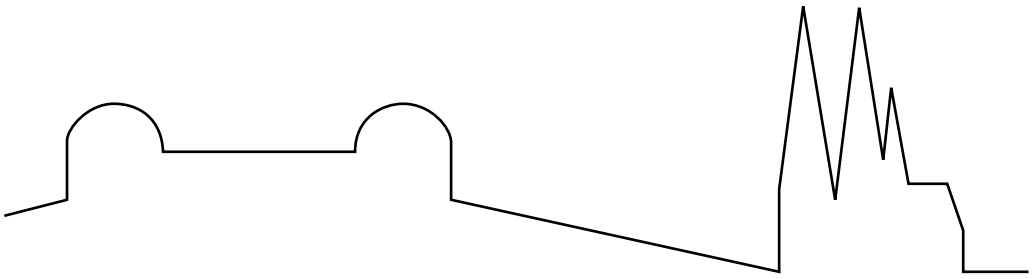
UPPSALA THESES IN COMPUTING SCIENCE
No. 17/93

Cycletrees: a Novel Class of Interconnection Graphs

Margus Veanes



Thesis for the Degree of
Licentiate of Philosophy

ISSN 0283-359X

UPMAIL
Computing Science Department
Uppsala University
Box 311
S-751 05 UPPSALA
Sweden

Abstract

We introduce a new class of graphs that we call *cycletrees*. A cycletree includes a basic binary tree and has a unique Hamiltonian cycle. We argue that cycletrees reflect the communication patterns of several common parallel programming paradigms and can be used in various fields of parallel computation. Several minimality, optimality and planarity properties are proved for cycletrees. A subclass of cycletrees is identified as *natural* cycletrees through an inductive definition. Methods for generating natural cycletrees, given either a set of vertices, a cycle or a basic binary tree, are presented. We present an inexpensive and simple routing algorithm for natural cycletree networks. A superfast parallel algorithm is presented, which dynamically establishes the shortest path router data for the given router algorithm. We compare cycletrees with other interconnection graphs that have been designed for partly similar reasons.

Acknowledgements

I wish to thank all those who have helped me with my thesis. In particular, I would like to thank my supervisor Jonas Barklund, and my colleague Sven-Olof Nyström for invaluable comments on earlier versions of this thesis. Special thanks goes to Håkan Millroth for providing me with literature that has proved to be of great relevance during the course of this work. I also want to thank Marianne Ahrne for checking the English language.

Finally, I am most grateful to my wife Katrine for all the support and encouragement I have received through the past years.

Contents

List of Figures

Chapter 1

Introduction

A *cycletree* is a graph which includes a basic binary tree and has a unique Hamiltonian cycle. A *minimal* cycletree has the minimal number of edges that are not part of the binary tree.

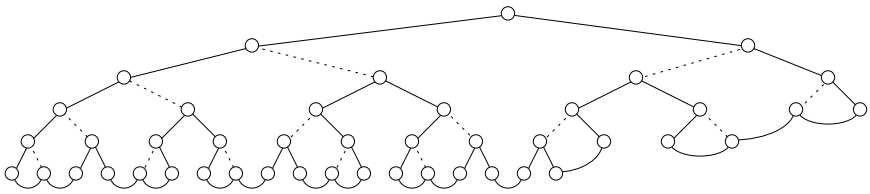


Figure 1.1: *A cycletree. The curved lines correspond to edges that are not part of the binary tree and the dashed lines correspond to edges that are not part of the cycle.*

We argue that a cycletree is well suited as an interconnection graph of a network of processing entities or nodes that are used to execute parallel programs. Communication between arbitrary nodes is possible, but we favour the following communication patterns between the nodes. (Let the nodes be numbered from 1 to N.)

1. Broadcasting or distributing data from node 1 to all the other nodes.
2. Collecting or combining data from all nodes to node 1.
3. Communication between nodes i and $i + 1$, $1 \leq i < N$, and possibly between nodes 1 and N.

These communication patterns occur frequently in many parallel programming paradigms, as we shall see.

1.1 Parallel programming paradigms

The following programming paradigms and techniques are frequently used in parallel computation; the above communication patterns occur often.

1. Binary tree paradigm.
2. Growing by doubling paradigm.
3. Compute-aggregate-broadcast technique.
4. Parallel divide-and-conquer technique [50, 51].

5. Systolic paradigm based on a linear or circular array.

6. Master-slave paradigm.

A widely recognized interconnection graph that supports the communication patterns 1 and 2 and the first four paradigms is the binary tree [17, 30, 46, 66]. Clearly, an interconnection graph that supports the third communication pattern and the fifth paradigm is a circular array (an N-cycle, a ring). For a survey of paradigms 1–5 and related algorithms see, e.g., Chaudhuri [17]. Let us examine the last two paradigms more closely.

1.1.1 The master-slave paradigm

The master-slave paradigm (process farming [32], pure agenda parallel approach [15]) is a technique that is often used when the computation at hand can be divided into several independent subcomputations. Problems that can be computed in this way are often called 'embarrassingly parallel'. Both binary trees and circular or linear arrays are, due to their simple structure, often used as virtual interconnection graphs of the master and the slaves.

1.1.2 The systolic paradigm

The systolic paradigm [37, 59] is, arguably, the most widely used paradigm for non-shared memory parallel computation. The concept of systolic arrays was introduced by H. T. Kung [46, chapter 8 by H. T. Kung]. In a systolic algorithm the entire computation is divided into subcomputations, each of which is assigned to a node. There is a flow of data through the nodes and the operations are pipelined to balance input/output, computation and processing. Systolic algorithms are required to have the following properties.

- Locality of computation and communication. Each node can communicate with only a subset of its neighbours.
- Regular communication structure.

Computation can proceed either synchronously or asynchronously. Asynchronous systolic algorithms are also known as *soft-systolic* or *wavefront* algorithms.

Several systolic algorithms have been developed for linear and circular arrays. In general, problems that belong to the pipeline complexity class [36] can be solved efficiently on such structures. Matrix-vector multiplication, recurrence evaluation and priority queues are good examples [36, 42].

Systolic algorithms are often expected to have linear best-case relative speedup. The run time for such algorithms is generally determined by the cost for starting the algorithm, called *latency*, and the number of input values. Distributing or broadcasting initial values and collecting the results can impede the relative speedup considerably if such operations are not supported by the underlying network. Thus, if a linear or circular array structure is used by the algorithm, an additional binary tree structure of the nodes would clearly be favourable.

Wavefront algorithms

A wide range of parallel numerical algorithms can be described as wavefront algorithms. The execution of such algorithms can be visualized as a two-dimensional grid of subcomputations or cells. The computation in each cell is data driven and the computational activity starts in the upper left corner and proceeds down and right towards the lower right corner, creating a computational wavefront through the

grid (see Figure 1.2). The concept of computational wavefront was introduced by S. Y. Kung [38] and is described in several other sources [16, 34, 37, 39].

The similarity comparison of DNA sequences [26] is a good representative of this category. This problem occurrs in the field of genetics as part of the Human Genome project [22] and is one of the important real world problems of today.

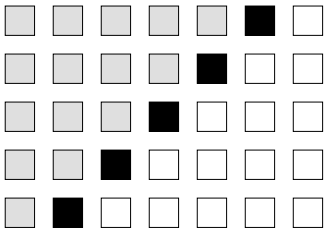


Figure 1.2: *The wavefront principle. The grey cells have already been computed and the black cells are currently active.*

The point we want to make is that even though such algorithms can often be best understood and visualized as a two-dimensional grid, the best implementation can often be achieved on a linear or circular array. The reason for that should be obvious: every cell is inactive during most of the computation, so implementation on a two-dimensional grid of processors (one cell per processor, see Figure 1.2) would result in poor efficiency. An implementation on a linear or circular array (a row of cells per processor, see Figure 1.2) would not impede the relative speedup. The sequence comparison problem is a good example, an excellent study has been presented by Carriero and Gelernter [15].

1.1.3 Affinity between different paradigms

Often a problem is not solved by using just one paradigm, and the resulting algorithm can be a mixture of different paradigms. For example, the sequence comparison problem: if one sequence is to be compared with several other sequences (a database) then one could either

1. do those comparisons in parallel,
2. parallelize each comparison, or
3. combine 1 and 2.

In the first case the master-slave paradigm is used, in the second case the systolic paradigm is used, and in the third case a mixture of both is used. There are several factors to be considered when choosing the best alternative [15].

1.2 Parallelization of sequential programs

Another important area of parallel computation deals with automatic parallelization of sequential programs. Clearly, in that case there are no obvious choices of parallel programming paradigms. Different approaches can be employed, e.g., using a master-slave technique for parallelizing sequential divide-and-conquer algorithms [21].

A vast amount of research has been done in the area of parallelizing repetition, usually in the form of sequential loops, see for example Wolfe [69] for some recent achievements and directions. Often the data dependencies (if any) among the parallel subcomputations have a local nature, e.g., iteration i requires a value produced by

iteration $i-1$ (assuming a vectorization of the loop). The resulting computation often has a systolic nature [44] or is 'embarrassingly parallel', in which case there are no data dependencies among the subcomputations and would in many cases benefit from a linear or circular array structure and a binary tree structure of the network, as discussed above.

In particular, we are interested in parallelization of logic programs. Extracting parallelism in repetition has recently received serious attention in the logic programming community. The two basic approaces that are currently being studied are Reform, and bounded quantifications.

1.2.1 Reform

The fundamental programming structure for repetition in Horn clause based logic programming is recursion. The Reform inference principle [65] offers a method for uncovering parallelism in recursive logic programs. A novel compilation method [47, 48], based on Reform, can be used to compile recursion over inductively defined data structures to bounded iteration over vectors. Well-known techniques can then be used to run the iterations in parallel.

The data dependencies (if any) that arise most frequently between successive iterations in a Reform parallel execution of linear recursive logic programs are local. A cycletree has been recognized by the Reform group [10] as an ideal interconnection network for a nonshared memory implementation of a Reform engine.

1.2.2 Bounded quantifications

An extension to Horn clause based logic programming, that can be used to express repetition, is quantification. Bounded quantified formulas over finite domains can be realized as loops on sequential machines or as parallel threads on parallel machines. In particular, bounded quantifications map well onto machines supporting the data parallel model of computation.

Experience of programming with bounded quantifications indicates that all-to-one, one-to-all, and local communication patterns are the most important ones in such computations and would thereby benefit from a cycletree structure in a nonshared memory implementation [5, 6].

1.3 Properties of interconnection graphs

The theoretically ideal network of N nodes has an interconnection graph that is a complete graph of N vertices. Clearly, such a network is prohibitively expensive to realize for large N (see Figure 1.3). The total number of edges must be reduced while maintaining a structure that supports simple and efficient routing for the particular area of applications for which the network is designed.

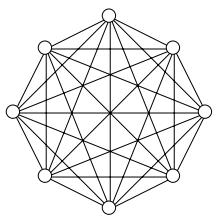


Figure 1.3: *The ideal network.*

Some parameters that are commonly used to characterize networks are: number of edges, degree of the network (maximum degree of the nodes), diameter (the largest distance between any two nodes), average distance, symmetry, edge- and node connectivities, extensibility, and reliability [1, 2, 7, 53, 54, 68].

In this thesis the main issue is to combine a circular array and a binary tree into a single graph, in order to provide efficient communication for the above communication patterns. In doing so,

- the lowest possible degree,
- the smallest possible number of edges,[1] and
- extensibility

are the parameters that we consider to be most important. We want to emphasize the importance of extensibility: a cycletree can be constructed inductively to include any basic binary tree. As an extreme example, Figure 1.4 shows a cycletree which matches the cyclic-order odd-even transposition bilinear sorter [40].

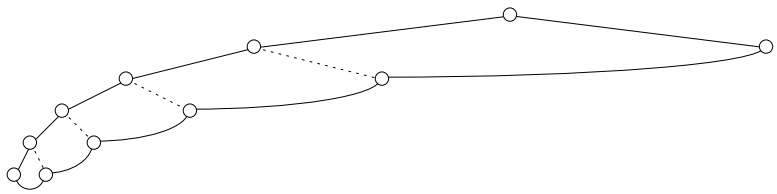


Figure 1.4: *An extreme example of a cycletree.*

Several interconnection graphs have been presented that, in one way or another, include a full binary tree and a circular array of the vertices, e.g., X-trees [19] and completely linked trees [30], both of which are supergraphs of a full cycletree, and thus are more complex and expensive. Recently, a network called a ringtree [70], which is a special case of a cycletree (a full cycletree), has been put forward. The results presented [70] are special cases of the more general results presented in Chapter 2 of this thesis.

1.3.1 Embedding

An *embedding* of a guest graph G in a host graph H is a one-to-one association of the vertices of G with the vertices of H, together with a specification of paths in H connecting the images of the endpoints of each edge of G. The *dilation* of the embedding is the maximum length of such paths. The *expansion* of the embedding is the ratio $|V_H|/|V_G|$, where V_H (V_G) is the number of vertices in H (G) [55].

We do not address embedding of cycletrees in other networks, or VLSI, in this thesis. It should be noted, however, that the low degree and the minimal number of edges of certain cycletrees makes it possible to embed cycletrees in other networks, so that the resulting embedding has a minimal expansion and dilation with respect to binary tree and circular array embeddability.

Embedding of binary trees and full binary tree based networks, like X-trees, sneptrees, de Bruijn networks and completely linked trees, in other networks like hypercubes, meshes, rings and butterflies, and VLSI arrays has been studied extensively [25, 30, 31, 41, 45, 49, 55, 56, 58, 67, 71, 72]. Several of those techniques and results apply directly to cycletrees.

[1] When the cycletree under consideration fulfills certain criteria; see chapter 2.

1.3.2 Mapping

When G has more vertices than H embedding is, by definition, not possible, since several vertices of G would have to be associated with one vertex of H. In such cases G has to be *mapped* in H. In the literature mapping is divided into *grouping* (also called *contraction*), *placement* and *routing* (also called *wiring* or *path mapping*), see for example Shen [60] for definitions (where the first alternatives are used). Grouping and placement jointly, are usually referred to as *scheduling*.

One can define *mapping* of G in H more formally as follows. Given a decomposition of G into disjoint subgraphs, such that the number of those subgraphs does not exceed the number of vertices in H (grouping above), let G' be a graph which has a vertex for each such subgraph and an edge between two vertices if there is an edge between the corresponding subgraphs of G. A mapping of G in H is then an embedding (placement and routing above) of G' in H.

The problem of finding the optimal mapping is known as the *mapping problem*. The mapping problem in its most general form is computationally equivalent to the graph isomorphism problem, see Bokhari [13] for a formal proof. Polynomial time algorithms for solving the graph isomorphism problem are not known [18]. A *topological mapping* of G in H (an embedding of G in H with dilation 1), if such exists, is an example of an optimal mapping. The topological mapping has been investigated for some regular guest graphs and particular host graphs, e.g., CHiP lattices [62] and hypercubes [11, 14, 57].

The approaches for obtaining suboptimal solutions, concerning either static mapping [12, 20, 60, 61], or dynamic (adaptive) mapping [35], vary. The most important properties of the guest graph, determining the outcome of any solution, are the number of edges and the degree (and regularity when topological mapping is the issue) of the guest graph.

We believe that the nice inductive structure of cycletrees makes them an attractive alternative to simple binary trees in *dynamic* embedding and mapping strategies, yet to be investigated for cycletrees. Dynamic embedding of binary trees (in hypercubes and butterflies) have been investigated, e.g., by Bhatt, Chung, Leighton and Rosenberg [11]. Clearly, such strategies are important in the context of parallel divide-and-conquer techniques [51]. The additional edges in cycletrees would considerably widen the range of applicable parallel programming paradigms when combined with dynamic embedding or mapping techniques.

1.4 Related networks

In this section we give a brief survey of the related network topologies that have been put forward. For an overview of other static- and dynamic connection topologies see, e.g., Almasi and Gottlieb [2].

A linear array by itself is a powerful interconnection network for many problems [9] and is used for example in the Warp computer [3]. Another circular array based interconnection graph is the chordal ring [4].

A binary tree is, for example, a common feature of the DADO architectures [63], and is used, among others, in the Bentley and Kung machine [8]. The Leiserson machine [27, 42] uses the completely linked binary tree structure (also called a semi X-tree [27]). In the Bentley and Kung machine and in the Leiserson machine, the internal nodes are used for routing only.

An interesting class of interconnection graphs is the X-trees [19]. The X-tree structure is used for example in the Ottman, Rosenberg and Stockmeyer machine [52]. Hypertrees [23] are similar to X-trees. Some types of threaded trees are classified as X-trees and have been studied by Despain and Patterson [19]. The threaded X-tree

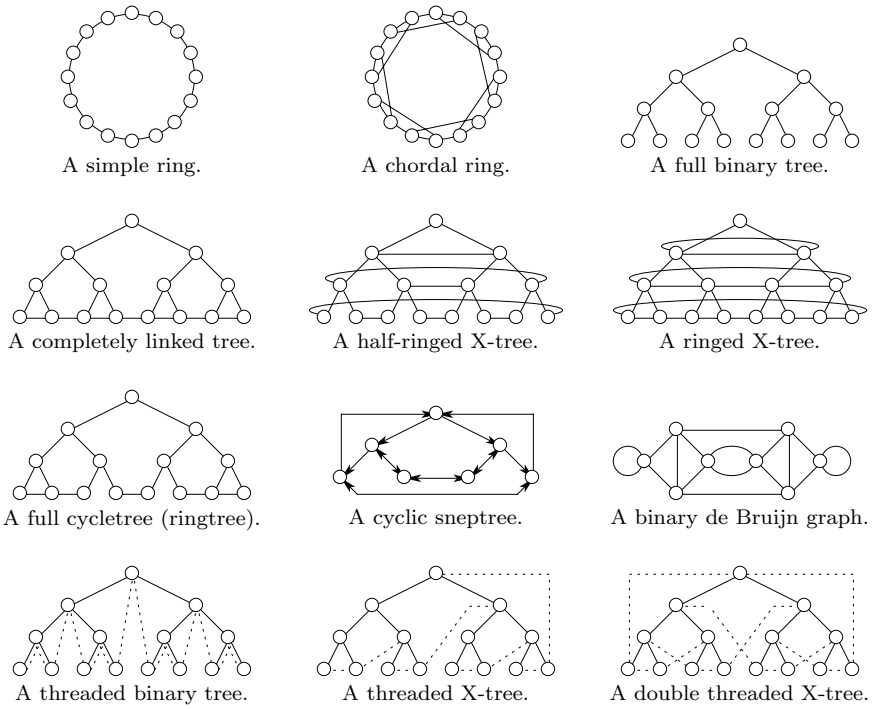


Figure 1.5: *Some examples of related networks.*

in Figure 1.5 is actually a preorder threaded binary tree (assuming left and right as shown in the figure) with an extra thread from the rightmost leaf to the root and belongs to the class of cycletrees. The threaded binary tree in the lower left corner of Figure 1.5 is, what is commonly understood by a threaded binary tree [33], i.e., it is threaded in inorder (or symmetric order).

Other binary tree based interconnection networks are sneptrees [45], de Bruijn graphs [33, 58], and ringtrees [70]. The de Bruijn network of degree 4 is one type of sneptree. Ringtrees are identical with full cycletrees.

Another class of binary tree (and butterfly) related networks are the fat-trees [43]. The interconnection graph of the latest model, CM-5, of the Connection Machine is based on a fat-tree. The earlier models are hypercube based [64].

1.5 Outline of the thesis

In Chapter 2 we define and analyse cycletrees from a graph theoretical point of view and prove some basic properties of cycletrees. In Chapter 3 we treat cycletrees as networks of communicating processes. We show how cycletrees can be constructed recursively, and we present a shortest path routing algorithm for certain cycletrees, namely those which contain no edges between vertices at levels l and $l+2$, or *cross-level* edges. We summarize our results in Chapter 4.

Chapter 2

Cycletrees

In this chapter we define formally a class of interconnection graphs called *cycletrees*. We show how certain cycletrees, which we call *natural cycletrees* can be defined inductively. A *minimal* natural cycletree is shown to have the minimum possible number of nontree edges. We also give a formula for the exact number of edges in an *optimal* natural cycletree.

2.1 Preliminaries

We assume that the reader is familiar with elementary concepts in graph theory [24]. Cycletrees are treated as *undirected simple* graphs. Let $G = \langle V, E \rangle$ be a graph. Throughout the thesis we will assume that G is simple and undirected. We will sometimes denote the edge-set of G by E_G and the vertex-set by V_G. By writing $E_{G_1 \oplus G_2}$ we mean $E_{G_1} \oplus E_{G_2}$ where '$\oplus$' is some set operation. An edge in E_G is written as a pair (a, b) where a and b are two vertices in V_G. (Note that $(a, b) = (b, a)$, since G is undirected.) We use the notation $G[X_1, X_2, \ldots, X_n]$, to indicate that each X_i, $i \in \{1, \ldots, n\}$, is a certain partial subgraph of G; if X_i is an isolated vertex then we simply write X_i for that vertex.

Following Almasi and Gottlieb [2], we use the term *binary tree* for a tree T which consists of a single vertex, called its *root*, plus 0 or 2 disjoint binary trees, called its immediate subtrees. The root of T has degree 0 or 2, the *internal* vertices (other than the root) have degree 3 and the *external* vertices or *leaves* have degree 1. If a binary tree is not an isolated vertex, i.e., when the root has degree 2, then we call it a *basic* binary tree. We identify the root r of a binary tree T by $T[r]$. Note that a basic binary tree is not the same as an *extended binary tree* [33], although the concepts are related. (See Knuth [33, pp. 309 and 315], where binary trees as defined above are called b-trees.)

2.2 Basic definitions and theorems

A *Hamiltonian circuit* or *path* in a graph G is a circuit or path which visits all the vertices of G exactly once. We call the partial subgraph of G traversed by a Hamiltonian circuit or path of G simply a *cycle* or *chain* in G, respectively. We identify the terminals r and s of a chain C by $C[r, s]$. We have a straightforward classification of those graphs we call cycletrees.

Definition 1 (*Cycletrees*) Let G be a graph. Then $G[C, T]$, $V_C = V_T = V_G$, $E_G = E_{C \cup T}$, is a *cycletree* if T is a basic binary tree and C is a unique cycle in G. ∎

Let $G[C,T]$ be a cycletree. We say that an edge α, $\alpha \in E_G$, is a *tree edge* (with respect to T), if $\alpha \in E_T$; a *cycle edge*, if $\alpha \in E_C$; a *nontree edge*, if $\alpha \in E_{C-T}$; a *noncycle edge*, otherwise.

We will next show how to construct certain cycletrees inductively. We do so by giving an inductive definition of *natural cycletrees* and show then that natural cycletrees are indeed cycletrees. First, let us illustrate the form of a natural cycletree (see Figure 2.1). The edges represented by the solid lines form a unique cycle. The edges represented by straight lines form a basic binary tree. The dashed lines represent the edges that are not part of the cycle. The curved lines represent the edges that are not part of the basic binary tree.

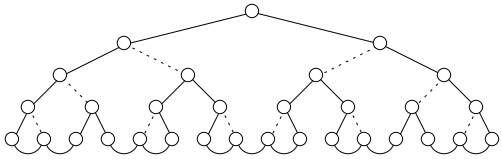


Figure 2.1: *A full natural cycletree of 31 vertices.*

It is easy to see that the "left" and "right" halves of the natural cycletree in Figure 2.1 are symmetric. We can also recognize a regular pattern in the structure of one half. This structure is not only present in full natural cycletrees. We begin with an inductive definition of a *chaintree* $H[r,s]$ where r and s are two vertices of H. We will later show that there exists a chain $C[r,s]$ in H and that there exists no other chain $C'[r,s]$ in H, such that $C \neq C'$. A natural cycletree is then constructed from two chaintrees.

Definition 2 (*Chaintrees*)

1. Let r be a vertex. Then $\langle\{r\},\ \emptyset\rangle[r,r]$ is a chaintree.
2. Let $H_i[r_i,s_i]$, $i \in \{1,2,3\}$, be disjoint chaintrees, and r and s two distinct vertices not in $V_{H_1\cup H_2\cup H_3}$. Then
   (a) $\langle V_{H_1} \cup \{r,s\},\ E_{H_1} \cup \{(r,r_1),(r,s),(s_1,s)\}\rangle[r,s]$, and
   (b) $\langle V_{H_1\cup H_2\cup H_3} \cup \{r,s\},$
   $E_{H_1\cup H_2\cup H_3} \cup \{(r,r_1),(r,s),(s,r_2),(s,r_3),(s_1,s_2)\}\rangle[r,s_3]$

   are chaintrees (see Figure 2.2).

The only chaintrees are those given by clauses 1 and 2. ∎

We can now construct a natural cycletree from two chaintrees as follows.

Definition 3 (*Natural cycletrees*) Let $H_1[r_1,s_1]$ and $H_2[r_2,s_2]$ be disjoint chaintrees, and r a vertex not in $V_{H_1\cup H_2}$. Then

$$\langle V_{H_1\cup H_2} \cup \{r\},\ E_{H_1\cup H_2} \cup \{(r,r_1),(r,r_2),(s_1,s_2)\}\rangle$$

is a natural cycletree. No graph is a natural cycletree unless it can be constructed as above. ∎

Definition 3 is illustrated by Figure 2.3. An example of a possible natural cycletree is shown by Figure 2.4. We have the following theorem.

Theorem 1 *The maximum degree of natural cycletrees is 3.*

Proof. Immediate from Definition 2 and Definition 3. ∎

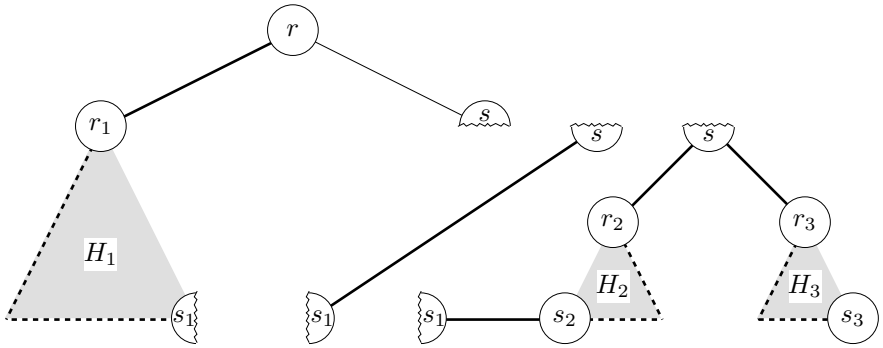


Figure 2.2: *Inductive construction of chaintrees, cases 2(*a*) and 2(*b*) of Definition 2. The bold lines illustrate a chain.*

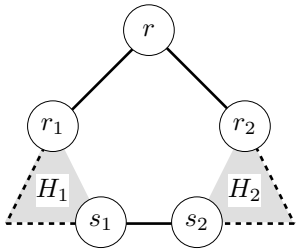


Figure 2.3: *Construction of natural cycletrees. The bold lines illustrate a cycle.*

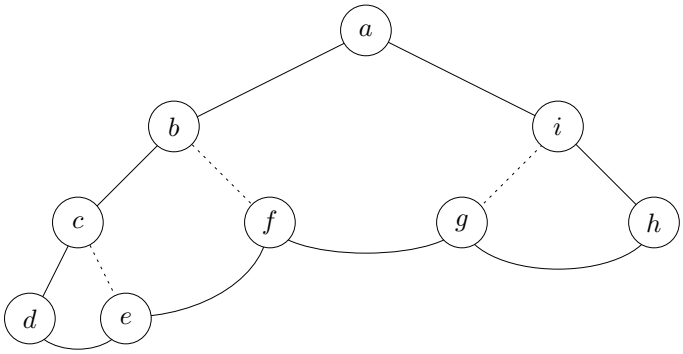


Figure 2.4: *A natural cycletree.*

The observant reader has probably noticed that we cannot always uniquely identify the basic binary tree in a natural cycletree, the smallest example to illustrate this is when the natural cycletree is a triangle; another example is the natural cycletree in Figure 2.6. This is, however, only a minor detail and we deal with it in Section 2.4.

We will now prove that all natural cycletrees are cycletrees. We will use the following lemma.

Lemma 1 *Let $H[r, v]$ be a chaintree. Then H contains a chain $C[r, v]$ and there exists no other chain $C'[r, v]$ in H such that $C' \neq C$.*

Proof. We prove the lemma by induction over chaintrees.

Base case: When H is an isolated vertex, $r = v$, then trivially C, $C = H$, exists and is unique.

Induction case: Let $H_i[r_i, s_i]$, $i \in \{1, 2, 3\}$, be chaintrees as in Definition 2.2. Assume that $C_i[r_i, s_i]$ is a chain in each H_i and that lemma holds for every H_i. We prove the lemma for cases 2(a) and 2(b) of Definition 2.

Case 2(a): Clearly, $C[r, s]$, where $s = v$, must use the edges (r, r_1) and (s_1, s) (see Figure 2.2), and thus the chain through H_1 must have r_1 and s_1 as terminals. According to the induction hypothesis, C_1 is such a unique chain. Consequently,

$$C = \langle V_H,\ \{(r, r_1), (s_1, s)\} \cup E_{C_1} \rangle$$

is a chain in H and there exists no other chain $C'[r, s]$, $C' \neq C$, in H.

Case 2.(b): Clearly, $C[r, s_3]$, where $s_3 = v$, must use the edges (r, r_1), (s_1, s_2), (r_2, s) and (s, r_3) (see Figure 2.2), and thus the chains through H_1, H_2 and H_3 must have r_i and s_i as terminals. According to the induction hypothesis, C_1, C_2 and C_3 are the corresponding unique chains. Consequently,

$$C = \langle V_H,\ \{(r, r_1), (s_1, s_2), (r_2, s), (s, r_3)\} \cup E_{C_1 \cup C_2 \cup C_3} \rangle$$

is a chain in H and there exists no other chain $C'[r, s_3]$, $C' \neq C$, in H.

According to the induction principle we have proved the lemma for all chaintrees. ∎

We can now easily prove the following theorem by using Lemma 1.

Theorem 2 *A natural cycletree G is a cycletree $G[C, T]$.*

Proof. Let G, r, $H_1[r_1, s_1]$ and $H_2[r_2, s_2]$ be as in Definition 3. As r has degree 2, both of the edges (r, r_1) and (r, r_2) must be part of any cycle of G. Clearly the edge (s_1, s_2) must also be used. Thus, the chains through H_1 and H_2 must be $C_1[r_1, s_1]$ and $C_2[r_2, s_2]$, repectively. By using Lemma 1, C_1 and C_2 exist and are unique. Consequently

$$C = \langle V_G,\ \{(r, r_1), (s_1, s_2), (r_2, r)\} \cup E_{C_1 \cup C_2} \rangle$$

is the unique cycle of G. Hence G is a cycletree. ∎

Note that the choice of vertex r in Theorem 2 does not affect the proof, since for any vertex of degree 2 both its edges must be used in any cycle. Thus, by removing any such vertex (and the corresponding edges) we must obtain a chain between the two remaining terminals of the removed edges in the rest of the graph. If such a chain is unique, then so is the cycle. We get the following corollary from Theorem 2.

Corollary 2.1 *Let T be a basic binary tree, such that $|V_T| > 3$, and v an arbitrary internal vertex of degree 3 in T. Let v_1 and v_2 be the sons of v and w the father of v. Then there exists a natural cycletree that uses the edges (w, v) and (v, v_1) as cycle edges and a natural cycletree that uses the edges (w, v) and (v, v_2) as cycle edges.*

Proof. Immediate by using Theorem 2, Definition 2 and Definition 3. Let $r = v$ in Definition 2. Clearly, either $v_1 = r_1$ or $v_2 = r_1$, since the binary trees with roots r_1 and s, formed in Definition 2, can be arbitrary (see Figure 2.2). ∎

All cycletrees are not natural cycletrees. Let us, just once, illustrate a cycletree which is not a natural cycletree (see Figure 2.5). Another example of a non-natural cycletree is the threaded X-tree in Figure 1.5.

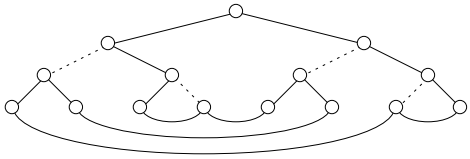


Figure 2.5: *A cycletree which is not a natural cycletree.*

Our main concern throughout the rest of the thesis is the treatment of natural cycletrees. We will from now on, unless otherwise stated, by cycletrees mean natural cycletrees.

2.3 Minimality and optimality

We will in the following sections address the issue of the number of edges of a cycletree. We will prove that a cycletree can be used to combine any basic binary tree and a cycle in such a way that the resulting number of edges is the smallest possible. In that case the cycletree is called *minimal*.

We also address the issue of the number of edges in a cycletree $G[C, T]$, which is constructed with a given set of vertices so that T has the minimal total path length and G has the minimum number of edges. In that case G is called *optimal*. We also give a formula for the exact number of edges of an optimal cycletree.

2.3.1 Minimal cycletrees

Assume that we have a basic binary tree T, and that we want to obtain a cycle C, such that $V_C = V_T$, by adding as few edges as possible to T. Now the question is: can we construct a cycletree that fulfills this property? This is indeed the case, which will be proved shortly. Note that the uniqueness of the cycle in a cycletree does not imply this property in general, as is illustrated with the next example.

Example 2.3.1 Consider the cycletree G in Figure 2.6. The noncycle edges are (b, c) and (i, g). Clearly it has the same set of tree edges with respect to the binary tree $T[a]$ as the cycletree in Figure 2.4, but the total number of edges in Figure 2.6 is less than in Figure 2.4. Note also that G has a different set of (non)tree edges with respect to the binary tree $T'[e]$. The (non)cycle edges form, however, always a unique subset of E_G. ∎

Definition 4 (*Minimality*) We say that a cycletree $G[C, T]$ is *minimal* for T, if for any other cycletree $G'[C', T]$, $|E_G| \leq |E_{G'}|$. ∎

For example, the cycletree in Figure 2.6 is minimal. Consider a given basic binary tree T. We will prove that no graph that extends T to also contain a Hamiltonian cycle has fewer edges than a minimal cycletree.

To begin with, let us consider *any* cycle C', such that $V_{C'} = V_T$. Let $S[r]$ be a subtree of T and let v be the father of r, if $S \neq T$; any vertex not in V_T, otherwise.

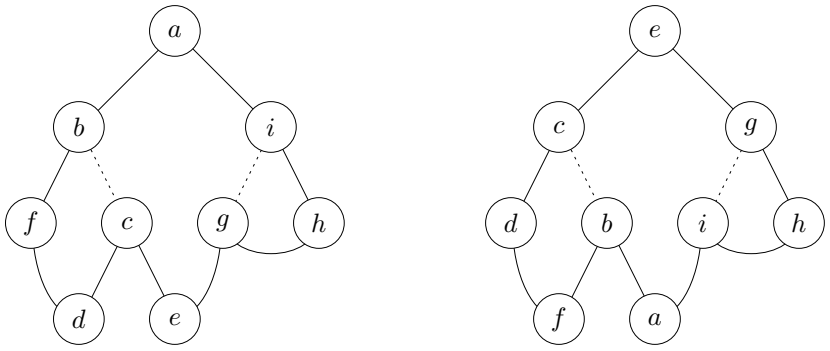


Figure 2.6: *A minimal (and optimal) cycletree that contains more than one basic binary tree.*

Clearly, either (r, v) is, or is not, in $E_{C'}$. We will denote by $\mathrm{nc}(S)$ and $\overline{\mathrm{nc}}(S)$, respectively, the minimum possible number of edges in S that cannot participate in C' in each case. Thus $\overline{\mathrm{nc}}(T)$ is the theoretical lower bound for $|E_{T-C'}|$. We will use the following lemma.

Lemma 2 *Let $G[C, T]$ be a minimal cycletree for T. Then $\overline{\mathrm{nc}}(T) = |E_{T-C}|$.*

Proof. Let $S[r]$ be a subtree of T. Let v be the father of r, if $S \neq T$; any vertex not in V_T, otherwise. Then

$$|E_{S-C}| = \begin{cases} \mathrm{nc}(S), & \text{if } (r, v) \in E_C; \\ \overline{\mathrm{nc}}(S), & \text{otherwise.} \end{cases} \tag{2.1}$$

We prove the lemma by proving Property 2.1 and Property 2.2

$$\mathrm{nc}(S) \leq \overline{\mathrm{nc}}(S) + 1 \leq \mathrm{nc}(S) + 1 \tag{2.2}$$

by induction over binary trees (subtrees of T).

Base case. If r is a leaf of T then $|E_{S-C}| = \mathrm{nc}(S) = \overline{\mathrm{nc}}(S) = 0$, and $0 \leq 1 \leq 1$.

Induction case. Let $S_1[r_1]$ and $S_2[r_2]$ be the immediate subtrees of S and assume that Property 2.1 and Property 2.2 hold for S_1 and S_2.

We know from Definition 2, Definition 3, and Theorem 2 that exactly one of (r, r_1) and (r, r_2), say (r, r_1), is in C, if (r, v) is in C; both (r, r_1) and (r, r_2) are in C, otherwise. By using the induction hypothesis (Property 2.1 holds for S_1 and S_2), we get that

$$|E_{S-C}| = \begin{cases} \mathrm{nc}(S_1) + \overline{\mathrm{nc}}(S_2) + 1, & \text{if } (r, v) \in E_C; \\ \mathrm{nc}(S_1) + \mathrm{nc}(S_2), & \text{otherwise.} \end{cases} \tag{2.3}$$

Furthermore,

$$\mathrm{nc}(S_1) + \overline{\mathrm{nc}}(S_2) \leq \overline{\mathrm{nc}}(S_1) + \mathrm{nc}(S_2) \tag{2.4}$$

must hold, or else, by using Corollary 2.1, there would exist another cycletree that uses a larger subset of E_S as cycle edges, including the edge (r, r_2), and G would not be minimal for T.

Now, let C' be *any* cycle such that $V_{C'} = V_T$. Clearly, if (v, r) is in C' then at most one of the edges (r, r_1) and (r, r_2) can be in C'. Accordingly, if (v, r) is not in C' then both (r, r_1) and (r, r_2) can be in C'. According to the induction hypothesis, $\mathrm{nc}(S_i) \leq \overline{\mathrm{nc}}(S_i) + 1$, $i \in \{1, 2\}$. Thus, by using Property 2.4,

$$\mathrm{nc}(S) = \mathrm{nc}(S_1) + \overline{\mathrm{nc}}(S_2) + 1, \tag{2.5}$$

$$\overline{\mathrm{nc}}(S) = \mathrm{nc}(S_1) + \mathrm{nc}(S_2). \tag{2.6}$$

Now, it follows trivially from Property 2.3, Property 2.5 and Property 2.6 that Property 2.1 holds for S. Furthermore, by using the induction hypothesis, $\mathrm{nc}(S_1) \leq \overline{\mathrm{nc}}(S_1) + 1$, and Property 2.5 and Property 2.6, we get that $\overline{\mathrm{nc}}(S) \leq \mathrm{nc}(S)$. By using the induction hypothesis, $\overline{\mathrm{nc}}(S_2) \leq \mathrm{nc}(S_2)$, and Property 2.5 and Property 2.6, we get that $\mathrm{nc}(S) \leq \overline{\mathrm{nc}}(S) + 1$. Hence Property 2.2 holds for S.

According to the induction principle, we have proved Property 2.1 and Property 2.2 for all subtrees of T and in particular for T itself. Now, it is an immediate consequence of Property 2.1 that $\overline{\mathrm{nc}}(T) = E_{T-C}$. ∎

We can now easily prove the following theorem, by using Lemma 2.

Theorem 3 *Let G' be any graph such that a basic binary tree T and a cycle C', $V_{C'} = V_T$, are partial subgraphs of G'. Let $G[C,T]$ be a minimal cycletree for T. Then $|E_G| \leq |E_{G'}|$.*

Proof. We know, by definition, that $\overline{\mathrm{nc}}(T) \leq |E_{T-C'}|$. By using Lemma 2, $|E_{T-C}| = \overline{\mathrm{nc}}(T)$, we get that $|E_{T-C}| \leq |E_{T-C'}|$. We know that $|E_G| = |E_{T-C}| + |E_C|$, since there are no superfluous edges in G, and trivially that $|E_{G'}| \geq |E_{T-C'}| + |E_{C'}|$. Hence $|E_G| \leq |E_{G'}|$, since $|E_C| = |E_{C'}|$. ∎

2.3.2 Optimal cycletrees

Let us assume that one wants to construct a cycletree $G[C,T]$ given only a set of vertices. Clearly, constructing T to have the minimal *total path length* has several advantages. For example if T is used as the interconnection graph of a process network then the total communication path length of T is minimized. At this point it is worth noting that we are not giving top priority to minimizing the average distance between an arbitrary pair of vertices of G. The reason why we introduced cycletrees in the first place was to provide efficient communication for parallel computations that *generally* need T for "global communication" and C for "local communication".

The *level* of a vertex v of $T[r]$ is the path length[1] of the shortest path from r to v. The *depth* of T is the maximum level of T. We say that a binary tree T of depth D is *tree-complete* if all the leaves of T are at levels D and $D-1$. We say that a cycletree $G[C,T]$ is tree-complete (with respect to T) if T is tree-complete.

We know that the total path length of a basic binary tree T is minimal if T is tree-complete. (The relation to complete extended binary trees is obvious, see Knuth [33].) When constructing a cycletree $G[C,T]$, we also want to keep the number of additional edges at minimum. This suggests the following definition.

Definition 5 (*Optimality*) Let $G[C,T]$ be a tree-complete cycletree. We say that G is *optimal* if $|E_G| \leq |E_{G'}|$ for any other tree-complete cycletree $G'[C',T']$, $V_{G'} = V_G$. ∎

Clearly, if a cycletree $G[C,T]$ is optimal then it is both minimal for T and tree-complete. (The converse is in general not true, i.e., a cycletree that is both tree-complete and minimal need not be optimal.)

Theorem 4 *Let G be an optimal cycletree, $N = |V_G|$ and $K = \lfloor \log_2(N+1) \rfloor$. Then*

$$|E_G| = \begin{cases} (3N-1)/2 - \lfloor (2^K+1)/3 \rfloor, & \text{if } N > 4\lfloor (2^K+1)/3 \rfloor - 1; \\ N - 1 + \lfloor (2^K+1)/3 \rfloor, & \text{otherwise.} \end{cases} \tag{2.7}$$

Proof. See Appendix A.1. ∎

[1] Number of edges used by the path.

Let $G[C,T]$, $N = |V_G|$, be a cycletree where T is a full binary tree, i.e., $N+1$ is a power of 2. Then Formula 2.7 reduces to

$$N - 1 + \lfloor (N+2)/3 \rfloor, \tag{2.8}$$

which was also shown by Xie and Ge [70] to be the number of edges in a ringtree. One can easily verify that the cycletree in Figure 2.6 is optimal, by using Formula 2.7. We also get the following corollary from Theorem 4.

Corollary 4.1 *Let G be any graph that has a tree-complete basic binary tree T and a cycle C, $V_C = V_T$, as partial subgraphs. Let N and K be as above. Then G has at least as many edges as the number given by Formula 2.7.*

Proof. Immediate by using Theorem 3, since optimal cycletrees are by definition also minimal. ∎

As an example of the use of the above formulas, assume that one has a full cycletree of N vertices and wants to double the size of it to $2N+1$. Then how many additional edges are required? By using Formula 2.8, we obtain that $N+2\lfloor (N+3)/6 \rfloor$ additional edges are required. Thus, doubling the number of vertices roughly doubles the number of edges in a cycletree.

2.4 Ordered cycletrees

In this section we will introduce some new concepts that will be used extensively throughout the rest of the thesis. Let $G[C,T[r]]$ be a cycletree. We know that C is unique. That allows us to *enumerate* the vertices of G in a simple manner and by those means define an ordering on G.

Let $N = |V_G|$ and $P = (r, s_2, \ldots, s_N, r)$ be a Hamiltonian circuit of G. We say that r has *address* 1 and each s_i has address i, $2 \leq i \leq N$. (Clearly, there exist two such enumerations of G with respect to r; we may choose either of those.) We will from now on identify a vertex with its address, i.e., we will say vertex a instead of vertex with address a. Vertex 1, i.e, the root of T, will also be called the root of G. Let $S[a]$ be a subtree of T, such that a is not a leaf. Let $S_1[l]$ and $S_2[h]$ be the immediate subtrees of S, such that $l < h$. We call S_1 the *left* subtree of S and S_2 the *right* subtree of S.

Now, let $S[a]$ be a proper subtree of T and v the father of a. It is an immediate consequence of Definition 2 and Definition 3 that S was traversed in one of the following ways.

1. If vertex a is the immediate successor of vertex v on P, i.e., $a - 1 = v$, then S was traversed by visiting a first.

2. If vertex a is the immediate predecessor of vertex v on P, i.e., $(a \bmod N)+1 = v$, then S was traversed by visiting a last.

3. Otherwise S was traversed by traversing the left subtree of S (if any), visiting a, and traversing the right subtree of S (if any).

We call S a *pre-tree*, a *post-tree*, or an *in-tree*, in each case, respectively. Vertex a is called accordingly, a *pre-vertex*, a *post-vertex* or an *in-vertex*. In general, we call *pre*, *post* and *in* the *mark* of the corresponding subtree or vertex. The above cases are illustrated in Figure 2.7. Throughout the rest of the thesis, '−', '0' and '+' will be used in figures to represent the marks *pre*, *in* and *post*, respectively.

Let $S[a]$ be a subtree of T, such that a is not a leaf of T. Let $S_1[l]$ and $S_2[h]$ be the left and right subtrees of S, respectively. We have the following relations between left and right, and the marks.

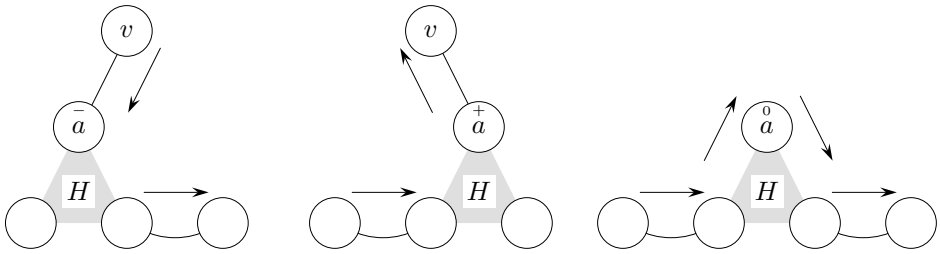


Figure 2.7: *Traversal of S; H is the subgraph of the cycletree that is induced by V_S.*

- If a is the root of G then l is a pre-vertex and h a post-vertex.
- If a is a pre-vertex then l is a pre-vertex and h an in-vertex.
- If a is an in-vertex then l is a post-vertex and h a pre-vertex.
- If a is a post-vertex then l is an in-vertex and h a post-vertex.

We say that T is ordered in *cycle order*. We illustrate the new concepts in Figure 2.8. (If we had enumerated the cycle in Figure 2.8 in the other direction we would have

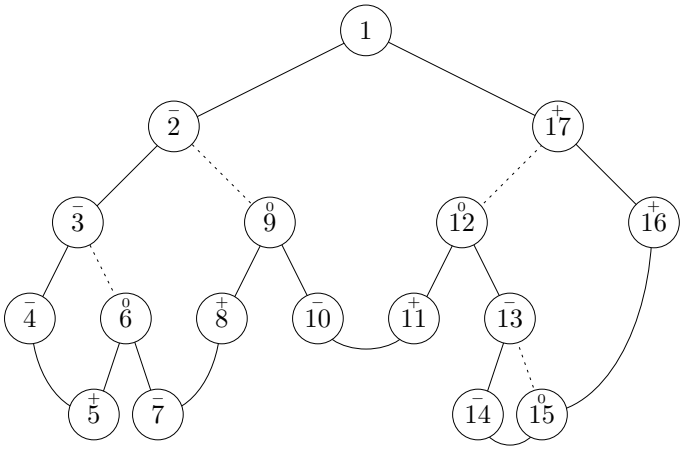


Figure 2.8: *An ordered cycletree.*

got a mirror image of Figure 2.8, with pre-trees and post-trees switched.)

2.4.1 Related concepts

The choice of names containing 'pre', 'post' and 'in' has a historical background in ordered binary trees. In fact, there is a connection between ordered cycletrees and *threaded* binary trees [33]. One can give a mutually recursive definition of *cycle order traversal* of an ordered binary tree, by using the above points, and obtain an ordered binary tree that is threaded in cycle order. By removing the orientation of the arcs in such a tree, a cycletree is obtained. Some types of threaded trees, to be used as networks, have been studied by Despain and Patterson [19] and are illustrated in Figure 1.5.

Chapter 3

Cycletrees as networks

In this chapter we will treat cycletrees as networks of communicating nodes. We will present an algorithm that recursively configures a cycletree from a given number of nodes. We next treat the problem of routing in cycletrees. Shortest path routing criteria are defined for each node. A "systolic" algorithm, that is superfast for tree-complete cycletrees, is presented for establishing the shortest path routing criteria dynamically.

3.1 Basic definitions

We have up to now studied some basic properties of cycletrees. Let us now turn to the more practical issue of using cycletrees as interconnection graphs in the MP-RAM model, see for example Almasi and Gottlieb [2] for a formal treatment of the various computational models.

We will think of our RAMs simply as *nodes*. Each node has a unique address between 1 and N, where N is the total number of nodes. When saying node a we will mean the node with address a.

The *interconnection graph* has a vertex corresponding to each node in the ensemble. An edge in the graph indicates that the nodes at either end can communicate via a bidirectional channel or *link*. Nodes so connected are called *neighbours*.[1]

3.2 Constructing cycletrees

Let us assume having only a collection of N nodes that we want to configure so that the corresponding interconnection graph is a cycletree. As there exists a vast number of possible cycletrees having N vertices, we assume that certain constraints are given and must be satisfied, e.g., tree-completeness. (Note that it is necessary that N is odd and $N \geq 3$, in order that it be possible to construct a basic binary tree of N vertices.)

Let $G[C, T]$ be the cycletree to be constructed. Let S be a subtree of T, which has mark m and n vertices. The constraints are assumed to be given in form of a definition for the relation 'split':

$$\text{split}(m, n, l, r) \Leftrightarrow \textit{constraints}$$

[1] We will be referring to the vertices and the edges of an interconnection graph as nodes and links, when we have that interpretation in mind.

where l is the number of vertices in the left subtree of S and r is the number of vertices in the right subtree of S.[2]

Example 3.2.1 If we want G to be tree-complete then we can define 'split' as follows.

$$\begin{aligned} \text{split}(m,n,l,r) \quad \Leftrightarrow \quad & l = \min(n - 2^{\lfloor \log_2(n+1) \rfloor - 1}, 2^{\lfloor \log_2(n+1) \rfloor} - 1) \wedge \\ & r = n - 1 - l \end{aligned}$$

We do not need the mark in this case. ∎

Before turning to the algorithm, let us show how, for a given subtree $S[a]$, the addresses of a's left and right sons can be calculated in a "top down" manner. Let $S_1[a_1]$ and $S_2[a_2]$ be the left and right subtrees of S, respectively. We know from Section 2.4 that the vertices of S_1 shall precede in cycle order those of S_2. We know also the following.

- If a is a pre-vertex then it is the first (in cycle order) vertex of S.
- If a is an in-vertex then it is between S_1 and S_2.
- If a is a post-vertex then it is the last vertex of S.

For example, if a is a pre-vertex then $a_1 = a + 1$ and a_2 can be calculated as $a_2 = a + 1 + L + L_2$ (see Figure 3.1).

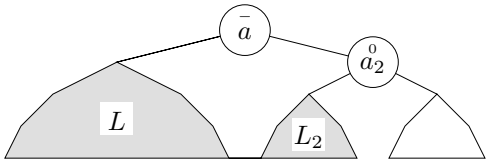


Figure 3.1: *Calculating the address of the right son of a pre-vertex.*

We can now present the algorithm. The algorithm constructs a cycletree recursively, using the above properties. As it is a trivial matter to construct a cycle, one is assumed to be given.

Algorithm 1 Let C be a cycle of N vertices; N is odd and $N \geq 3$. An enumeration of the vertices is assumed. The algorithm constructs an ordered cycletree $G[C,T] = \langle V_C, E_C \cup E \rangle$, by adding the required noncycle edges E to C, in such a way that 'split' holds for every subtree of T.

[**Top level.**] The root of G, vertex 1, has the pre-vertex 2 as its left son and the post-vertex N as its right son. Calculate values of L, R, L_1, R_1, L_2 and R_2 that satisfy

$$\text{split}(\mathit{root}, N, L, R) \wedge \text{split}(\mathit{pre}, L, L_1, R_1) \wedge \text{split}(\mathit{post}, R, L_2, R_2).$$

Now E is given by

$$E = \text{cfg}(\mathit{pre}, 2, L_1, R_1) \cup \text{cfg}(\mathit{post}, N, L_2, R_2),$$

where $\text{cfg}(m, a, L, R)$ is computed as the set of noncycle edges to form the subtree of T, which has root a, mark m, L vertices in its left subtree, and R vertices in its right subtree.

[2]Note that $\text{split}(m,n,l,r)$ can be nondeterministic in l and r. It could, in one extreme case, be defined so that l is any odd part of n up to $n-2$, which would correspond to "no constraints".

[**Base case.**] If $L = 0$ then a is an external vertex; return $\emptyset$.

[**Recursive case.**] If $L > 0$ then a is an internal vertex. We have three cases to consider, depending on the mark, m, of a. Let a_1 and a_2 be the left and right sons of a, respectively.

[**pre-vertex**] If a is a pre-vertex, $m = \mathit{pre}$, then $a_1 = a + 1$ is a pre-vertex and a_2 is an in-vertex. Calculate values of L_1, R_1, L_2 and R_2 that satisfy

$$\text{split}(\mathit{pre}, L, L_1, R_1) \ \wedge \text{split}(\mathit{in}, R, L_2, R_2).$$

Let $a_2 = a + L + L_2 + 1$ (see Figure 3.1); return

$$\{(a, a_2)\} \cup \text{cfg}(\mathit{pre}, a_1, L_1, R_1) \cup \text{cfg}(\mathit{in}, a_2, L_2, R_2).$$

[**post-vertex**] If a is a post-vertex, $m = \mathit{post}$, then $a_2 = a - 1$ is a post-vertex and a_1 is an in-vertex. Calculate values of L_1, R_1, L_2 and R_2 that satisfy

$$\text{split}(\mathit{in}, L, L_1, R_1) \ \wedge \text{split}(\mathit{post}, R, L_2, R_2).$$

Let $a_1 = a - R - R_1 - 1$; return

$$\{(a, a_1)\} \cup \text{cfg}(\mathit{in}, a_1, L_1, R_1) \cup \text{cfg}(\mathit{post}, a_2, L_2, R_2).$$

[**in-vertex**] If a is a in-vertex, $m = \mathit{in}$, then its left son $a_1 = a - 1$ is a post-vertex and its right son $a_2 = a + 1$ is a pre-vertex. Calculate values of L_1, R_1, L_2 and R_2 that satisfy

$$\text{split}(\mathit{post}, L, L_1, R_1) \ \wedge \text{split}(\mathit{pre}, R, L_2, R_2).$$

Return

$$\text{cfg}(\mathit{post}, a_1, L_1, R_1) \cup \text{cfg}(\mathit{pre}, a_2, L_2, R_2).$$

∎

Termination of Algorithm 1 is guaranteed by the fact that the third argument of 'cfg' is strictly less in each recursive call of 'cfg'. Thus, eventually, the base case must be reached. For example, by running Algorithm 1 with $N = 9$ and assuming a definition of 'split' as given in Example 3.2.1, the algorithm produces a cycletree as shown in Figure 3.2. The dashed lines correspond to the edge-set E in the algorithm.

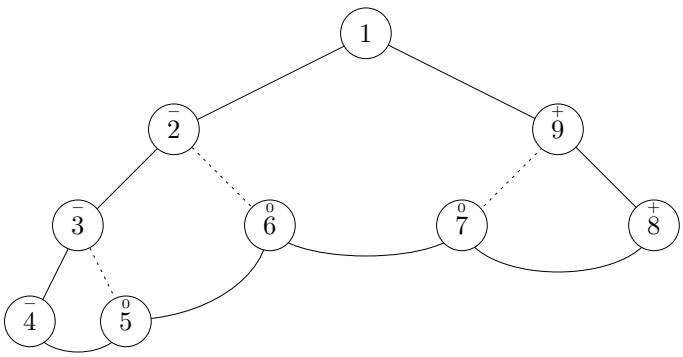


Figure 3.2: *A tree-complete cycletree of 9 vertices.*

3.3 Routing in cycletrees

In this section we show how to route data from one arbitrary node to another in a cycletree network. We shall give an algorithm that when presented with two nodes a and b produces a path from a to b along which data can be routed. We ignore issues of redundant paths that can be used, for example, to split the load between nodes that communicate heavily. We will also assume that all communication links in our idealized model have the same cost.

Evidently, a naive algorithm would be to use the cycle whenever a and b are two consecutive nodes, and to use the tree otherwise. That would result in a poor use of the cycletree. Consider, for example, a cycletree network that has the interconnection graph as in Figure 3.2; we would get a path of length 3 between the nodes 7 and 2, although the shortest path between these nodes has length 2.

Instead, we will treat a cycletree as a whole. Our aim is to obtain an algorithm that is both *simple* and *inexpensive* to use, and that for each pair of nodes a and b produces a *shortest* path from a to b.

In the following we will first present a routing algorithm. The algorithm will rely on certain static values associated with each node, which we call the *router data* of the node.

We then show how to define the router data so that the algorithm fulfills the shortest path requirement. We say that such router data are *optimal*. We will restrict the possible set of cycletrees to those which contain no edges between nodes at levels l and $l+2$, which we call *cross-level* edges; e.g., the cycletree in Figure 2.8 has a cross-level edge between nodes 15 and 16.

Finally, we present a parallel algorithm that dynamically establishes the optimal router data of each node of a cycletree having no cross-level edges. Assuming N nodes configured as a cycletree, the algorithm uses $\mathrm{O}(N)$ memory and $\mathrm{O}(K)$ time, where K is the depth of the cycletree. Thus, the algorithm is *superfast* for tree-complete cycletrees, since $K = \mathrm{O}(\log N)$ in that case.

3.3.1 Routing algorithm for cycletrees

Let G be a cycletree with N nodes and let a be some node in G. We will use $f(a)$, $l(a)$ and $r(a)$ to denote the following neighbours of a.

- $f(a)$ is the father of a, unless a is the root of G in which case it is undefined.
- $l(a)$ is the left son of a, if a is an internal node; node $a-1$, otherwise.
- $r(a)$ is the right son of a, if a is an internal node; node $(a \bmod N)+1$, otherwise.

Note that $l(a) = f(a)$ or $r(a) = f(a)$ if a is an external pre-node or an external post-node, respectively. When producing a path $(a, x, \ldots, b)$ from node a to node b, $a \neq b$, where x is a neighbour of a, the algorithm uses the given router data $\mathrm{lmin}(a)$, $\mathrm{lmax}(a)$, $\mathrm{rmin}(a)$ and $\mathrm{rmax}(a)$ to select x.

Algorithm 2 (*Router*). Let G be a cycletree and let (a, b), $a \neq b$, be an ordered pair of nodes in G. The algorithm produces a path from a to b.

[**Select neighbour.**] Select a neighbour x of a by using the router data of a.

$$x = \begin{cases} l(a), & \text{if } \mathrm{lmin}(a) \leq b \leq \mathrm{lmax}(a); \\ r(a), & \text{if } \mathrm{rmin}(a) \leq b \leq \mathrm{rmax}(a); \\ f(a), & \text{otherwise.} \end{cases}$$

[**Iterate.**] If $x = b$ then terminate else produce the path from x to b accordingly. ∎

As discussed previously, our aim was to obtain a routing algorithm that is both simple and inexpensive. Evidently, this is the case for Algorithm 2. Now we must find the router data of each node so that the algorithm terminates.

The necessary (since the neighbour x is chosen deterministically) and sufficient (since there are only a finite number of nodes) condition for termination of Algorithm 2 is that no node occurs twice on the produced path.

Let $G[C,T]$ be a cycletree and let a be a vertex in G. A naive solution, which was mentioned earlier, would be to let $\mathrm{lmin}(a)$ and $\mathrm{lmax}(a)$ be the minimum- and maximum vertices, respectively, in the subtree $S'[l(a)]$ of T. Accordingly, one could let $\mathrm{rmin}(a)$ and $\mathrm{rmax}(a)$ be the minimum and maximum vertices, respectively, in the subtree $S'[r(a)]$ of T.

The best solution is, ofcourse, if the router data are optimal, i.e., when a shortest path from vertex a to any other vertex b goes through: $l(a)$, if $\mathrm{lmin}(a) \leq b \leq \mathrm{lmax}(a)$; $r(a)$, if $\mathrm{lmin}(a) \leq b \leq \mathrm{lmax}(a)$; $f(a)$, otherwise.

It turns out that it is not possible to find the optimal router data of the nodes of an arbitrary cycletree. However, it is sufficient to disallow cross-level edges in order to be able to do so, e.g., all tree-complete cycletrees fall into this category, neither does the cycletree in Figure 1.1 have any cross-level edges.

3.3.2 Planar properties of cycletrees

We shall next investigate some further properties of cycletrees that will subsequently be used to determine the optimal router data of the nodes of a cycletree having no cross-level edges.

Definition 6 (*Admissibility*) Let $G[C,T]$ be any graph in the class of cycletrees (possibly a non-natural cycletree). We say that G is *admissible* if G is planar and there exists a plane[3] graph of G such that E_C is the contour[4] of the infinite region. We say that the corresponding plane graph is admissible. ∎

For example the non-natural cycletree in Figure 2.5 is not admissible. The non-natural cycletree (threaded X-tree) in Figure 1.5 is admissible however.

It is easy to prove that all (natural) cycletrees are planar graphs, which has also been illustrated with *plane* graphs of several cycletrees. As another example, Figure 3.3 is an admissible plane graph of a cycletree, R_ω in the figure is the infinite region; the finite regions are R_1, R_2, R_3, R_{12} and R_{13}.

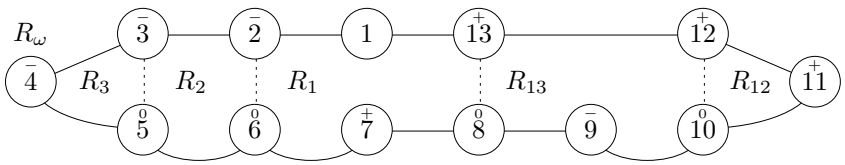


Figure 3.3: *An admissible plane graph of a cycletree.*

Theorem 5 *Let G be a (natural) cycletree. Then G is admissible.*

Proof. Follows easily from Definition 2, Definition 3, Lemma 1 and Theorem 2 (see Figure 2.2 and Figure 2.3). ∎

We get the following two useful theorems.

[3] A planar depiction of a planar graph. In general there exist several plane graphs for a planar graph.

[4] Set of edges surrounding a region.

Theorem 6 *The contours of two adjacent (finite[5]) regions of an admissible plane graph of a cycletree share exactly one edge, which is a noncycle edge.*

Proof. Let $G[C,T]$ be a cycletree. We know, due to Theorem 5, that G is admissible. Beginning with C (C has one region), each noncycle edge that is added must, due to the admissibility criterion, split a region into two regions, and thus becomes the only shared edge between those two regions. ∎

Theorem 7 *The contour of every region of an admissible plane graph of a cycletree includes exactly one nontree edge.*

Proof. Let $G[C,T]$ be a cycletree. The contour of a region of G must include at least one nontree edge (as T by itself has no regions). According to Theorem 6, none of those edges can be in the contour of another region of G. There are, by using Euler's formula[6], exactly r,

$$r = |E_G| - |V_G| + 1 = \overbrace{|E_T|}^{|V_G|-1} + |E_{C-T}| - |V_G| + 1 = |E_{C-T}|,$$

regions in G, i.e., as many regions as there are nontree edges. Consequently, every region must have exactly one nontree edge in its contour. ∎

As a consequence of Theorem 6, the set of contours of the regions of any plane depiction of a cycletree is always the same, regardless of the choice of depiction, as long as the admissibility criterion is satisfied.[7]

As a consequence of Theorem 7 and that the terminals of every nontree edge are the leaves of the basic binary tree, the contour D of a region of a cycletree $G[C,T]$ begins and ends with the root of some subtree S of T. Contour D includes the rightmost branch of the left subtree of S and the leftmost branch of the right subtree of S. We call those the *left side* and the *right side* of D, respectively. The root of S, which is neither on the left nor the right side of D, is called the *top* of D.

It is easy to see that each pre- and in-vertex belongs to the left side of some contour and each post- and in-vertex belongs to the right side of some contour. Furthermore, the top of a contour is always either a pre-vertex, a post-vertex or the root of the cycletree.

Example 3.3.1 Let D_1 be the contour of region R_1, of the cycletree in Figure 3.4 (or Figure 3.3). We have that

$$D_1 = \{(1,2),(2,6),(6,7),(7,8),(8,13),(13,1)\}.$$

The vertices 2 and 6 (none of which is a post-vertex) are on the left side of D_1, and the vertices 7, 8 and 13 (none of which is a pre-vertex) are on the right side of D_1. Vertex 1 is the top of D_1. To illustrate Theorem 6, e.g., R_1 and R_{13} are adjacent regions having only the noncycle edge $(8,13)$ in common. ∎

We define a *descendent* of vertex a to be a itself or a descendent of vertex b if a is at level l, b is at a level greater than l, and there is an edge between a and b. We call a an *ascendent* of b if b is a descendent of a.

We denote the set of descendents of a by desc(a). Note that a leaf b can be a descendent of another leaf a if there is a "descending" edge from a to b, e.g., vertex 7 of the cycletree in Figure 3.4 is a descendent of vertex 6, even though vertex 6 is not its father in the basic binary tree.

[5] Assumed from now on, unless otherwise stated.

[6] Euler's formula is more commonly written $r = |E| - |V| + 2$, where r takes into account also the infinite region.

[7] Note that Theorem 6 and Theorem 7 hold for all graphs in the class of cycletrees that are admissible (not only natural cycletrees). The theorems hold for example for the threaded X-tree in Figure 1.5.

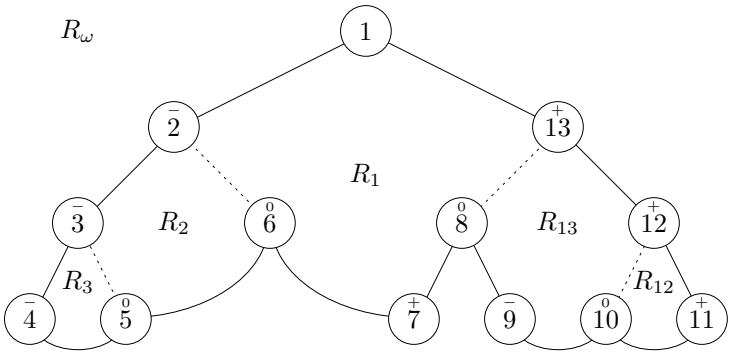


Figure 3.4: *Another admissible planar depiction of the cycletree in Figure 3.3.*

3.3.3 Optimal router data

Armed with the necessary definitions and theorems, we can now attack the problem of finding the optimal router data of a vertex a of a cycletree $G[C, T]$ having no cross-level edges.

Let us try to grasp the intuition first, considering only $\mathrm{lmin}(a)$ and $\mathrm{lmax}(a)$ (treatment of $\mathrm{rmin}(a)$ and $\mathrm{rmax}(a)$ is analogous). Assume also that a is internal and, to start with, on the right side of the contour D of a region R (see Figure 3.5).

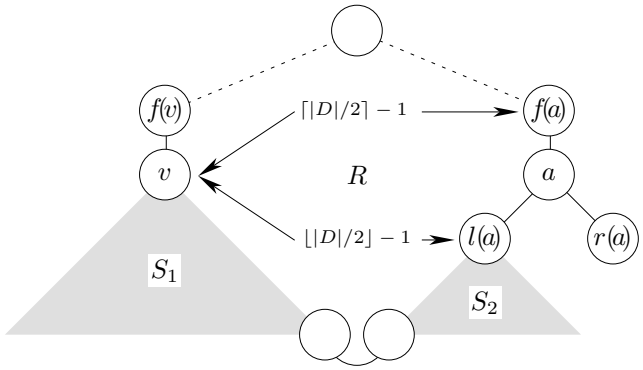


Figure 3.5: *An internal vertex a on the right side of the contour D of a region R. Dashed lines correspond to zero or more edges, each solid line corresponds to a single edge.*

Let v be the vertex on D such that a shortest path, P say, from $l(a)$ to v is shorter than (or as short as) the shortest path, P' say, from $f(a)$ to v, and P' is at most 1 step longer than P. Theorem 6 implies that a shortest path between any two vertices on the same contour must follow that contour. Thus[8] $|P| = \lfloor |D|/2 \rfloor - 1$.

Now, since no cross-level edges are allowed, v must be on the left side of D (simple arithmetics), and a shortest path from a to any descendent of v (in S_1) or a descendent of $l(a)$ (in S_2) goes through $l(a)$. For the same reason, a shortest path from a to any vertex outside S_1 and S_2 goes through either $r(a)$ or $f(a)$. Now, due to cycle ordering,

$$\mathrm{desc}(v) \cup \mathrm{desc}(l(a)) = \{\, x \mid \min(\mathrm{desc}(v)) \leq x \leq \max(\mathrm{desc}(l(a))) \,\},$$

and thus, intuitively $\mathrm{lmin}(a) = \min(\mathrm{desc}(v))$ and $\mathrm{lmax}(a) = \max(\mathrm{desc}(l(a)))$ should be optimal.

[8]Note that $|P|$ ($|D|$) is the number of edges in P (D), i.e., the length of P (D).

If a does not belong to the right side of any contour, i.e., a is a pre-vertex, then $\mathrm{lmin}(a)$ should clearly be $\min(\mathrm{desc}(l(a)))$, which is vertex $a+1$.

Note that the above reasoning holds *only under the assumption that the cycletree contains no cross-level edges*. It is easy to find counterexamples otherwise. We will next treat the problem more formally and handle internal and external vertices separately.

Optimal router data of internal vertices

Let us denote by $x \xrightarrow{s} y$ a shortest path from vertex x to vertex y in G.

Definition 7 ($\circ$) Let a be a vertex of G. We let $\circ a$ and $a\circ$ be the following vertices of G. If a is a pre-vertex then $\circ a = l(a)$. Otherwise a is on the right side of a contour D and we let $\circ a$ be the vertex on D such that

$$|f(a) \xrightarrow{s} \circ a| - 1 \leq |l(a) \xrightarrow{s} \circ a| \leq |f(a) \xrightarrow{s} \circ a| \tag{3.1}$$

(see Figure 3.5 where $v = \circ a$). If a is a post-vertex then $a\circ = r(a)$. Otherwise a is on the left side of a contour D and we let $a\circ$ be the vertex on D such that

$$|f(a) \xrightarrow{s} a\circ| - 1 \leq |r(a) \xrightarrow{s} a\circ| \leq |f(a) \xrightarrow{s} a\circ| \tag{3.2}$$

(see Figure 3.5 where $v\circ = a$). ∎

Example 3.3.2 Let us look at the internal vertex 8 of the cycletree in Figure 3.4. We get $\circ 8 = 2$ and $8\circ = 10$ (see Figure 3.6). It is easy to see that any descendent of vertex 2, 8 or 10 is closer to vertex 8 via one of its sons than via its father, and any other vertex (in the dashed area) is closer to vertex 8 via its father. In fact, $\mathrm{lmin}(8) = 2$, $\mathrm{lmax}(8) = 7$, $\mathrm{rmin}(8) = 9$ and $\mathrm{rmax}(8) = 10$ are optimal. ∎

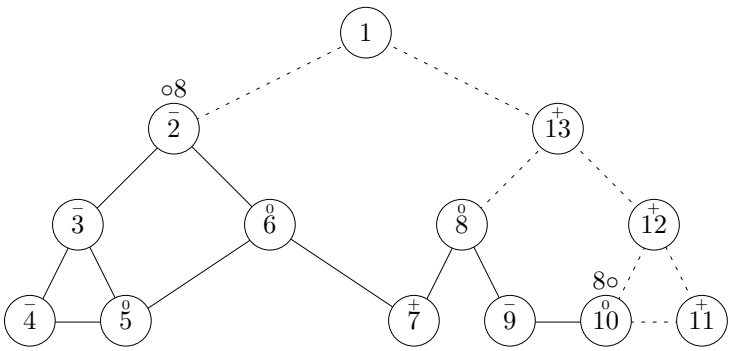


Figure 3.6: *The subgraph of the cycletree, induced by the descendents of vertices 8, 2 and 10 is drawn with solid lines.*

Summing up, we have the following definitions.

Definition 8 (*Internal router data*) Let G be a cycletree having no cross-level edges and let a be an internal vertex of G. We define the router data of a as follows.

$$\mathrm{lmin}(a) = \min(\mathrm{desc}(\circ a)), \tag{3.3}$$
$$\mathrm{lmax}(a) = \max(\mathrm{desc}(l(a))), \tag{3.4}$$
$$\mathrm{rmin}(a) = \min(\mathrm{desc}(r(a))), \tag{3.5}$$
$$\mathrm{rmax}(a) = \max(\mathrm{desc}(a\circ)), \tag{3.6}$$

∎

Optimal router data of external vertices

The above router data need not be optimal for external vertices. This can easily be confirmed by looking at the cycletree in Figure 3.4. Vertex 10 is on the right side of the contour of region R_{13}. We get that $\min(\mathrm{desc}(\circ 10)) = \min(\mathrm{desc}(8)) = 7$, but for example vertex 5 is closer to vertex 10 via vertex 9 than via vertex 12.

Let a be an external vertex of a cycletree G having no cross-level edges and let $N = |V_G|$. Trivially, we must have $\mathrm{lmax}(a) = l(a)$ and $\mathrm{rmin}(a) = r(a)$. It is also easy to see that, whenever a is a descendent of $l(a)$ or $r(a)$, then $\mathrm{lmin}(a) = 1$ or $\mathrm{rmax}(a) = N$, respectively, is optimal. (Note that a cannot be *both* a descendent of $l(a)$ *and* a descendent of $r(a)$.) If $l(a)$ or $r(a)$ is a descendent of a then $\mathrm{lmin}(a) = \min(\mathrm{desc}(\circ a))$ or $\mathrm{rmax}(a) = \max(\mathrm{desc}(a\circ))$, repectively, is optimal. The reasoning in the last case is the same as if a had been internal (see Figure 3.5).

Assume now that $l(a)$ is at the same level as a (see Figure 3.7), and let us find the optimal value of $\mathrm{lmin}(a)$ (treatment of $\mathrm{rmax}(a)$, when $r(a)$ and a are at the same level, is analogous).

Let z be the least vertex that has $l(a)$ as its maximum descendent. Then z must be the left son of $f(z)$, or else z would not be *least*. Thus z is either a pre-vertex or an in-vertex. If z is an in-vertex then it is on the right side of the contour D of a region (R' in the figure). Vertex $*z$ in the figure denotes vertex $\circ z$, if D has an odd number of vertices; $r(\circ z)$, otherwise. (The formal definition is given below.) The subgraph S_1 is induced by $\mathrm{desc}(*z)$ and the subgraph S_2 is induced by $\mathrm{desc}(z)$. Any vertex in S_1 or S_2 is now closest to a via $l(a)$, and any vertex outside S_1 and S_2 is closest to a via $f(a)$ or $r(a)$.

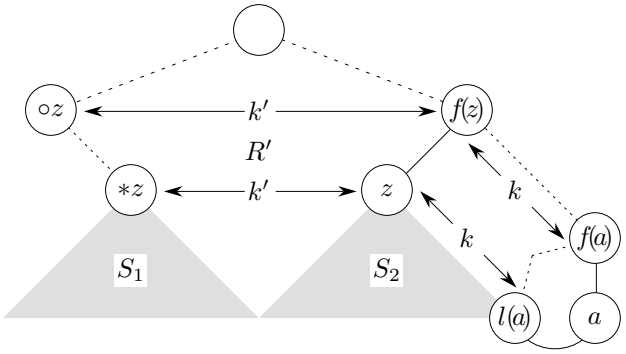


Figure 3.7: *An external vertex a at the same level as $l(a)$. Vertex z is the minimum vertex such that $l(a) = \max(\mathrm{desc}(z))$; $k = |l(a) \xrightarrow{\mathrm{s}} z|$ and $k' = |z \xrightarrow{\mathrm{s}} *z|$.*

Definition 9 ($*$) Let v be a vertex of G. We let $*v$ and $v*$ be the following vertices of G. If v is a pre-vertex then $*v = v$. Otherwise v is on the right side of a contour D and we let $*v$ be the vertex on D such that

$$|v \xrightarrow{\mathrm{s}} *v| = |f(v) \xrightarrow{\mathrm{s}} \circ v|. \tag{3.7}$$

We let, accordingly, $v* = v$ if v is a post-vertex. Otherwise v is on the left side of a contour D and we let $v*$ be the vertex on D such that

$$|v \xrightarrow{\mathrm{s}} v*| = |f(v) \xrightarrow{\mathrm{s}} v\circ|. \tag{3.8}$$

∎

Summing up, we have the following definitions.

Definition 10 (*External router data*) Let G be a cycletree having no cross-level edges and let a be an external vertex of G. We define the router data of a as follows.

$$\text{lmin}(a) = \begin{cases} 1, & \text{if } a \in \text{desc}(l(a)); \\ \text{min}(\text{desc}(\circ a)), & \text{if } l(a) \in \text{desc}(a); \\ \text{min}(\text{desc}(*z)), & \text{otherwise, where } z = \text{min}\{\, x \mid l(a) = \text{max}(\text{desc}(x)) \,\}, \end{cases} \tag{3.9}$$

$$\text{lmax}(a) = l(a), \tag{3.10}$$

$$\text{rmin}(a) = r(a), \tag{3.11}$$

$$\text{rmax}(a) = \begin{cases} N, & \text{if } a \in \text{desc}(r(a)); \\ \text{max}(\text{desc}(a\circ)), & \text{if } r(a) \in \text{desc}(a); \\ \text{max}(\text{desc}(z*)), & \text{otherwise, where } z = \text{max}\{\, x \mid r(a) = \text{min}(\text{desc}(x)) \,\}, \end{cases} \tag{3.12}$$

∎

Example 3.3.3 Let us consider vertex 10 in Figure 3.4, and let us calculate lmin(10) first. As vertices 10 and $l(10) = 9$ are at the same level, the third case of Formula 3.9 must apply. Now, the least vertex that has 9 as its maximum descendent is vertex 8. We know that $*8 = 6$. The minimum descendent of 6 is 5. Thus lmin(10) = 5 (see Figure 3.8). We calculate rmax(10), in a similar fashion, to be 11, by using the third case of Formula 3.12. Clearly, a shortest path to any other vertex (in the dashed area) goes through vertex 12. ∎

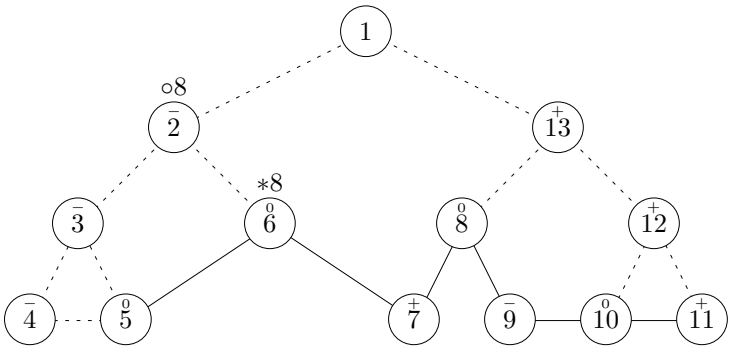


Figure 3.8: *The subgraph induced by* $\{\, x \mid 5 \le x \le 11 \,\}$ *is shown with solid lines.*

Theorem 8 *Let G be a cycletree network having no cross-level links and let the router data of each node in G have the values given by Definition 8 and Definition 10 in the internal and external case, respectively. Then Algorithm 2 terminates and produces a shortest path between any pair of nodes in G.*

Proof. See Appendix A.2. ∎

3.4 Establishing optimal router data dynamically

We will next treat the problem of dynamically establishing the optimal router data in a cycletree network. We assume that we are given a network of N nodes configured as a cycletree without cross-level edges. We present a parallel algorithm that establishes the optimal router data in each node. We assume that we are given $O(N)$ processors, each having $O(1)$ memory. Let the depth of the cycletree be K. We will show that the algorithm runs in $O(K)$ time units, and thus is logarithmic in N if the cycletree is tree-complete.

In the algorithm each node is seen as a process that consists of smaller (possibly concurrent) subprocesses, fair scheduling of which is assumed. We assume a *rendezvous* communication model between nodes. An edge beteen two vertices in the interconnection graph corresponds to a bidirectional synchronous channel or *link* between the corresponding nodes. Thus a communication event takes place when both parts are ready.

The algorithm is described from the local viewpoint of one node and the same algorithm is run in every node of the cycletree. Thus, the algorithm in one node is in itself a subprocess of the node, which upon termination has established the optimal router data of that node. The optimal router data can then be used, as described by Algorithm 2, to obtain optimal communication when running further programs.

We will start by giving a brief overview of the algorithm. We then give a detailed description of the algorithm and illustrate the algorithm with an example. Finally, we argue for its correctness and calculate its parallel complexity. (It could provide easier understanding of the algorithm if sections 3.4.1, 3.4.2 and 3.4.3 were studied simultaneously.)

3.4.1 Overview of the algorithm

The information that is assumed to be known in each node a is: whether a is internal or external, mark of a, (and the address of a)[9]. Upon termination $lmin = \mathrm{lmin}(a)$, $lmax = \mathrm{lmax}(a)$, $rmin = \mathrm{rmin}(a)$ and $rmax = \mathrm{rmax}(a)$. The algorithm consists of 5 steps.

Step 1. Initialization. Fetch the mark of $f(a)$. If either a or $f(a)$ is a pre-node then a is on the left side of a contour D. Otherwise run the "opposite" algorithm in a (roughly: left and right are switched). The rest of the algorithm is described as if a is a pre-node or $f(a)$ is a pre-node. (If a is an in-node then it is also on the right side of another contour D'.)

Step 2. Calculate α as the number of ascendents of a, other than a itself, on the same side of D, δ as the difference between the length of a's side of D and the length of the opposite side of D. If a is an in-node then calculate also δ' as the corresponding value with respect to D'.
(*Subglobal view: nodes on the left and the right sides of a contour exchange information.*)

Step 3. Calculate *mind* as $\min(\mathrm{desc}(a))$, *maxd* as $\max(\mathrm{desc}(a))$, *lmax* and *rmin* (as above). If a is external then it uses δ and δ' to determine its immediate ascendent(s) and descendent(s).
(*Global view: information is propagated upwards from descendent to ascendent.*)

Step 4. (Kernel 1.) Calculate *lmin* as $\min(\mathrm{desc}(\circ a))$, and *rmax* as $\max(\mathrm{desc}(a\circ))$. Also, calculate *rmax** as $\max(\mathrm{desc}(a*))$ if a is an in-node. The router data are now optimal, unless $r(a)$ or $l(a)$ is at the same level as a.
(*Subglobal view: nodes on the left and the right sides of a contour exchange information.*)

Step 5. (Kernel 2.) The optimal values for *lmin* and *rmax* are calculated if a is external.
(*Global view: information is propagated downwards from ascendent to descendent.*)

An important property

There is one important property of cycletrees that follows easily from Theorem 6 and Theorem 7 and is used in steps 2 and 4. Namely, the left and the right sides

[9] As before, we will identify a node directly with its address.

of a contour and the nontree edge connecting those sides constitute a path that is *edge-disjoint* with the corresponding paths with respect to other contours.

In steps 2 and 4 all communication takes place within such a subpath of a contour in a systolic manner. We can therefore describe those steps "subglobally" by looking at each contour separately. See Figure 3.9 that illustrates the property for the cycletree shown previously in Figure 1.1.

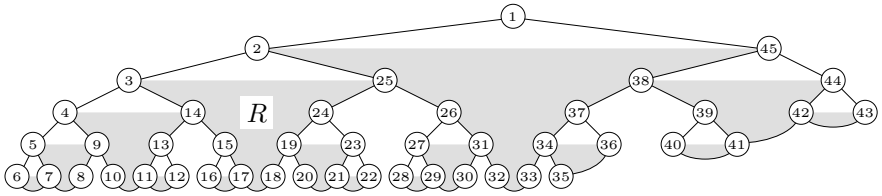


Figure 3.9: *The grey area of a region is spanned by the left side, the right side and the nontree edge of the contour of that region. No two grey areas share an edge.*

Before turning to the formal treatment of the algorithm, let us illustrate Step 4 subglobally by extracting the subpath of the contour spanning the grey area of region R in Figure 3.9. See Figure 3.10.

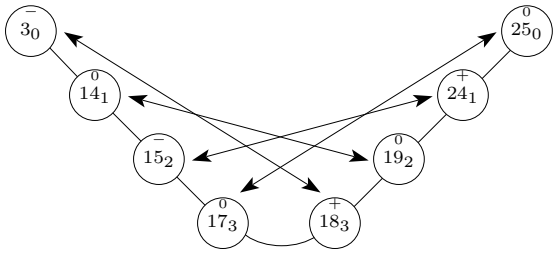


Figure 3.10: *The subpath of the contour spanning the grey area of region R in Figure 3.9.*

An arrow between two nodes x and y in Figure 3.10 shows that x and y need a value from each other. In particular, node 14 needs the value of max(desc(19)) from node 19 for its *rmax*, and node 19 needs the value of min(desc(14)) from node 14 for its *lmin*. (Those values were calculated during Step 3.)

In order to accomplish such an exchange of information in a systolic manner, the values of α and δ are used (those values were calculated during Step 2). The number of ascendents of a node, other than the node itself, that are on the same side of the contour of region R is shown as an index of that node's address in the figure. This value (except for node 25) is the value of α in the corresponding node.

For example, during Step 4, node 15 needs to bypass 2 ($= \alpha$) values from node 17 to node 14, before it receives the value for its *rmax* from node 17 (originally sent by node 24). Concurrently, node 15 sends the value of its minimum descendent to node 17 and after that bypasses 2 ($= \alpha$) values from node 14 to node 17.

3.4.2 The algorithm

We shall now present a detailed description of the algorithm. The algorithm is described as it is executed in one node a. We name the neighbours $f(a)$ (if any), $l(a)$ and $r(a)$, by f, l and r, respectively.

Let b be a neighbour of a. We denote a send operation of the value of an expression e to b by $b!e$, and a receive operation of a value from b that is saved in variable x

by $b?x$. Furthermore, we will denote two concurrent subprocesses P and Q of the algorithm by $P \parallel Q$, and two sequential subprocesses, P followed by Q, by $P\,;\,Q$. We adopt the convention that ';' binds more tightly than '‖'. We can thus drop the parentheses in $(P\,;\,Q) \parallel R$ and write it as $P\,;\,Q \parallel R$.

(The algorithm can almost trivially be translated into either CSP [28] or OCCAM [32].)

Algorithm 3 Let a be a node in a network of N nodes configured as a cycletree with no cross-level edges. Let m be the mark of a and let I be a flag which is true, if a is an internal node; otherwise false. Upon termination the variables *lmin*, *lmax*, *rmin* and *rmax* hold the optimal values of the router data lmin(*a*), lmax(*a*), rmin(*a*) and rmax(*a*), respectively. At the top level the algorithm is described by the process

$$\textbf{if } a = 1 \textbf{ then } \mathit{S1}' \textbf{ else } (\mathit{S1}\,;\,\mathit{S2}\,;\,\mathit{S3}\,;\,\mathit{S4}\,;\,\mathit{S5}).$$

[**Step 1**] calculates *mf*, the mark of f. If $a = 1$ then a has no father and it acts like a pre-node for its left son and like a post-node for its right son. If $a = 1$ then the router data are calculated in *S1′* and the algorithm terminates. (The values for *lmax* and *rmin* are sent from l and r during *S3*.)

$$\begin{aligned} \mathit{S1}' &= (l!\mathit{Pre} \parallel r!\mathit{Post})\,;\,(\mathit{lmin}, \mathit{rmax} := 2, N \parallel l?\mathit{lmax} \parallel r?\mathit{rmin}) \\ \mathit{S1} &= (f?\mathit{mf} \parallel \textbf{if } I \textbf{ then } (l!m \parallel r!m))\,;\,\mathit{Switch} \end{aligned}$$

Switch: If either a or f is a pre-node then let $\mathit{one} := 1$ and $\mathit{limit} := N$. Otherwise run the "opposite" algorithm in a: let $\mathit{one} := -1$, $\mathit{limit} := 1$ and switch the names, r with l, *lmin* with *rmax* and *lmax* with *rmin*.

[**Step 2**] calculates α and δ; also δ' if a is an in-node.

$$\begin{aligned} \mathit{S2} &= (\textbf{if } m = \mathit{mf} \textbf{ then } \mathit{S21} \textbf{ else } \mathit{S22}) \parallel \\ &\quad\ (\textbf{if } m = \mathit{In} \textbf{ then } \mathit{S23}) \end{aligned}$$

S21 or *S22* calculates α and δ, and *S23* calculates δ' (δ' is defined only if a is an in-node). In *S21*, a has no ascendents, other than a itself, on the same side of the contour. In *S22*, a has at least one such ascendent. (See Figure 3.11.)

$$\begin{aligned} \mathit{S21} &= \alpha := 0\,;\, r!1 \parallel r?x\,;\, \delta := 1 - x \\ \mathit{S22} &= (\mathit{S221} \parallel \mathit{S222})\,;\, \delta := \alpha + 1 - x \\ \mathit{S221} &= f?\alpha\,;\, r!(\alpha + 1) \\ \mathit{S222} &= r?x\,;\, f!(x - 1) \\ \mathit{S23} &= l!1 \parallel l?y\,;\, \delta' := 1 - y \end{aligned}$$

[**Step 3**] calculates *maxd*, *mind*, *lmax* and *rmin*.

$$\mathit{S3} = \mathit{S31}\,;\, f!\mathit{maxd} \parallel \mathit{S32}\,;\, \mathit{S33}\,;\, \textbf{if } \neg I \wedge \delta > 0 \textbf{ then } r!\mathit{mind}$$

In *S31*, the value of *maxd* is received from r if r is a descendent of a, else a is its own greatest descendent. The value of *maxd* is then sent to f. In *S32*, x is received from l if l is a descendent of a, x is set to a otherwise. In *S33*, x is used to calculate *mind*, *lmax* and *rmin*. Finally, the value of *mind* is sent to r if r is an ascendent of a.

$$\begin{aligned} \mathit{S31} &= \textbf{if } I \vee \delta < 0 \textbf{ then } r?\mathit{maxd} \textbf{ else } \mathit{maxd} := a \\ \mathit{S32} &= \textbf{if } I \vee (m = \mathit{In} \wedge \delta < 0) \textbf{ then } l?x \textbf{ else } x := a \\ \mathit{S33} &= \textbf{if } I \wedge m \neq \mathit{In} \textbf{ then } \mathit{mind}, \mathit{lmax}, \mathit{rmin} := a, x, x + \mathit{one} \\ &\quad\ \textbf{else } \mathit{mind}, \mathit{lmax}, \mathit{rmin} := x, a - \mathit{one}, a + \mathit{one} \end{aligned}$$

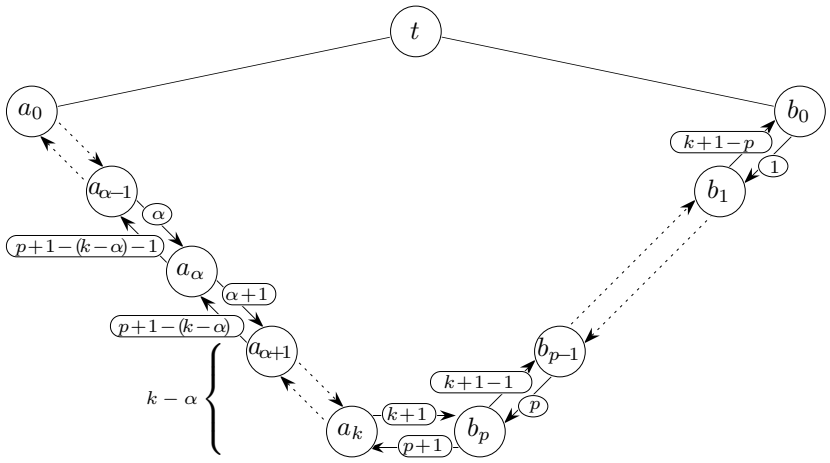


Figure 3.11: *Subglobal view of Step 2. In node a_α the value of $p+1-(k-\alpha)$ is received as x and δ is calulated as $\delta = \alpha+1-x = k-p$. The "opposite" algorithm is executed in nodes $b_1, \ldots, b_p$, and either in b_0 and a_0 if t is a post-node, or in b_0 if t is the root of the cycletree.*

[**Step 4**] calculates *rmax* and *lmin*; also *rmax** if a is an in-node. The value of *rmax** is needed in Step 5. The value of *rmax* (*lmin*) is now optimal, unless r (l) is at the same level as a.

$$\begin{aligned} S4 &= (\textbf{if } I \vee \delta \leq 0 \textbf{ then } (S41\,;\, S41^*) \textbf{ else } (rmax := limit\,;\, S42)) \parallel \\ &\quad (\textbf{if } m = In \textbf{ then } S43 \textbf{ else } lmin := a + one) \\ S43 &= l!maxd \parallel l?lmin \end{aligned}$$

In *S41* and *S41**, *rmax* and *rmax**, respectively, are calculated (see Figure 3.12).

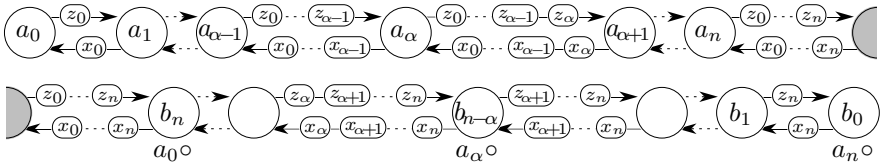


Figure 3.12: *Subglobal view of Step 4 of Algorithm 3; $(a_0, \ldots, a_k)$ and $(b_0, \ldots, b_p)$ are the left and right sides of a contour, respectively, and $n = \min(k, p)$; $z_i = \min(\mathrm{desc}(a_i))$ and $x_i = \max(\mathrm{desc}(a_i\circ))$. The grey node is either a_k, if $k = p+1$; b_p, if $p = k+1$; otherwise nonexistent.*

If r is an ascendent of a then the optimal value of *rmax* is trivially *limit* (see Definition 3.12). Furthermore, *rmax** is not needed in Step 5, since a cannot be the maximum ascendent that has the same least descendent as a, and *S42* is executed to simply pass α values from f to r and from r to f.

The value of *lmin* is calculated concurrently.

The following works only because no cross-level edges are allowed, otherwise deadlock would occur. In *S41*, a sequence of α values bypass a from r to f and the $(\alpha+1)$'st value x from r is assigned to *rmax*. The α'th value from r (if any) is in x'. Now, if a is an in-node then $\alpha > 0$ and we know that $x' = \max(\mathrm{desc}(f\circ))$ and *rmax** is calculated by using that

$$\max(\mathrm{desc}(a*)) = \begin{cases} \max(\mathrm{desc}(f\circ)) = x', & \text{if } |C| \text{ is even } (\delta \neq 0); \\ \max(\mathrm{desc}(a\circ)) = x, & \text{otherwise.} \end{cases}$$

Concurrently, the value of *mind* is sent to r and α values bypass a from f to r.

$$\begin{aligned}
S41 &= S411 \parallel S412 \\
S411 &= z := mind\ ;\ FtoR(\alpha)\ ;\ r!z \\
S412 &= r?x\ ;\ RtoF(\alpha)\ ;\ rmax := x \\
S41* &= \textbf{if}\ \delta = 0\ \textbf{then}\ rmax^* := x\ \textbf{else}\ rmax^* := x' \\
S42 &= f?z\ ;\ FtoR(\alpha-1)\ ;\ r!z \parallel r?x\ ;\ RtoF(\alpha-1)\ ;\ f!x \\
RtoF(k) &= ;_{1\le i\le k}(x' := x\ ;\ (f!x' \parallel r?x))_i \\
FtoR(k) &= ;_{1\le i\le k}(z' := z\ ;\ (r!z' \parallel f?z))_i
\end{aligned}$$

[**Step 5**] calculates the optimal values of *rmax* (*lmin*) if a is an external node at the same level as r (l).

$$S5 = \textbf{if}\ I\ \textbf{then}\ S51\ \textbf{else}\ (\textbf{if}\ m \neq In\ \textbf{then}\ S52\ \textbf{else}\ S53)$$

If a is internal then the following is done in *S51*. If $\delta < 1$ (then max(desc(a)), say b, and $r(b)$ can be at the same level), and if a and f are pre-nodes then *mind* (equals min(desc($*a$))) is sent to r. If $m \neq mf$ and $\delta < 1$ then a value is passed from f to r. Thus, that value is eventually received by b, and is from there sent to $r(b)$ as the optimal value of lmin($r(b)$) if b is at the same level as $r(b)$.

Accordingly, if a is an in-node and $\delta' < 1$ then *rmax**, which was calculated in *S4*, is sent to l and is eventually received by l(min(desc(a))) (or discarded by min(desc(a)) if l(min(desc(a))) is not at the same level).

S52 and *S53* work similarly (see Figure 3.13 for *S53*).

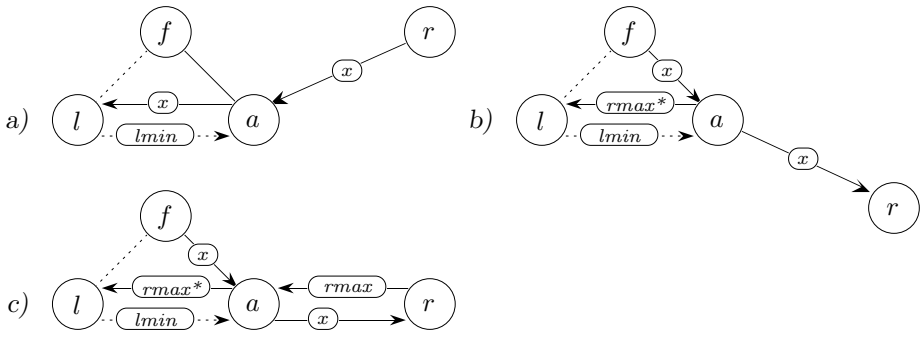


Figure 3.13: *Illustration of S53; a) $\delta = 1$; b) $\delta = -1$; c) $\delta = 0$. In all cases, if $\delta' = 0$, i.e., a is at the same level as l, then lmin is received from l as illustrated with the dashed arrow.*

$$\begin{aligned}
S51 &= (\textbf{if}\ \delta < 1\ \textbf{then}\ (\textbf{if}\ m = mf\ \textbf{then}\ r!mind\ \textbf{else}\ (f?x\ ;\ r!x)))\ \parallel \\
&\quad (\textbf{if}\ m = In \wedge \delta' < 1\ \textbf{then}\ l!rmax^*) \\
S52 &= (\textbf{if}\ \delta < 1\ \textbf{then}\ (\textbf{if}\ m = mf\ \textbf{then}\ r!mind\ \textbf{else}\ (f?x\ ;\ r!x)))\ \parallel \\
&\quad (\textbf{if}\ \delta \ge 0\ \textbf{then}\ r?y)\ ;\ (\textbf{if}\ \delta = 0\ \textbf{then}\ rmax := y) \\
S53 &= (\textbf{if}\ \delta' = 0\ \textbf{then}\ l?lmin)\ \parallel \\
&\quad (\textbf{if}\ \delta > 0\ \textbf{then}\ r?x\ ;\ l!x)\ ; \\
&\quad (\textbf{if}\ \delta < 0\ \textbf{then}\ (f?x\ ;\ r!x \parallel l!rmax^*))\ ; \\
&\quad (\textbf{if}\ \delta = 0\ \textbf{then}\ (f?x\ ;\ r!x \parallel l!rmax^* \parallel r?rmax))
\end{aligned}$$

∎

Example 3.4.1 Let us illustrate Algorithm 3 with a cycletree of 9 nodes, as shown in Figure 3.14. During Step 1 is *mf* calculated in each node. The "opposite" algorithm is executed in nodes 4, 7, 8 and 9.

In Step 2 we get a flow of data through the cycletree. For example, in node 5, $\alpha = 1$, $\delta = \alpha + 1 - 1 = 1$ (level difference between nodes 6 and 7), and $\delta' = 1 - 0 = 1$ (level difference between nodes 4 and 3).

During Step 3 the minimum and maximum descendents are calculated. For example, nodes 7 and 9 receive 6 as their least descendent. Note that, at this point the root receives its values for *lmax* and *rmin* (see *S1′*).

During Step 4 most nodes receive the optimal values for *lmin* and *rmax*. Nodes 7 and 8 are exceptions. In node 7 the current value of *rmax* is not known to be optimal and in node 8 the current value of *lmin* is not known to be optimal (those values are emphasized in the figure). In node 5 $rmax^* = 7$, and in node 7 (the "opposite" algorithm) $lmin^* = 4$.

During Step 5 node 8 receives the optimal value for *lmin*, which is the value of $lmin^*$ sent by node 7. Node 7 receives a new value for its *rmax* from node 8. The other events have no effect on the current values (the values received by nodes 4 and 6 are discarded). ∎

3.4.3 Correctness

We will now prove in a relatively informal manner that each step of the algorithm terminates and produces the values it is supposed to, or in one word, that the algorithm is *correct*. Certain parts will be skipped, for example Step 1, where a proof seems superfluous.

Let us first introduce some useful concepts. Let $X_0, X_1, \ldots, X_k$, $k \geq 0$, be concurrent processes forming a *chain*, i.e., X_i uses one output channel which is the only input channel used by X_{i+1} for $0 \leq i < k$. We denote it by

$$X_0 \gg X_1 \gg \cdots \gg X_k.$$

The above chain is called a *pipe* [29], if X_0 uses one input channel and X_k uses one output channel; we call it an *isolated pipe* if X_0 uses no input channel and X_k uses no output channel. A pipe (and thus an isolated pipe) has the nice property that it does not *deadlock* if none of the elements do [29].

The following notations will be used. A process X which is executed by the "opposite" algorithm is denoted by $\overline{X}$. The index of a process denotes the node where the corresponding process is executed.

Step 2. Let us look subglobally at one contour D of the cycletree. Let $(a_0, \ldots, a_k)$ be the left side of D and $(b_0, \ldots, b_p)$ the right side of D (see Figure 3.11). The execution of *S2* initially forms an isolated pipe from a_0 to b_0 and from b_0 to a_0

$$\begin{array}{cl} & X1_{a_0} \gg S221_{a_1} \gg \cdots \gg S221_{a_k} \gg \overline{S222}_{b_p} \gg \cdots \gg \overline{S222}_{b_1} \gg Y2_{b_0} \\ \| & X2_{a_0} \ll S222_{a_1} \ll \cdots \ll S222_{a_k} \ll \overline{S221}_{b_p} \ll \cdots \ll \overline{S221}_{b_1} \ll Y1_{b_0}, \end{array}$$

where

$$\begin{array}{rcl} X1 \,\|\, X2 & = & S21, \text{ if } a_0 \text{ is a pre-node; } \overline{S23} \text{ otherwise,} \\ Y1 \,\|\, Y2 & = & \overline{S21}, \text{ if } b_0 \text{ is a post-node; } S23 \text{ otherwise.} \end{array}$$

Let P be any one of the two chains. The first element of P sends a value to the next element and terminates. The rest of P then forms an isolated pipe which reduces similarly until the last element of P receives a value and terminates.

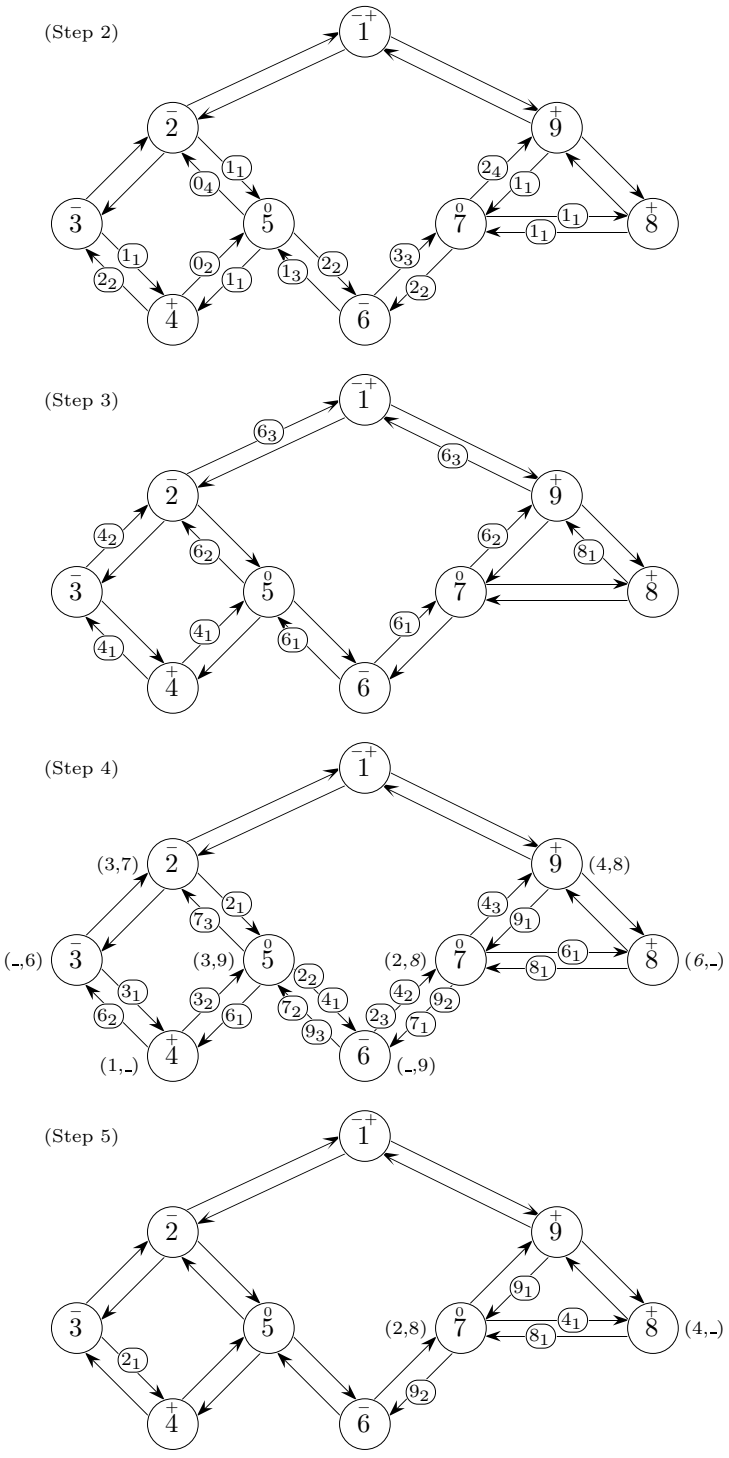


Figure 3.14: *Sample execution of Algorithm 3. An arrow denotes a unidirectional channel between two neighbours. A communication event between two neighbours is shown by a label on the corresponding channel. The indices of the events show their chronological ordering. The values of* $(lmin, rmax)$ *upon completion of Step 4 are illustrated.*

It is not hard to see, by examining the code of Step 2 and Figure 3.11, that correct values for α and δ have been established in each node.

Step 3. Termination of *S3* is guaranteed by the fact that in each node a, process X_a, where X is $S1'$, if a is the root; *S3*, if a or $f(a)$ is a pre-node; $\overline{S3}$, otherwise, receives a value concurrently from all immediate descendents of a (if any), and, unless a is the root, sends a value concurrently to all immediate ascendents of a. There are no cycles in the corresponding global logical process network, which implies termination [29], since X neither deadlocks nor livelocks.

It is not hard to see, by examining the code of *S3*, which uses the cycle order properties of cycletrees to calculate *maxd*, *mind*, *lmax* and *rmin*, that the calculated values are correct.

Step 4. Let D, $(a_0, \ldots, a_k)$ and $(b_0, \ldots, b_p)$ be as above. The execution of *S4* forms initially, in a manner similar to the execution of *S2*, an isolated pipe from a_0 to b_0 and from b_0 to a_0. Let $n = \min(k, p)$. We have the isolated pipes

$$\begin{aligned} & X1_{a_0} \gg S411_{a_1} \gg \cdots \gg S411_{a_n} \gg Z1 \gg \overline{S412}_{b_n} \gg \cdots \gg \overline{S412}_{b_1} \gg Y2_{b_0} \\ \| \quad & X2_{a_0} \ll S412_{a_1} \ll \cdots \ll S412_{a_n} \ll Z2 \ll \overline{S411}_{b_n} \ll \cdots \ll \overline{S411}_{b_1} \ll Y1_{b_0}, \end{aligned}$$

where

$$\begin{aligned} X1 \parallel X2 &= S41, \text{ if } a_0 \text{ is a pre-node; } \overline{S43}, \text{ otherwise,} \\ Y1 \parallel Y2 &= \overline{S41}, \text{ if } b_0 \text{ is a post-node; } S43, \text{ otherwise,} \\ Z1 \parallel Z2 &= S42_{a_{n+1}}, \text{ if } k = p+1; \ \overline{S42}_{b_{n+1}}, \text{ if } p = k+1; \text{ none, otherwise.} \end{aligned}$$

The "none" case means that $S411_{a_n} \gg \overline{S412}_{b_n}$ and $S412_{a_n} \ll \overline{S411}_{b_n}$ are pipes. Note that $-1 \leq k - p \leq 1$, since cross-level edges are disallowed.

Let P be any one of the above isolated pipes, and let $m = \max(k, p)$. First, P_i and P_{i+1}, $0 \leq i \leq m$, communicate and P_0 terminates. Next, P_i and P_{i+1}, $1 \leq i \leq m$, communicate and P_1 terminates. Let $m = m-1$. The rest of P then forms an isolated pipe which reduces in the same manner until $m = -1$. (See also Figure 3.12.)

Now, in every node a_i, $0 \leq i \leq n$, $rmax_{a_i}$ has been assigned the value of $maxd_{b_{n-i}}$, which was originally sent by b_{n-i}. It is easy to calculate that $b_{n-i} = a_i\circ$, and thus $rmax_{a_i} = maxd_{a_i\circ} = \max(\mathrm{desc}(a_i\circ))$ which is correct. It is straightforward to see that also the other values are correct.

Step 5. The execution of *S5*, viewed globally, initially forms a set of isolated pipes. Let

$$P_{a_0} \gg P_{a_1} \gg \cdots \gg P_{a_n}$$

be an element in that set. There are two symmetric cases. One of those is that a_0 is the *minimum* node which has a_n as its *maximum* descendent (possibly a_0 itself), i.e.,

$$a_0 = \min\{\, x \mid \max(\mathrm{desc}(x)) = a_n \,\}.$$

Then a_0 is the left son of its father and a_0 has no other immediate ascendents. Furthermore, $a_{i+1} = r(a_i)$ and $a_{i+1} \in \mathrm{desc}(a_i)$, for $0 \leq i < n$. Thus, a_n is either the rightmost son of a_0, or if the rightmost son of a_0 is not the maximum descendent of itself then (let $z = a_0$) node $l(a)$ as illustrated in Figure A.5.

Initially P_{a_0} sends the value of $\min(\mathrm{desc}(*a_0))$ to P_{a_1} (unless $n = 0$). That value is either the value of $mind_{a_0}$, if a_0 and $f(a_0)$ are pre-nodes; the value of $lmin^*_{a_0}$, otherwise, i.e., if a_0 is an in-node and executes the "opposite" algorithm. The subsequent elements of the chain pass that value on to their immediate successors and terminate, until P_{a_n} is reached. Evidently, the chain terminates.

Finally, P_{a_n} sends that value to node $r(a_n)$ if a_n and $r(a_n)$ are at the same level. In node $r(a_n)$ the received value is assigned to $lmin_{r(a_n)}$ and is now optimal according to Formula 3.9 and thus correct.

3.4.4 Complexity analysis

Let $G[C,T]$ be a cycletree with no cross-level edges. Let $N = |V_G|$, and let K be the depth of T. As we let G be the interconnection graph that defines the topology of our MP-RAM model, we trivially associate each node of G with a processor in the model. Thus we assume a *balanced distribution* of an *identity-size problem* [2].

We show that Algorithm 3 runs in $\mathrm{O}(K)$ time and requires $\mathrm{O}(1)$ memory per processor. The latter statement can be confirmed simply by observing that the number of variables per node (each capable of holding one datum[10]), used by the algorithm, is constant.

Let us now treat the time complexity of the algorithm. In our idealized model, a communication event of one datum between any two neighboring nodes costs $\mathrm{O}(1)$ time units. We treat the algorithm stepwise and denote the time taken by each step i by $T^i(N)$.

Clearly $T^1(N) = \mathrm{O}(1)$. We know from the discussion in the previous section that $T^3(N) = \mathrm{O}(K)$ and $T^5(N) = \mathrm{O}(K)$. We know also from the previous section that $T^2(N) = \mathrm{O}(K)$ and $T^4(N) = \mathrm{O}(K)$, since the size of the largest contour is $\mathrm{O}(K)$. The overall time cost is therefore $T(N) = \mathrm{O}(K)$. The most interesting case is when G is tree-complete. In that case we know that $K = \mathrm{O}(\log N)$, and thus $T(N) = \mathrm{O}(\log N)$.

[10] An integer between -1 and N.

Chapter 4

Conclusions and future work

Cycletrees reflect the communication patterns of several common parallel programming paradigms. In the general case, however, the communication patterns are not directly identifiable, e.g., in computations resulting from automatic parallelization. The most common communication patterns that arise in parallel computations are, arguably, supported by a binary tree structure and a circular array structure.

We have shown in Chapter 2 that a cycletree includes any basic binary tree, has a unique Hamiltonian cycle and that the maximum degree of a natural cycletree is 3, which is clearly the lowest possible. We have also shown that a cycletree, if minimal, has the smallest possible number of nontree edges. Thus a cycletree can be used to realize both a basic binary tree structure and a circular array structure for the lowest possible cost, from an embedding or mapping point of view.

Natural cycletrees have an appealing inductively defined structure. In Chapter 3 we first showed with Algorithm 1 how to construct a (natural) cycletree recursively by using the inductive nature of its structure. This algorithm provides the outlines of how cycletree networks can be configured dynamically.

We then presented an inexpensive and simple router, Algorithm 2, for natural cycletrees and showed through Algorithm 3 how to obtain a shortest path communication harness in a natural cycletree network, thus providing optimal communication in cases when communication is not local.

We showed that Algorithm 3 is superfast for tree-complete cycletrees, and thus, in combination with dynamic configuration of cycletrees, obtaining a shortest path communication harness is usually (in the interesting cases) inexpensive.

Supported by the discussion in Chapter 1, we believe that cycletrees can be used in several areas of parallel computation to support, in the general case, efficient communication in parallel computations on nonshared memory machines, in particular when the exact communication patterns of the computation are not known.

Future work

The work can be pursued in several directions from here on. There are several open questions to be be investigated.

How well can cycletrees be embedded or mapped in various host graphs like hypercubes, meshes and butterflies, or VLSI? Based on the extensive work that has been done on embedding binary trees and binary tree based networks in other networks and VLSI, the answer to that question seems promising.

How well do cycletrees perform in practice, when used as (either virtual or actual) interconnection graphs in parallel computations where one-to-all, all-to-one and local communication patterns occur generally? Cycletrees could be studied in the context

of data parallel computations, e.g., in the context of bounded quantifications and Reform.

What are the potential bottlenecks in cycletrees for arbitrary communication patterns? To what cost, and how, can cycletrees be extended to "fat cycletrees" in order to remove those?

Which problems can be solved in parallel by making a direct use of the cycletree structure, and how should parallel algorithms be designed for such problems?

What other graph theoretical properties do cycletrees employ? Does the class of cycletrees include graphs that are not natural cycletrees but merit equal attention?

Bibliography

[1] Agrawal, D. P., Janakiram, V. K. and Pathak G. C., Evaluating the performance of multicomputer configurations, *IEEE Computer* 19,5 (1986).

[2] Almasi, G. S. and Gottlieb, A., *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, 1989.

[3] Annaratone, M. *et al.*, The Warp computer: architecture, implementation and performance, *IEEE Transactions on Computers* C-36,12 (1987).

[4] Arden, B. W. and Lee, H., Analysis of chordal ring network, *IEEE Transactions on Computers*, C-30:291–295 (1981).

[5] Arro, H., Barklund, J. and Bevemyr, J., Parallel bounded quantifications – preliminary results, UPMAIL Technical Report No. 74, Computing Science Department, Uppsala University, 1992.

[6] Barklund, J. and Millroth, H., Providing iteration and concurrency in logic programs through bounded quantifications, in: H. Tanaka (ed.), *Proc. Intl. Conf. on Fifth Generation Computer Systems*, Ohmsha, Tokyo, 1992.

[7] Beivide, R., Herrada, E., Balcázar, J. L. and Arruabarrena, A., Optimal distance networks of low degree for parallel computers, *IEEE Transactions on Computers* 40,10 (1991).

[8] Bentley, J. L. and Kung, H. T., A tree machine for searching problems, in: *Proc. Intl. Conf. on Parallel Processing*, 1979.

[9] Bentley, J. L. and Ottmann, T., On the power of one dimensional vectors of processors, in: H. Noltemeier (ed.), *Proc. Graph-Theoretic Concepts in Computer Science, LNCS 100*, Springer-Verlag, Berlin, 1980.

[10] Bevemyr, J., Lindgren, T., Millroth, H. and Tärnlund, S.-Å., Personal communication.

[11] Bhatt, S. N., Chung, F. R. K., Leighton, F. T. and Rosenberg, A. L., Optimal simulations of tree machines, in: *27th IEEE Symp. on Foundations of Computer Science*, 1986.

[12] Boillat, J. E. and Kropf P. G., A fast distributed mapping algorithm, in: H. Burkhart (ed.), *Proc. CONPAR 90 – VAPP IV, LNCS 457*, Springer-Verlag, Berlin, 1990.

[13] Bokhari, S. H., 1981, On the mapping problem, *IEEE Transactions on Computers*, C-30,3 (1981).

[14] Brandenburg, J. E. and Scott, D. E., Embedding of communication trees and grids into hypercubes, *Intel iPSC User Group*, 1 (1986).

[15] Carriero, N. and Gelernter, D., *How to Write Parallel Programs, A First Course*, MIT Press, Mass., 1990.

[16] Chandy, K. M. and Misra, J., *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, Mass., 1988.

[17] Chaudhuri, P., *Parallel Algorithms: Design and Analysis*, Prentice Hall, Sydney, 1992.

[18] Corneil, D. and Read, R. C., The graph isomorphism disease, *Journal of Graph Theory*, 1:339–363 (1977).

[19] Despain, A. M. and Patterson, D. A., X-tree: a tree structured multi-processor computer architecture, in: *Proc. IEEE 5th Annual Symposium on Computer Architecture*, 1978.

[20] Eklund, P., Hierarchical wiring in multigrids, in: *Proc. CONPAR 90 - VAPP IV, LNCS 457*, Springer-Verlag, Berlin, 1990.

[21] Freisleben, B. and Kielmann, T., Automatic parallelization of divide-and-conquer algorithms, in: *CONPAR 92 - VAPP V*, Springer-Verlag, Berlin, 1992.

[22] Frenkel, K. A., The Human Genome project and informatics, *Communications of the ACM*, 34,11 (1991).

[23] Goodman, J. R. and Sequin, C. H., Hypertree: a multiprocessor interconnection topology, Computer Science Technical Report 4227, Dept. Elec. Eng. and Comp. Sci., Univ. of California, Berkeley, 1981.

[24] Golumbic, M. C., *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[25] Gordon, D., Efficient embedding of binary trees in VLSI arrays, *IEEE Transactions on Computers*, C-36:1009–1018 (1987).

[26] Gotoh, O., An improved algorithm for matching biological sequences, *Journal of Molecular Biology*, 162:705–708 (1982).

[27] Goyal, P. and Narayanan, T. S., Dictionary machine with improved performance, *The Computer Journal*, 31,6 (1988).

[28] Hoare, C. A. R., Communicating sequential processes, *Communications of the ACM*, 21,8 (1978).

[29] Hoare, C. A. R., *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[30] Horowitz, E. and Zorat, A., The binary tree as an interconnection network: applications to multiprocessor systems and VLSI, *IEEE Transactions on Computers*, C-30:247–253 (1981).

[31] Hromkovitč, J., Müller, V., Sýkora, O. and Vrťo, I., On embedding interconnection networks into rings of processors, in: *Proc. PARLE 92*, Springer-Verlag, Berlin, 1992.

[32] Jones, G. and Goldsmith, M., *Programming in occam 2*, Prentice Hall, London, 1988.

[33] Knuth, D. E., *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, Second Edition, Addison-Wesley, Reading, Mass., 1973.

[34] Krishnamurthy, E. V., *Parallel Processing: Principles and Practice*, Addison-Wesley, Sydney 1989.

[35] Krämer, O. and Mühlenbein, H., Mapping strategies in message based multiprocessor systems, in: *Proc. PARLE 87, LNCS 258*, Springer-Verlag, Berlin, 1987.

[36] Kung, H. T., The structures of parallel algorithms, in: M. Yovits (ed.), *Advances in Computers 19*, Academic Press, New York, 1980.

[37] Kung, S. Y., *VLSI Array Processors*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[38] Kung, S. Y., VLSI array processor for signal processing, in: *MIT Conf. on Adva. Research on I. C.*, MIT Press, Mass., 1980.

[39] Kung, S. Y., Arun, K. S., Bhaskar Rao, D. V. and Hu, Y. H., A matrix data flow language/architecture for parallel matrix operations based on computational wavefront concept, in: H. T. Kung, B. Sproull and G. Steele (eds.), *VLSI Systems and Computations*, Springer-Verlag, Berlin, 1981.

[40] Lam, S. P. S., A novel sorting array processor, in: *Proc. CONPAR 92 - VAPP V, LNCS 634*, Springer-Verlag, Berlin, 1992.

[41] Latifi, S. and El-Amawy, A., Efficient approach to embed binary trees in 3-D rectangular arrays, *IEEE Processors*, 137,2 (1990).

[42] Leiserson, C. E., Systolic priority queues, Report CMU-CS-79-115, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1979. (Also in: *Proc. Caltech VLSI Conf.*, 1979)

[43] Leiserson, C. E., Fat-Trees: universal networks for hardware-efficient supercomputing, *IEEE Transactions on Computers*, C-34,10 (1985).

[44] Lengauer, C., Towards systolizing compilation: an overview, in: *Proc. PARLE 89, LNCS 366*, Springer-Verlag, Berlin, 1989.

[45] Li, P. P. and Martin, A. J., The sneptree – a versatile interconnection network, in: K. Hwang, S. M. Jakobs and E. E. Swartzlander (eds.), *Proc. IEEE Intl. Conf. on Parallel Processing*, 1986.

[46] Mead, C. and Conway, L., *Introduction to VLSI systems*, Addison-Wesley, Reading, Mass., 1980.

[47] Millroth, H., Reforming Compilation of Logic Programs, Ph.D. Thesis, Computing Science Department, Uppsala University, 1990.

[48] Millroth, H., Reforming compilation of logic programs, in: *International Logic Programming Symposium*, MIT Press, Cambridge, Mass., 1991.

[49] Monien, B. and Sudborough, I. H., Simulating binary trees on hypercubes, in: J. H. Reif (ed.), *Proc. AWOC 88 VLSI Algorithms and Architectures, LNCS 319*, Springer-Verlag, Berlin, 1988.

[50] Mou, Z. G., A Formal Model for Divide-and-Conquer and its Parallel Realization, Ph.D. thesis, Research Report YALEU/DCS/RR–795, Dept. of Comp. Sci., Yale Univ., 1990.

[51] Nelson, P. A. and Snyder, L., Programming paradigms for nonshared memory parallel computers, in: L. H. Jamieson, D. B. Gannon and R. J. Douglass (eds.), *The Characteristics of Parallel Algorithms*, MIT Press, Cambridge, Mass., 1987.

[52] Ottman, T. A., Rosenberg, A. L. and Stockmeyer, L. J., A dictionary machine for VLSI, *IEEE Transactions on Computers*, C-31,9 (1982).

[53] Reed, D. A. and Grunwald, D. C., The performance of multicomputer interconnection networks, *IEEE Computer*, 20,6 (1987).

[54] Reed, D. A. and Schwetman, H. D., Cost-performance bounds for multimicrocomputer networks, *IEEE Transactions on Computers*, C-32,1 (1983).

[55] Rosenberg, A. L., Graph embeddings 1988: recent breakthroughs, new directions, in: J. H. Reif (ed.), *Proc. AWOC 88 VLSI Algorithms and Architectures, LNCS 319*, Springer-Verlag, Berlin, 1988.

[56] Ruzzo, W. L. and Snyder, L., Minimum edge length planar embeddings of trees, in: H. T. Kung, B. Sproull and G. Steele (eds.), *VLSI Systems and Computations*, Springer-Verlag, Berlin, 1981.

[57] Saad, Y. and Schultz, M. H., Topological properties of hypercubes, Research Report RR-389, Yale University, 1985.

[58] Samatham, M. R. and Pradhan, D. K., The de Bruijn multiprocessor network: a versatile parallel processing and sorting network for VLSI, *IEEE Transactions on Computers*, 38,4 (1989).

[59] Shapiro, E., Systolic programming: a paradigm of parallel processing, in: E. Shapiro (ed.), *Concurrent Prolog, Collected Papers Vol. 1*, MIT Press, Cambridge, Mass., 1987.

[60] Shen, H., 1991, Efficient Design and Implementation of Parallel Algorithms, Ph.D. Thesis, Dept. of Comp. Sci., Åbo Akademi, 1991.

[61] Shen, H., Self-adjusting mapping: a heuristic mapping algorithm for mapping parallel programs on to transputer networks, *The Computer Journal*, 35,1 (1992).

[62] Snyder, L., Introduction to the configurable, highly parallel computer, *IEEE Computer*, 15,2 (1982).

[63] Stolfo, S. J., Initial performance of the DADO2 prototype, *IEEE Computer*, 20,1 (1987).

[64] Tucker, L. W. and Robertson, G. G., Architecture and applications of the Connection Machine, *IEEE Computer*, 21,8 (1988).

[65] Tärnlund, S.-Å., Reform, draft, Computing Science Department, Uppsala University, 1991.

[66] Ullman, J. D., *Computational Aspects of VLSI*, Computer Science Press, Rockville, Md., 1984.

[67] Valiant, L. G., Universality considerations for VLSI circuits, *IEEE Transactions on Computers*, C-30:135–140 (1981).

[68] Wittie, L. D., Communication structures for large networks of microcomputers, *IEEE Transactions on Computers*, C-30:264–273 (1981).

[69] Wolfe, M., New program restructuring technology, in: *Proc. Parallel Computation, First Intl. ACPC Conf., LNCS 591*, Springer-Verlag, Berlin, 1991.

[70] Xie, X. and Ge, Y., An optimal structure that accomodates both a ring and a binary tree, in: A. Bode (ed.), *Proc. Distributed Memory Computing, LNCS 487*, Springer-Verlag, Berlin, 1991.

[71] Youn, H. Y. and Singh, A. D., On implementing large binary tree architectures in VLSI and WSI, *IEEE Transactions on Computers*, 38,4 (1989).

[72] Zienicke, P., Embeddings of treelike graphs into 2-dimensional meshes, in: Möhring (ed.), *Proc. WG'90 Graph-Theoretic Concepts in Computer Science, LNCS 484*, Springer-Verlag, Berlin, 1990.

Appendix A

Proofs

A.1 Number of edges in an optimal cycletree

Let $G[C,T]$ be an optimal cycletree, let $N = |V_G|$. As T is a tree-complete basic binary tree, T is full up to level $K-1$, i.e., T has 2^l vertices at level l, $0 \leq l < K$, and R number of internal vertices at level $K-1$, where

$$\begin{aligned} K &= \lfloor \log_2(N+1) \rfloor, \\ 2R &= N + 1 - 2^K. \end{aligned}$$

Let us by I_m denote the number of noncycle edges at a full level[1] m of a cycletree, and by J_m that of a chaintree. Then

$$I_m = 2J_{m-1}, \quad m > 0,$$

and, by using Definition 2, we obtain the following linear recurrence equation for J_m:

$$\begin{aligned} J_0 &= 0, \\ J_1 &= 1, \\ J_m &= J_{m-1} + 2J_{m-2}, \quad m > 1. \end{aligned}$$

By using standard techniques we get the following solution:

$$J_m = \frac{2^m - (-1)^m}{3} = \left\lfloor \frac{2^m + 1}{3} \right\rfloor.$$

Let R' be the number of noncycle edges at level K. We know that R' must be minimal, since G is optimal and the number of noncycle edges at levels m, $1 \leq m < K$, are fixed by I_m. There are R internal vertices at level $K-1$ and for each of these vertices exactly one edge is a noncycle edge (assume that $K > 1$). Thus $R - R'$ of these edges are at level $K-1$. We get the following formula for R', since $R - R'$ can be at most I_{K-1},

$$R' = \begin{cases} R - I_{K-1}, & \text{if } R - I_{K-1} \geq 0; \\ 0, & \text{otherwise.} \end{cases}$$

We know that $R = (N+1)/2 - 2^{K-1}$ and $I_{K-1} = 2J_{K-2}$, thus

$$\begin{aligned} R - I_{K-1} &= \frac{N+1}{2} - 2^{K-1} - 2\frac{2^{K-2} - (-1)^{K-2}}{3} \\ &= \frac{N+1}{2} - 2\frac{2^K - (-1)^K}{3} \\ &= \frac{N+1}{2} - I_{K+1}. \end{aligned}$$

[1] The level of an edge (r,s), where r is the father of s, is the level of s.

Thus the following is an equivalent formula for R':

$$R' = \begin{cases} (N+1)/2 - I_{K+1}, & \text{if } N \geq 2I_{K+1} - 1; \\ 0, & \text{otherwise.} \end{cases}$$

Now, the total number of edges of G is the number of cycle edges, which is N, plus the number of noncycle edges at all levels. Thus

$$\begin{aligned}
|E_G| &= N + \sum_{m=1}^{K-1} I_m + R' \\
&= N + 2\sum_{m=0}^{K-2} J_m + R' \\
&= N + 2\sum_{m=0}^{K-2} \frac{2^m - (-1)^m}{3} + R' \\
&= N + \frac{2}{3}\left(\sum_{m=0}^{K-2} 2^m - \sum_{m=0}^{K-2} (-1)^m\right) + R' \\
&= N + \frac{2}{3}\left(2^{K-1} - 1 - \frac{(-1)^{K-1} - 1}{-2}\right) + R' \\
&= N + \frac{2^K - (-1)^K}{3} - 1 + R' \\
&= N + J_K - 1 + R' \\
&= \begin{cases} N + J_K - 1 + (N+1)/2 - 2J_K, & \text{if } N \geq 4J_K - 1; \\ N + J_K - 1, & \text{otherwise.} \end{cases} \\
&= \begin{cases} (3N-1)/2 - J_K, & \text{if } N \geq 4J_K - 1; \\ N - 1 + J_K, & \text{otherwise.} \end{cases} \\
&= \begin{cases} (3N-1)/2 - \lfloor (2^K+1)/3 \rfloor, & \text{if } N \geq 4\lfloor (2^K+1)/3 \rfloor - 1; \\ N - 1 + \lfloor (2^K+1)/3 \rfloor, & \text{otherwise.} \end{cases}
\end{aligned}$$

A.2 Optimality of router data

Let G be a cycletree with no cross-level edges, and let a be any vertex of G. We will prove that the router data as given by formulas 3.3 and 3.4 in the internal case, and 3.9 and 3.10 in the external case, are optimal. (Proof of the optimality of formulas 3.5, 3.6, 3.11 and 3.12 is analogous, due to symmetry of G, just switch pre- and post vertices). We must prove that for any vertex x of G such that $x \neq a$,

$$|l(a) \xrightarrow{s} x| \leq |f(a) \xrightarrow{s} x|$$

when $\text{lmin}(a) \leq x \leq \text{lmax}(a)$, and

$$|l(a) \xrightarrow{s} x| \geq |f(a) \xrightarrow{s} x|$$

when $x < \text{lmin}(a)$.

Internal vertices

In the following we will assume that a is an internal vertex of G and x is any other vertex of G.

Pre-vertex (or root)

When a is a pre-vertex (or the root) then a is on the right side of no contour. In that case $\circ a = l(a)$, by definition. As a is a pre-vertex (or the root), so is $l(a)$. The vertices between $\text{lmin}(a)$ and $\text{lmax}(a)$ are all the descendents of $l(a)$ and are trivially closer to a via $l(a)$. Any other vertex less than $l(a)$, other than a itself, must be closer to a via $f(a)$.

In- or Post-vertex

Assume now that a is an in- or a post-vertex. In that case a belongs to the right side of the contour D of a region R. Let t be the top of D. See Figure A.1. Let S be the subgraph of G induced by

$$\{x \mid \text{lmin}(a) \leq x \leq \text{lmax}(a)\} = \text{desc}(\circ a) \cup \text{desc}(l(a)).$$

The subgraph S is shown as S_1 and S_2 in the figure.

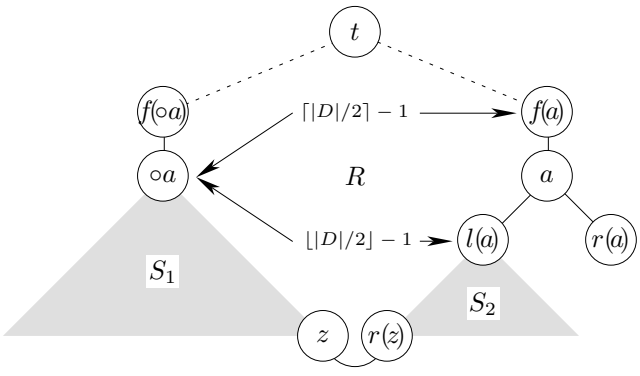


Figure A.1: *Vertex a is internal.*

We know that $\circ a$ is always on the left side of D, since a is internal. Let z be the rightmost son of $\circ a$. This condition holds because cross-level edges are not allowed, i.e., whenever $|D|$ is odd, z and $r(z)$ must be at the same level, and whenever $|D|$ is even, the levels must differ by one.

Assume x is in S. We prove that $|f(a) \xrightarrow{s} x| \geq |l(a) \xrightarrow{s} x|$. Unless $l(a)$ is visited, in which case the proof is trivial, we know, due to Theorem 6, that any shortest path from $f(a)$ to x must visit t by following the contour. As $\circ a$ is a descendent of t and x is a descendent of $\circ a$, a shortest path from t to x goes through $\circ a$. We have that

$$|f(a) \xrightarrow{s} x| = |f(a) \xrightarrow{s} \circ a \xrightarrow{s} x|.$$

According to Definition 3.1 $|l(a) \xrightarrow{s} \circ a| \leq |f(a) \xrightarrow{s} \circ a|$. Thus

$$|f(a) \xrightarrow{s} x| \geq |l(a) \xrightarrow{s} \circ a \xrightarrow{s} x| \geq |l(a) \xrightarrow{s} x|.$$

Assume $x < \mathrm{lmin}(a)$. We prove that $|l(a) \xrightarrow{s} x| \geq |f(a) \xrightarrow{s} x|$. Unless $f(a)$ is visited, in which case the proof is trivial, any shortest path from $l(a)$ to x must visit z by following the contour through the nontree edge $(z, r(z))$. Furthermore, any shortest path from z to x must visit $\circ a$. This follows from that $\circ a$ is an in-vertex or a pre-vertex, in both cases a shortest path from any right descendent of $\circ a$ to x must visit $\circ a$. We get that

$$|l(a) \xrightarrow{s} x| = |l(a) \xrightarrow{s} \circ a \xrightarrow{s} x|.$$

Let us consider two cases now, assuming that $|D|$ is either odd or even. If $|D|$ is odd then $f(\circ a)$ must be on the left side of D, with $\circ a$ as its right son. As x is not a descendent of $\circ a$, we know that $|f(\circ a) \xrightarrow{s} x| \leq |\circ a \xrightarrow{s} x|$. We know also that $|f(a) \xrightarrow{s} f(\circ a)| = |l(a) \xrightarrow{s} \circ a|$, since $|D|$ is odd. Thus

$$\begin{aligned} |l(a) \xrightarrow{s} x| &\geq |l(a) \xrightarrow{s} \circ a| + |f(\circ a) \xrightarrow{s} x| \\ &= |f(a) \xrightarrow{s} f(\circ a)| + |f(\circ a) \xrightarrow{s} x| \\ &\geq |f(a) \xrightarrow{s} x|. \end{aligned}$$

If $|D|$ is even then $|f(a) \xrightarrow{s} \circ a| = |l(a) \xrightarrow{s} \circ a|$, and thus

$$|l(a) \xrightarrow{s} x| = |f(a) \xrightarrow{s} \circ a \xrightarrow{s} x| \geq |f(a) \xrightarrow{s} x|.$$

External vertices

We will in the following assume that a is an external vertex of G. We have three cases to consider, according to Formula 3.9. We can exclude the case when a is a pre-vertex because then $l(a) = f(a)$ and the proof is trivial. Thus, we assume that a is either a post- or an in-vertex. In either case a must be on the right side of the contour D of a region R. Let t be the top of D.

Case 1: a is a descendent of $l(a)$.

In that case $\mathrm{lmin}(a) = 1$, i.e., all the vertices less than a are closer to a through $l(a)$. This situation is illustrated by Figure A.2.

We know that $|D|$ must be even, $k = |D|/2 - 1$ in the figure. The case is really trivial, as any vertex x, $x < a$, must be to the left of a. Thus in the worst case a shortest path from $l(a)$ to x visits t, and is as long as from $f(a)$ to x, e.g., when $x = 1$.

Case 2: $l(a)$ is a descendent of a.

In this case $\mathrm{lmin}(a) = \min(\mathrm{desc}(\circ a))$. See Figure A.3. Also in that case must $|D|$ be even, since cross-level edges are disallowed. We have that $\circ a$ is the left son of t, $k = |D|/2 - 1$ in the figure.

The subgraph induced by $\mathrm{desc}(\circ a)$ is shown as S. Clearly, if x is in S then a shortest path from $f(a)$ to x, unless visiting $l(a)$, must visit t and $\circ a$. We get that

$$|f(a) \xrightarrow{s} x| = k + |\circ a \xrightarrow{s} x| = |l(a) \xrightarrow{s} \circ a \xrightarrow{s} x| \geq |l(a) \xrightarrow{s} x|.$$

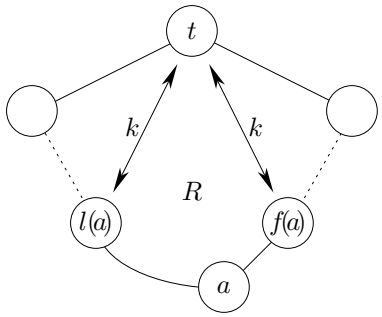


Figure A.2: *Vertex a is a leaf and a is a descendent of $l(a)$.*

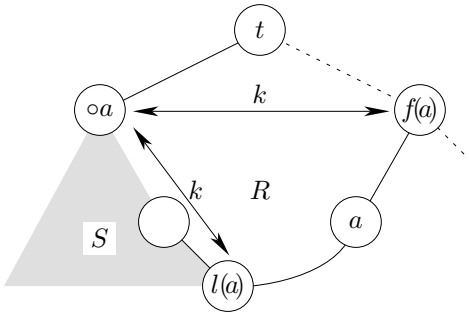


Figure A.3: *Vertex a is a leaf and $l(a)$ is a descendent of a.*

Now, if $x < \mathrm{lmin}(a)$ then any shortest path from $l(a)$ to x, unless visiting $f(a)$, must visit $\circ a$, and thus

$$|l(a) \xrightarrow{\mathrm{s}} x| = k + |\circ a \xrightarrow{\mathrm{s}} x| = |f(a) \xrightarrow{\mathrm{s}} \circ a \xrightarrow{\mathrm{s}} x| \geq |f(a) \xrightarrow{\mathrm{s}} x|.$$

Case 3: a and $l(a)$ are at the same level.

Let z be the minimum vertex such that $l(a) = \max(\mathrm{desc}(z))$. Then $\mathrm{lmin}(a) = \min(\mathrm{desc}(*z))$, according to Formula 3.9. If z is not a pre-vertex then it is on the right side of the contour D' of a region R', as illustrated by Figure A.4 where $k' = \lfloor(|D'| - 1)/2\rfloor$.

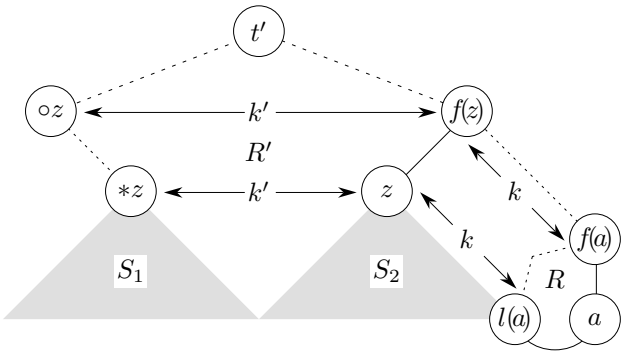


Figure A.4: *Vertex a is a leaf at the same level as $l(a)$ and z is the minimum vertex such that $l(a) = \max(\mathrm{desc}(z))$.*

The subgraph S induced by

$$\{x \mid \mathrm{lmin}(a) \leq x < a\} = \mathrm{desc}(*z) \cup \mathrm{desc}(z)$$

is shown as S_1 and S_2 in the figure. Note that if z is a pre-vertex then $*z = z$ and $S_1 = S_2$. It is illustrated by Figure A.4 that $|l(a) \xrightarrow{s} z| = |f(a) \xrightarrow{s} f(z)| = k$. Actually, $f(a)$ is a descendent of $f(z)$ and k is the level difference between those vertices, as it is between z and $l(a)$. Furthermore, a shortest path between z and $f(a)$ has length $k+1$. It is easy to calculate that if a is the left son of $f(a)$ then $z = \circ a$ and the above holds. If a is the right son of $f(a)$ then $z = \circ a = l(a)$ only if z is either a pre-vertex or $l(l(a))$ is not an ascendent of $l(a)$, otherwise the case must be one of those illustrated by Figure A.5. It is easy to see that the above reasoning holds also in those cases.

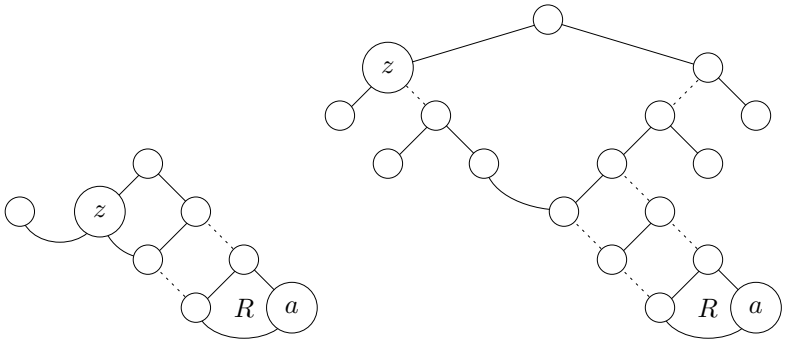


Figure A.5: *Two possible cases of z (either internal or external) if $z \neq \circ a$.*

Assume that x is in S. Let us first assume that x is in S_2. We know, based on the above reasoning, that

$$|f(a) \xrightarrow{s} x| \geq 1 + |l(a) \xrightarrow{s} x|.$$

Now, assume that z is not a pre-vertex and x is in S_1, otherwise $S_1 = S_2$ and we are done. A shortest path from $f(a)$ to x, unless visiting z (in which case a shortest path from $l(a)$ to z is clearly shorter and we are done), goes through t', $\circ z$ and $*z$. Thus, see Figure A.4,

$$\begin{aligned} |f(a) \xrightarrow{s} x| &= |f(a) \xrightarrow{s} f(z) \xrightarrow{s} \circ z \xrightarrow{s} *z \xrightarrow{s} x| \\ &= k + k' + |\circ z \xrightarrow{s} *z \xrightarrow{s} x| \\ &= |l(a) \xrightarrow{s} z \xrightarrow{s} *z| + |\circ z \xrightarrow{s} *z \xrightarrow{s} x| \\ &\geq |l(a) \xrightarrow{s} x|. \end{aligned}$$

Assume that $x < \mathrm{lmin}(a)$. If z is a pre-vertex then $\mathrm{lmin}(a) = z$ and any vertex less than z is clearly closest to a via $f(a)$. Assume the opposite. Then a shortest path from $l(a)$ to x, unless visiting $f(a)$ and $f(z)$, goes through z and $*z$. Thus

$$\begin{aligned} |l(a) \xrightarrow{s} x| &= |l(a) \xrightarrow{s} z \xrightarrow{s} *z \xrightarrow{s} x| \\ &= k + k' + |*z \xrightarrow{s} x| \\ &= |f(a) \xrightarrow{s} f(z) \xrightarrow{s} \circ z| + |*z \xrightarrow{s} x|. \end{aligned}$$

Let us consider two cases now: either $|D'|$ is odd or even. If $|D'|$ is odd then $*z = \circ z$ and trivially

$$|l(a) \xrightarrow{s} x| = |f(a) \xrightarrow{s} f(z) \xrightarrow{s} \circ z \xrightarrow{s} x| \geq |f(a) \xrightarrow{s} x|.$$

If $|D'|$ is even, then $\circ z$ must be on the left side of D' with $*z$ as its right son. As x is not a descendent of $*z$, we know that $|\circ z \xrightarrow{s} x| \leq |*z \xrightarrow{s} x|$, and thus

$$|l(a) \xrightarrow{s} x| \geq |f(a) \xrightarrow{s} f(z) \xrightarrow{s} \circ z| + |\circ z \xrightarrow{s} x| \geq |f(a) \xrightarrow{s} x|,$$

which completes the proof.