(x, z)+ 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 *Chldrn(x)= Ø,* or if there is more than one choice of *y* for which *w(x, y)+ height(y)* is maximum, *best_child(x)* is undefined.
3. We define *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. *secondary_down_path(x)* is defined to be the longest path in T_x that starts at *x* does not pass through *best_child(x).* If *Normal_Chldrn(x)= Ø,* we define *secondary_down_path(x)* to be the trivial path at *x.* Let *secondary height(x)=* W_T *(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 ecc_{Tx} (y)

We first introduce some additional notation.

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

Lemma 6.1 For any process x

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

(b) *Spine(x) is a chain.*

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

$$\begin{aligned} depth(y) &< depth(p(y)) \\ &< W\ (\sigma) \\ &< W\ (\tau\ ) \\ &\leq ecc_{Tx}\ (y) \end{aligned}$$

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

Now, suppose that y is not the best child of $p(y)$. There exists $z \in Chldrn(p(x))$, where $z = y$ *and* $depth(z)+ w(z, p(y)) \geq depth(y)+ w(y, p(y))$. Let σ be the longest path from z to a leaf of T_z, and *let* $\tau =(y, p(y)) + (p(y),z)+ \sigma.$ Then

$$\begin{aligned} depth(y) &< depth(y) + w(y, p(y)) \\ &\leq depth(z) + w(z, p(y)) \\ &< W(\tau) \\ &\leq ecc_{Tx}(y) \end{aligned}$$

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

Part (b) follows immediately from (a).

Continuing our list of terms, we let

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

In Figure 6.2-1 and Figure 6.2-2, we illustrate some of these definitions.

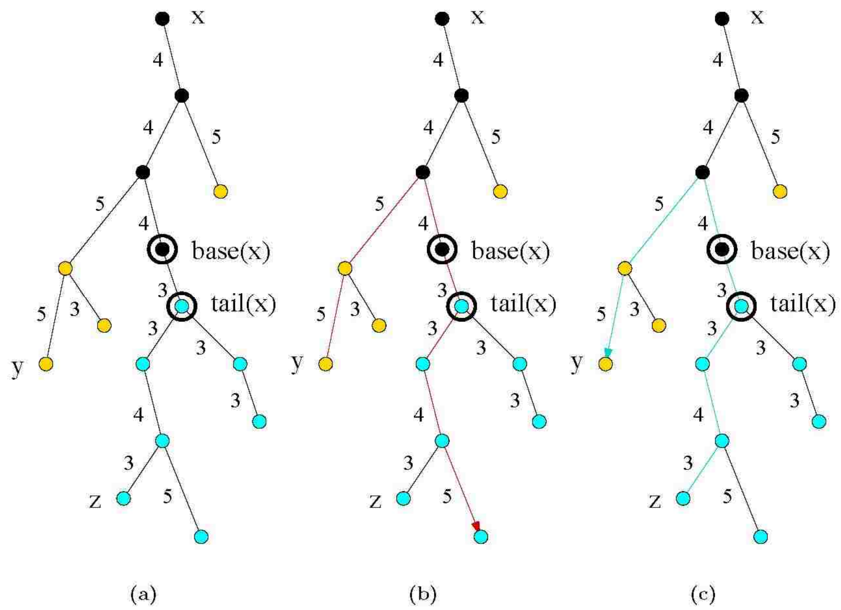


Figure 6.2-1 : Example of T_x, where *tail (x)* is defined.Processes of *Spine(x)* are solid black, processes of *Fan(x)* are cyan, and other processes of *Up(x)* are gold. The red path in (b) has length *$ecc_{Tx}(y)$*, and the cyan path in (c) has length *$ecc_{Tx}(z)$.*

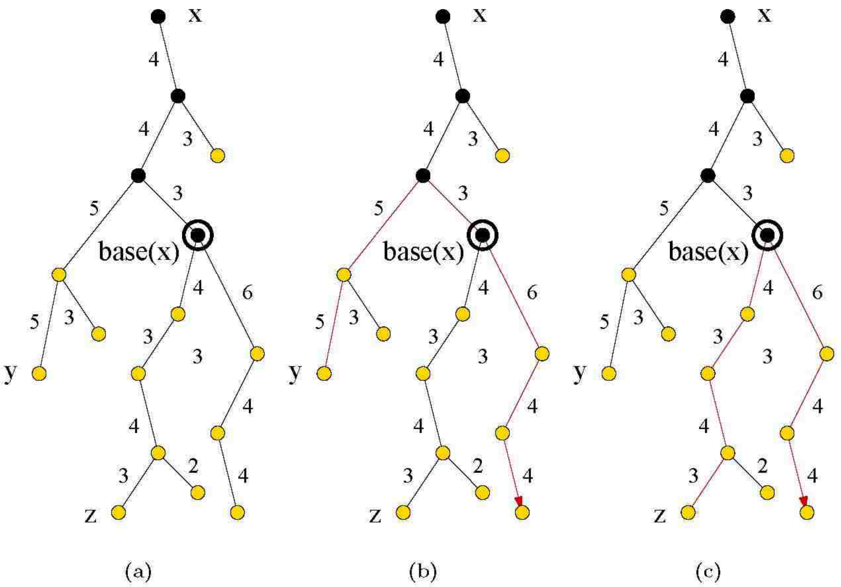


Figure 6.2-2 : Example of T_x, where *tail (x)* is undefined. Processes of *Spine(x)* are solid black, and processes of *Up(x)* are gold. The red path in (b) has length *$ecc_{Tx}(y)$*, and the red path in (c) has length *$ecc_{Tx}(z)$.*

We now characterize *longest_path*$_{Tx}$ *(y)*, the path in T_x from y whose weight is ecc$_{Tx.}$ Recall that *down_path(z)= longest_path*$_{Tz}$ *(z)* for any process *z, i.e., W (down path(z)= depth(z).*

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

(a) $u \in$ *Spine(x).*

(b*) If y* $\in$ *Fan(x), then longest_path*$_{Tx}$ *(y) =path (y,u) + secondary_down_path (u).*

(c) *If y'*$\in$ *Fan(l), then longest_path*$_{Tx}$ *(y)= path(y, u)+ down_path(u).*

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

We now prove (b). By the definition of *u, p(u)* does not lie on *longest_path*$_{Tx}$ *(y), s*ince *best_child(u)* lies on *path(y, u), path(u, s)= secondary_down_path(u),* and we are done.

We now prove (c). If $y \in$ *Spine(l),* then $u = y$ and *longest_path*$_{Tx}$ *(y)= down_path(y),* and we are done. Otherwise, let v be the first member of *Spine(x)* in *path(y, u).* Pick $z \in$ *Chldrn(v)* $\cap$ *path(y, v).* Then $z \neq$ *best_child(v)* since *y'*$\notin$ *Fan(x).* Thus, *down_path(v)= longest_path*$_{Tx}$ *(v)* does not contain *z,* and hence *longest_path*$_{Tx}$ *(y)= path(y, v)+ 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$_{max}$, 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 *critical_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.

Lemma 6.3

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

(b) $y \in$ *Up(l) if and only if critical_level(x)* $\leq l \leq$ *level(x).*

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

$$ecc_{Tx'}(y) \geq ecc_{Tx}(y) > depth(y)$$

and hence $y \in$ *Up(l')* by definition. Part (b) follows immediately.

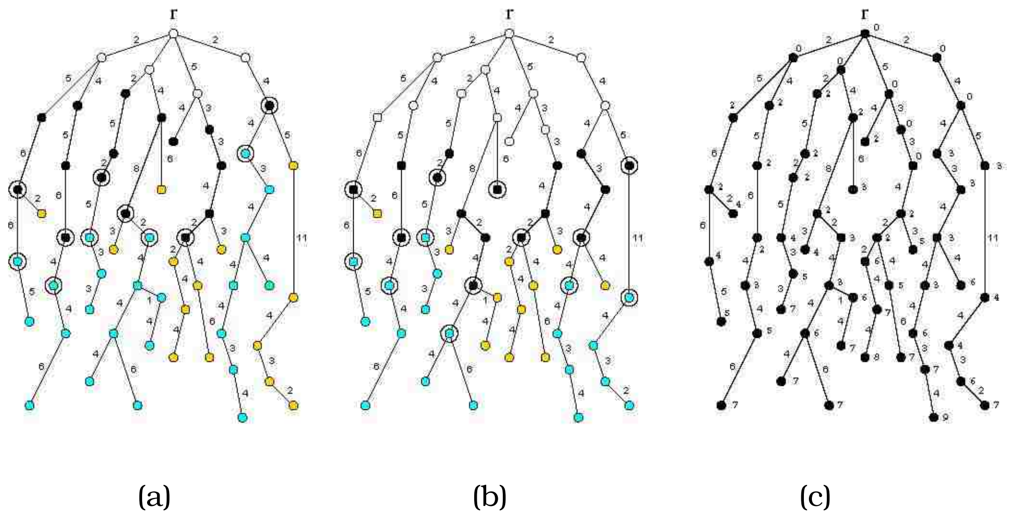


Figure 6.2-3 : illustration of Lemma 6.2
In the tree shown, the weights of the edges are proportional to vertical distance in the figure. *Spine(l)* is the union of chains headed by all processes at level *l*. Processes of *Spine(l)* are solid black. Processes of *Fan(l)* are filled in cyan; other processes of *Up(l)* are filled in gold. Processes in *Base(l)* and *Tail(l)* are circled in black. *Spine(2), Up(2),* and *Fan(2)* are shown in (a). *Spine(3), Up(3),* and *Fan(3)* are shown in (b). Note that *Up(3)* $\subseteq$ *Up(2).* The values of *critical_level* are shown in (c). Note that $x \in Spine(l)$ if and only if $critical\_level(x) \le l \le level(x)$.

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

```
for $0 \le l \le d$ in decreasing order do {Wave $l$}
    for all $x$ such that $level(x) = l$ concurrently do
        for all $y \in T_x - x$ in top down order do
            $p \leftarrow p(y)$
            $\mu(y, x) \leftarrow \max\{\mu(p, x) + w(y, p), \eta(y)\}$
            if $\mu(y, x) \le depth(y)$ then
                $critical\_level(y) \leftarrow l$
            end if
        end for
    end for
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 LINEAR$_{\max}$

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

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

The optimization phase is a dynamic programming algorithm, where $up\_package(y, l)$ 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 $local\_cost(y, l) = min\ \{w(y, y')+ depth(y'): y' \in Swap\ N\ (y, l)\}$.

1. For $y \in Up(l)$, $up\_package(y, l)$ contains

   (a) $min\_up\_cost(y, l) = \min\ \{local\_cost(z, l)+ W\ (y, z): z \in T_y\}$.

2. If $y \in Spine(y)$, *then* $up\_package(y)$ contains

   (a) $min\_normal\_cost(y, l) = \min\ \{ local\_cost(z, l)+ W\ (y, z): z \in T_y$ *and* $z \neq T_{best\ child(y)}\}$

   (b) $min\_fan\_cost(y, l) = \min\ \{local\_cost(z, l)+ W\ (y, z): z \in T_y \cap$

Fan(l)}. (Note that min_fan_cost(y, l)= ∞ *if Ty* $\cap$ *Fan(l)=* $\emptyset$*.)*

(c) *subtree_mincost(y, l)* = min *{local_cost(z, l)+ ecc*$_{Ty}$ *(z): z* $\in T_y$*. }*

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.

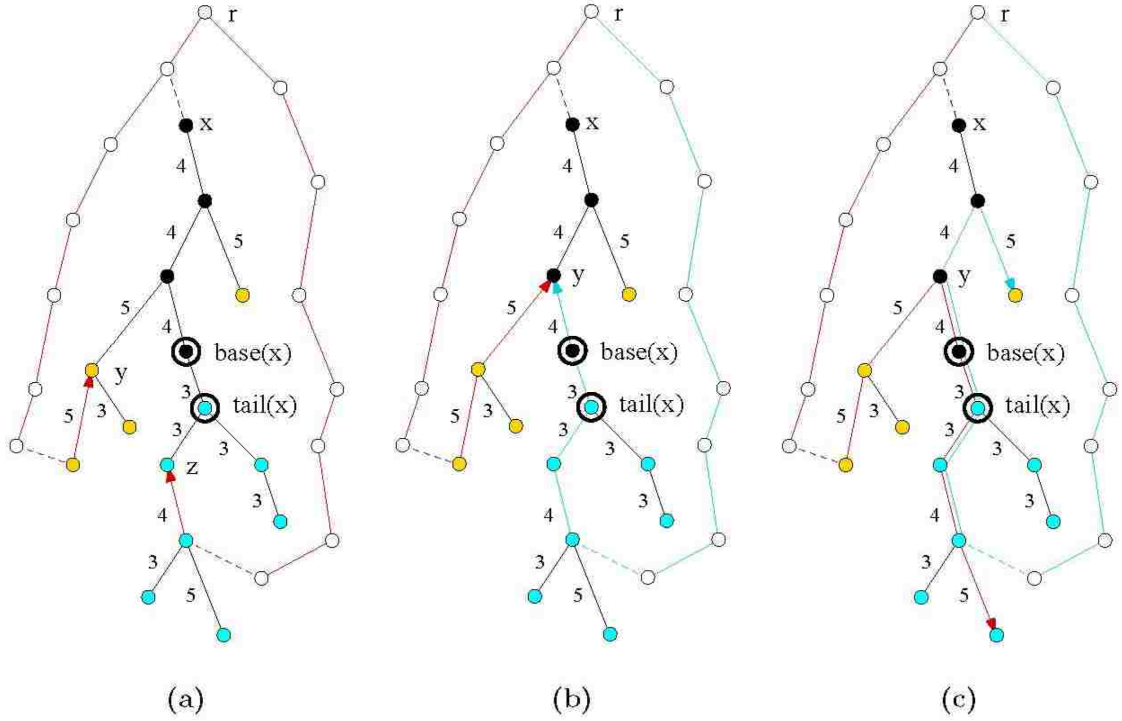


Figure 6.3-1 : *swap_edge_cost, min_up_cost and min_fan_cost*
Functions in up_package of descendants of *x*, where level (x) = *l* = 2. In (a), we show *min_up_cost(y, 2)* and *min_up_cost(z, 2)*, where *y, z* ∈ *Up(2).* In (b), we show *min_normal_cost(y, 2)* and *min_fan_cost(y, 2), where y* ∈ *Spine(2).*In (c), we show two paths whose lengths are candidates for *swap_edge_cost(x) = 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 LINEAR$_{max}$.

```
$local\_cost(y, l) = \min \{w(y, y') + depth(y') : y' \in Swap\ N\ (y, l)\}$
**if** $y \in Up(l)$ **then**
    $min\_up\_cost(y, l) \leftarrow \min \begin{cases} local\_cost(y, l) \\ \min \{min\_up\_cost(z, l) : z \in Chldrn(y)\} \end{cases}$
**else** $\{y \in Spine(l)\}$
    $min\_normal\_cost(y, l) \leftarrow \min \begin{cases} local\_cost(y, l) \\ \min \{min\_up\_cost(z,l) + w(y,z) : z \in Normal\_Chldrn(y)\} \end{cases}$
    if $best\_child(y)$ is defined then
        $z \leftarrow best\ child(y)$
        if $z \in Spine(l)$ then
            $min\_fan\_cost(y, l) \leftarrow min\_fan\_cost(z, l) + w(z, y)$
        else
            $min\_fan\_cost(y, l) \leftarrow min\_up\_cost(z, l) + w(z, y)$
        end if
        $subtree\_mincost(y, l) \leftarrow \min \begin{cases} min\_normal\_cost(y, l) + height(y) \\ min\_fan\_cost(y,l) + secondary\_height\ (y) \\ subtree\_mincost(z, l) \end{cases}$
    else {$y = base(x)$, and $tail(x)$ undefined}
        $min\_fan\_cost(y, l) \leftarrow \infty$
        $subtree\_mincost(y, l) \leftarrow min\_normal\_cost(y, l) + height(y)$
    end if
end if
```

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

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* $F_{max}(T, r, e, e') = W\ (longest\_path_{Tx}(z)) + w(z, z') + depth(z')$, 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 $y \in$ *Spine(l)* and if *Fan(l)* $\cap$ *Tx* = $\emptyset$, then *up_package(y, l)* also needs only one variable, namely *subtree_mincost(y, l),* which is the minimum value of ecc_{Ty} *(z)+ w(z, z')+ depth(z')* over all *{z, z'}* $\in$ *SwapEdges(x)* such that $z \in T_y$.

If $y \in$ *Spine(l)* and *Fan(l)* $\cap$ T_x = $\emptyset$, then *up_package(y, l)* needs two variables, *subtree_mincost(y, l),* as described above, and *min_fan_cost(y, l),* which is the minimum value of *W (path(z, y))* + *w(z, z')+ depth(z')* over all *{z, z'}* $\in$ *SwapEdges(x)* such that $z \in T_y \cap$ *Fan(l).*

At most one of those two values will be needed to compute *swap_edge_cost(x).* If *subtree_mincost(y, l)* ≥ *min_fan_cost(y, l)+μ(y, x),* then *min_fan_cost(y, l)* could be discarded; otherwise *subtree_mincost(y, l)* could be discarded. But since *y* does not know the value of *μ(y, x),* it cannot discard either.

6.4 Overview of LINEARdiam

Recall that $F_{diam}(T, r, e, e')= ecc_{Tx}(y)+ w(y, y')+ ecc_{T\sim x}(y')$, *where e = {x, p(x)}, y* $\in T_x$, *and {y, y'}* $\in$ *SwapEdges(x).* $\mathrm{LINEAR}_{\mathrm{diam}}$ has all the complexity of $\mathrm{LINEAR}_{\mathrm{max}}$, since it must handle the impossibility of calculating ecc_{Tx} *(y)* during the optimization phase; it also has additional complexity due to the impossibility of calculating $ecc_{T\sim x}$ *(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 $ecc_{Tx}(y)+ w(y, y')+ ecc_{T\sim x}(y')$ has the largest value:

1. $depth(y)+ w(y, y')+ restr\_ecc(y')$
2. $\mu(y, x)+ w(y, y')+ restr\_ecc(y')$
3. $depth(y)+ w(y, y')+ secondary\_ecc(y')$
4. $\mu(y, x)+ w(y, y')+ secondary\_ecc(y')$

where $restr\_ecc(y')$ *and* $secondary\_ecc(y')$, defined below, are computed during the preprocessing phase.

Otherwise, we only need to use $critical\_level(y)$ to choose among the two formulas

1. $depth(y)+ w(y, y')+ ecc_{Tx}(y')$
2. $\mu(y, x)+ w(y, y')+ ecc_{Tx}(y')$

That decision can be made at the time that $up\_package(y, l)$ is computed, for $l = level(x)$, 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 LINEAR$_{\text{diam}}$

The preprocessing phase of LINEAR$_{\text{diam}}$ computes all the same variables as the preprocessing phase of 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. *branch(x)*, provided $x \neq r$, which is defined to be that value of i such that $x \in S_i$. After *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 $h_i = w(c_i, r) + height(c_i)$. The value i is then broadcast to all processes in S_i.
2. h_1, h_2, *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 *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 $x = best\_child(p(x))$.

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

   $$chain\_level(x) = \begin{cases} level(x) & \text{if } x = best\_child(p) \text{ and} \\ & chain\_level(p) = level(p) \\ chain\_level(p) & \text{otherwise} \end{cases}$$

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

   The values of $local\_\mu(x)$ are computed in a broadcast wave, using the definition

$$local\_\mu(x) = \begin{cases} 0 & \text{if } x = ci \\ w(x, p(x)) + max\ \{local\_\mu(p(x)), \eta(x)\} & \text{otherwise} \end{cases}$$

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

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

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:

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

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

   The *full eccentricity* of x.

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

   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. $$restr\_ecc(x) = \begin{cases} ecc_T - S_2(x) & \text{if } x \in S_1 \\ ecc_T - S_1(x) & \text{otherwise} \end{cases}$$

The *restricted eccentricity* of *x*. We compute the restricted eccentricity of all *x* as follows.

$$restr\_ecc(x) = \begin{cases} max \begin{Bmatrix} local\_\varphi(x) \\ secondary\_ecc(x) \end{Bmatrix} & if\, \mathrm{x} \in \mathrm{S1} + \mathrm{S2} \\ secondary\_ecc(x) & otherwise \end{cases}$$

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 *secondary_ecc(x),* that path need not contain *r*.)

Figure 6.5-1 below illustrates the definitions of *local_φ(x), ecc*$_T$ *(x), and restr_ecc(x).*

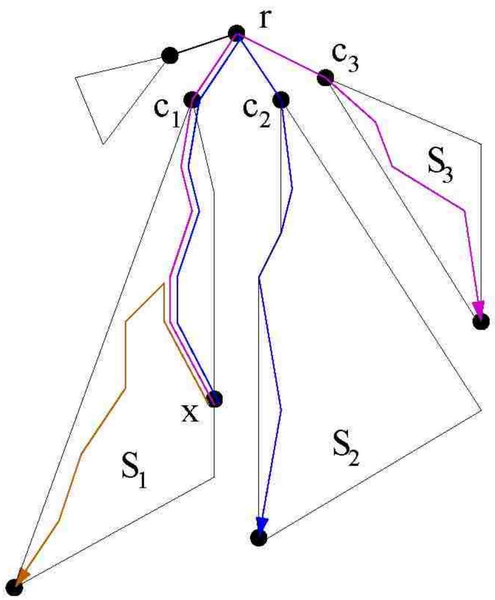


Figure 6.5-1 : restricted, local and full eccentricity.
If $x \in S1$, the local eccentricity of x is the length of the longest path from x in S_1, shown in brown. The full eccentricity of x is the length of the longest path from x to a point in S_2, shown in blue. The restricted eccentricity of x is either the length of the longest path from x to a point in S_3, shown in magenta, or *local_φ(x)*, whichever is greater.

6.6 Special Levels

We define *special_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 $ecc_{T\sim x}(y')$ during the optimization phase of $\text{LINEAR}_{\text{diam}}$.

For any $x \in C_1$, we define

$$A(x)=\{ y' \notin S1 : restr\_ecc(y') \geq depth(y')+ avoid(x) \}$$

$$B(x)= \{y' \notin S1 : restr\_ecc(y') < depth(y')+ avoid(x) \}$$

For any $x \in C_2$, we define

$$A(x) = y' \in S1 : restr\_ecc(y') \geq depth(y') + avoid(x)$$

$$B(x) = y' \in S1 : restr\_ecc(y') < depth(y') + avoid(x)$$

Finally, let

$$A(l) = \bigcup \{A(x): level(x) = l\}$$

$$B(l) = \bigcup \{B(x): level(x) = l\}$$

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.

```
initialize $special\_level(y) \leftarrow \infty$ for all $y \in S_1$
for all $x \in C_2$ in bottom-up order do
    $l = level(x)$
    for all $z \in C_2$ which are ancestors of $x$ in bottom-up order do
        copy $avoid(x)$ to $z$
    end for
    copy $avoid(x)$ to $r$
    for all $y \in S_1$ in top down order do
        copy $avoid(x)$ to $y$
        if $restr\ ecc(y) < depth(y) + avoid(x)$ then
            $special\_level(y) \leftarrow l$
        end if
    end for
end for
```

Table 6.6-1 : Special Level Phase for S_1

The code for computing *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 *special_level(y)* are erased after the wave passes.

The value of *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 *special_level(y)*= ∞ when the phase is done.

The code for computing *special_level(x)* for $x' \in S_1$, given in Table 6.6-2, is very similar.

```
 1: initialize $special\ level(y) \leftarrow \infty$ for all $y \notin S1$
 2: for all $x \in C_1$ in bottom-up order do
 3:     $l = level(x)$
 4:     for all $z \in C_1$ which are ancestors of $x$ in bottom-up order do
 5:         copy $avoid(x)$ to $z$
 6:     end for
 7:     for all $y \notin S_1$ in top down order do
 8:         copy $avoid(x)$ to $y$
 9:         if $restr\_ecc(y) > depth(y) + avoid(x)$ then
10:             $special\_level(y) \leftarrow l$
11:         end if
12:     end for
13: end for
```

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 *special_level(x).*

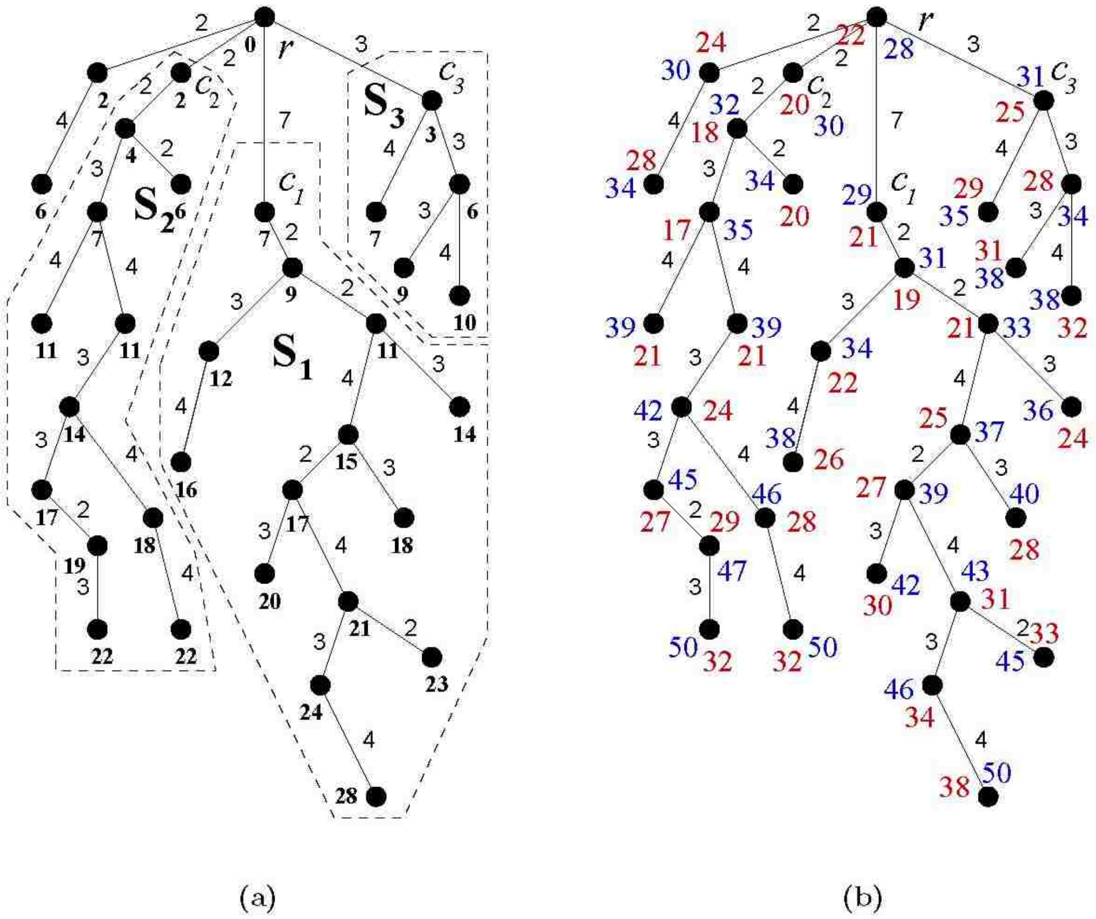


Figure 6.6-1 : Computation of Special Levels.
(a) shows the depth of all processes, as well as subtrees S_1, S_2, and S_3.
(b) shows *ecc_T (x)* in blue and *restr_ecc(x)* in red for all *x*.

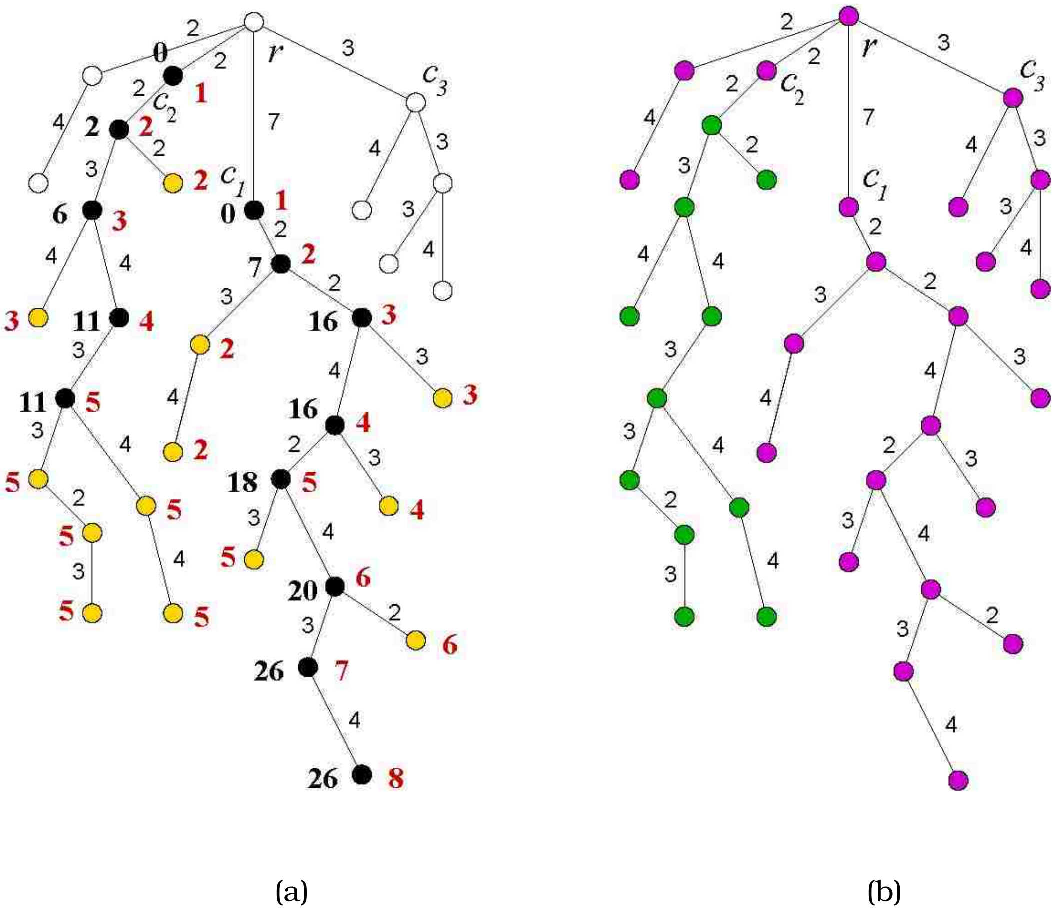
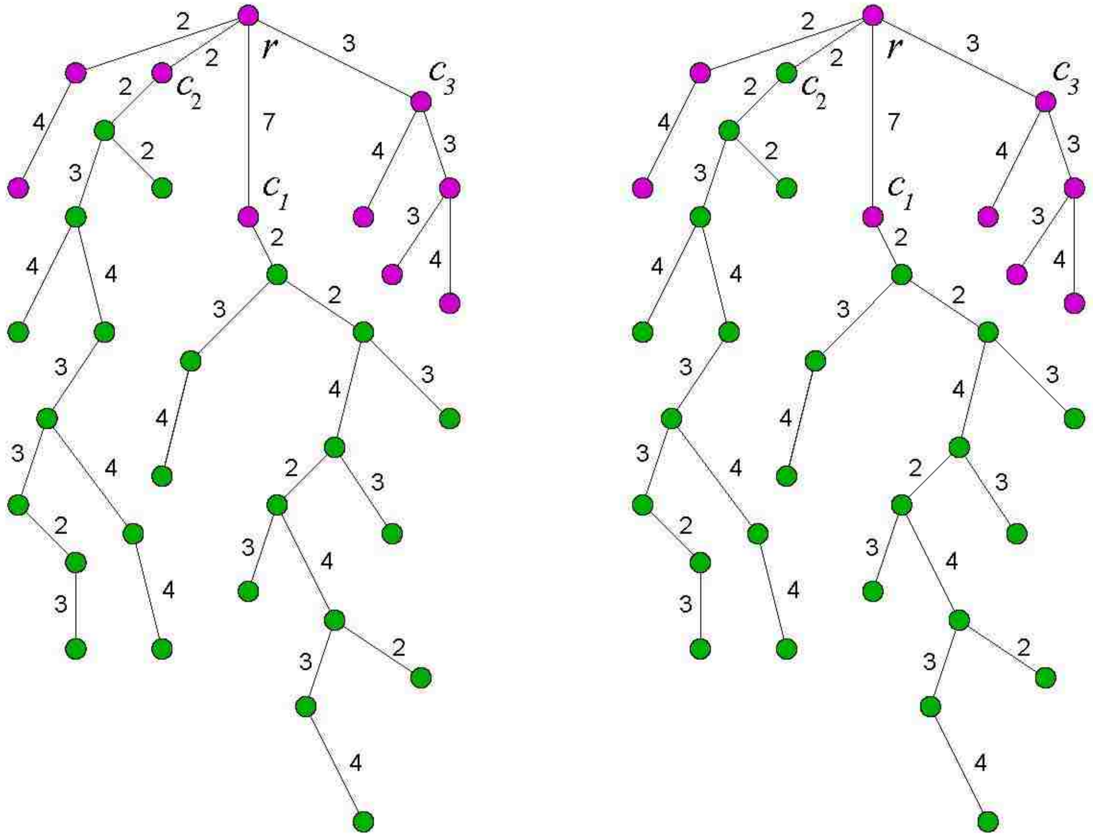
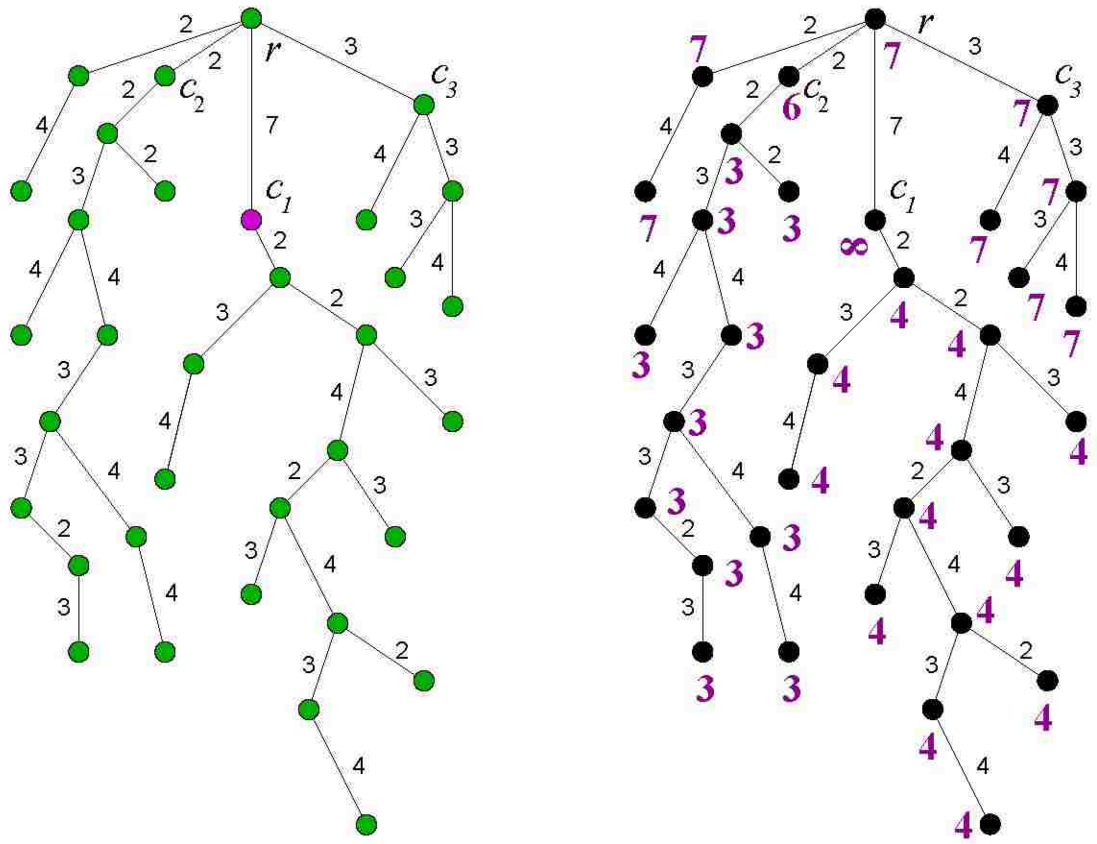


Figure 6.6-1 : (Continued): (c) shows *avoid(x)* for all $x \in C_1 + C_2$ in black, and *chain_level(x)* for all $x \in S_1 + S_2$ in red. Processes of $C_1 + C_2$ are black, and other processes of $S_1 + S_2$ are gold. (d) shows processes of the set A(3) in magenta, and processes of the set B(3) in green.



(c) (d)

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.



(e)

(f)

Figure 6.6-1 : (Continued): For any $l \geq 7$, A(l)= {c₁}, 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:*

$$ecc_{T\text{-}x}\,(y') = \begin{cases} ecc_T\,(y') \text{ if } y' \in C(y, l) \\ restr\_ecc(y') \text{ if } y' \in D(y, l) \\ depth(y') + avoid(x) \text{ if } y' \in E(y, l) \end{cases}$$

The partition is implemented by y as follows. For any $y' \in Swap\_N\,(y, l)$:

- If $y \in S_1$ and $chain\_level(y) \geq l$ then
  - $y' \in C(y, l)$ if $y' \in S1$.
  - $y' \in D(y, l)$ if $y' \notin S1$ and *special level*$(y') > l$.
  - $y' \in E(y, l)$ if $y' \notin S1$ and *special level*$(y') \leq l$.
- If $y \in S_2$ and $chain\_level(y) \geq l$ then
  - $y' \in C(y, l)$ if $y' \notin S_1$.
  - $y' \in D(y, l)$ if $y' \in S_1$ and *special level*$(y') > l$.
  - $y' \in E(y, l)$ if $y' \in S_1$ and *special level*$(y') \leq l$.
- If $y \in S_1 + S_2$ *and 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

```
for $1 \le l \le d$ do
  for all $y$ such that $level(y) \ge l$ in bottom up order do
    $local\_costC(y, l) = min \{w(y, y')+ ecc_{Tx} (y'): y' \in C(y, l)\}$
    $local\_costD(y, l) = min \{w(y, y')+ restr\ ecc(y'): y' \in D(y, l)\}$
    $local\_costE(y, l) = min \{w(y, y\ I)+ depth(y'): y' \in D(y, l)\}$
    if $y \in Up(l)$ then
        $min\_up\_costC(y, l) \leftarrow \min \begin{cases} local\_costC(y, l) \\ min\{min\_up\_costC(z, l): z \in Chldrn(y)\} \end{cases}$
        $min\_up\_costD(y, l) \leftarrow \min \begin{cases} local\_costDy, l) \\ \mathrm{m}in\{min\_up\_costD(z, l): z \in Chldrn(y)\} \end{cases}$
        $min\_up\_costE(y, l) \leftarrow \min \begin{cases} local\_costE(y, l) \\ min\{min\_up\_costE(z, l): z \in Chldrn(y)\} \end{cases}$
     else $\{y \in Spine(l)\}$
        $min\_normal\_costC(y,l) \leftarrow \min \begin{cases} local\_costC(y, l) \\ \min\{min\_up\_costC(z,l):z \in Normal\_Chldrn(y)\} \end{cases}$
        $min\_normal\_costD(y,l) \leftarrow \min \begin{cases} local\_costD(y, l) \\ \min\{min\_up\_costD(z,l):z \in Normal\_Chldrn(y)\} \end{cases}$
        $min\_normal\_costE(y,l) \leftarrow \min \begin{cases} local\_costE(y, l) \\ \min\{min\_up\_costE(z,l):z \in Normal\_Chldrn(y)\} \end{cases}$
          if $best\_child(y)$ is defined then
             $z \leftarrow best\ child(y)$
             if $z \in Spine(l)$ then
                 $min\_fan\_costC(y, l) \leftarrow min\_fan\_costC(z, l)+ w(z, y)$
                 $min\_fan\_costD(y, l) \leftarrow min\_fan\_costD(z, l)+ w(z, y)$
                 $min\_fan\_costE(y, l) \leftarrow min\_fan\_costE(z, l)+ w(z, y)$
             else
                 $min\_fan\_costC(y, l) \leftarrow min\_up\_costC(z, l)+ w(z, y)$
                 $min\_fan\_costD(y, l) \leftarrow min\_up\_costD(z, l)+ w(z, y)$
                 $min\_fan\_costE(y, l) \leftarrow min\_up\_costE(z, l)+ w(z, y)$
             end if
```

```
25:             $subtree\_mincostC(y, l) \leftarrow \min \begin{cases} min\_normal\_costC(y, l) + height(y) \\ min\_fan\_costC(y, l) + secondary\_height(y) \\ subtree\ mincostC(z, l) \end{cases}$
26:             $subtree\_mincostD(y, l) \leftarrow \min \begin{cases} min\_normal\_costD(y, l) + height(y) \\ min\_fan\_costD(y, l) + secondary\_height(y) \\ subtree\ mincostD(z, l) \end{cases}$
27:             $subtree\_mincostE(y, l) \leftarrow \min \begin{cases} min\_normal\_costE(y, l) + height(y) \\ min\_fan\_costE(y, l) + secondary\_height(y) \\ subtree\ mincostE(z, l) \end{cases}$
28:         else
29:             $min\_fan\_costC(y, l) \leftarrow \infty$
30:             $min\_fan\_costD(y, l) \leftarrow \infty$
31:             $min\_fan\_costE(y, l) \leftarrow \infty$
32:             $subtree\_mincostC(y, l) \leftarrow min\_normal\_costC(y, l) + height(y)$
33:             $subtree\_mincostD(y, l) \leftarrow min\_normal\_costD(y, l) + height(y)$
34:             $subtree\_mincostE(y, l) \leftarrow min\_normal\_costE(y, l) + height(y)$
35:         end if
36:       end if
37:       if $level(y) = l$ then
38:         $swap\_edge\_cost(y) \leftarrow \min \begin{cases} subtree\_mincostC(y, l) \\ subtree\_mincostD(y, l): \\ subtree\_mincostE(y, l) \end{cases}$
39:      end if
40:   end for
41: end for
```

Table 6.7-1 : Optimization Phase of LINEAR$_{\text{diam}}$

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 $ecc_{T\sim x}\ (y')$ in LINEAR$_{\text{diam}}$. 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 6.3-1corresponds 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 LINEAR$_{\text{diam}}$

Finally, we summarize the algorithm LINEAR$_{\text{diam}}$ in Table 6.7-2 below. The time complexity of each phase, and hence of LINEAR$_{\text{diam}}$, is $O(h)$, and no more than $O(\delta_x)$ 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 : LINEAR$_{\text{diam}}$

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'* $\notin$ *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 stared 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_{sum}.

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

BIBLIOGRAPHY

[1] H. Attiya and J. Welch. 2004, Distributed Computing: Fundamentals, Simulations, and Advanced Topics.

[2] H. Booth and J. Westbrook. 1994, A linear algorithm for analysis of minimum spanning and shortest-path trees of planar graphs. Algorithmica 11(4), pp. 341-352.

[3] C. Cheng, I. Cimet and S. Kumar. A protocol to maintain a minimum spanning tree in a dynamic topology. Presented at Symposium Proceedings on Communications Architectures and Protocols.

[4] B. Das and M. C. Loui. 2008, Reconstructing a minimum spanning tree after deletion of any node. Algorithmica 31(4), pp. 530-547.

[5] B. Dixon, M. Rauch and R. E. Tarjan. 1992, Verification and sensitivity analysis of minimum spanning trees in linear time. SIAM Journal on Computing 21pp. 1184-1184.

[6] P. Flocchini, A. M. Enriques, L. Pagli, G. Prencipe and N. Santoro. 2006, Point-of-failure shortest-path rerouting: Computing the optimal swap edges distributively. IEICE Trans. Inf. Syst. 89(2), pp. 700-708.

[7] P. Flocchini, T. Enriquez, L. Pagli, G. Prencipe and N. Santoro. 2007, Distributed computation of all node replacements of a minimum spanning tree. Euro-Par 2007 Parallel Processing pp. 598-607.

[8] P. Flocchini, L. Pagli, G. Prencipe, N. Santoro and P. Widmayer. 2008, Computing all the best swap edges distributively. Journal of Parallel

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

[9] B. Gfeller, N. Santoro and P. Widmayer. 2007, A distributed algorithm for finding all best swap edges of a minimum diameter spanning tree. Distributed Computing pp. 268-282.

[10] M. Grotschel, C. L. Monma and M. Stoer. 1995, Design of survivable networks. Handbooks in Operations Research and Management Science 7pp. 617–672.

[11] D. B. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning trees. Presented at Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures.

[12] E. Korach, D. Rotem and N. Santoro. 1984, Distributed algorithms for finding centers and medians in networks. ACM Transactions on Programming Languages and Systems (TOPLAS) 6(3), pp. 401.

[13] E. Nardelli, G. Proietti and P. Widmayer. 2004, Nearly linear time minimum spanning tree maintenance for transient node failures. Algorithmica 40(2), pp. 119-132.

[14] E. Nardelli, G. Proietti and P. Widmayer. 2003, Swapping a failing edge of a single source shortest paths tree is good and fast. Algorithmica 35(1), pp. 56-74.

[15] E. Nardelli, G. Proietti and P. Widmayer. 1998, Finding all the best swaps of a minimum diameter spanning tree under transient edge failures. Algorithms—ESA'98 pp. 1-1.

[16] R. E. Tarjan. 1979, Applications of path compression on balanced

trees. Journal of the ACM (JACM) 26(4), pp. 690-715.

[17] Y. H. Tsin. 1988, On handling vertex deletion in updating minimum spanning trees. Information Processing Letters 27(4), pp. 167-168.

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. Yoohwan Kim, Ph.D
Committee Member, Dr. Emma Regentova, Ph.D.