Louisiana State University
LSU Digital Commons

LSU Doctoral Dissertations | Graduate School

2005

Broadcast in sparse conversion optical networks using fewest converters

Tong Yi
Louisiana State University and Agricultural and Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_dissertations

Part of the Computer Sciences Commons

Recommended Citation

Yi, Tong, "Broadcast in sparse conversion optical networks using fewest converters" (2005). *LSU Doctoral Dissertations*. 2591.
https://digitalcommons.lsu.edu/gradschool_dissertations/2591

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Digital Commons. For more information, please contactgradetd@lsu.edu.

BROADCAST IN SPARSE CONVERSION OPTICAL NETWORKS USING FEWEST CONVERTERS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Tong Yi

B.S. Huazhong University of Science and Technology, 1993
M.S. Huazhong University of Science and Technology, 1998

December 2005

Acknowledgements

It is a pleasure to thank my advisor, Prof. Sukhamay Kundu, for his guidance, encouragement, and patience. He clarified the concepts, algorithms, and writing. I admire for his devotion to research and teaching.

I thank my committee members. Prof. Bert Boyce served as my minor advisor and gave feedback quickly. Prof. Jianhua Chen encouraged me constantly and helped me in writing when Prof. Kundu was on Sabbatical. Prof. Donald Kraft is helpful as the graduate director of the Department of Computer Science. Prof. Stephen Shipman served as the Dean's representative in the final defense and simplified an example.

Special thanks for Prof. Guoli Ding and Prof. Bogdan Oporowski for their crucial help in mathematical proofs and their generousness in their time.

I am in debt to Prof. Charles Grenier and Prof. Harlon Traylor for spending so many painstaking hours in reading and revising the drafts of my dissertation. They raised questions, encouraged, and complemented me, without which the dissertation could not be finished in time.

My thanks go to Prof. Ahmed A. El-Amawy and Stefan Pascu for the discussions of optical networks.

I am grateful to Prof. James Oxley for instructing me how to teach and write mathematics. I appreciate for his meticulous supervision of my project in the course of communicating mathematics.

Many teachers helped me in many instances. In particular, I must thank Profs Leonard Richardson, William Adkins, Daniel Cohen, Charles Delzell, Robert Lax,

Robert Perlis, Jerome Hoffman, Ambar Sengupta, Richard Litherland, Paul van Wamelen, and Peter Wolenski. My thank also goes to Dean Guillermo Ferreyra.

I thank Mrs. Phoebe Rouse for the helpful training on teaching undergraduate courses.

I appreciate Dr. Norma Travis and Ms. Jeri Hebert for their patient help in rectifying my English pronunciation.

I thank Mr. Guojun Jin and Dr. Deborah Agarwal for their help when I was working for an internship in the Lawrence Berkeley National Laboratory of Berkeley in the summer of 2001. Dr. Agarwal's stress on the importance of presentation motivated me to take the course of communicating mathematics.

The computer system managers and staffs of the Department of Computer Science and of the Department of Mathematics are very helpful. In particular, I thank Lynette Jackson, Vera Watkins, Wan Lee, Elias Khalaf, Jeff Sheldon, and Zed Pobre.

During these years, I am very lucky to get incredible help from many people. In particular, I am grateful for Prof. Harlon Traylor and Doris Traylor for the caring as a local host family. I thank Lucille Parsons for her generous accommodations and for waiting late for me every night, she is like a grandma to me. I appreciate Prof. Charles Grenier, Virginia Grenier, and Cheryl Grenier for the unexpected help in so many instances.

My thanks go to my friends: Jeremy Aikin, Maiia Bakhova, Wei Cai, Liqun Fang, Jian Guan, Yixin Luo, Chao Meng, Goderdzi Pruidze, Hairui Tu, Jie Wu, Hong Yin, Wudong Luo, Faming Zhu, Huifang Zhang, Benhou Yan, Xiaotuan Liu, Zhihua Xiao, and Min Zhan.

My thanks also go to my teachers in China: Prof. Zhengding Lu, Prof. Yansheng Lu, Prof. Liping Pang, Yanan Zhang, Liucun Yu, and Ping Li.

I thank the Department of Computer Science and the Department of Mathematics at Louisiana State University for the generous financial aids. This work was partially supported by NSA grant H98230-05-1-0081.

Finally, I must thank my grandmother Xiuzhen Li and my late grandfather Deji Su for their loves and caring. I am forever in debt to my mom Huizhen Su, dad Ruhao Yi, and brother Hai Yi for their continuous love, support, and encouragement.

Table of Contents

Acknowledgements ii

List of Tables vii

List of Figures viii

List of Symbols x

Abstract xii

1 Introduction **1**
- 1.1 Sparse Conversion WDM Optical Networks 1
- 1.2 The Converter Usage Problem (CUP) 1
- 1.3 The CUP in Tree Networks 2
- 1.4 The CUP in Networks of Bounded Treewidth 3
  - 1.4.1 Graphs of Bounded Treewidth 3
  - 1.4.2 Tree Decompositions of Some Common Graphs 5
  - 1.4.3 A Property of Tree Decomposition 8
  - 1.4.4 The CUP in Networks of Bounded Treewidth 9
- 1.5 Related Works 10
- 1.6 Overview of the Dissertation 11

2 Broadcast in Optical Tree Networks Using Fewest Converters **12**
- 2.1 Problem Description 12
- 2.2 Theorems 13
- 2.3 An Algorithm 24
  - 2.3.1 A Pseudocode 24
  - 2.3.2 An Example 28
  - 2.3.3 Complexity Analysis 30

3 Find a Minimum Wavelength-dominating Set in Optical Networks with Bounded Treewidth **32**
- 3.1 Problem Description 32
- 3.2 General Approach for Solving the MWDSP 33
- 3.3 Notations and Definitions 37
- 3.4 Theorems 41
- 3.5 An Algorithm 59
  - 3.5.1 An Example 59
  - 3.5.2 Complexity Analysis 60

3.5.3 Experiment Results 64

4 Summary and Future Work 69
4.1 Summary 69
4.2 Future Work 70

References 72

Appendix: Code of btw.c 78

Vita 127

List of Tables

2.1 Some concepts of rooted trees 15

2.2 Some variables used in Pseudocode CONVERTER-USAGE 24

2.3 Calculation of $P_\lambda(v)$ in a bottom-up fashion 28

3.1 Calculation of $f(2, 0, \tau, \Lambda_p, \Lambda_r)$. 60

3.2 Calculation of $f(3, 0, \tau, \Lambda_p, \Lambda_r)$. 61

3.3 Calculation of $f(1, 0, \tau, \Lambda_p, \Lambda_r)$. 61

3.4 Calculation of $f(1, 1, \tau, \Lambda_p, \Lambda_r)$. 62

3.5 Calculation of $f(1, 2, \tau, \Lambda_p, \Lambda_r)$. 63

List of Figures

1.1 A graph of bounded treewidth 2 and some of its tree decompositions. 4

1.2 A graph of bounded treewidth 2 and two of its tree decompositions. 5

1.3 A tree decomposition of a tree. 6

1.4 A tree decomposition of a circle. 7

1.5 Tree decompositions of $n \times n$ grids when $n = 1, 2$. 7

1.6 A tree decomposition of a graph contains its vertex-cut information. 8

2.1 An illustration of the dynamic formula for NC-nodes. 17

2.2 An illustration of the dynamic formula for C-nodes. 22

2.3 Calculation of $P_\lambda(v)$ for non-root v in a bottom-up fashion. 27

2.4 A flow chart to calculate $P_\lambda(v)$. 27

2.5 Assign wavelengths for the network in Table 2.3. 30

3.1 The dominating set problem in an optical network N. 34

3.2 General approach to solve the MWDSP 35

3.3 Wavelength information. 36

3.4 An illustration of t_i and $c(t)$. 38

3.5 An illustration of $T(t, d)$, $\overline{B(t, d)}$, and $B(t, d)$. 39

3.6 Calculation of $f(t, d, \tau, \Lambda_p, \Lambda_r)$ when $d = 0$. 43

3.7 An illustration of $\tau \subseteq X_t$, $\tau' \subseteq X_{t_d}$, and $\tau \cap \tau' = \tau \cap X_{t_d} = \tau' \cap X_t$. 45

3.8 An illustration of Theorem 3.4. 45

3.9 Relationships among variables. 47

3.10 An illustration of $\Lambda_p = W_\tau \cup (W_S \cap W_{X_t})$. 48

3.11 An illustration of Λ'_p, Λ'_r, Λ''_p, and Λ''_r. 49

3.12 An illustration of $\Lambda_r = (\Lambda'_r \cup \Lambda''_r) - \Lambda_p$. 50

3.13 When $\Lambda_p = \varnothing$, a solution is feasible if and only if $|\Lambda_r| = 1$. 60

3.14 An example of NSF-14 network. 68

4.1 A hypergraph G and one of its branch-decompositions T. 70

List of Symbols

(u, v): the link between nodes u and v, page 12

$\lambda_{\text{best}}(v)$: a wavelength corresponding to the usage of a minimum number of converters in T_v for broadcasting in T_v, page 24

$\lambda_{\text{select}}(\text{parent}(v), v)$: the wavelength assigned for link $(\text{parent}(v), v)$ in an optimal wavelength assignment, page 24

$\overline{B(t,\, d)}$: defined as $\bigcup\{X_i : i \in V(T(t,\, d))\}$, page 38

$\text{parent}(v)$: the parent of node v in a rooted tree, page 13

$\text{useConv}(\lambda, v)$: Given v and λ, where v is a C-node and λ is a wavelength assigned for link $(\text{parent}(v), v)$, to broadcast in T_v using a minimum number of C-nodes in T_v, whether we shall use the converter at v or not, page 24

$B(t,\, d)$: defined as $\overline{B(t,\, d)} - X_t$, page 38

C: a set of C-nodes, page 12

C-node: a node that has a converter, page 12

$c(t)$: the number of children of node t in a rooted tree, page 37

E: the set of links, page 12

NC-node: a node that does not have a converter, page 12

$P_\lambda(v)$: the minimum number of C-nodes that are put in use in T_v for broadcasting in T_v, when messages are sent to v by $\text{parent}(v)$ in wavelength λ, page 14

$P_\lambda^{+0}(v)$: the minimum number of C-nodes in T_v needed to be used for broadcasting in T_v, when the wavelength in link $(\text{parent}(v), v)$ is λ and there is no converter at v. Applicable only when v is a C-node, page 20

$P_\lambda^{+1}(v)$: the minimum number of C-nodes in T_v needed to be used for broadcasting in T_v, when the wavelength in link $(\text{parent}(v), v)$ is λ and there is a converter at v. Applicable only when v is a C-node, page 20

$T = (V, C, E, W)$: an optical tree network, page 12

$T(t,\, d)$: for $t \in V(T)$ and $0 \leq d \leq c(t)$, if $d = 0$, then define $T(t,\, d)$ to be node t itself; otherwise, $T(t,\, d)$ is the induced subtree of T on t, the first d children of t, and those children's descendants, page 38

T_v: the induced subtree of T at v and all its descendants, page 13

V: the set of nodes, page 12

$W(u, v)$: the set of available wavelengths in link (u, v), page 12

W_U: defined as $\bigcup_{u \in U} W_u$, where U is a subset of nodes, page 37

W_u: the set of wavelengths on all the links incident at node u, page 37

X_t: a subset of graph nodes corresponding to node t of a tree decomposition of the graph, page 3

convSet: a minimum size subset of C-nodes used for broadcasting in the network, page 24

CUP: acronym of *Converter Usage Problem*, which aims to use a minimum number of C-nodes so that each node can send messages to all the others, page 2

minNumConv: the minimum number of C-nodes used for broadcasting in the network, page 24

○ : an NC-node, that is, a node *not* eligible to have a converter, page 12

◎ : an NC-node serving as the root, page 13

□ : a C-node, that is, a node eligible to have a converter, page 12

▨ : a used C-node, that is, a C-node whose converter is put in use, page 12

Abstract

Wavelengths and converters are shared by communication requests in optical networks. When a message goes through a node without a converter, the outgoing wavelength must be the same as the incoming one. This constraint can be removed if the node uses a converter. Hence, the usage of converters increases the utilization of wavelengths and allows more communication requests to succeed. Since converters are expensive, we consider sparse conversion networks, where only some specified nodes have converters. Moreover, since the usage of converters induces delays, we should minimize the use of available converters.

The *Converters Usage Problem* (CUP) is to use a minimum number of converter so that each node can send messages to all the others (broadcasting). In this dissertation, we study the CUP in sparse conversion networks.

We design a linear algorithm to find a wavelength assignment in tree networks such that, with the usage of a minimum number of available converters, every node can send messages to all the others. This is a generalization of [35], where each node has a converter. Our algorithm can assign wavelengths efficiently and effectively for one-to-one, multicast, and broadcast communication requests.

A converter wavelength-dominates a node if there is a uniform wavelength path between them. The *Minimal Wavelength Dominating Set Problem* (MWDSP) is to locate a minimum number of converters so that all the other nodes in the network are wavelength-dominated. We use a linear complexity dynamic programming algorithm to solve the MWDSP for networks with bounded treewidth. One such solution provides a low bound for the optimal solution to the CUP.

Chapter 1
Introduction

1.1 Sparse Conversion WDM Optical Networks

Bandwidth-demanding applications, like e-commerce and multimedia conferencing, are growing rapidly. With the employment of *wavelength division multiplexing* (WDM) technology, which enables multiple users to simultaneously send signals in distinct wavelengths of a fiber link and thus increases bandwidth, optical networks have become the preeminent choice for meeting this ever-expanding bandwidth requirement.

To communicate in a WDM optical network, one sets up a fiber path between a source and a destination and assigns a wavelength for each link in the path. For any intermediate node v in the path, if there is no wavelength converter placed at v, then the outgoing wavelength must be the same as the incoming one. On the other hand, with a converter placed at v, we may use any available wavelength in the outgoing link of node v. [1]

Hence, the use of converters enhances wavelength (link) utilization; however, it also induces high costs or long delays. To resolve this dilemma, we restrict the converters to some specified locations. That is, only the specified nodes in a network have converters. This is referred to as *sparse conversion WDM optical network*.

1.2 The Converter Usage Problem (CUP)

An increasingly popular request is *broadcast* (also called one-to-all), where a source node sends messages to all the other nodes in the network. In particular, we are interested in a special case of broadcast which enables any node to send

[1] We assume that a converter can transform an incoming wavelength to any outgoing one.

messages to all the others, as in the cases of multimedia conferencing, multi-player computer games, and remote teaching.

The *converter usage problem* (CUP) that we consider in this dissertation is to use a minimum number of converters so that each node can communicate with all the others in a sparse conversion network. In particular, we study the CUP in tree networks and networks of bounded treewidth.

1.3 The CUP in Tree Networks

To broadcast in a tree network, it is necessary and sufficient to place a converter at each node whose incident links are not assigned the same wavelength. Hence, given a wavelength assignment, the locations of converters are determined. Thus, the goal of the CUP is to assign a wavelength for each link such that a minimum number of converters are needed for broadcasting. We solve the above problem in $O(|W| \cdot |V|)$, where $|W|$ is the maximum number of wavelengths in each link and $|V|$ is the number of nodes in the network. Note that $|W|$ is bounded, so the algorithm is linear with respect to the number of nodes.

The algorithm proposed in this dissertation can be used to assign wavelength efficiently and effectively for one-to-one communication requests after the selection of routes. The algorithm also works for multicast (also called one-to-many) requests, where a source sends messages to multiple nodes simultaneously. An approach to implement multicast is to build a tree to connect the source and its destinations, as proposed in [55], then to assign a wavelength for each link in the tree using our algorithm.

An important property of a tree is, after the removal of an arbitrary non-terminal node (one with more than a neighbor), the tree is broken into two parts, each of which is a connected subtree. A similar property holds for a tree decomposition T of a graph G: after the removal from G the subset of graph nodes specified

in an arbitrary non-terminal tree node in T, the graph is broken into two or more components (connected subgraphs). For details, see Example 1.1 in Section 1.4.3.

Hence, it is natural to generalize the CUP from tree networks to networks of bounded treewidth k, which will be discussed in the following section.

1.4 The CUP in Networks of Bounded Treewidth

1.4.1 Graphs of Bounded Treewidth

Treewidth is a property of a graph. The smaller the treewidth, the more similar is a graph to a tree. For example, the treewidth of a tree or a forest is one. Before we introduce the concept of treewidth, we shall introduce tree decomposition.

Given a graph G, let $V(G)$ be its node set and $E(G)$ be its link (edge) set. A *tree decomposition* of graph G is a pair $(T, \{X_t | t \in V(T)\})$, where T is a tree and $\{X_t | t \in V(T)\}$ is a family of subsets of $V(G)$, one for each node of T, such that

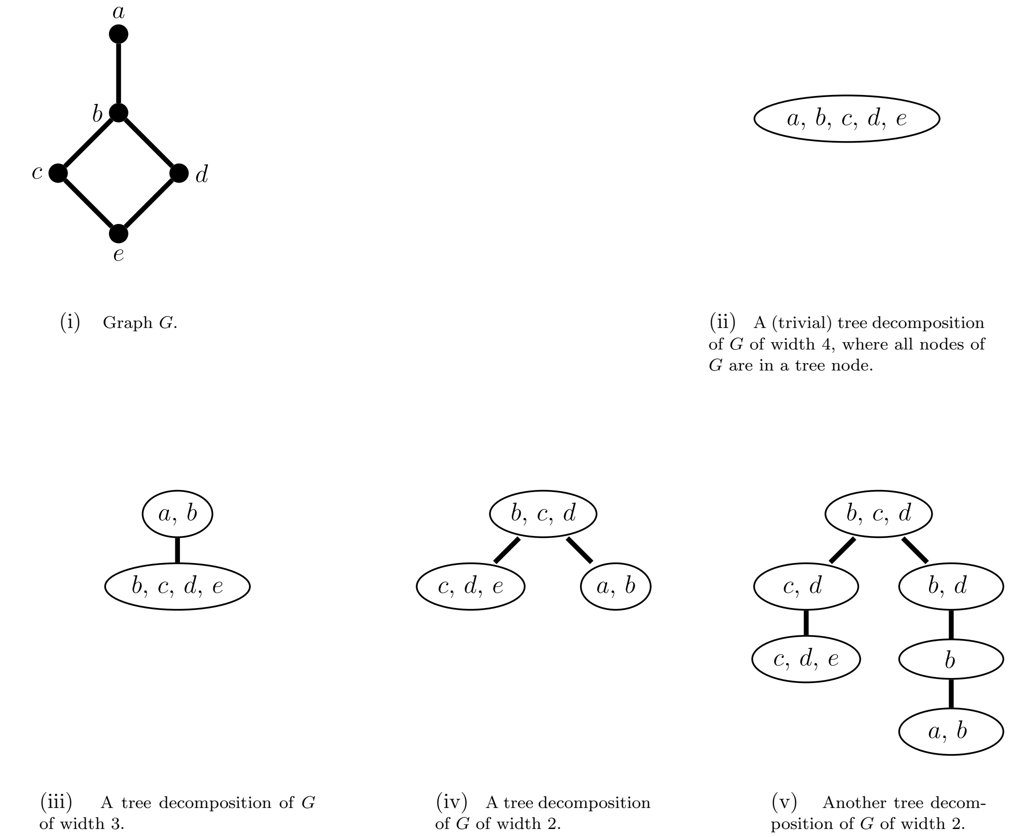
(**T1**) Both ends of each link (edge) in G are included in some node in tree T. That is, for every link $(u, v) \in E(G)$, there is a $t \in V(T)$ such that both u and v are in X_t. Note that (**T1**) implies that each node in G is included in some node in T.

(**T2**) If j is on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$. Equivalently, for every v in G, all the nodes in T that contains v form a connected subtree in T.

The *width* of a tree decomposition is $\max_{t \in V(T)} |X_t| - 1$. In general, for a given graph, tree decompositions are not unique. See Figures 1.1 and 1.2. The *treewidth* of graph G is the minimum width over all tree decompositions of G, see [10]. The notion of treewidth was introduced by Robertson and Seymour in [51].

If the treewidth of G is smaller than or equal to a constant k, then we say G has a bounded treewidth k. See Figures 1.1 and 1.2 for graphs and bounded treewidth 2.

For an arbitrary graph, to find its treewidth is $\mathcal{NP}$-complete. However, given a fixed number k, there are polynomial-time algorithms to decide whether the treewidth of a graph is bounded by k or not. Thus, we shall focus on graphs of bounded treewidth.



(i) Graph G.

(ii) A (trivial) tree decomposition of G of width 4, where all nodes of G are in a tree node.

(iii) A tree decomposition of G of width 3.

(iv) A tree decomposition of G of width 2.

(v) Another tree decomposition of G of width 2.

FIGURE 1.1: A graph of bounded treewidth 2 and some of its tree decompositions.

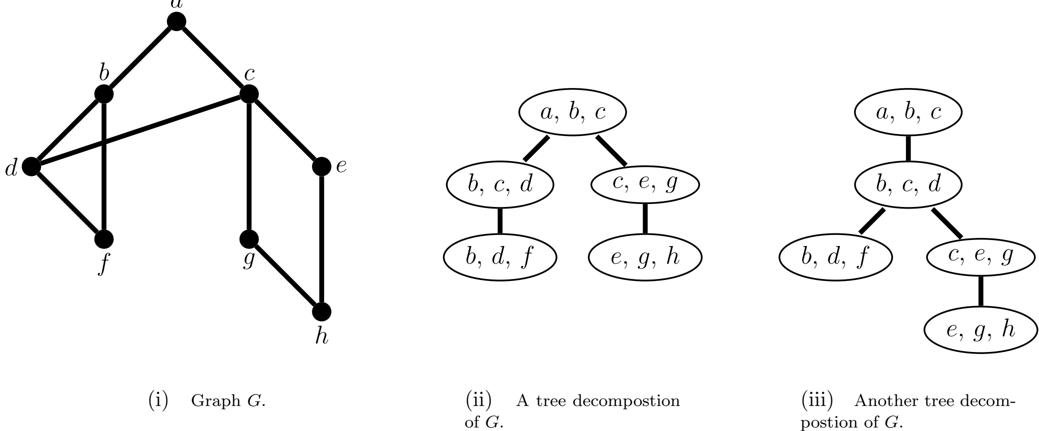


FIGURE 1.2: A graph of bounded treewidth 2 and two of its tree decompositions.

1.4.2 Tree Decompositions of Some Common Graphs

We first state some simple results on tree decompositions, then give some tree decompositions of common graphs.

Let $\{C_i\}$ be the set of bicomponents (2-connected components) in a graph G. Then $\text{treewidth}(G) = \max\{\text{treewidth}(C_i)\}$. This follows from the fact that C_i are arranged in the form of a tree, and one can obtain a tree decomposition of G by combining tree decompositions of each C_i in the tree structure that describes the way that C_i's are connected to each other. For example, in Figure 1.2(i), the bicomponents are $C_1 = \{a, b, c, d, f\}$ and $C_2 = \{c, e, g, h\}$, and C_1 and C_2 are connected to each other with exactly one common node c. It is clear that $\text{treewidth}(G) = \max\{\text{treewidth}(C_1), \text{treewidth}(C_2)\} = 2$.

We may think that tree decomposition is a generalization of bicomponent decomposition in that both decompositions are tree structures, and in general, in tree decomposition, an X_t (the set of nodes of G contained in a tree node of a tree decomposition of G) does not need to induce a connected subgraph on G; also, between two X_t's, there can be more than one common node.

Tree decompositions of a tree and a circle are shown in Figures 1.3 and 1.4,

respectively. It is clear that the treewidth of any tree is one, while the treewidth of any circle is two. Here is an approach to obtain a tree decomposition of width 2 for a circle. Remove a node from the circle, say node a in Figure 1.4(i), do a tree decomposition in the remaining graph, which is a path, then add the removed node back to every tree node of that decomposition.

From the above way of obtaining a tree decomposition for a circle, we observe that, for any graph G and one of its node v, we have

$$\text{treewidth}(G - v) \leq \text{treewidth}(G) \leq 1 + \text{treewidth}(G - v).$$

The second inequality can be explained as follows: take any node v out, find an optimal tree decomposition of the remaining graph $G - v$, then add v to that optimal decomposition, we obtain a tree decomposition of G, though not necessarily optimal, with width no larger than $(1 + \text{treewidth}(G - v))$. The inequality then follows immediately.

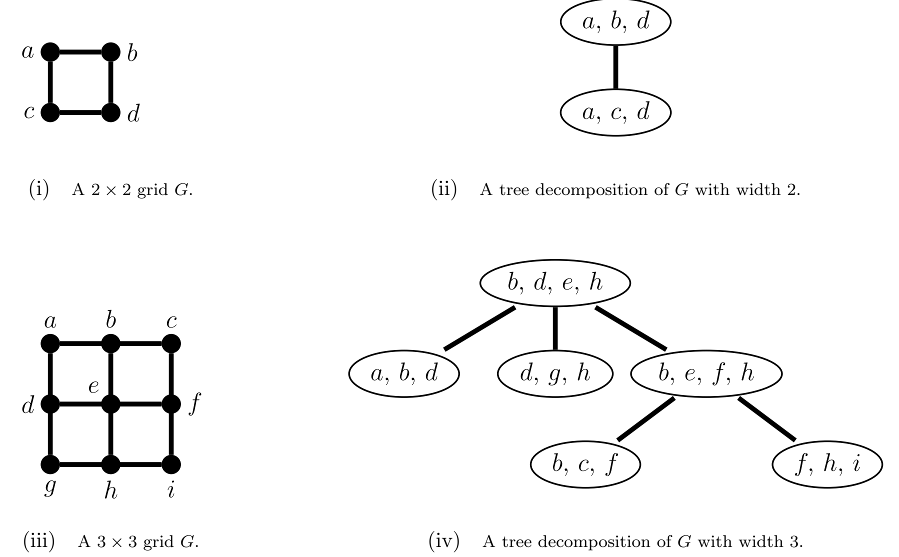
(i) Tree T.　　(ii) A tree decomposition of T.

FIGURE 1.3: A tree decomposition of a tree.

(i) A circle G. (ii) A tree decomposition of G.

FIGURE 1.4: A tree decomposition of a circle.

Note that an $n \times n$ grid (also called $n \times n$ wall), where $n \geq 1$, has treewidth n. It is worthy noting that $n \times n$ grids are simple graphs with big treewidths. See Figure 1.5. Also, treewidth may be larger than the maximum degree of all nodes. For example, a 5×5 grid has treewidth 5, which is larger than the maximum degree 4 of the graph.



(i) A 2×2 grid G. (ii) A tree decomposition of G with width 2.

(iii) A 3×3 grid G. (iv) A tree decomposition of G with width 3.

FIGURE 1.5: Tree decompositions of $n \times n$ grids when $n = 1, 2$.

1.4.3 A Property of Tree Decomposition

A *vertex-cut* [2] of nodes u and v are a set of nodes whose removal disconnects u and v. For example, set $\{b\}$ is a vertex-cut for nodes a and e in the graph of Figure 1.6(i), so is $\{c, d\}$.

A tree decomposition of graph G maintains vertex-cut information for the nodes in G, as shown in Example 1.1.

Example 1.1. In Figure 1.6(ii), node a in G is included in $X_3 = \{a,\ b\}$ of T, while node e in G is located in $X_2 = \{c,\ d,\ e\}$ of T. Note that, in tree T, node 1 is on the path from 2 to 3, while neither a nor e of G is in $X_1 = \{b,\ c,\ d\}$. Observe that X_1 contains vertex-cuts $\{b\}$ and $\{c,\ d\}$ for nodes a and e. Said differently, after the removal from G the nodes specified in X_1, that is, nodes b, c, and d, graph G is broken into two components, one contains a, and the other contains e.

Similarly, in Figure 1.6(iii), node $X_2 = \{c,\ d\}$ in T' contains a vertex-cut, which is X_2 itself, for $e \in X_4 = \{c,\ d,\ e\}$ and $b \in X_5 = \{b\}$.

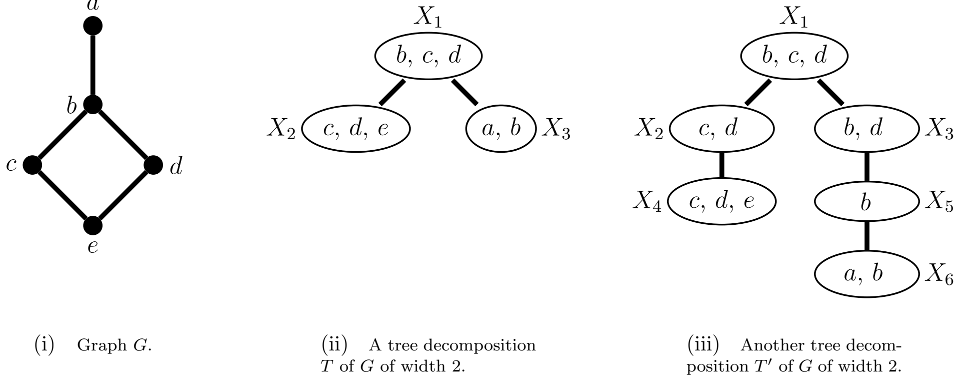


FIGURE 1.6: A tree decomposition of a graph contains its vertex-cut information.

The above observation can be generalized. Let j be on the path from i to k in

[2] A vertex is also called a node.

T. Suppose that $u \in X_i$ and $v \in X_k$. If u, $v \notin X_j$, then X_j contains a vertex-cut for u and v. This property reflects structural similarities between trees and tree decompositions of graphs.

1.4.4 The CUP in Networks of Bounded Treewidth

In a network modelled as a simple graph (one without loops or multiple links) of bounded treewidth k, for some fixed number k, each node is either a C-node (one with a converter) or an NC-node (one without a converter). Each link is assigned an available wavelength. Furthermore, we assume that all links with the same wavelength form a connected subgraph.

Node u *wavelength-dominates* v if u is a C-node and the converter at u is put in use, furthermore, there is a uniform-wavelength $u, v-$path, that is, all the links are of the same wavelength.

Given the above network model, the CUP is formulated as the *minimum connected wavelength-dominating set problem* (MCWDSP), where connected means that, between any two dominating nodes u and v, there is either a uniform-wavelength path, or a path in which uniform-wavelength segments are joined by some dominating nodes.

Due to the complexity of the above CUP, in this dissertation, we consider the *minimum wavelength-dominating set problem* (MWDSP). That is, given a network N of bounded treewidth k with a tree decomposition of N, find a minimum cardinality subset S of converters such that, each node in the network is either in S or is wavelength-dominated by a node in S. One such solution provides a low bound for the CUP.

1.5 Related Works

A related problem to the CUP is the converter placement problem (CPP), whose goal is to locate a minimum number of nodes to install converters so that a given set of communication requests (or traffic loads) can be satisfied.

For CPP, in simple topologies, like bus and ring, an algorithm is presented in [62] to minimize the blocking probability with a minimum number of converters. For more complex topologies, one normally relies on extensive simulation using heuristic methods [65, 33, 67]. The above works only consider one-to-one communication requests (that is, a source sends messages to exactly one destination).

The CUP aims to use a minimum number of available converters, which are decided by the CPP, for broadcasting requests. In [52], the authors provide an approximate algorithm to locate a minimum number of converters such that, with wavelength converters placed at these nodes, broadcast can be supported. A linear algorithm is proposed in [35] to use a minimum number of converters for broadcasting in an optical tree network. This dissertation generalizes the result in [35] by considering a *sparse conversion* optical tree network, that is, only a fraction of the nodes are eligible to have converters.

Dominating set problem is very important and has wide applications. For a general graph, this problem is $\mathcal{NP}$-complete. However, if we consider the graphs with bounded treewidth (also called partial k-trees), then it can be solved be in linear time, see [3] and [20]. More examples on problems other than dominating sets on bounded treewidth graphs can be found in [8], [64], [56], and so on. For a complete survey on algorithms on bounded treewidth graphs, see [10].

The CUP is modelled as the *connected wavelength-dominating set* problem in graph theory, where the nodes with converters are classified as potential dominating nodes. The difference between dominating and wavelength-dominating is that, in

dominating set problem, a dominating node can only dominate its neighbors; while in wavelength-dominating set problem, only converter nodes are eligible to be dominating nodes, furthermore, a dominating node may dominate non-neighbors as long as there is a unique-wavelength path to reach each of those non-neighbors.

A possible way to attack the CUP in bounded treewidth networks is Monadic second-order logic (MSOL) [21]. We do not take this approach, since ad-hoc dynamic programming algorithms normally results in smaller constant complexity algorithm. Furthermore, we want to tune on the programs so that, for this specific CUP, the run-time complexity will not be very high, since this is an application problem in networking.

1.6 Overview of the Dissertation

In Chapter 2, we design a linear complexity dynamic programming algorithm to solve the CUP in tree networks with sparse conversion.

In Chapter 3, we solve the CUP in networks of bounded treewidth by a linear complexity dynamic programming algorithm.

In the last chapter, we summarize the dissertation and outline future works. We attach the source code to implement the algorithm of Chapter 3 in Appendix.

Chapter 2

Broadcast in Optical Tree Networks Using Fewest Converters

2.1 Problem Description

In the CUP (Converter Usage Problem) discussed in [35], a converter can be located at *any* node. In this chapter, we generalize the CUP in tree networks by assuming that the converters are restricted to *some* designated nodes only.

To broadcast in tree networks, it is necessary and sufficient to have a converter at every node whose incident links are not assigned the same wavelength. Before we formulate the CUP in tree networks, we shall introduce some notations as follows.

A node is called a *C-node* if it has a converter; otherwise, it is an *NC-node*. A C-node is said to be *used* if its converter is put in use. We adopt the following symbols throughout this dissertation.

○ an NC-node.

□ a C-node.

⧅ a used C-node, that is, a C-node whose converter is put in use.

Denote an optical tree network as $T = (V, C, E, W)$, where V is the set of nodes, $C \subseteq V$ is the set of C-nodes, E denotes the set of fiber links (u, v), with $u, v \in V$. and W is a mapping such that $W(u, v)$ represents the set of available wavelengths in link (u, v). When $C = V$, the problem is reduced to the CUP in [35].

The CUP that we examine in this chapter is outlined as follows: given an optical

tree network $T = (V, C, E, W)$, assign a wavelength for each link such that, after we use the converters in a minimum number of nodes in set C, every node in T can send messages to all the others.

This chapter is structured as follows. In Section 2.2, we prove dynamic programming formulas used to find the minimum number of converters needed for the network. In Section 2.3, we provide Algorithm CONVERTER-USAGE.

2.2 Theorems

We may assume that:

(**A1**) Every terminal node is an NC-node. Since a terminal node v is connected to only one neighbor, a converter at v performs no wavelength conversion. Thus no terminal node can be included in an optimal solution.

For the remainder of this chapter, it will be convenient to treat T as a rooted tree.

(**A2**) Select a terminal node as the root. (This is done just to avoid the special processing needed when the root has more than one child.) According to this assumption, the root is an NC-node, and is denoted by ◎.

We now introduce some notations for rooted trees. There is a parent-child relationship between each pair of adjacent nodes: the node that is closer (with shorter distance) to the root is a parent, the other, a child. A descendant of a node is similarly defined. For example, in network T' of Table 2.1, the parent of node c is b, while the children of node c are d and e. Given a node v, we denote the parent of v as parent(v).

For a node v, let T_v be the induced subtree of T at v and all its descendants, where *induced* means that, for nodes u and w in T_v, link (u, w) is included in T_v if and only if (u, w) is in T. The following is an approach to obtain T_v. If v is a

non-root node, then we remove the link between v and its parent, thus break T into two subtrees, T_v is the one containing v. If node v is the root, then $T_v = T$. For example, we have $T'_a = T'$, as shown in Table 2.1. This property will be used in the proof of Theorem 2.8.

Given a node v, its parent, children, and descendants, along with T_v, may vary according to the different choice of the root. See Table 2.1.

Now we define $P_\lambda(v)$. For each wavelength $\lambda \in W(\text{parent}(v), v)$, when messages are sent to v by parent(v) in wavelength λ, let $P_\lambda(v)$ be the minimum number of C-nodes that are put in use in T_v for broadcasting in T_v, If $\lambda \notin W(\text{parent}(v), v)$, then we assign $P_\lambda(v) = \infty$.

By Assumption (**A2**), we choose a terminal node as the root. This helps to avoid the special processing needed when the root has more than one child. Furthermore, this will save us from calculating $P_\lambda(r)$, for root r. The reasons are listed as follows.

(i) The root has no parent, or, parent(r) is empty. It can be thought that the root initializes the communication by sending messages to all the others.

(ii) What is more important, once we have the information of $P_\lambda(u)$, where u is the only child of the root (since we choose a terminal node to be the root), we can find the minimum number of C-nodes used for broadcasting in the network. The details are shown in the proof of Theorem 2.8.

TABLE 2.1: Some concepts of rooted trees

A unrooted tree network T	Tree network T' rooted at node a	Tree network T'' rooted at node h
T	T'	T''
the parent of c	b	d
the children of c	d and e	b and e
the descendants of c	d, e, f, g, and h	b, e, a, and g
T_c: the induced subtree of T at node c and all its descendants	T'_c of T'	T''_c of T''
T_a: the induced subtree of T at node a and all its descendants	T'_a of T' equals T'.	T''_a of T'' is a single node.

Hence we shall focus on how to calculate $P_\lambda(v)$ for a non-root node v. When v is a terminal node, that is, an NC-node with only one neighbor, we use Proposition 2.1; when v is an NC-node with more than one neighbor, we apply Theorem 2.3. When v is a C-node, the formula is given in Theorem 2.6. the minimum number of converters needed for broadcasting is provided by Theorem 2.8.

As an initialization step, for each non-root node v and each wavelength λ, we set $P_\lambda(v) = \infty$.

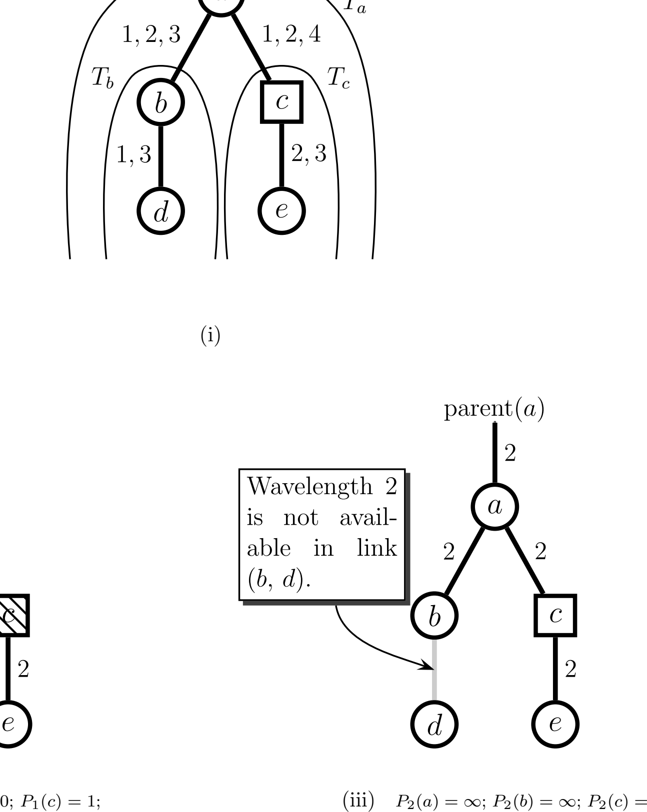
Proposition 2.1. If v is a terminal node, then, for every $\lambda \in W(\text{parent}(v), v)$, we have $P_\lambda(v) = 0$.

Proof. When a message is transmitted to v from its parent in some available wavelength λ of link $(\text{parent}(v), v)$, node v receives the message. Note that T_v is v itself when v is a terminal node, thus a broadcast in T_v is carried out trivially. Evidently there is no need to use any converter at the node in T_v. By definition of $P_\lambda(v)$, we have $P_\lambda(v) = 0$, as desired. □

For example, in Figure 2.1(i), we have $P_1(d) = 0$ and $P_3(d) = 0$. Note that $P_2(d) = \infty$ by the initialization step. Intuitively, this can be explained as follows: since wavelength 2 is not available in link (b, d), we cannot use wavelength 2 to broadcast in T_d. Hence, no matter how many converters we use at the C-nodes in T_d, even if this number is arbitrarily large (say ∞), still no broadcast can be carried out.

Next we shall check the following example before we provide a formula to calculate $P_\lambda(v)$, when v is an NC-node with more than one neighbor.

Example 2.2. Calculate $P_1(a)$, $P_1(b)$, and $P_1(c)$ in the network shown in Figure 2.1(i). Find the relationships among them. Do the same for $P_2(a)$, $P_2(b)$, and $P_2(c)$.



(i)

(ii) $P_1(a) = 1$; $P_1(b) = 0$; $P_1(c) = 1$; $P_1(a) = P_1(b) + P_1(c)$.

(iii) $P_2(a) = \infty$; $P_2(b) = \infty$; $P_2(c) = 0$; $P_2(a) = P_2(b) + P_2(c)$.

FIGURE 2.1: An illustration of the dynamic formula for NC-nodes.

When a message comes to b from a in wavelength 1, node b can use the same wavelength to reach d. Hence $P_1(b) = 0$.

When a message comes to c from a in wavelength 1, the only available wavelengths for link (c, e) are 2 and 3, whichever we choose, it is different from the incoming wavelength 1 of node c, hence it is necessary (and also sufficient) to use the converter at node c. Thus $P_1(c) = 1$.

Next we calculate $P_1(a)$. Since there is no wavelength shared by all the links in T_a, we need to use at least a converter in T_a. On the other hand, when a message arrives at node a from its parent in wavelength 1, use the converter at c is sufficient for broadcasting in T_a, as shown in Figure 2.1(ii). Therefore $P_1(a) = 1$. Thus we have $P_1(a) = P_1(b) + P_1(c)$.

Now we calculate $P_2(a)$. First we find the value for $P_2(b)$. When a message arrives at b from node a in wavelength 2, no broadcast can be carried out in T_b. Hence $P_2(b) = \infty$. It is clear that $P_2(c) = 0$. We can verify that $P_2(a) = \infty$. Thus $P_2(a) = P_2(b) + P_2(c)$, as shown in Figure 2.1(iii).

The above equation can be generalized and is stated in Theorem 2.3.

Theorem 2.3. If v is a non-root NC-node, then, for each $\lambda \in W(\text{parent}(v), v)$, we have

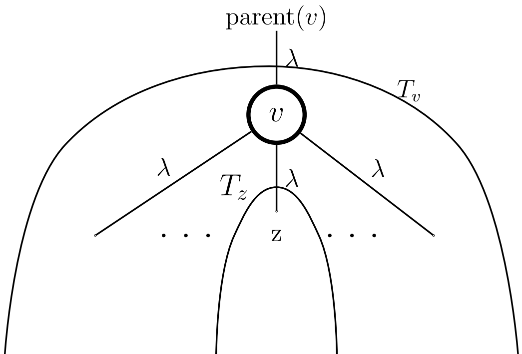
$$P_\lambda(v) = \sum_{z\text{: child of } v} P_\lambda(z).$$

That is, the minimum number of C-nodes used in T_v to broadcast in T_v is the sum of the minimum number of C-nodes used in T_z to broadcast in T_z, where z is a child of v. In another word, if we think of T_z as a branch of T_v, then we obtain the minimum number of C-nodes needed to be used in T_v by adding up the corresponding minimum number for each branch of T_v.

Proof. There are two cases to be considered, as shown in the following.

(**Case 1**) If a message arrives at node v in wavelength λ cannot be broadcasted in T_v, then $P_\lambda(v) = \infty$. Furthermore, there is a child y of v such that the message cannot reach all the nodes in T_y, which implies that $P_\lambda(y) = \infty$. Hence $P_\lambda(v) = \sum_{z\text{: child of } v} P_\lambda(z) = \infty$ holds. For example, when λ is not shared by all the links incident at v, we have $P_\lambda(v) = \infty$ and $\sum_{z\text{: child of } v} P_\lambda(z) = \infty$.

(**Case 2**) If a message arriving at node v in wavelength λ can be broadcasted in T_v, then $P_\lambda(v) < \infty$. In this situation, it is necessary for λ to be shared by all the links incident at v. Thus, when a message comes from parent(v) to v in wavelength λ, the message will continue to move from v to each of its children z in wavelength λ, as shown in the following figure.



If a total of $P_\lambda(z)$ of C-nodes in T_z are put in use, then the message can be transmitted to all the nodes in T_z. Thus a total of $\sum_{z\text{: child of } v} P_\lambda(z)$ converters is sufficient for broadcasting in T_v. Hence, we have $P_\lambda(v) \leq \sum_{z\text{: child of } v} P_\lambda(z)$.

It remains to show that the equality holds. Suppose not. Then, for a given optimal solution using $P_\lambda(v)$ converters in T_v, let $S(z)$ be the number of put-in-use C-nodes in T_z that contribute to $P_\lambda(v)$. Hence $P_\lambda(v) = \sum_{z\text{: child of } v} S(z)$. Since $P_\lambda(v) < \sum_{z\text{: child of } v} P_\lambda(z)$ by assumption, we have $\sum_{z\text{: child of } v} S(z) < \sum_{z\text{: child of } v} P_\lambda(z)$. Therefore, there is some child x of v such that $S(x) < P_\lambda(x)$. Note that, when λ is assigned for link (v, x), variable $S(x)$ is the number of C-nodes used for broadcasting in T_x, a contradiction to the minimality of $P_\lambda(x)$. $\square$

Now we consider the case when v is a C-node. When a message arrives at v from parent(v) in wavelength λ, we shall decide whether the converter at v should be used or not.

We shall introduce two notations before delving into the analysis. Given a C-node v and the incoming wavelength λ in link (parent$(v), v$), let $P_\lambda^{+0}(v)$ be the minimum number of C-nodes in T_v needed to be used for broadcasting in T_v, when there is the converter at v is not used. Notation $P_\lambda^{+1}(v)$ is similarly defined except that the converter at v is used. It is clear that $P_\lambda(v) = \min\{P_\lambda^{+0}(v), P_\lambda^{+1}(v)\}$. The remaining job is to calculate $P_\lambda^{+0}(v)$ and $P_\lambda^{+1}(v)$.

Calculate $\mathbf{P_\lambda^{+0}(v)}$.

If the converter at v is not used, then v acts as an NC-node, By Theorem 2.3, we have the following lemma.

Lemma 2.4. $P_\lambda^{+0}(v) = \sum_{z\text{: child of } v} P_\lambda(z)$.

Calculate $\mathbf{P_\lambda^{+1}(v)}$.

If the converter at v is used, then the message may leave from v to each of its children z in any wavelength available in link (v, z). In particular, for each subtree T_z, we may select a wavelength $\lambda'' \in W(v, z)$, such that the number of C-nodes used in T_z is the minimum. That is, wavelength λ'' satisfies $P_{\lambda''}(z) = \min_{\lambda' \in W(v,z)} P_{\lambda'}(z)$.

By definition of $P_\lambda^{+1}(v)$, we have

$$P_\lambda^{+1}(v) \leq 1 + \sum_{z\text{: child of } v} \min_{\lambda' \in W(v,z)} P_{\lambda'}(z),$$

where the additional 1 represents for the converter at v is used.

In fact, the above inequality is an equation, as Lemma 2.5 states.

Lemma 2.5. $P_\lambda^{+1}(v) = 1 + \sum_{z\text{: child of } v} \min_{\lambda' \in W(v,z)} P_{\lambda'}(z)$.

Proof. It remains to show that the equality holds.

Suppose not. Let $S(z)$ be the number of those converters in T_z that contribute to $P_\lambda^{+1}(v)$. Then

$$P_\lambda^{+1}(v) = 1 + \sum_{z\text{: child of } v} S(z) \; < \; 1 + \sum_{z\text{: child of } v} \min_{\lambda' \in W(v,\, z)} P_{\lambda'}(z).$$

Equivalently, we have

$$\sum_{z\text{: child of } v} S(z) \; < \; \sum_{z\text{: child of } v} \min_{\lambda' \in W(v,\, z)} P_{\lambda'}(z).$$

Hence, there is a child x such that $S(x) < \min_{\lambda' \in W(v,\, x)} P_{\lambda'}(x)$, while both $S(x)$ and $\min_{\lambda' \in W(v,\, x)} P_{\lambda'}(x)$ are sufficient for broadcasting in T_x, a contradiction to the minimality of $\min_{\lambda' \in W(v,x)} P_{\lambda'}(x)$. □

From the above discussion, we obtain the following theorem.

Theorem 2.6. If v is a C-node, then

$$P_\lambda(v) = \min\{P_\lambda^{+0}(v),\; P_\lambda^{+1}(v)\}, \text{ where}$$

$$P_\lambda^{+0}(v) = \sum_{z\text{: child of } v} P_\lambda(z) \;\text{ and }\; P_\lambda^{+1}(v) = 1 + \sum_{z\text{: child of } v} \min_{\lambda' \in W(v,\, z)} P_{\lambda'}(z).$$

Proof. The assertion follows from the definitions of $P_\lambda^{+0}(v)$, $P_\lambda^{+1}(v)$, and $P_\lambda(v)$, along with Lemmas 2.4 and 2.5. □

Example 2.7. This example is used to illustrate Theorem 2.6. Calculate $P_1(a)$, $P_1^{+0}(a)$, and $P_1^{+1}(a)$ for the network shown in Figure 2.2(i). Find the relationships among them. Do the same for $P_2(a)$, $P_2^{+0}(a)$, and $P_2^{+1}(a)$.

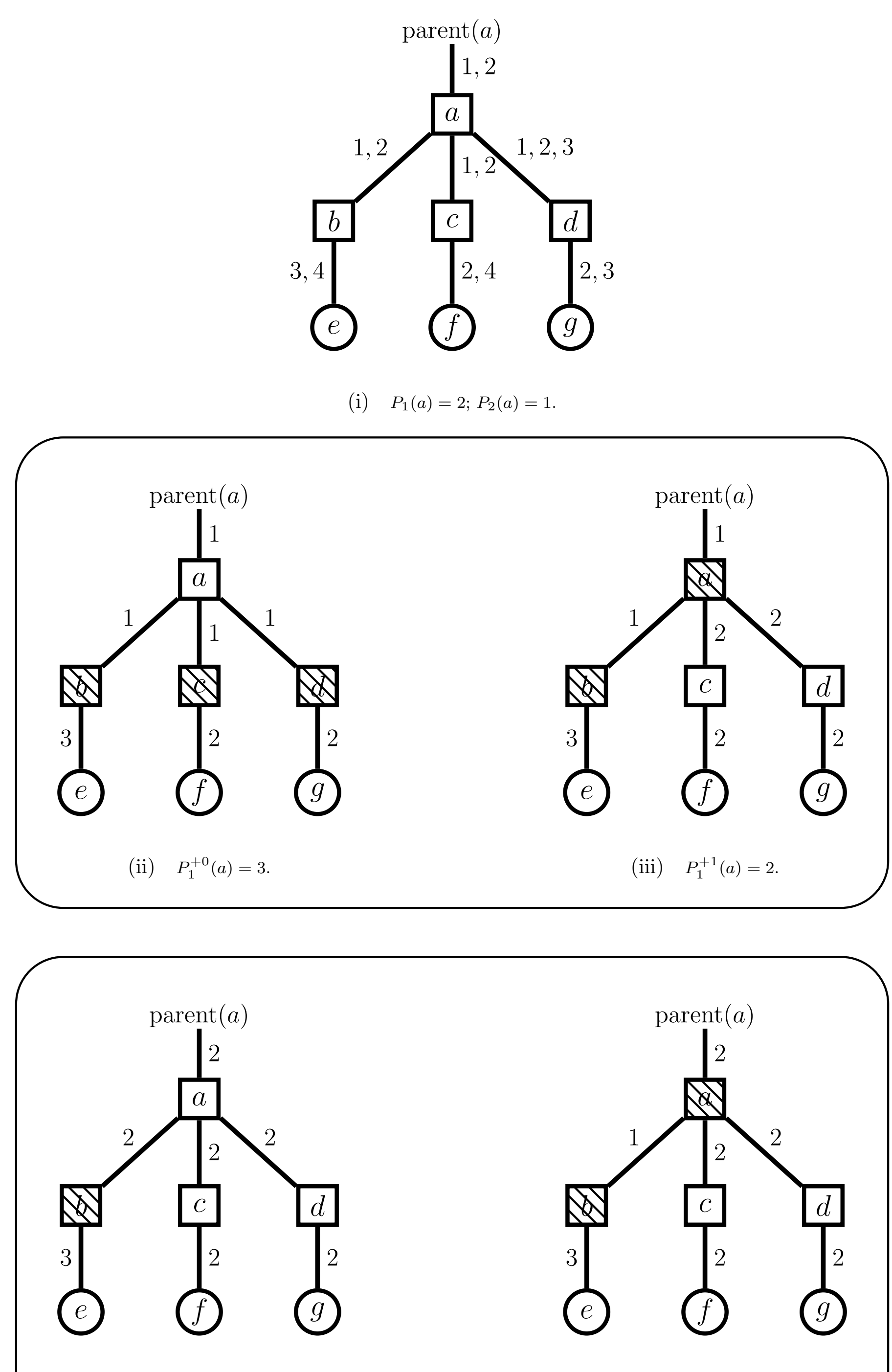


FIGURE 2.2: An illustration of the dynamic formula for C-nodes.

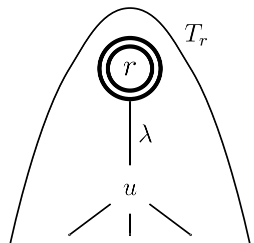
When a message comes to a in wavelength 1, at least two C-nodes are needed to be used, and two converters are sufficient, see Figure 2.2(iii), hence $P_1(a) = 2$. On the other hand, if we do not use the converter at a, then a minimum of three converters are needed and therefore $P_1^{+0}(a) = 3$, as shown in Figure 2.2(ii); Similarly, we obtain $P_1^{+1}(a) = 2$, as shown in Figure 2.2(iii). Thus $P_1(a) = \min\{P_1^{+0}(a),\ P_1^{+1}(a)\}$ holds, as desired.

Similarly, when a message comes to a in wavelength 2, at least one C-node, say node b, needs to have a converter, and one converter is sufficient. Thus $P_2(a) = 1$. On the other hand, we have $P_2^{+0}(a) = 1$ and $P_2^{+1}(a) = 2$, as shown in Figures 2.2(iv) and 2.2(v), respectively. Again, $P_2(a) = \min\{P_2^{+0}(a),\ P_2^{+1}(a)\}$ holds.

Now we are ready to find the minimum number of C-nodes used to broadcast in the whole network. If there is only one node in the network, then no converter is needed. Without loss of generality, we may assume that the tree network has two or more nodes, so that a node can be a child of some other node. By assumption (**A2**), root r is chosen from a terminal node and thus has only one child.

Theorem 2.8. For a network T with at least two nodes, the minimum number of converters needed to broadcast in T is $\min_{\lambda \in W(r,u)} P_\lambda(u)$, where u is the child of r.

Proof. Note that $W(r, u)$ is the set of all the possible wavelengths that can be used to send messages from r to u. Furthermore, $T = T_r$. See the following diagram. The assertion then follows from the definition of $P_\lambda(u)$.



□

As a remark of Theorem 2.8, if $\min_{\lambda \in W(r,u)} P_\lambda(u) = \infty$, then there is no solution for the CUP; that is, no broadcast can be carried out for the network.

2.3 An Algorithm

We use Algorithm CONVERTER-USAGE to find an *optimal* wavelength assignment for the network, where optimal means that a minimum number of C-nodes needed to used for broadcasting in the network.

2.3.1 A Pseudocode

We first introduce some variables used in the pseudocode.

TABLE 2.2: Some variables used in Pseudocode CONVERTER-USAGE

Name	Description
convSet	a minimum size subset of C-nodes used for broadcasting in the network
minNumConv	the minimum number of C-nodes used for broadcasting in the network
useConv(λ, v)	Given v and λ, where v is a C-node and λ is a wavelength assigned for link (parent$(v), v)$, to broadcast in T_v with a minimum number of used C-nodes in T_v, whether we use the converter at v or not. If useConv(λ, v)=TRUE, then we use the converter at v and depict v as a hatched box $\boxed{v}$; otherwise, the converter at v is not used, and v is drawn as a blank box $\boxed{v}$.
$\lambda_{\text{best}}(v)$	a wavelength that leads to the minimum $P_\lambda(v)$, for a given node v.
$\lambda_{\text{select}}(\text{parent}(v), v)$	the wavelength assigned for link (parent$(v), v)$ in an optimal wavelength assignment

Algorithm CONVERTER-USAGE (Comments are in brackets [ ].)

Input: An optical tree network $T = (V, C, E, W)$.

Output: Locate a minimum size set convSet of C-nodes for broadcasting and a corresponding wavelength assignment for each link.

(1) [Divide the nodes by layers.] Select a terminal node as the root r and let r be in the first layer. Divide other nodes by layers, according to their distances to r. Let L be the maximum layer of all the nodes.

(2) [Initialize $P_\lambda(v)$ for all non-root nodes v.] For (each non-root node v) and for (each wavelength λ) let $P_\lambda(v) = \infty$.

(3) [Calculate $P_\lambda(v)$ for all non-root nodes v in a bottom-up fashion.] For (layer $i = L$ down-to $i = 2$) and for (each node v at layer i) do the following:

(3.1) [Calculate $P_\lambda(v)$ when v is a leaf.] If v is a leaf, then, for (each $\lambda \in W(\text{parent}(v), v)$), set $P_\lambda(v) = 0$.

(3.2) [Calculate $P_\lambda(v)$ when v is a non-leaf NC-node.] Otherwise, if (v is an NC-node), then, for each $\lambda \in W(\text{parent}(v), v)$, set $P_\lambda(v) = \sum_{z\text{: child of } v} P_\lambda(z)$.

(3.3) [Calculate $P_\lambda(v)$ when v is a non-leaf C-node.] Otherwise, let $P_\lambda^{+1}(v) = 1 + \sum_{z\text{: child of } v} P_{\lambda_{\text{best}}(z)}(z)$.

For (each $\lambda \in W(\text{parent}(v), v)$) do:

(3.3.1) Let $P_\lambda^{+0}(v) = \sum_{z\text{: child of } v} P_\lambda(z)$.

(3.3.2) If $(P_\lambda^{+0}(v) \leq P_\lambda^{+1}(v))$, then set $P_\lambda(v) = P_\lambda^{+0}(v)$ and $\text{useConv}(\lambda, v) = \text{FALSE}$.

(3.3.3) Otherwise, set $P_\lambda(v) = P_\lambda^{+1}(v)$ and $\text{useConv}(\lambda, v) = \text{TRUE}$.

(3.4) Choose $\lambda_{\text{best}}(v)$ leading to the minimum value of $P_\lambda(v)$.

(4) [Now v is the child of r.] Set minNumConv $= P_{\lambda_{\text{best}}(v)}(v)$. If minNumConv $= \infty$, then exit with a message "No broadcast can be carried out in T."

(5) [Locate a total of minNumConv C-nodes to use and record them in set convSet; and, in a top-bottom fashion, assign a wavelength for each link to obtain this optimal usage.] Set convSet $= \varnothing$ and $\lambda_{\text{select}}(v) = \lambda_{\text{best}}(v)$.

For (level $i = 2$ to $i = L - 1$) and for (each node v at level i) do the following:

(5.1) If v is a leaf, then skip to the next node.

(5.2) Otherwise, if (node v is a C-node) and (useConv$(\lambda_{\text{select}}(v), v) =$ TRUE), then do the following:

(5.2.1) Add v to convSet.

(5.2.2) For each child z of v, set $\lambda_{\text{select}}(z) = \lambda_{\text{best}}(z)$.

(5.3) Otherwise, for each child z of v, set $\lambda_{\text{select}}(z) = \lambda_{\text{select}}(v)$.

In Step (3), we calculate $P_{\lambda}(v)$ for every non-root v, in a bottom-up fashion, as shown in Figure 2.3.

Figure 2.4 shows a flow chart of the calculation of $P_{\lambda}(v)$ and $\lambda_{\text{best}}(v)$, for a given v.

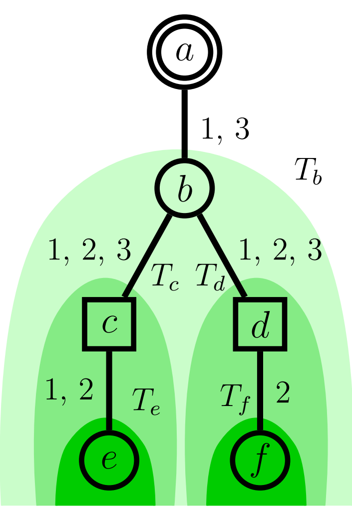


FIGURE 2.3: Calculation of $P_\lambda(v)$ for non-root v in a bottom-up fashion.

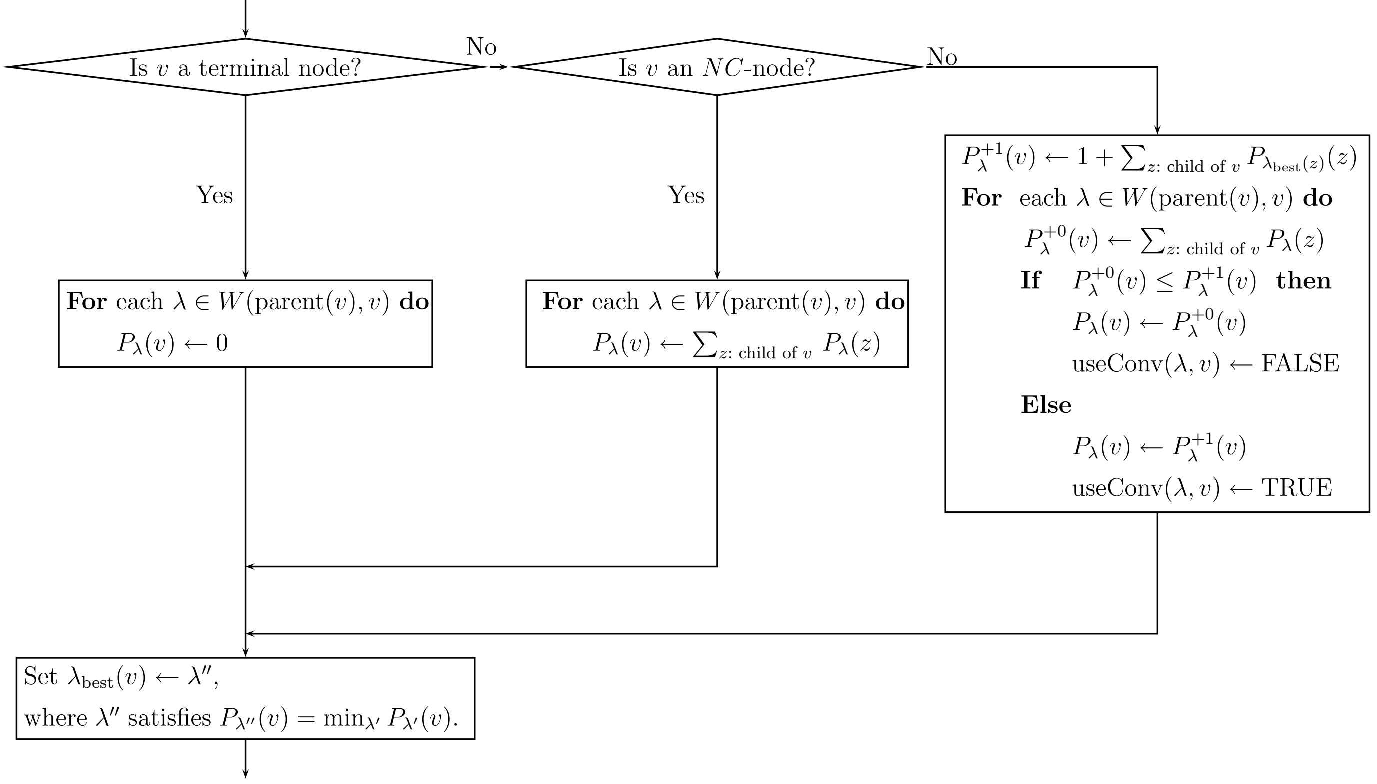


FIGURE 2.4: A flow chart to calculate $P_\lambda(v)$.

TABLE 2.3: Calculation of $P_\lambda(v)$ in a bottom-up fashion

	layer	v	$P_1(v)$; useConv$(1, v)$	$P_2(v)$; useConv$(2, v)$	$P_3(v)$; useConv$(3, v)$
A tree network T	L4	e	$\lambda_{\text{best}}(e)$ 0	0	∞
	L4	f	∞	$\lambda_{\text{best}}(f)$ 0	∞
	L3	c	$\lambda_{\text{best}}(c)$ 0; FALSE	0; FALSE	1; TRUE
An optimal solution for T	L3	d	1; TRUE	$\lambda_{\text{best}}(d)$ 0; FALSE	1; TRUE
	L2	b	$\lambda_{\text{best}}(b)$ 1	∞	2

2.3.2 An Example

Example 2.9. Table 2.3 illustrates Step (3) of Algorithm CONVERTER-USAGE.

For each $\lambda \notin W(\text{parent}(v), v)$, we have $P_\lambda(v) = \infty$ from the initialization step. For example, $P_3(e) = \infty$.

If v is a leaf, then for all $\lambda \in W(\text{parent}(v), v)$, we set $P_\lambda(v) = 0$, according to Proposition 2.1. For example, $P_1(e) = 0$ and $P_2(e) = 0$.

The best incoming wavelength $\lambda_{\text{best}}(v)$ for node v is the one that results in the minimum $P_\lambda(v)$ value. For example, $\lambda_{\text{best}}(e) = 1$.

When v is a C-node, we calculate $P_\lambda(v)$ and useConv(λ, v). As an example, take $\lambda = 3$ and let v be node c. According to Theorem 2.6, we have

$P_3(c) = \min\{P_3^{+0}(c),\ P_3^{+1}(c)\}$. Since e is the only child of c and $\lambda_{\text{best}}(e) = 1$, we have $P_3^{+0}(c) = \sum_{z\text{: child of } c} P_3(z) = P_3(e) = \infty$, and $P_3^{+1}(c) = 1 + \sum_{z\text{: child of } c} P_{\lambda_{\text{best}}(z)}(z) = 1 + P_1(e) = 1$. It is clear that $P_3^{+1}(c) < P_3^{+0}(c)$, hence $\text{useConv}(3, c) = \text{TRUE}$.

For a non-terminal NC-node, we apply Theorem 2.3. For example, $P_1(b) = P_1(c) + P_1(d) = 0 + 1 = 1$ and $P_3(b) = P_3(c) + P_3(d) = 1 + 1 = 2$. where c and d are the children of b.

Since $\min_\lambda P_\lambda(b) = 1$, by Theorem 2.8, we assert that a minimum of one converter is needed.

Next we use Example 2.10 to illustrate how to assign a wavelength for each link so that an optimal converter usage is achieved.

Example 2.10. We start from the child of the root, work layer by layer in a top-bottom fashion, applying the following rules to assign a wavelength from node v to each of its children.

(**R1**) If v is the child of the root, then $\lambda_{\text{select}}(v) = \lambda_{\text{best}}(v)$.

(**R2**) If v is an NC-node, or, if v is a C-node whose converter is not put in use, then, for each child z of v, the incoming wavelength of z is the same as that of parent(z). That is, set $\lambda_{\text{select}}(z) = \lambda_{\text{select}}(v)$.

(**R3**) If the converter at a C-node v is used, then v can use the respective best available wavelength $\lambda_{\text{best}}(z)$ to reach each of its children z. That is, set $\lambda_{\text{select}}(z) = \lambda_{\text{best}}(z)$.

We start from node b, the only child of root a. Note that $\lambda_{\text{best}}(b) = 1$. Hence $\lambda_{\text{select}}(b) = 1$ by (**R1**), as shown in Fig. 2.5(i).

Since b is an NC-node, by (**R2**), the message must be transmitted to b's children c and d in wavelength 1, we have $\lambda_{\text{select}}(c) = 1$ and $\lambda_{\text{select}}(d) = 1$. Note that useConv$(1, c)$ = FALSE, while useConv$(1, d)$ = TRUE, hence node c is drawn as a blank box and node d is depicted as a hatched one. See Fig. 2.5(ii).

We apply (**R2**) on node c, and find that $\lambda_{\text{select}}(e) = 1$, as shown in Fig. 2.5(iii).

We use (**R3**) on node d, and assert that $\lambda_{\text{select}}(f) = 2$, see Fig. 2.5(iv).

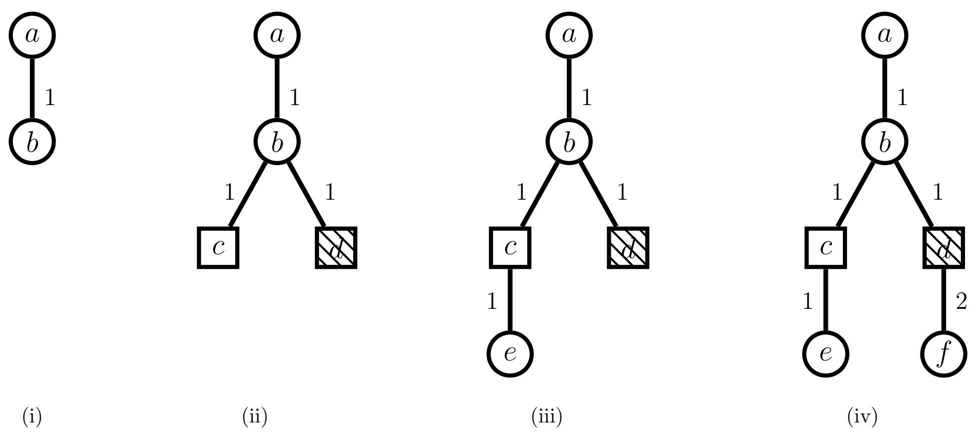


FIGURE 2.5: Assign wavelengths for the network in Table 2.3.

2.3.3 Complexity Analysis

Let $|W| \geq 1$ be the maximum number of wavelengths in each link, while $|V|$ and $|E|$ are the number of nodes and links in the network, respectively.

For the first two steps of Algorithm CONVERTER-USAGE, it takes $O(|E|)$ to divide the nodes by layers and $O(|W| \cdot |V|)$ to initialize $P_\lambda(v)$.

We now consider the time to calculate $P_\lambda(v)$ for all non-root v. It takes more time to calculate $P_\lambda(v)$ for a C-node than for an NC-node. Hence it suffices to find the time to calculate $P_\lambda(v)$ when v is a C-node.

Let numChild(v) be the number of children of node v. For a fixed wavelength

λ, it takes numChild(v) additions to obtain $P_\lambda^{+0}(v) = \sum_{z\text{: child of } v} P_\lambda(v)$ and one comparison between $P_\lambda^{+0}(v)$ and $P_\lambda^{+1}(v)$ to get $P_\lambda(v)$. Therefore, to obtain $P_\lambda(v)$ for all λ, a total of $\big(\text{numChild}(v) + 1\big) \cdot |W|$ operations (additions and comparison) are needed. In addition, it takes $\big(\text{numChild}(v) + 1\big)$ additions to get $P_\lambda^{+1}(v) = 1 + \sum_{z\text{: child of } v} P_{\lambda_{\text{best}}(z)}(z)$, which needs to be calculated only once for each v, since $P_{\lambda'}^{+1}(v) = P_{\lambda''}^{+1}(v)$ for any $\lambda', \lambda'' \in W(\text{parent}(v), v)$. Combine this with the previous result, we assert that, to get $P_\lambda(v)$ for all λ, when v is fixed, at most $\big((\text{numChild}(v) + 1) \cdot |W| + \text{numChild}(v) + 1\big)$ operations are needed. In short, in Step (3), the time to calculate $P_\lambda(v)$, for every λ and v, is $O(|W| \cdot |V|)$.

It takes $O(1)$ to run Step (4).

Note that, in Step (5), it takes $O(1)$ to assign a wavelength for each link. Thus, to find an optimal wavelength assignment for all links takes $O(|E|)$.

In summary, the time complexity of Algorithm CONVERTER-USAGE is $O(|W| \cdot |V| + |E|)$. Note that, in a tree network, we have $|V| = |E| + 1$. Hence the above expression can be simplified as $O(|W| \cdot |V|)$. Since $|W|$ is bounded, we conclude that the time complexity for the algorithm is $O(|V|)$.

Chapter 3

Find a Minimum Wavelength-dominating Set in Optical Networks with Bounded Treewidth

3.1 Problem Description

In this chapter, we consider the converter usage problem (CUP) in an optical network, which is modelled as a simple graph (one without loops and multiple links) of bounded treewidth k, for some fixed positive integer k.

In the network, each node is either a C-node (one with a converter) or an NC-node (one without a converter). Each link is assigned an available wavelength. Furthermore, we assume that all links with the same wavelength form a connected subgraph, as in [52]. One such network is illustrated in Figure 3.1(i).

A *uniform-wavelength* path is one whose links are of the same wavelength, say, paths $c - e - h$ and $c - g - h$ in Figure 3.1(i).

Node u *wavelength-dominates* (abbreviated as *dominates*) v if u is a C-node and the converter at u is put in use, furthermore, there is a uniform-wavelength $u, v-$path. If u dominates v, then u is called a *dominating node* and v is said to be *dominated*. For example, in Figure 3.1(ii), node c dominates a, d, e, g, and h. Equivalently, when the converter at c is put in use, nodes a, c, d, e, g, and h can send messages to each other.

We may call the set of C-nodes whose converters are put in use a dominating set. To enable each node to send messages to all the others, it is necessary, though not sufficient, to find a dominating set S, which is a subset of C-nodes, such that,

for each node v, either v is in S, or v is dominated by a node in S. For example, in Figure 3.1(iii), let $\{b, c\}$ be a dominating set, that is, the converters at b and c are put in use. Then, all the other nodes in the network are dominated. Nodes a, b, d, and f can send messages to each other; so are a, c, d, e, g, and h. However, node b cannot send messages to c.

If, in addition, we use the converter at node a, then each node in the network can send messages to all the others. See Figure 3.1(iv).

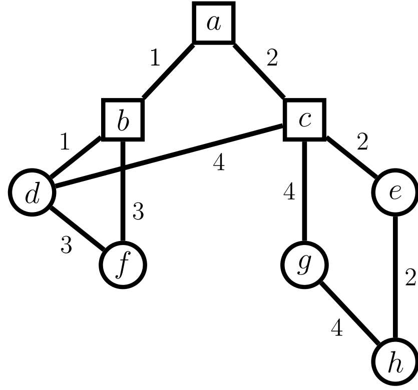
The CUP is modelled as the *minimum connected dominating set* problem, where connected means that, given two dominating nodes u and v, there is either a uniform-wavelength $u, v-$path or a path in which uniform-wavelength segments joined by some dominating nodes (for example, in Figure 3.1(iv), uniform-wavelength segments b–a and a–c are joined by dominating node a to form path b–a–c.).

In this dissertation, we shall focus on the *minimum wavelength-dominating set* problem (MWDSP), described as follows. Given a network of bounded treewidth k, find a minimum cardinality subset S of C-nodes such that, each node in the network is either in S or is dominated by a node in S. One such solution provides a low bound for the CUP.

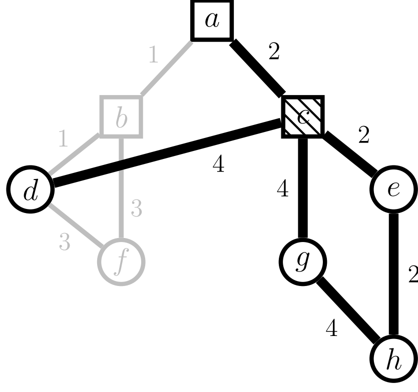
3.2 General Approach for Solving the MWDSP

In a tree decomposition T of G, Select a root in T. Work in a bottom-up fashion using T and find a minimum dominating set in each induced subgraph of G. See Figure 3.2.

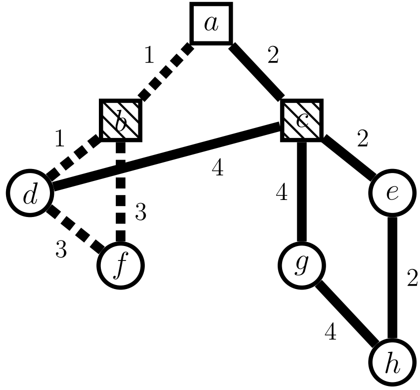
Note that, in Figure 3.2(iv), when we combine the nodes in X_1 and X_2, it is sufficient to broadcast in the subgraph induced on the nodes of $X_1 \cup X_2 = \{b, c, d, e\}$ with the converter at node b. The usage of that converter will activate



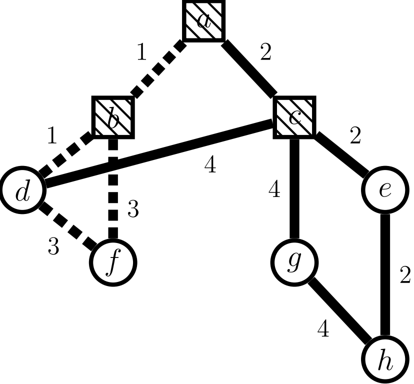
(i) An optical network N of bounded treewidth 2, where a, b, and c are C-nodes.



(ii) Node c dominates a, d, e, g, h through the non-gray paths.

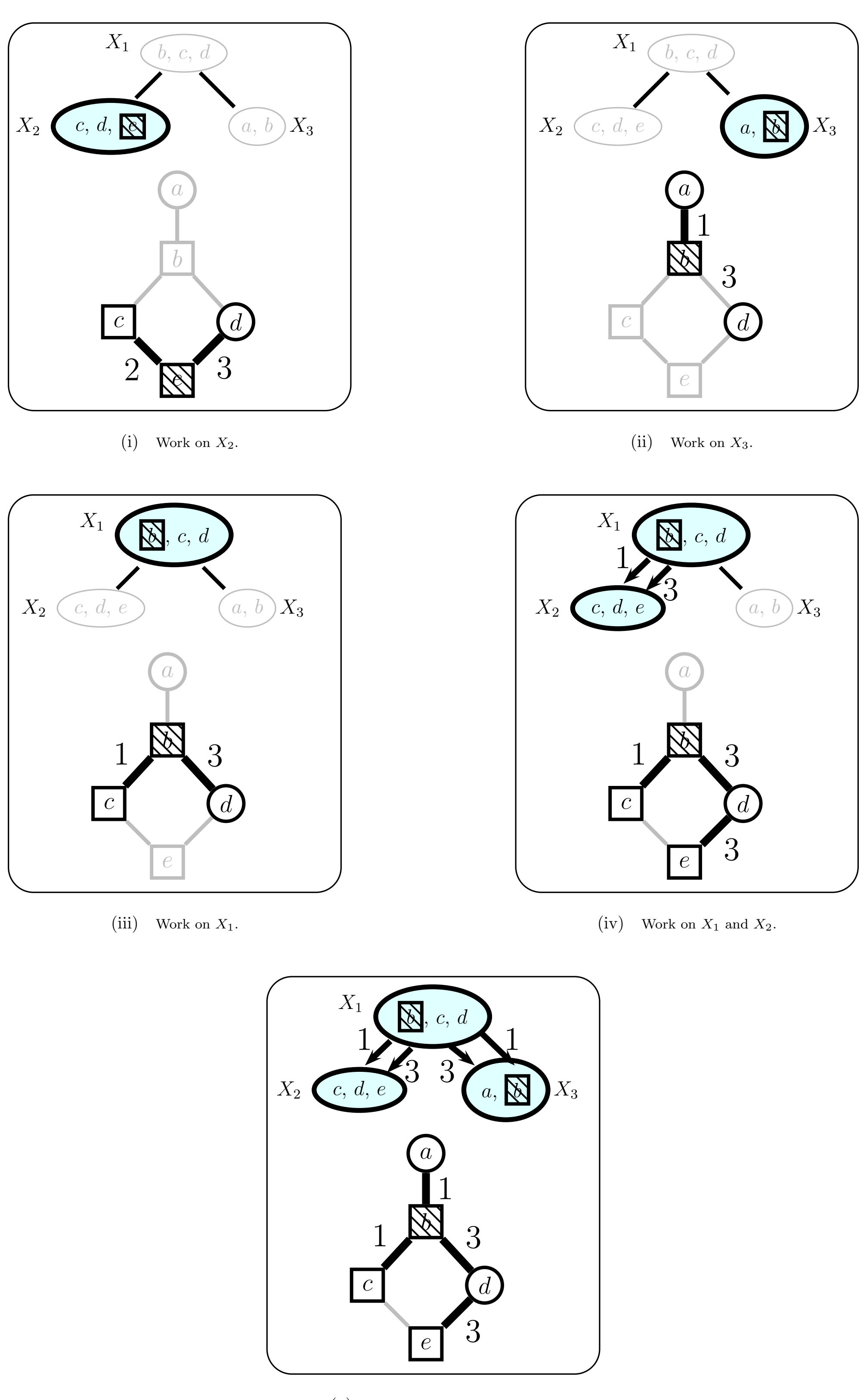


(iii) Nodes a, b, d, and f can send messages to each other; so are a, c, d, e, g, and h. However, b cannot send messages to c.



(iv) Each node can send messages to all the others.

FIGURE 3.1: The dominating set problem in an optical network N. Some tree decompositions of N are shown in Figure 1.2.



(i) Work on X_2.

(ii) Work on X_3.

(iii) Work on X_1.

(iv) Work on X_1 and X_2.

(v) Work on X_1, X_2, and X_3.

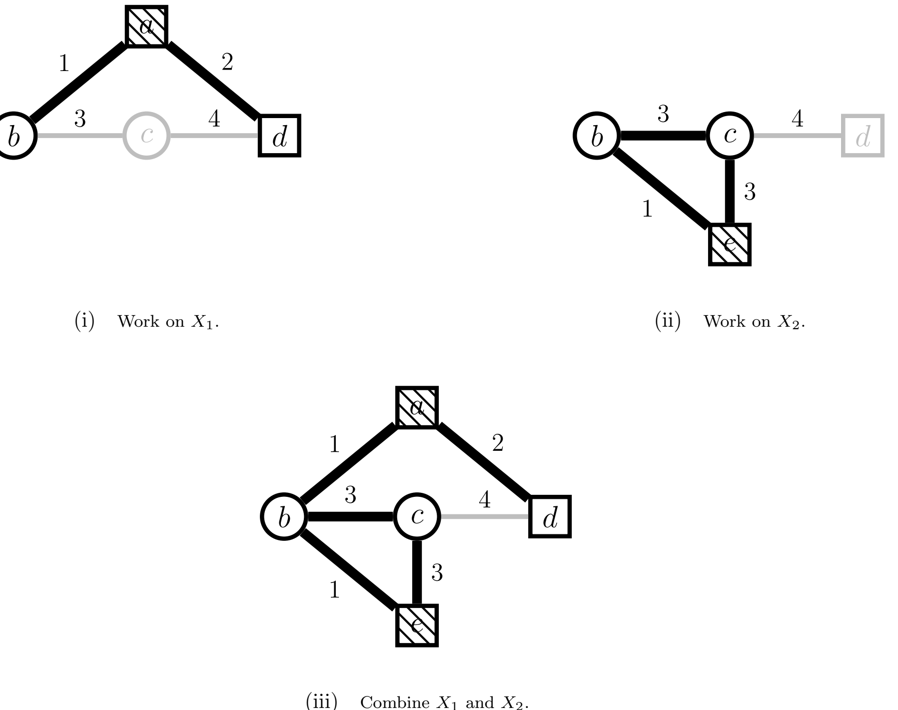
FIGURE 3.2: General approach to solve the MWDSP

wavelength 1 and 3, node e can be reached by wavelength 3, even though e is not the immediate neighbor of b.

Hence, the difference between the MWDSP and the original minimum dominating set problem (MDSP) is that, in MWDSP, a node v can dominates non-neighbors through the wavelengths incident at v; while in the MDSP, a node can only dominates its neighbors. Hence, when we design dynamic programming formulas for the MWDSP, we need to conside the role of wavelengths.

The following example illustrates what kind of wavelength information we need to use in designing the formulas.

Example 3.1. Let graph G be shown in Figure 3.3. Suppose that T is composed of $X_1 = \{a, b, c, d\}$ and $X_2 = \{b, c, d, e\}$. Take X_2 as the root of T.



(i) Work on X_1.

(ii) Work on X_2.

(iii) Combine X_1 and X_2.

FIGURE 3.3: Wavelength information.

Node a activates (provides) wavelengths 1 and 2, requests additional wavelength 3 or 4 to dominate node c, as shown in Figure 3.3(i).

Node e provides wavelengths 1 and 3, requests additional wavelength 2 or 4 to dominate node d. See Figure 3.3(ii).

Neither a nor e is a feasible solution for the corresponding subgraph; however, $\{a, e\}$ is a minimum dominating set, as shown in Figure 3.3(iii).

The above example shows that we shall use parameters to record information like provided wavelength and request wavelength. For details, see the definition of f in Section 3.3.

3.3 Notations and Definitions

Let G be a simple graph of bounded treewidth k, where k is a fixed positive integer. Denote C as a subset of $V(G)$ that have converters, as before.

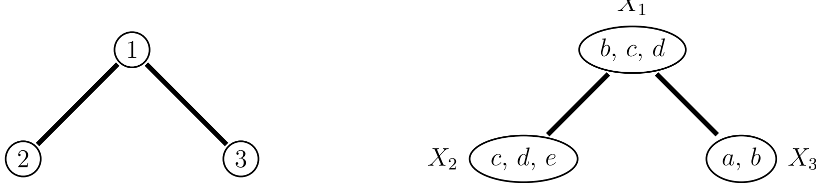
Let Λ be a finite set of wavelengths and W be a mapping from $E(G)$ to 2^{Λ}, the set of all subsets of Λ. For each $\lambda \in \Lambda$, we assume that all the links with wavelength λ form a connected subgraph.

Given a node u of G, let $W_u = \bigcup_{(u,\,v)\in E(G)} W(u,\, v)$, the set of wavelengths on all the links incident at u. For each subset of nodes $U \subseteq V(G)$, let $W_U = \bigcup_{u\in U} W_u$. A subset $\Lambda' \subseteq \Lambda$ *covers* U if $\Lambda' \cap W_u \neq \varnothing$ for all $u \in U$.

For example, in Figure 3.1(i), we have $W_a = \{1,\, 2\}$, $W_b = \{1,\, 3\}$, and $W_c = \{2,\, 4\}$. Therefore, $W_{\{a,b\}} = \{1,\, 2,\, 3\}$ and $W_{\{b,c\}} = \{1,\, 2,\, 3,\, 4\}$. The wavelength set $\Lambda' = \{1,\, 2,\, 3\}$ covers all nodes in the network except g, while $\Lambda' = \{2,\, 3,\, 4\}$ covers every node.

Let $(T,\, \{X_t : t \in V(T)\})$ be a tree decomposition of G as described in Section 1.4.1. We fix a root for T. For each node $t \in V(T)$, denote the number of children of t as $c(t)$. We label the children of t as $t_1,\, t_2,\, \cdots,\, t_{c(t)}$, in some order.

See Figure 3.4.



Let $t = 1$. Then $c(t) = 2$. If we label nodes 2 and 3 as the first and the second child of node 1, respectively, then $t_1 = 2$ and $t_2 = 3$.

Let $t = 2$. Then $c(t) = 0$.

Let $t = 3$. Then $c(t) = 0$.

FIGURE 3.4: An illustration of t_i and $c(t)$.

For $t \in V(T)$ and $0 \leq d \leq c(t)$, define $T(t, d)$ to be node t itself if $d = 0$; otherwise, let $T(t, d)$ be the induced subtree of T on t, the first d children of t — labelled as $t_1, t_2, \cdots, t_d$ — and those children's descendants. Let $\overline{B(t, d)} = \bigcup\{X_i : i \in V(T(t, d))\}$ and $B(t, d) = \overline{B(t, d)} - X_t$. It is clear that $\overline{B(t, 0)} = X_t$ and $B(t, 0) = \varnothing$, for any node t in T. See Figure 3.5.

Next we define $f(t, d, \tau, \Lambda_p, \Lambda_r)$ used in the formulas in Section 3.4. The specifications of t, d, τ, Λ_p, and Λ_r are explained as follows.

(1) $t \geq 1$, a node in T;

(2) $d \geq 0$, the first d children of t in T;

(3) $\tau \subseteq X_t \cap C$, a subset of C-nodes in X_t whose converters are put in use;

(4) $W_\tau \subseteq \Lambda_p \subseteq W_{X_t}$, the subset of wavelengths that are activated by all the put-in-use converters in $\overline{B(t, d)}$ (including those in τ) and are visible [1] from

[1]A wavelength λ is visible from a set U if λ is incident at some node of U.

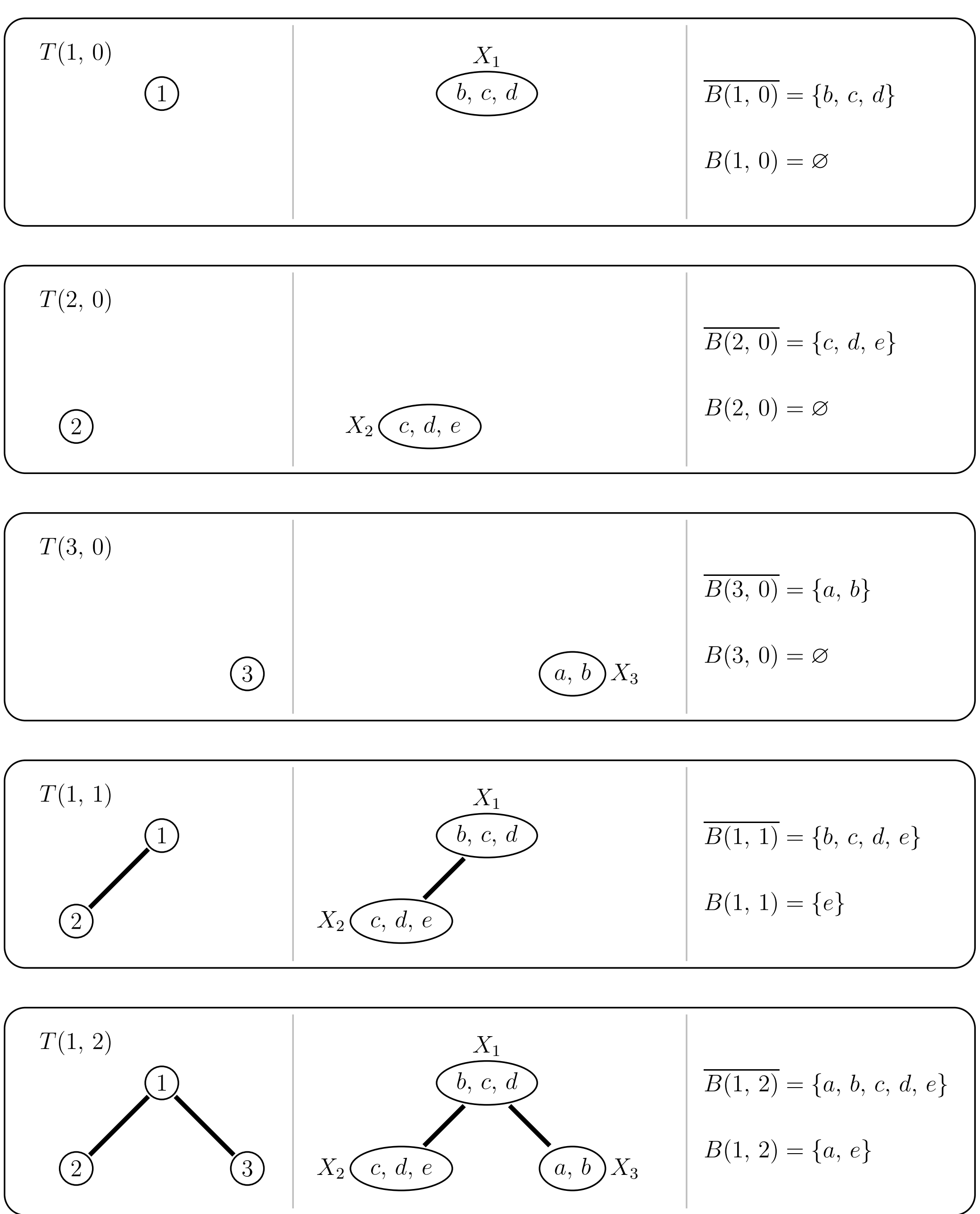


FIGURE 3.5: An illustration of $T(t,\,d)$, $\overline{B(t,\,d)}$, and $B(t,\,d)$.

X_t. We may think Λ_p as the set of wavelengths provided by the converters inside $\overline{B(t,\, d)}$ to cover nodes outside $\overline{B(t,\, d)}$.

(5) $\Lambda_r \subseteq W_{X_t}$, a subset of wavelengths that are activated by all the put-in-use converters outside $\overline{B(t,\, d)}$ and are visible from X_t. We may think Λ_r as the set of wavelengths required from the converters outside $\overline{B(t,\, d)}$ to cover nodes inside $\overline{B(t,\, d)}$.

(6) In addition, Λ_p and Λ_r satisfy

(6.1) $\Lambda_p \cup \Lambda_r$ covers all nodes in X_t; [2]

(6.2) $\Lambda_p \cap \Lambda_r = \varnothing$. We only request the activated wavelengths from the put-in-use converters outside $\overline{B(t,\, d)}$ when the inside ones cannot provide.

Given t, d, τ, Λ_p, and Λ_r that satisfy (1)-(6), we define

$$f(t,\, d,\, \tau,\, \Lambda_p,\, \Lambda_r) = \min |\tau \cup S| = |\tau| + \min |S|,$$

where S is a subset of C-nodes in $B(t,\, d)$ whose converters are put in use. That is, S satisfies

(7) $S \subseteq B(t,\, d) \cap C$ and therefore $\tau \cap S = \varnothing$.

Furthermore, S needs to satisfy

(8) $W_{\tau \cup S} \cup \Lambda_r$ covers $\overline{B(t,\, d)}$. This condition requires S to be sufficiently large.

(9) $\Lambda_p = W_\tau \cup (W_S \cap W_{X_t}) = W_{\tau \cup S} \cap W_{X_t}$. This condition specifies that S should not be too big.

[2] This is implied by (8)-(9).

If t, d, τ, Λ_p, and Λ_r do not satisfy (1)-(6) or if there is no S satisfying (7)-(9), then $f(t,\, d,\, \tau,\, \Lambda_p,\, \Lambda_r)$ is defined to be ∞.

In short, $|\tau| + |S|$ is a finite-value candidate for the minimum value $f(t,\, d,\, \tau,\, \Lambda_p,\, \Lambda_r)$ if and only if

$$\tau \subseteq X_t \cap C,^{3} \tag{3.3.1}$$

$$S \subseteq B(t,\, d) \cap C,^{4} \tag{3.3.2}$$

$$\Lambda_p = W_\tau \cup (W_S \cap W_{X_t}),^{5} \tag{3.3.3}$$

$$\Lambda_r \subseteq W_{X_t},^{6} \tag{3.3.4}$$

$$\Lambda_r \cap \Lambda_p = \varnothing,^{7} \tag{3.3.5}$$

$$W_{\tau \cup S} \cup \Lambda_r \text{ covers } \overline{B(t,\, d)}.^{8} \tag{3.3.6}$$

3.4 Theorems

When $d = 0$, we calculate $f(t,\, d, \tau,\, \Lambda_p,\, \Lambda_r)$ by Lemma 3.2. When $d \geq 1$, we use Theorem 3.4. The solution to the minimum cardinality dominating set problem is

---

[3] Listed as (3).
[4] Listed as (7).
[5] Listed as (8), which implies (4).
[6] Listed as (5).
[7] Listed as (6.2).
[8] Listed as (9). Combining Equations (3.3.3) and (3.3.6), we have (6.1).

provided by Theorem 3.8.

Lemma 3.2. For $d = 0$, we have

$$f(t,\,0,\tau,\,\Lambda_p,\,\Lambda_r) = \begin{cases} |\tau| & \text{if } \tau \subseteq X_t \cap C,\ \Lambda_p = W_\tau,\ \Lambda_r \subseteq W_{X_t}, \\ & \quad \Lambda_p \cap \Lambda_r = \varnothing, \text{ and } \Lambda_p \cup \Lambda_r \text{ covers } X_t; \\ \infty & \text{otherwise.} \end{cases}$$

Proof. This assertion follows from the following facts:

(1) $B(t,\,0) = \varnothing$, which implies that $S = \varnothing$;

(2) $\overline{B(t,\,0)} = X_t$.

□

Example 3.3. In the network shown in Figure 3.6, let $t = 1$. Then $X_t = \{b,\,c,\,d\}$ and $W_{X_t} = \{1,\,2,\,3\}$.

Take $d = 0$. Then $\overline{B(1,\,0)} = X_1$ and $B(1,\,0) = \varnothing$, which imply that the converters used in X_1 are exactly those used in $\overline{B(1,\,0)}$.

Suppose that we do not use any converter in X_1. Then $\tau = \varnothing$ and so $\Lambda_p = \varnothing$. If wavelengths 1 and 3 are activated from some converters outside $\overline{B(t,\,d)}$, that is, $\Lambda_r = \{1,\,3\}$, then all the nodes in X_1 can be covered. Hence, we have $f(1,\,0,\,\varnothing,\,\varnothing,\,\{1,\,3\}) = 0$.

Suppose that the converters at b and c are put in use. Then $\tau = \{b,\,c\}$ and so $W_\tau = \{1,\,2,\,3\}$. By the specification of Λ_p stated in (4), we have

$$\{1,\,2,\,3\} = W_\tau \subseteq \Lambda_p \subseteq W_{X_t} = \{1,\,2,\,3\}.$$

Hence $\Lambda_p = \{1, 2, 3\}$ and covers every nodes in X_t. Therefore, we do not need any wavelength outside $\overline{B(t, d)}$. Equivalently, $\Lambda_r = \varnothing$. Then $f(1, 0, \{b, c\}, \{1, 2, 3\}, \varnothing) = |\{b, c\}| = 2$ follows.

Similarly, we have

$f(2, 0, \{c\}, \{1, 2\}, \{3\}) = 1$ and

$f(2, 0, \{c, e\}, \{1, 2, 3\}, \varnothing) = 2.$

Now we find the value of $f(2, 0, \{c\}, \{1, 2, 3\}, \varnothing)$.

Since $\tau = \{c\}$, we have $W_\tau = \{1, 2\}$, which is not equal to $\Lambda_p = \{1, 2, 3\}$, by Lemma 3.2, we have

$f(2, 0, \{c\}, \{1, 2, 3\}, \varnothing) = \infty.$

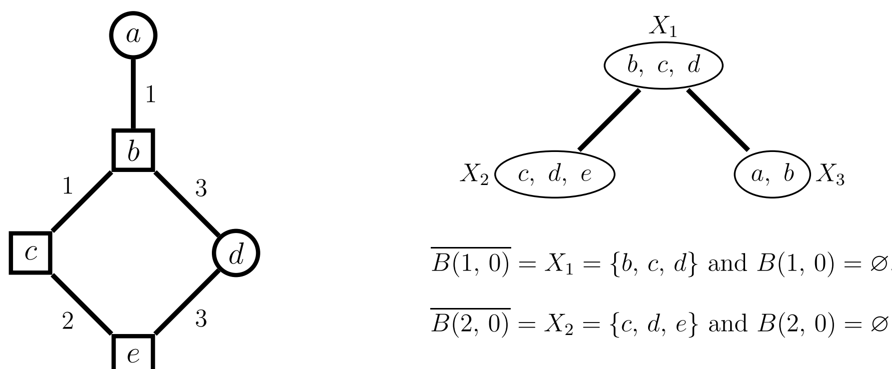


FIGURE 3.6: Calculation of $f(t, d, \tau, \Lambda_p, \Lambda_r)$ when $d = 0$.

Theorem 3.4. For any given $\tau \subseteq X_t$ and $\Lambda_p, \Lambda_r \subseteq W_{X_t}$, when $d \geq 1$, we have

$$f(t, d, \tau, \Lambda_p, \Lambda_r) = \min\{ f(t, d-1, \tau, \Lambda''_p, \Lambda''_r) + f(t_d, c(t_d), \tau', \Lambda'_p, \Lambda'_r) - |\tau \cap \tau'| \}, \tag{3.4.1}$$

where the minimum is taken over all τ', Λ'_p, Λ'_r, Λ''_p, Λ''_r such that

$$\tau \cap X_{t_d} = \tau' \cap X_t, \text{[9]} \tag{3.4.2}$$

$$\Lambda'_r \subseteq W_{X_t}, \text{[10]} \tag{3.4.3}$$

$$\Lambda_p = \Lambda''_p \cup (\Lambda'_p \cap W_{X_t}), \tag{3.4.4}$$

$$\Lambda_r = (\Lambda'_r \cup \Lambda''_r) - \Lambda_p. \tag{3.4.5}$$

Note that there can be several possible τ' to satisfy Equation (3.4.2), and for each τ', there can be several combinations of Λ'_p, Λ''_p, Λ'_r, and Λ''_r to fulfill Equations (3.4.4) and (3.4.5).

Example 3.5. Equation (3.4.2) specifies which $\tau \subseteq X_t$ and $\tau' \subseteq X_{t_d}$ can be combined together in Equation (3.4.1). See Figure 3.7.

Example 3.6. The following examples illustrates Theorem 3.4.

Consider the tree decomposition in Figure 3.8(ii). let $t = 1$ and $d = 1$. Then $X_t = \{b, c, d\}$ and $X_{t_d} = \{c, d, e\}$, When $\tau = \{c\}$, to satisfy Equation (3.4.2), either $\tau' = \{c\}$ or $\tau' = \{c, e\}$.

Furthermore, take $\Lambda_p = \{1, 2, 3\}$ and $\Lambda_r = \varnothing$. The feasible combinations of Λ'_p, Λ''_p, Λ'_r, and Λ''_r are listed in the following.

---

[9] This is a simplification of $\tau \cap (X_{t_d} \cap X_t) = \tau' \cap (X_{t_d} \cap X_t)$, which means the set of C-nodes in $X_{t_d} \cap X_t$ whose converters are put-in-use.

[10] To reach nodes in $\overline{B(t_d,\, c(t_d))}$, a wavelength activated from converters outside $\overline{B(t_d,\, c(t_d))}$ must be in W_{X_t}.

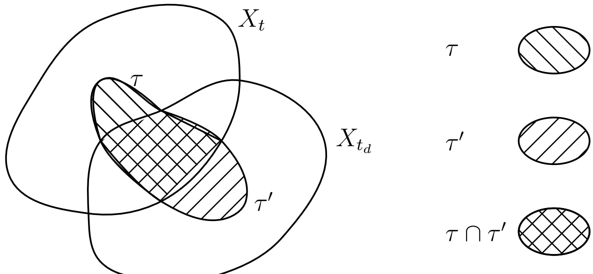
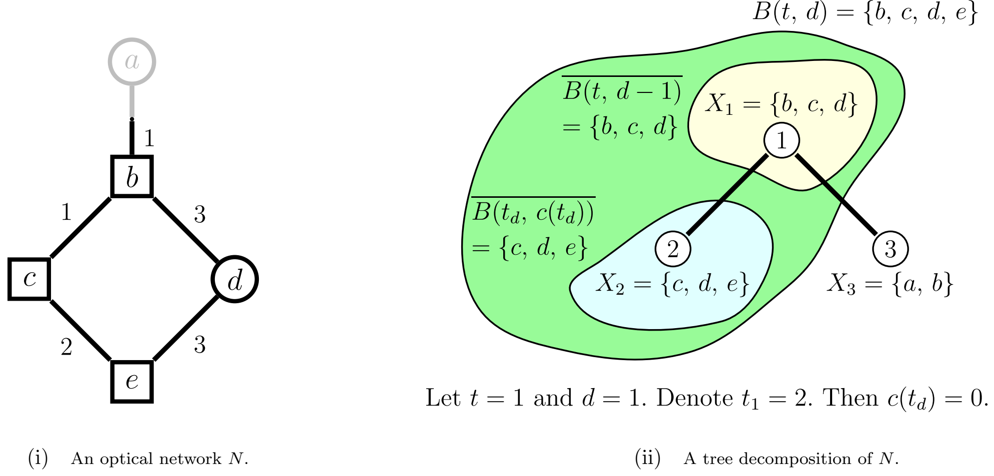


FIGURE 3.7: An illustration of $\tau \subseteq X_t$, $\tau' \subseteq X_{t_d}$, and $\tau \cap \tau' = \tau \cap X_{t_d} = \tau' \cap X_t$.

A possible combination is $\tau' = \{c\}$, $\Lambda'_p = \{1, 2\}$, $\Lambda'_r = \{3\}$, $\Lambda''_p = \{1, 2, 3\}$, and $\Lambda''_r = \varnothing$, which contribute to a value of $f(1, 0, \{b, c\}, \{1, 2, 3\}, \varnothing) + f(2, 0, \{c\}, \{1, 2\}, \{3\}) - |\{b, c\} \cap \{c\}| = 2 + 1 - 1 = 2$.

Similarly, $f(1, 0, \{b, c\}, \{1, 2, 3\}, \varnothing) + f(2, 0, \{c, e\}, \{1, 2, 3\}, \varnothing) - |\{b, c\} \cap \{c, e\}| = 2 + 2 - 1$.

The above two are the possible finite-value candidates for $f(1, 1, \{b, c\}, \{1, 2, 3\}, \varnothing)$, whose value equals $\min\{2, 3\} = 2$.



(i) An optical network N.

(ii) A tree decomposition of N.

FIGURE 3.8: An illustration of Theorem 3.4.

Next we shall prove Theorem 3.4 by applying Lemma 3.7.

Lemma 3.7. Given two linearly ordered sets X and Y, if for any $x \in X$, there is a $y \in Y$ such that $x \geq y$, then $\min X \geq \min Y$.

Proof. It is clear. □

We now prove Theorem 3.4.

First, we shall prove

$$f(t, d, \tau, \Lambda_p, \Lambda_r) \geq \min\{ f(t, d-1, \tau, \Lambda''_p, \Lambda''_r) + f(t_d, c(t_d), \tau', \Lambda'_p, \Lambda'_r) - |\tau \cap \tau'| \}.$$

Given a finite-value candidate $|\tau| + |S|$ for $f(t, d, \tau, \Lambda_p, \Lambda_r)$, that is, τ, S, Λ_p, and Λ_r satisfy Equations (3.3.1)-(3.3.6). We shall prove that $|\tau| + |S|$ is greater than or equal to some element in the set $\{ f(t, d-1, \tau, \Lambda''_p, \Lambda''_r) + f(t_d, c(t_d), \tau', \Lambda'_p, \Lambda'_r) - |\tau \cap \tau'| \}$.

Let $\{\tau', S', \Lambda'_p, \Lambda'_r\}$ and $\{S'', \Lambda''_p, \Lambda''_r\}$ be defined as follows. See Figures 3.9 – 3.11.

$$\tau' = (\tau \cup S) \cap X_{t_d}, \tag{3.4.6}$$

$$S' = S \cap B(t_d, c(t_d)), \tag{3.4.7}$$

$$\Lambda'_p = W_{\tau'} \cup (W_{S'} \cap W_{X_{t_d}}), \tag{3.4.8}$$

$$\Lambda'_r = (\Lambda_r \cup \Lambda''_p) \cap W_{X_{t_d}} - \Lambda'_p. \tag{3.4.9}$$

$$S'' = S \cap B(t,\, d-1), \tag{3.4.10}$$

$$\Lambda''_p = W_\tau \cup (W_{S''} \cap W_{X_t}), \tag{3.4.11}$$

$$\Lambda''_r = \Lambda_r \cup (\Lambda'_p \cap W_{X_t}) - \Lambda''_p. \tag{3.4.12}$$

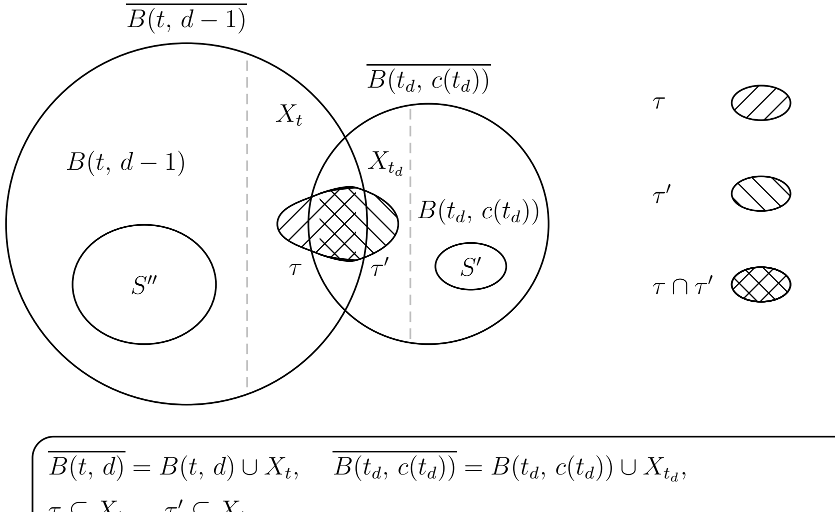


FIGURE 3.9: Relationships among variables.

Step 1: We show that Equations (3.4.2)-(3.4.5) hold.

Proof. Equation (3.4.2) follows from Equation (3.4.6) and $S \cap X_t = \varnothing$.

Equation (3.4.3) holds due to Equations (3.4.9), together with $\Lambda_r \subseteq W_{X_t}$ and $\Lambda''_p \subseteq W_{X_t}$.

Next, we verify that Equation (3.4.4) holds.

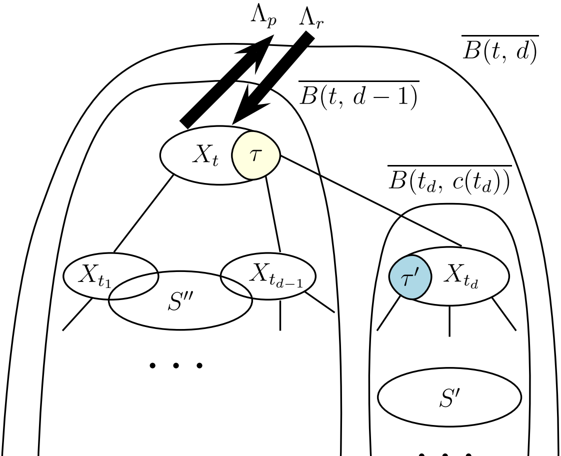


FIGURE 3.10: An illustration of $\Lambda_p = W_\tau \cup (W_S \cap W_{X_t})$, where $S = S' \cup S'' \cup (\tau' - \tau)$, $\Lambda_r \subseteq W_{X_t}$, and $\Lambda_p \cap \Lambda_r = \varnothing$.

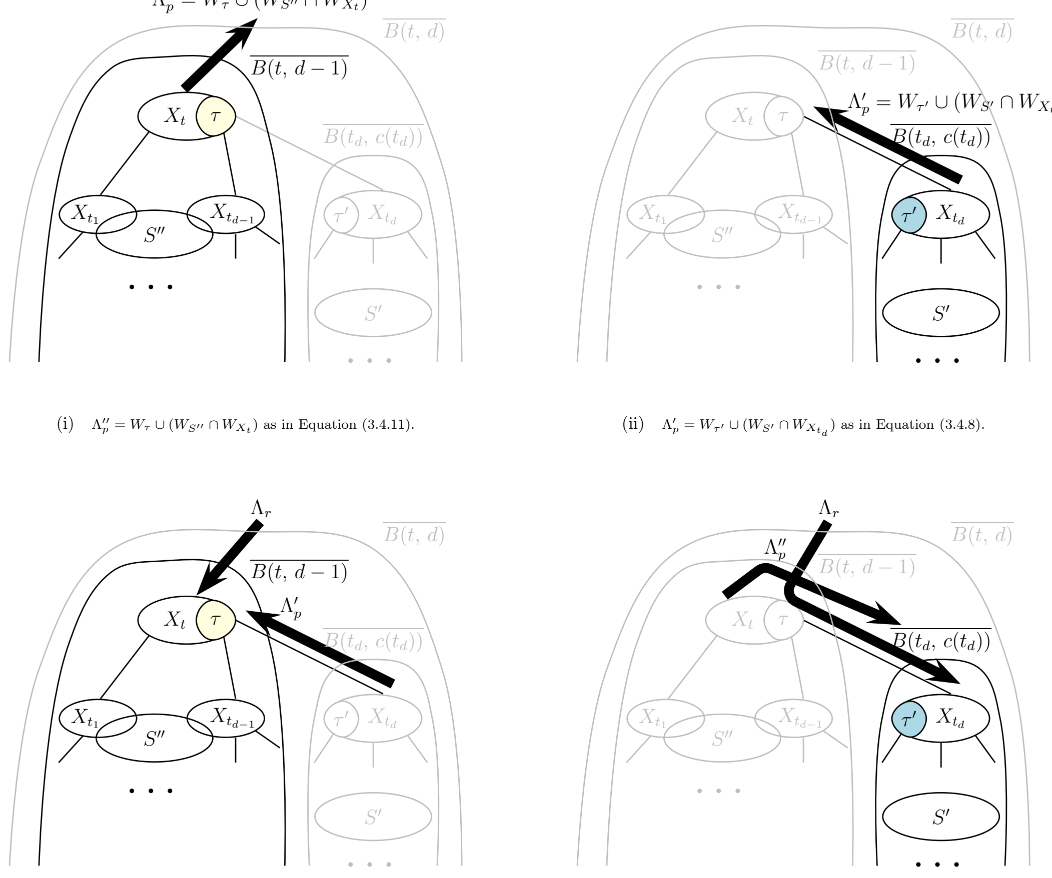
By Equation (3.3.3), we have $\Lambda_p = (W_\tau \cup W_S) \cap W_{X_t}$. Partitioning S into S', S'', and $\tau' - \tau$, we have $\Lambda_p = (W_\tau \cup W_{\tau'-\tau} \cup W_{S'} \cup W_{S''}) \cap W_{X_t}$, which can be rewritten as $(W_{\tau \cup S''} \cap W_{X_t}) \cup (W_{\tau' \cup S'} \cap W_{X_t})$. According to Equation (3.4.11), we have $\Lambda_p'' = W_{\tau \cup S''} \cap W_{X_t}$, thus $\Lambda_p = \Lambda_p'' \cup (W_{\tau' \cup S'} \cap W_{X_t})$.

It remains to show that $W_{\tau' \cup S'} \cap W_{X_t}$ equals $\Lambda_p' \cap W_{X_t}$. Since X_{t_d} contains the vertex-cut from $v \in \overline{B(t_d,\, c(t_d))}$ to a node $u \in X_t$ (as illustrated in Example 1.1), we assert that, some node of X_{t_d} must be located in each $u, v-$path. Thus, whenever wavelength $\lambda \in W_{\tau' \cup S'} \cap W_{X_t}$, we have $\lambda \in W_{X_{t_d}}$. Hence $W_{\tau' \cup S'} \cap W_{X_t} \subseteq W_{X_{t_d}}$, which implies that $W_{\tau' \cup S'} \cap W_{X_t} = W_{\tau' \cup S'} \cap W_{X_{t_d}} \cap W_{X_t}$.

By Equation (3.4.8), $\Lambda_p' = W_{\tau' \cup S'} \cap W_{X_{t_d}}$, and so $W_{\tau' \cup S'} \cap W_{X_t} = \Lambda_p' \cap W_{X_t}$ follows.

We illustrate Equation (3.4.5) in Figure 3.12 and then give a formal proof as follows.

We shall first prove that $\Lambda_r \cap \Lambda_p' = \varnothing$. By Equation (3.3.5), we have $\Lambda_r \cap \Lambda_p = \varnothing$. From Equation (3.4.4), we get $\Lambda_p' \cap W_{X_t} \subseteq \Lambda_p$. Hence $\Lambda_p' \cap W_{X_t} \cap \Lambda_r = \varnothing$. Note that,



(i) $\Lambda''_p = W_\tau \cup (W_{S''} \cap W_{X_t})$ as in Equation (3.4.11).

(ii) $\Lambda'_p = W_{\tau'} \cup (W_{S'} \cap W_{X_{t_d}})$ as in Equation (3.4.8).

(iii) $\Lambda''_r = \Lambda_r \cup (\Lambda'_p \cap W_{X_t}) - \Lambda''_p$ as in Equation (3.4.12).

(iv) $\Lambda'_r = (\Lambda_r \cup \Lambda''_p) \cap W_{X_{t_d}} - \Lambda'_p$ as in Equation (3.4.9).

FIGURE 3.11: An illustration of Λ'_p, Λ'_r, Λ''_p, and Λ''_r.

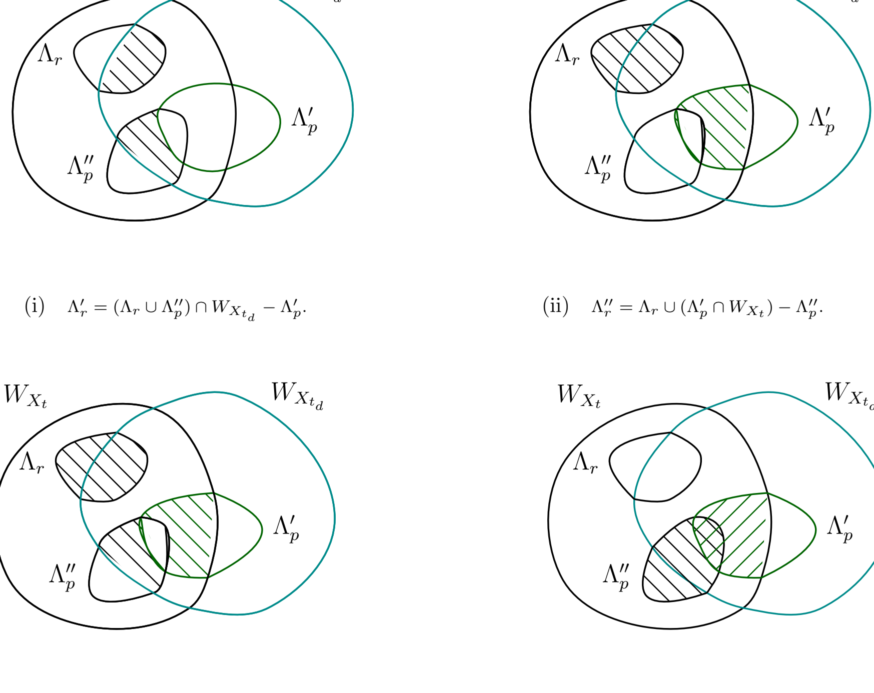


FIGURE 3.12: An illustration of $\Lambda_r = (\Lambda'_r \cup \Lambda''_r) - \Lambda_p$.

by Equation (3.3.4), we have $\Lambda_r \subseteq W_{X_t}$. Therefore $\Lambda'_p \cap W_{X_t} \cap \Lambda_r = \Lambda'_p \cap \Lambda_r = \varnothing$.

Also by Equation (3.4.4), we assert that $\Lambda_r \cap \Lambda''_p = \varnothing$.

Next we shall prove that $\Lambda_r = (\Lambda'_r \cup \Lambda''_r) - \Lambda_p$.

Replacing Λ'_r with $(\Lambda_r \cup \Lambda''_p) \cap W_{X_{t_d}} - \Lambda'_p$ and substituting Λ''_r for $\Lambda_r \cup (\Lambda'_p \cap W_{X_t}) - \Lambda''_p$, we have

$$(\Lambda'_r \cup \Lambda''_r) - \Lambda_p = ((\Lambda_r \cup \Lambda''_p) \cap W_{X_{t_d}} - \Lambda'_p) \cup (\Lambda_r \cup (\Lambda'_p \cap W_{X_t}) - \Lambda''_p) - \Lambda_p.$$

Rewrite the above equation as

$$(\Lambda'_r \cup \Lambda''_r) - \Lambda_p = (\Lambda_r \cap W_{X_{t_d}} - \Lambda'_p) \cup (\Lambda''_p \cap W_{X_{t_d}} - \Lambda'_p) \cup (\Lambda_r - \Lambda''_p) \cup (\Lambda'_p \cap W_{X_t} - \Lambda''_p) - \Lambda_p.$$

We now simplify $(\Lambda_r \cap W_{X_{t_d}} - \Lambda'_p) \cup (\Lambda_r - \Lambda''_p)$ in the right-hand side of the above equation. By Equation (3.4.4), we have $\Lambda_r \cap \Lambda''_p = \varnothing$ and so $\Lambda_r - \Lambda''_p = \Lambda_r$. Applying

$$(\Lambda_r \cap W_{X_{t_d}} - \Lambda'_p) \cup (\Lambda_r - \Lambda''_p) = (\Lambda_r \cap W_{X_{t_d}} - \Lambda'_p) \cup \Lambda_r = \Lambda_r,$$

we have

$$(\Lambda'_r \cup \Lambda''_r) - \Lambda_p = \Lambda_r \cup (\Lambda''_p \cap W_{X_{t_d}} - \Lambda'_p) \cup (\Lambda'_p \cap W_{X_t} - \Lambda''_p) - \Lambda_p.$$

By Equation (3.3.5), we have $\Lambda_r \cap \Lambda_p = \varnothing$.

Furthermore, $\Lambda''_p \cap W_{X_{t_d}} - \Lambda'_p \subseteq \Lambda''_p \subseteq \Lambda_p$ and

$\Lambda'_p \cap W_{X_t} - \Lambda''_p \subseteq \Lambda'_p \cap W_{X_t} \subseteq \Lambda_p$ holds.

Thus we have $(\Lambda'_r \cup \Lambda''_r) - \Lambda_p = \Lambda_r$. □

Step 2: Show that $|\tau|+|S''|$ is a finite-value candidate for $f(t,\, d-1,\, \tau,\, \Lambda''_p,\, \Lambda''_r)$. That is, we need to verify Properties (3.4.13)-(3.4.18).

$$\tau \subseteq X_t \cap C, \tag{3.4.13}$$

$$S'' \subseteq B(t,\, d-1) \cap C, \tag{3.4.14}$$

$$\Lambda''_p = W_\tau \cup (W_{S''} \cap W_{X_t}), \tag{3.4.15}$$

$$\Lambda''_r \subseteq W_{X_t}, \tag{3.4.16}$$

$$\Lambda''_r \cap \Lambda''_p = \varnothing, \tag{3.4.17}$$

$$W_{\tau \cup S''} \cup \Lambda''_r \text{ covers } \overline{B(t,\, d-1)}. \tag{3.4.18}$$

Proof. Equation (3.4.13) holds by the choice of τ in Equation (3.3.1).

Equation (3.4.14) is implied by Equations (3.3.2) and (3.4.10).

Equation (3.4.15) follows from Equation (3.4.11).

Equation (3.4.16) follows from Equations (3.4.3) and (3.4.12).

Equation (3.4.17) is implied by Equation (3.4.12).

To verify Property (3.4.18), it suffices to prove that, for each $v \in \overline{B(t,\, d-1)}$, if node v is not covered by $W_{\tau \cup S''}$, then v is covered by Λ''_r.

By assumption, v is not covered by $W_{\tau \cup S''}$, hence v is covered by $W_{\tau' \cup S'} \cup \Lambda_r$. Furthermore, since $\Lambda''_p \subseteq W_{\tau \cup S''}$, we learn that v is not covered by Λ''_p. Said

differently, v is covered by $W_{\tau' \cup S'} \cup \Lambda_r - \Lambda''_p$, which can be written as $(W_{\tau' \cup S'} - \Lambda''_p) \cup (\Lambda_r - \Lambda''_p)$. There are two possibilities, considered in the following.

(i) Suppose that v is covered by $\Lambda_r - \Lambda''_p$. By Equation (3.4.12), we learn that $\Lambda_r - \Lambda''_p$ is a subset of Λ''_r. Hence v is covered by Λ''_r.

(ii) Suppose that v is covered by some wavelength $\lambda \in W_{\tau' \cup S'} - \Lambda''_p$. Then v is dominated by some node $u \in \tau' \cup S'$. Note that X_{t_d} contains a vertex-cut between u and v, hence a node in X_{t_d} must be located in every $u, v-$path. That is, $\lambda \in W_{X_{t_d}}$. Similarly, $\lambda \in W_{X_t}$. Thus, we have $\lambda \in (W_{\tau' \cup S'} - \Lambda''_p) \cap W_{X_{t_d}} \cap W_{X_t}$. So v is covered by $(W_{\tau' \cup S'} - \Lambda''_p) \cap W_{X_{t_d}} \cap W_{X_t}$, which equals $W_{\tau' \cup S'} \cap W_{X_{t_d}} \cap W_{X_t} - \Lambda''_p$. Note that $W_{\tau' \cup S'} \cap W_{X_{t_d}} = \Lambda'_p$. Hence v is covered by $\Lambda'_p \cap W_{X_t} - \Lambda''_p \subseteq \Lambda''_r$.

In short, $\overline{B(t,\, d-1)}$ is covered by $W_{\tau \cup S''} \cup \Lambda''_r$. $\square$

Step 3: Show that $|\tau'| + |S'|$ is a candidate for $f(t_d,\, c(t_d),\, \tau',\, \Lambda'_p,\, \Lambda'_r)$. That is, we need to verify Properties (3.4.19)-(3.4.24).

$$\tau' \subseteq X_{t_d} \cap C, \tag{3.4.19}$$

$$S' \subseteq B(t_d,\, c(t_d)) \cap C, \tag{3.4.20}$$

$$\Lambda'_p = W_{\tau'} \cup (W_{S'} \cap W_{X_{t_d}}), \tag{3.4.21}$$

$$\Lambda'_r \subseteq W_{X_{t_d}}, \tag{3.4.22}$$

$$\Lambda_r' \cap \Lambda_p' = \varnothing, \tag{3.4.23}$$

$$W_{\tau' \cup S'} \cup \Lambda_r' \text{ covers } \overline{B(t_d,\, c(t_d))}. \tag{3.4.24}$$

Proof. Equation (3.4.19) is implied by Equation (3.4.6) and $\tau \cup S \subseteq C$.

Equation (3.4.20) is implied by Equation (3.4.7) and $S \subseteq C$.

Equation (3.4.21) follows from Equation (3.4.8).

Equations (3.4.22) and (3.4.23) are implied by Equation (3.4.9).

To verify Property (3.4.24), it suffices to prove that, for each $v \in \overline{B(t_d,\, c(t_d))}$, if node v is not covered by $W_{\tau' \cup S'}$, then v is covered by Λ_r'.

By assumption, v is not covered by $W_{\tau' \cup S'}$, hence v is covered by $W_{\tau \cup S''} \cup \Lambda_r$. Furthermore, since $\Lambda_p' \subseteq W_{\tau' \cup S'}$, we learn that v is not covered by Λ_p'. Said differently, v is covered by $W_{\tau \cup S''} \cup \Lambda_r - \Lambda_p'$, which can be written as $(W_{\tau \cup S''} - \Lambda_p') \cup (\Lambda_r - \Lambda_p')$. There are two possibilities, considered in the following.

(i) If v is covered by some wavelength $\lambda \in \Lambda_r - \Lambda_p'$, then $\lambda \in W_{X_{t_d}}$; otherwise, wavelength λ cannot reach $v \in \overline{B(t_d,\, c(t_d))}$. Therefore, node v is covered by $(\Lambda_r - \Lambda_p') \cap W_{X_{t_d}} \subseteq \Lambda_r \cap W_{X_{t_d}} - \Lambda_p' \subseteq \Lambda_r'$.

(ii) Suppose that v is covered by some wavelength $\lambda \in W_{\tau \cup S''} - \Lambda_p'$. Then v is dominated by some node $u \in \tau \cup S''$. Note that X_t contains a vertex-cut between nodes u and v, hence a node in X_t must be located in any u, v−path. That is, $\lambda \in W_{X_t}$. Similarly, $\lambda \in W_{X_{t_d}}$. Thus, we have $\lambda \in (W_{\tau \cup S''} - \Lambda_p') \cap W_{X_t} \cap W_{X_{t_d}}$. So v is covered by $(W_{\tau \cup S''} - \Lambda_p') \cap W_{X_t} \cap W_{X_{t_d}}$, which equals $W_{\tau \cup S''} \cap W_{X_t} \cap W_{X_{t_d}} - \Lambda_p'$. Note that $W_{\tau \cup S''} \cap W_{X_t} = \Lambda_p''$. Hence v is covered by $\Lambda_p'' \cap W_{X_{t_d}} - \Lambda_p' \subseteq \Lambda_r'$.

In summary, $\overline{B(t_d,\, c(t_d))}$ is covered by $W_{\tau' \cup S'} \cup \Lambda'_r$. $\square$

Step 4: Verify that $|\tau| + |S|$ is greater than or equal to some element in set $\{\, f(t,\, d-1,\, \tau,\, \Lambda''_p,\, \Lambda''_r) \,+\, f(t_d,\, c(t_d),\, \tau',\, \Lambda'_p,\, \Lambda'_r) \,-\, |\tau \cap \tau'| \,\}$.

We shall show first that

$$|\tau| + |S| = |\tau| + |S''| + |\tau'| + |S'| - |\tau \cap \tau'|.$$

Proof.

$$\begin{aligned} & |\tau| + |S| \\ = \;& |\tau| + |S'| + |S''| + |\tau' - \tau| \\ = \;& |\tau \cup \tau'| + |S'| + |S''| \\ = \;& |\tau| + |S''| + |\tau'| + |S'| - |\tau \cap \tau'|. \end{aligned}$$

From **Steps 2** and **3**, we learn that $|\tau| + |S''| \geq f(t,\, d-1,\, \tau,\, \Lambda''_p,\, \Lambda''_r)$ and $|\tau'| + |S'| \geq f(t_d,\, c(t_d),\, \tau',\, \Lambda'_p,\, \Lambda'_r)$. Thus, we have

$|\tau| + |S| \geq f(t,\, d-1,\, \tau,\, \Lambda''_p,\, \Lambda''_r) + f(t_d,\, c(t_d),\, \tau',\, \Lambda'_p,\, \Lambda'_r) - |\tau \cap \tau'|$. $\square$

Hence, by **Steps 1-4**, we assert that

$$f(t,\, d,\, \tau,\, \Lambda_p,\, \Lambda_r) \geq \min\{\, f(t,\, d-1,\, \tau,\, \Lambda''_p,\, \Lambda''_r) + f(t_d,\, c(t_d),\, \tau',\, \Lambda'_p,\, \Lambda'_r) - |\tau \cap \tau'| \,\}.$$

To finish the proof of Theorem 3.4, it remains to show that

$$f(t,\, d,\, \tau,\, \Lambda_p,\, \Lambda_r) \leq \min\{\, f(t,\, d-1,\, \tau,\, \Lambda''_p,\, \Lambda''_r) + f(t_d,\, c(t_d),\, \tau',\, \Lambda'_p,\, \Lambda'_r) - |\tau \cap \tau'| \,\}.$$

Let $|\tau| + |S''|$ be a solution of $f(t, d-1, \tau, \Lambda''_p, \Lambda''_r)$. Then τ and S'' satisfy Properties (3.4.13)-(3.4.18).

Let $|\tau'| + |S'|$ be a solution of $f(t_d, c(t_d), \tau', \Lambda'_p, \Lambda'_r)$. Then τ and S'' satisfy Properties (3.4.19)-(3.4.24).

Note that $|\tau| + |S''|$ and $|\tau'| + |S'|$ need to satisfy Equations (3.4.2)-(3.4.5) to be combined together to generate $f(t, d, \tau, \Lambda_p, \Lambda_r)$.

We construct S as follows,

$$S = S' \cup S'' \cap (\tau' - \tau). \tag{3.4.25}$$

Step 5: Show that $|\tau| + |S|$ is a candidate for $f(t, d, \tau, \Lambda_p, \Lambda_r)$. That is, we need to verify Equations (3.3.1)-(3.3.6).

Proof. Equation (3.3.1) follows from Equation (3.4.13).

We shall prove that Equation (3.3.2) holds.

By Equation (3.4.25), we have $S = S' \cup S'' \cup (\tau' - \tau)$, where

$S' \subseteq B(t_d, c(t_d)) \cap C \subseteq B(t, d) \cap C$,

$S'' \subseteq B(t, d-1) \cap C \subseteq B(t, d) \cap C$, and

$\tau' - \tau \subseteq B(t, d) \cap C$,

we conclude that $S \subseteq B(t, d) \cap C$.

We shall prove that Equation (3.3.3) holds.

By Equation (3.4.4), we have $\Lambda_p = \Lambda''_p \cup (\Lambda'_p \cap W_{X_t})$. By Equation (3.4.15), we get $\Lambda''_p = W_\tau \cup (W_{S''} \cap W_{X_t})$. By Equation (3.4.21), we learn that $\Lambda'_p = W_{\tau'} \cup$

$(W_{S'} \cap W_{X_{t_d}})$. Thus

$$\begin{aligned}\Lambda_p &= W_\tau \cup (W_{S''} \cap W_{X_t}) \cup ((W_{\tau'} \cup (W_{S'} \cap W_{X_{t_d}})) \cap W_{X_t}) \\ &= W_\tau \cup ((W_{S''} \cup W_{\tau'} \cup (W_{S'} \cap W_{X_{t_d}})) \cap W_{X_t}) \\ &= W_\tau \cup ((W_{S''} \cup W_{\tau'}) \cap W_{X_t}) \cup (W_{S'} \cap W_{X_{t_d}} \cap W_{X_t}).\end{aligned}$$

Note that $W_{S'} \cap W_{X_{t_d}} \cap W_{X_t} = W_{S'} \cap W_{X_t}$. Hence

$$\begin{aligned}\Lambda_p &= W_\tau \cup ((W_{S''} \cup W_{\tau'}) \cap W_{X_t}) \cup (W_{S'} \cap W_{X_t}) \\ &= W_\tau \cup ((W_{S''} \cup W_{\tau'} \cup W_{S'}) \cap W_{X_t}).\end{aligned}$$

Since $\tau \subseteq X_t$, we have $W_\tau \subseteq W_{X_t}$. Therefore

$$\begin{aligned}\Lambda_p &= (W_\tau \cup W_{S''} \cup W_{\tau'} \cup W_{S'}) \cap W_{X_t} \\ &= (W_\tau \cup W_{\tau'-\tau} \cup W_{S'} \cup W_{S''}) \cap W_{X_t} \\ &= W_\tau \cup (W_{\tau'-\tau} \cup W_{S'} \cup W_{S''}) \cap W_{X_t} \\ &= W_\tau \cup (W_S \cap W_{X_t}).\end{aligned}$$

Equation (3.3.4) follows from Equations (3.4.5), (3.4.3), and (3.4.16).

Equation (3.3.5) is implied by Equation (3.4.5).

It remains to prove Property (3.3.6).

For any $v \in \overline{B(t,\, d)}$, if $v \in \overline{B(t,\, d-1)}$, then v is covered by $W_{\tau \cup S''} \cup \Lambda''_r$; if $v \in \overline{B(t_d,\, c(t_d))}$, then v is covered by $W_{\tau' \cup S'} \cup \Lambda'_r$. Hence $W_{\tau \cup S''} \cup \Lambda''_r \cup W_{\tau' \cup S'} \cup \Lambda'_r$

covers $\overline{B(t,\, d)}$.

Note that $\Lambda_r = \Lambda'_r \cup \Lambda''_r - \Lambda_p$ and $\Lambda_p \subseteq W_{\tau\cup S''} \cup W_{\tau'\cup S'}$. Thus $W_{\tau\cup S''} \cup W_{\tau'\cup S'} \cup \Lambda'_r \cup \Lambda''_r$ equals $W_{\tau\cup S''} \cup W_{\tau'\cup S'} \cup \Lambda_r$, the latter can be further simplified as $W_{\tau\cup S} \cup \Lambda_r$.

Hence $W_{\tau\cup S} \cup \Lambda_r$ covers $\overline{B(t,\, d)}$. □

Step 6: Verify that

$$|\tau| + |S''| + |\tau'| + |S'| - |\tau \cap \tau'| = |\tau| + |S|.$$

Proof. Note that the left hand side can be rewritten as $|\tau| + |\tau' - \tau| + |S''| + |S'|$, which equals to $|\tau| + |S|$. □

Theorem 3.8. Let v be the root node in the tree decomposition and $c(v)$ be the number of children of v. If $\Lambda_p = \varnothing$ and $|\Lambda_r| = 1$, then each node can be reached by the wavelength specified in Λ_r and no converter is needed. Otherwise, the minimum number of converters needed for all-to-all communication in the network is $\min_{\Lambda_r=\varnothing}\{f(v,\, c(v),\, \tau,\, \Lambda_p,\, \Lambda_r)\}$.

Proof. We consider the following two cases.

(i) Suppose that $\Lambda_p = \varnothing$. Then no C-node used in the network. Otherwise, we would have $\Lambda_p \neq \varnothing$.

   (i-1) If $|\Lambda_r| = 1$, then all the nodes can be reached by the wavelength specified in Λ_r, and we do not need to use any C-node.

   (i-2) If $|\Lambda_r| > 1$, then we need additional C-nodes to convert the different wavelengths in Λ_r, which cannot be done.

(ii) Suppose that $\Lambda_p \neq \varnothing$.

By definition, we have $\Lambda_p \cap \Lambda_r = \varnothing$, If neither Λ_p nor Λ_r is empty, then we need additional C-nodes to activate the wavelengths in Λ_r, which cannot be done. In another word, when $\Lambda_p \neq \varnothing$, to obtain a feasible solution, we need $\Lambda_r = \varnothing$.

This asseration then follows immediately. □

Example 3.9. This example illustrates the case when $\Lambda_p = \varnothing$ in Theorem 3.8. We assume that the following Λ_p and Λ_r are for the root node of a tree decomposition.

In Figure 3.13(i), if $\Lambda_p = \varnothing$ and $|\Lambda_r| = |\{1\}| = 1$, then we do not need to use any converter in the network. This happens when every node is reached by wavelength 1.

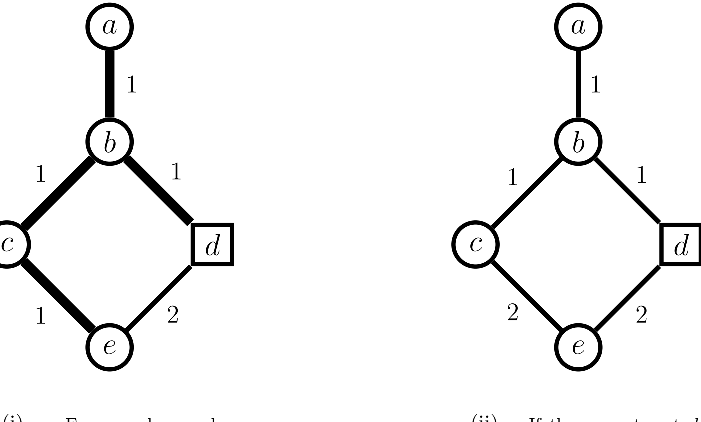
Figure 3.13(ii) differs from Figure 3.13(i) in the wavelength assigned to link (c, e). It is clear that no single wavelength can cover all the nodes in Figure 3.13(ii). Suppose that $\Lambda_p = \varnothing$ and $|\Lambda_r| = |\{1, 2\}| > 1$. On one hand, $\Lambda_p = \varnothing$ implies that no C-node in the network is put in use. On the other hand, two different wavelengths, say 1 and 2, are needed to reach all the nodes in the network. Hence, additional C-nodes are needed to convert wavelengths 1 and 2, which cannot be done. Said differently, in this network, the converter at node d must be put in use.

3.5 An Algorithm

When $d = 0$, we calculate $f(t, d, \tau, \Lambda_p, \Lambda_r)$ with Lemma 3.2. We combine the finite-value items in $B(t, d-1)$ and $B(t_d, c(t_d))$ to generate finite-value items for $B(t, d)$ by Theorem 3.4. An optimal solution is obtained from Theorem 3.8.

3.5.1 An Example

We illustrate the algorithm on the network of Figure 3.6, with infinite-value items omitted. In the solution column, we record the values of $f(t, d, \tau, \Lambda_p, \Lambda_r)$



(i) Every node can be reached by wavelength 1. No converter is needed.

(ii) If the converter at d is not used, then there is no feasible solution.

FIGURE 3.13: When $\Lambda_p = \varnothing$, a solution is feasible if and only if $|\Lambda_r| = 1$.

and $\tau \cup S$. The latter corresponds to the set of C-nodes used in $\overline{B(t,\, d)}$.

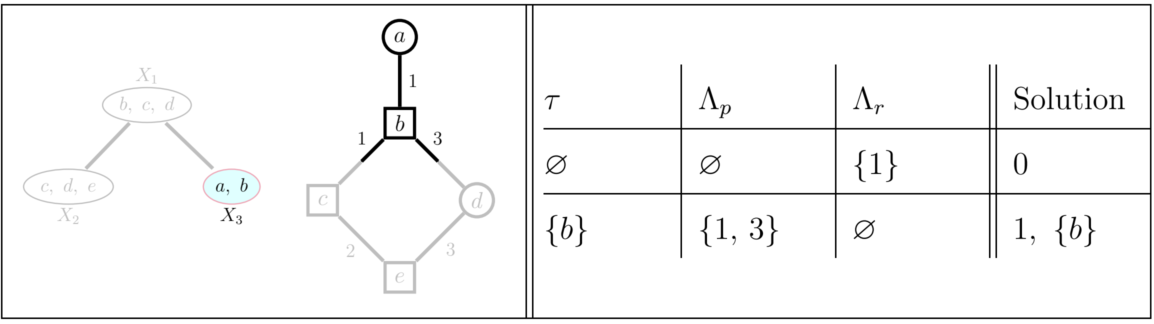
TABLE 3.1: Calculation of $f(2,\, 0,\, \tau,\, \Lambda_p,\, \Lambda_r)$.

τ	Λ_p	Λ_r	Solution
$\varnothing$	$\varnothing$	$\{1,\, 3\}$	0
$\varnothing$	$\varnothing$	$\{2,\, 3\}$	0
$\{c\}$	$\{1,\, 2\}$	$\{3\}$	$1,\ \{c\}$
$\{e\}$	$\{2,\, 3\}$	$\varnothing$	$1,\ \{e\}$
$\{c,\, e\}$	$\{1,\, 2,\, 3\}$	$\varnothing$	$2,\ \{c,\, e\}$

3.5.2 Complexity Analysis

Let $|W_t|$ be the maximum cardinality of each W_{X_t}. There are a maximum of 2^i possible sets of provided wavelength Λ_p, where $0 \leq i \leq |W_t|$. Applying the fact that $\Lambda_r \cap \Lambda_p = \varnothing$, we have at most $2^{|W_t|-i}$ possible sets Λ_r of request wavelengths. Hence, for a fixed set τ of C-nodes in each X_t, we have $\sum_{i=0}^{|W_t|} 2^{|W_t|-i} = 4^{|W_t|} - 2^{|W_t|}$

TABLE 3.2: Calculation of $f(3,\, 0,\, \tau,\, \Lambda_p,\, \Lambda_r)$.



τ	Λ_p	Λ_r	Solution
$\varnothing$	$\varnothing$	$\{1\}$	0
$\{b\}$	$\{1, 3\}$	$\varnothing$	$1,\ \{b\}$

TABLE 3.3: Calculation of $f(1,\, 0,\, \tau,\, \Lambda_p,\, \Lambda_r)$.

τ	Λ_p	Λ_r	Solution
$\varnothing$	$\varnothing$	$\{1, 3\}$	0
$\varnothing$	$\varnothing$	$\{2, 3\}$	0
$\{b\}$	$\{1, 3\}$	$\varnothing$	$1,\ \{b\}$
$\{c\}$	$\{1, 2\}$	$\{3\}$	$1,\ \{c\}$
$\{b, c\}$	$\{1, 2, 3\}$	$\varnothing$	$2,\ \{b, c\}$

TABLE 3.4: Calculation of $f(1,\ 1,\ \tau,\ \Lambda_p,\ \Lambda_r)$.

τ	Λ_p	Λ_r	Solution
$\varnothing$	$\varnothing$	$\{1,\ 3\}$	0
$\varnothing$	$\varnothing$	$\{2,\ 3\}$	0
$\varnothing$	$\varnothing$	$\{1,\ 2,\ 3\}$	0 *
$\varnothing$	$\{2,\ 3\}$	$\varnothing$	1, $\{e\}$
$\varnothing$	$\{2,\ 3\}$	$\{1\}$	1, $\{e\}$ †
$\{b\}$	$\{1,\ 3\}$	$\varnothing$	1, $\{b\}$
$\{b\}$	$\{1,\ 3\}$	$\{2\}$	1, $\{b\}$ ‡
$\{b\}$	$\{1,\ 2,\ 3\}$	$\varnothing$	2, $\{b,\ e\}$
$\{c\}$	$\{1,\ 2\}$	$\{3\}$	1, $\{c\}$
$\{c\}$	$\{1,\ 2,\ 3\}$	$\varnothing$	2, $\{c,\ e\}$
$\{b,\ c\}$	$\{1,\ 2,\ 3\}$	$\varnothing$	2, $\{b,\ c\}$

* $f(1,\ 1,\ \varnothing,\ \varnothing,\ \{1,\ 2,\ 3\}) = 0$ can be implied by $f(1,\ 1,\ \varnothing,\ \varnothing,\ \{1,\ 2\}) = 0$ or $f(1,\ 1,\ \varnothing,\ \varnothing,\ \{2,\ 3\}) = 0$.

† $f(1,\ 1,\ \varnothing,\ \{2,\ 3\},\ \{1\}\}) = 1$ can be implied by $f(1,\ 1,\ \varnothing,\ \{2,\ 3\},\ \varnothing) = 1$.

‡ $f(1,\ 1,\ \{b\},\ \{1,\ 3\},\ \{2\}\}) = 1$ can be implied by $f(1,\ 1,\ \{b\},\ \{1,\ 3\},\ \varnothing) = 1$.

TABLE 3.5: Calculation of $f(1,\ 2,\ \tau,\ \Lambda_p,\ \Lambda_r)$.

τ	Λ_p	Λ_r	Solution
$\varnothing$	$\varnothing$	$\{1,\ 3\}$	0
$\varnothing$	$\varnothing$	$\{1,\ 2,\ 3\}$	0 [§]
$\varnothing$	$\{2,\ 3\}$	$\{1\}$	1, $\{e\}$
$\{b\}$	$\{1,\ 3\}$	$\varnothing$	1, $\{b\}$
$\{b\}$	$\{1,\ 3\}$	$\{2\}$	1, $\{b\}$
$\{b\}$	$\{1,\ 2,\ 3\}$	$\varnothing$	2, $\{b,\ e\}$
$\{c\}$	$\{1,\ 2\}$	$\{3\}$	1, $\{c\}$
$\{c\}$	$\{1,\ 2,\ 3\}$	$\varnothing$	2, $\{c,\ e\}$
$\{b,\ c\}$	$\{1,\ 2,\ 3\}$	$\varnothing$	2, $\{b,\ c\}$

[§] $f(1,\ 2,\ \varnothing,\ \varnothing,\ \{1,\ 2,\ 3\}) = 0$ can be implied by $f(1,\ 2,\ \varnothing,\ \varnothing,\ \{1,\ 3\}) = 0$.

entries for possible Λ_p and Λ_r.

Let the width of a given tree decomposition be k. Then, for all t, we have $|X_t| \leq k+1$, which implies that there are 2^{k+1} choices for τ. Hence each table has $O(4^{|W_t|} \cdot 2^{k+1})$ entries. This is a theoretic analysis for the worst case. Experimental results show that there are far fewer entries.

We consider the time complexity to generate a table for $f(t,\, d,\, \tau,\, \Lambda_p,\, \Lambda_r)$ from $f(t,\, d{-}1,\, \tau,\, \Lambda''_p,\, \Lambda''_r)$ and $f(t_d,\, c(t_d),\, \tau',\, \Lambda'_p,\, \Lambda'_r)$. Since for each τ, either $\tau' = \tau \cap X_{t_d}$ or $\tau' = (\tau \cap X_{t_d}) \cup ((X_{t_d} - X_t) \cap C)$, for a fixed τ, it takes $O(4^{|W_t|} \cdot 4^{|W_t|})$ to generate an entry for $f(t,\, d,\, \tau,\, \Lambda_p,\, \Lambda_r)$, and a total of $O(4^{|W_t|} \cdot 4^{|W_t|} \cdot 2^{k+1}) = O(16^{|W_t|} \cdot 2^{k+1})$ for all τ.

In short, the total time complexity is $O(16^{|W_t|} \cdot 2^{k+1} \cdot n_t)$, where n_t is the number of tree nodes in a tree decomposition. By assumption, $|W_t|$ and k are bounded, so the time complexity is linear concerning the number of tree nodes.

3.5.3 Experiment Results

When $\tau \cap X_{t_d} = \tau' \cap X_t$, we add $f(t, d{-}1, \tau, \Lambda''_p, \Lambda''_r)$ to $f(t_d, c(t_d), \tau', \Lambda'_p, \Lambda'_r)$, and this summand serves as a candidate for $f(t, d, \tau, \Lambda'_p, \Lambda'_r)$. We call such an addition a *combination*.

Different tree decompositions may result in different number of combinations and table entries, but do not affect the size of optimal solutions. For instance, for the network shown in Subsection 3.5.1, in addition to the tree decomposition given in that Subsection, we also test on a trivial tree decomposition (in which a single tree node including all the graph nodes) and a tree decomposition consisting of $\{a, b\}$ and $\{b, c, d, e\}$. In each case, it can be seen that $\{b\}$ is an optimal solution. However, the number of table entries and combinations are different.

In the tree decomposition given in Subsection 3.5.1, we need a total of 24 table

entries and 32 combinations. For the other tree decompositions, the corresponding values are shown in the following. Though the tree decomposition in Subsection 3.5.1 needs more combinations, it is still an optimal one and is preferred, since we can break the graph into smaller pieces and study each piece throughly. In this particular example, the reason why the benefit of an optimal tree decomposition is not so obvious is that the total number of graph nodes is small (five only).

Given the tree decomposition shown in Figure 1.1(ii), we have the following output from the software we developed (see Appendix).

```
      parent = {a, b, c, d, e},  child = none

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Converters  Provide Wavelength   Request Wavelength   Partial Solution
----------------------------------------------------------------------
NULL        NULL                 {1, 3}               0
{b}         {1, 3}               NULL                 1, {b}
{c}         {1, 2}               {3}                  1, {c}
{b, c}      {1, 2, 3}            NULL                 2, {b, c}
{e}         {2, 3}               {1}                  1, {e}
{b, e}      {1, 2, 3}            NULL                 2, {b, e}
{c, e}      {1, 2, 3}            NULL                 2, {c, e}
{b, c, e}   {1, 2, 3}            NULL                 3, {b, c, e}
----------------------------------------------------------------------
Total number of table entries = 8.
Total number of combinations = 0.
```

The following is the output for the tree decomposition in Figure 1.1(iii).

```
      parent = {b, c, d, e},  child = none

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Converters  Provide Wavelength   Request Wavelength   Partial Solution
----------------------------------------------------------------------
NULL        NULL                 {1, 3}               0
NULL        NULL                 {2, 3}               0
{b}         {1, 3}               NULL                 1, {b}
{c}         {1, 2}               {3}                  1, {c}
```

```
{b, c}       {1, 2, 3}           NULL              2, {b, c}
{e}          {2, 3}              NULL              1, {e}
{b, e}       {1, 2, 3}           NULL              2, {b, e}
{c, e}       {1, 2, 3}           NULL              2, {c, e}
{b, c, e}    {1, 2, 3}           NULL              3, {b, c, e}
-----------------------------------------------------------------------

       parent = {a, b},  child = none

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Converters  Provide Wavelength   Request Wavelength   Partial Solution
-----------------------------------------------------------------------
NULL        NULL                 {1}                  0
{b}         {1, 3}               NULL                 1, {b}
-----------------------------------------------------------------------

       parent = {a, b},  child = {b, c, d, e}

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Converters  Provide Wavelength   Request Wavelength   Partial Solution
-----------------------------------------------------------------------
NULL        NULL                 {1, 3}               0
NULL        {1}                  {3}                  1, {c}
NULL        {3}                  {1}                  1, {e}
NULL        {1, 3}               NULL                 2, {c, e}
{b}         {1, 3}               NULL                 1, {b}
-----------------------------------------------------------------------
Total number of table entries = 16.
Total number of combinations = 8.
```

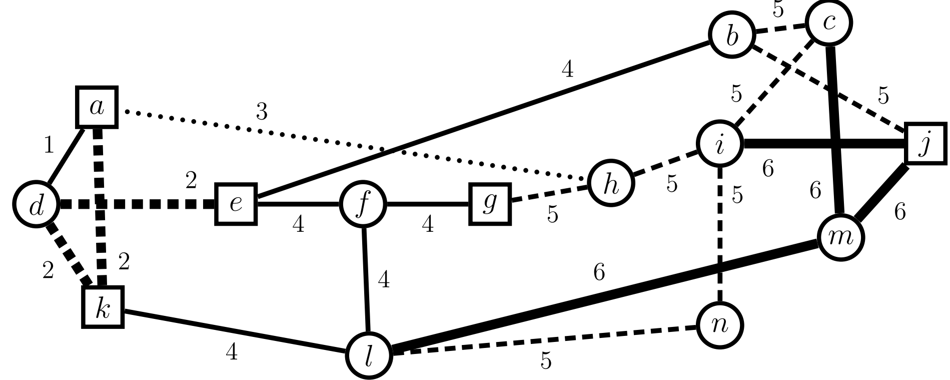
Besides the example given in the last subsection, we also work on NSF-14 network, shown in Figure 3.14(i), using tree decompositions in Figures 3.14(ii) and (iii). As a comment, the tree decomposition shown in Figure 3.14(ii) is optimal. It was proved in [2] that, a graph G has treewidth ≥ 3 if and only if G contains K_4 as a *minor* (graph H is a minor of graph G if H can be obtained from G by edge-deletions and edge-contractions). Take the induced subgraph on $\{b, c, d, e, f, g, h, i, k, l, m\}$ in Figure 3.14(i), then contract $\{e, d, k\}$ and

$\{b, c, g, h, i, m\}$, and we obtain a K_4. So the treewidth of G is at least 3. The tree decomposition we provide has width 3 and so is optimal.

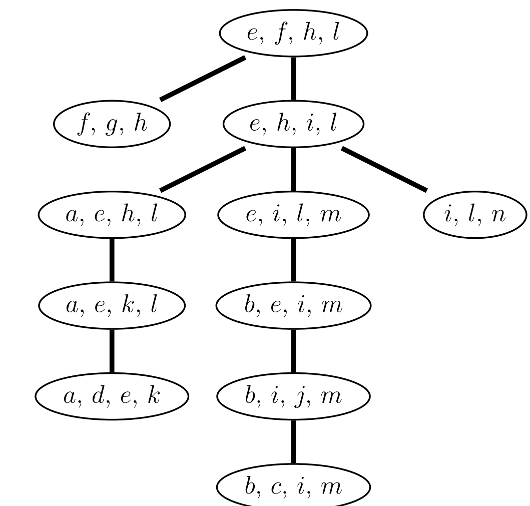
Here are some simple observations for an optimal solution. Denote the set of wavelengths on all the links incident at u as W_u. Let x and y be two converter nodes. If $W_x \subseteq W_y$, then no optimal solution can include node x. If $W_x = W_y$, then an optimal solution needs to include at most one of them. For example, in Figure 3.14(i), we have $W_e = W_k = \{2, 4\}$, and so either e or k is in an optimal solution, but not both.

Note that the addition of edge (e, k) with wavelength 2 or 4 does not affect the optimal solution, since the set of wavelengths incident at $\{e, j\}$ or $\{j, k\}$ is still $\{2, 4, 5, 6\}$, which covers all the nodes.

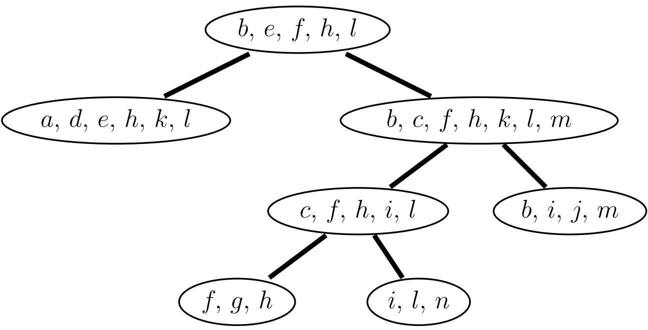
The bound given in Subsection 3.5.2 is theoretical. In practice, the total number of calculation is far less than it. For example, in the network illustrated in Figure 3.14(i), if we use a tree decomposition shown in Figure 3.14(ii), a total of 256 table entries and 567 combinations are needed to find an optimal solution. If we use a tree decomposition as in Figure 3.14(iii), a total of 102 table entries and 216 combinations are needed to find an optimal solution.



(i) NSF-14 network.



(ii) A tree decomposition of NSF-14 network.



(iii) Another tree decomposition of NSF-14 network.

FIGURE 3.14: An example of NSF-14 network with optimal solution $\{e, j\}$ and $\{j, k\}$ and two of its tree decompositions.

Chapter 4
Summary and Future Work

4.1 Summary

In this dissertation, we have presented a linear algorithm to find a wavelength assignment in tree networks such that, with the placement of a minimum number of converters, every node can send messages to all the others. The converters are restricted to a designated subset of nodes, which can be empty (that is, $C = \varnothing$), a proper subset (that is, $C \subset V$), or the whole set (that is, $C = V$). Our algorithm can assign wavelengths efficiently and effectively for one-to-one, multicast, and broadcast communication requests. For a network where the connections among nodes are known in advance (say, the 14-node NSFNET), after generating all spanning trees once using some algorithm such as the one in [25] and storing them in hardware, we can apply the above algorithm on each spanning tree and select one that needs a minimum number of converters. For general networks, the algorithm can serve as the basis of heuristic methods, as in the case of [35].

Next we focus on networks with bounded treewidth, which can be thought as a natural generalization of trees. We develop the concept of wavelength-domination. A converter wavelength-dominates a node if there is a uniform wavelength path between them. The Minimal Wavelength Dominating Set Problem (MWDSP) is to locate a minimum number of converters so that all the other nodes in the network are wavelength-dominated. We use a linear complexity dynamic programming algorithm to solve the MWDSP for networks with bounded treewidth. One such solution provides a low bound for the CUP for network with bounded treewidth.

4.2 Future Work

If we treat all the links with the same wavelength as a hyper-edge (one with two or more than two ends), then, to solve the CUP outlined in Chapter 3, we need to find a minimum connected dominating set in a hypergraph.

For a hypergraph H, we denote by $V(H)$ its node set and by $E(H)$ the set of its hyper-edges. A *branch decomposition* of a hypergraph H is a pair (T, ξ), where T is a tree with nodes of degree 1 or 3, and ξ is a bijection from $E(H)$ to the set of terminal nodes of T. The *order function* $\omega : E(T) \to 2^{V(H)}$ of a branch decompostion maps every edge e of T to a subset $\omega(e)$ of nodes, which contains all the nodes v with the following property: there are links f_1, $f_2 \in E(H)$ such that $v \in f_1 \cap f_2$ and the terminal nodes $\xi(f_1)$, $\xi(f_2)$ are in different components of $T - \{e\}$. The *width* of (T, ξ) equals $\max_{e \in E(T)} |\omega(e)|$. The *branch-width* of H is the minimum width over all branch decompositions of H. See [24].

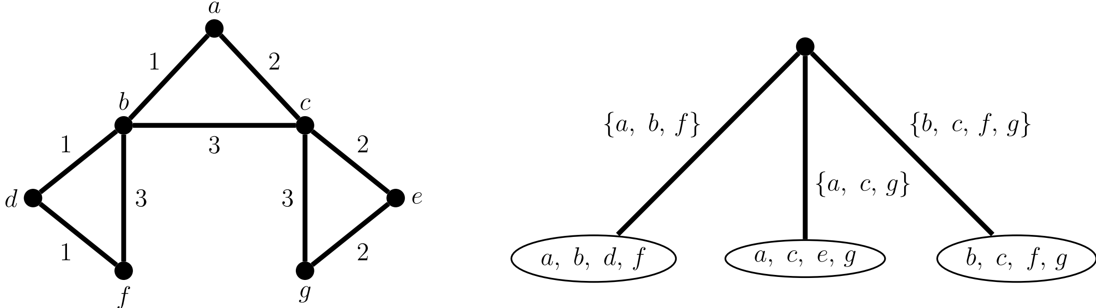


FIGURE 4.1: A hypergraph G and one of its branch-decompositions T.

The choice on solving a problem via tree decomposition or via branch decomposition normally do not affect the algorithm complexity. However, for some problems, branch-width is more suitable for actual implementations [24]. In [18],

the authors use branch-decomposition and branch-and-bound method to merge tours for graphs with more than 10,000 nodes.

We hope that, by using branch decomposition, we may be able to drop the assumption that all links in the same wavelengths form a connected subgraph. Furthermore, we want to generalize the result to the situation where the number of wavelengths for each link is unbounded.

References

[1] Jochen Alber, Michael R. Fellows, and Rolf Niedermeier, *Polynomial-time data reduction for dominating set*, Journal of the ACM (2004).

[2] Stefan Arnborg, Andrzej Proskurowski, and Derek G. Corneil, *Forbidden minors characterization of partial 3-trees*, Discrete Mathematics **80** (1990), 1–19. 3.5.3

[3] Stefan Arnborg and Andzej Proskurowski, *Linear time algorithms for NP-hard problems restricted to partial k-trees*, Discrete Applied Mathematics **23** (1989), 11–14. 1.5

[4] A. Auletta, I. Caragiannis, C. Kaklamanis, and P. Persiano, *Bandwidth allocation algorithms on tree-shaped all-optical networks with wavelength converters*, Proc. of the 4th Colloquium on Structural Information and Communication Complexity, SIROCCO, 1997, pp. 27–38.

[5] Evripidis Bampis and George N. Rouskas, *The scheduling and wavelength assignment problem in optical WDM networks*, Journal of lightwave technology **20** (2002), no. 5, 782–789.

[6] Dhritiman Banerjee and Biswanath Mukherjee, *A practical approach for routing and wavelength assignment in large wavelength-routed optical networks*, IEEE J. on Selected Areas in Communications **14** (1996), no. 5, 903–908.

[7] ______, *Wavelength-routed optical networks: linear formulation, resource budgeting tradeoffs, and a reconfiguration study*, IEEE/ACM Trans. Networking **8** (2000), no. 5, 598–607.

[8] M. W. Bern, E. L. Lawler, and A. L. Wong, *Linear time computation of optimal subgraphs of decomposable graphs*, Journal of Algorithms **8** (1987), 216–235. 1.5

[9] Dimitri Bertsekas and Robert Gallager, *Data networks*, Prentice-Hall, 1987.

[10] Hans Bodlaender, *Treewidth: algorithmic techniques and results*, Proc. 22nd International Symposium on Mathematical Foundations of Computer Science, MFCS'97, Lecture Notes in Computer Science, volume 1295, 1998, pp. 19–36. 1.4.1, 1.5

[11] Hans L. Bodlaender, *Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees*, Journal of Algorithms **11** (1990), 631–643.

[12] ______, *On reduction algorithms for graphs with small treewidth*, Workshop on Graph-Theoretic Concepts in Computer Science, 1993, pp. 45–56.

[13] Hans *L* Bodlaender, *A partial k-arboretum of graphs with bounded treewidth*, Theoretical Computer Science **209** (1998), 1–45.

[14] Bollobas, *Modern graph theory*, 1 ed., Spring, 1997.

[15] Liwei Chen and Eytan Modiano, *Efficient routing and wavelength assignment for reconfigurable WDM ring networks with wavelength converters*, IEEE/ACM Trans. Networking **13** (2005), no. 1, 173–186.

[16] Xiaowen Chu and Bo Li, *Dynamic routing and wavelength assignment in the presence of wavelength conversion for all optical networks*, IEEE/ACM Trans. Networking **13** (2005), no. 3, 704–714.

[17] Xiaowen Chu, Bo Li, and I. Chlamtac, *Wavelength converter placement for different RWA algorithms in wavelength-routed all-optical networks*, IEEE Transactions on Communications **51** (2003), 607–617.

[18] William Cook and Paul D. Seymour, *Tour merging via branch-decomposition*, INFORMS Journal on Computing **15** (2003), no. 3, 233–248. 4.2

[19] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver, *Combinatorial optimization*, John Wiley & Sons, 1998.

[20] D. G. Corneil and J. M. Keil, *A dynamic programming approach to the dominating set problem on k-trees*, SIAM J. Alg. Disc. Meth. **8** (1987), no. 4, 535–543. 1.5

[21] Bruno Courcelle, *The expression of graph properties and graph transformations in monadic second-order logic*, Handbook of graph grammars and computing by graph transformations, vol. 1: Foundations (G. Rozenberg, ed.), World Scientific, New-Jersey, London, 1997, pp. 313–400. 1.5

[22] Reinhard Diestel, *Graph theory*, 3 ed., Springer-Verlag, Heidelberg, 2005.

[23] Shimon Even, *Graph algorithm*, 1 ed., Computer Science Press, 1979.

[24] F. Fomin and D. Thilikos, *Dominating sets in planar graphs: Branch-width and exponential speed-up*, 2002. 4.2, 4.2

[25] Harold N. Garbow and Eugene W. Myers, *Finding all spanning trees of directed and undirected graphs*, SIAM J. Comput. **7** (1978), no. 3, 280–287. 4.1

[26] Michael R. Garey and David S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*, 1 ed., W.H.Freeman and company, 1979.

[27] Alan Gibbons, *Algorithmic graph theory*, 1 ed., Cambridge University Press, 1985.

[28] Jiong Guo, Rolf Niedermeier, and Daniel Raible, *Improved algorithms and complexity results for power domination in graphs*, Proc. 15th International Symposium on Fundamentals of Computation Theory (FCT) 2005, Aug. 17-20 2005.

[29] Magns M. Halldrsson, Jan Kratochvl, and Jan Arne Telle, *Independent sets with domination constraints*, ICALP '98: Proceedings of the 25th International Colloquium on Automata, Languages and Programming (London, UK), Springer-Verlag, 1998, pp. 176–187.

[30] Pinar Heggernes and Jan Arne Telle, *Partitioning graphs into generalized dominating sets*, Nordic J. of Computing **5** (1998), no. 2, 128–142.

[31] Xiao-Hua Jia, Ding-Zhu Du, Xiao-Dong Hu, He-Jiao Huang, and De-Ying Li, *Optimal placement of wavelength converters for guaranteed wavelength assignment in WDM networks*, IEICE transactions on communications **E85-B** (2002), no. 9, 1731–1739.

[32] Jason P. Jue, Dhritiman Banerjee, Byrav Ramamurthy, and Biswanath Mukherjee, *Optical components for WDM lightwave networks*, proceedings of the IEEE, vol. 85, Aug 1997, pp. 1274–1307.

[33] J. Kleinberg and A. Kumar, *Wavelength conversion in optical networks*, J. Algorithm **38** (2001), no. 1, 25–50. 1.5

[34] Donald E. Knuth, *The art of computer programming, volume 3*, 2 ed., Addison-Wesley, 1997.

[35] Sukhamay Kundu and Tong Yi, *Optimal converter placement in tree networks and a related heuristic for the general case*, Proc. 15th Int. Conf. Parallel and Distributed Computer Systems (PDCS), Sep. 2002. (document), 1.5, 2.1, 4.1

[36] Jean-François P. Labourdette, Georege W. Hart, and Anthony S. Acampora, *Branch-exchange sequences for reconfiguration of lightwave networks*, IEEE Transactions on Communications **42** (1994), no. 10, 2822–2832.

[37] Anil Maheshwari and Norbert Zeh, *I/o-efficient algorithms for graphs of bounded treewidth*, SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms (Philadelphia, PA, USA), Society for Industrial and Applied Mathematics, 2001, pp. 89–90.

[38] Biswanath Mukherjee, *WDM-based local lightwave networks – Part I: Single-hop systems*, IEEE network **6** (1992), no. 3, 12–26.

[39] ______, *WDM-based local lightwave networks – Part II: Multihop systems*, IEEE network **6** (1992), no. 4, 22–32.

[40] ______, *Optical communication networks*, 1 ed., Prentice-Hall, 1997.

[41] ______, *WDM optical communication networks: progress and challenges*, IEEE J. on Selected Areas in Communications **18** (2000), no. 10, 1810–1824.

[42] Biswanath Mukherjee, S. Ramamurthy, Dhritiman Banerjee, and Amarnath Mukherjee, *Some principles for designing a wide-area optical network*, IEEE/ACM Trans. Networking **4** (1996), 684–696.

[43] C. Siva Ram Murthy and Mohan Gurusamy, *WDM optical networks: Concepts, design, and algorithms*, 1 ed., Prentice-Hall, 2001.

[44] Hung *Q* Ngo, Dazhen Pan, , and Yuanyuan Yang, *Optical switching networks with minimum number of limited range wavelength converters*, Proc. 24rd conference of the IEEE Communications Society (INFOCOM 2005), Mar 2005.

[45] Asuman E. Ozdaglar and Dimitri P. Bertsekas, *Routing and wavelength assignment in optical networks*, IEEE/ACM Trans. Networking **11** (2003), no. 2.

[46] Charles C. Palmer and Aaron kershenbaum, *An approach to a problem in network design using genetic algorithms*, Networks **26** (1995), 151–163.

[47] Larry L. Peterson and Bruce S. Davie, *Computer networks: a system approach*, 2 ed., Morgan Kaufmann, 2000.

[48] B. Ramamurthy and B. Mukherjee, *Wavelength conversion in WDM networking*, IEEE J. Selected Areas of Communication **16** (1998), 1061–1073.

[49] Ramu Ramamurthy and Biswanath Mukherjee, *Fixed-alternate routing and wavelength conversion in wavelength-routed optical networks*, IEEE/ACM Trans. Networking **10** (2002), no. 3, 351–367.

[50] R. Ramaswami and K. N. Sivarajan, *Design of logical topologies for wavelength-routed optical networks*, IEEE J. on Selected Areas in Communications **14** (1996), no. 5, 840–851.

[51] N. Robertson and P.D. Seymour, *Graph minors II: Algorithmic aspects of tree-width*, J. Algorithms **7** (1986), 309–322. 1.4.1

[52] L. Ruan, D. Du, X. Hu, X. Jia, D. Li, and Z. Sun, *Converter placement supporting broadcast in WDM optical networks*, IEEE Trans. Comput. **50** (2001), 750–758. 1.5, 3.1

[53] Lu Ruan, *Multicast and broadcast routing in WDM optical networks*, Ph.D. thesis, University of Minnesota, Jul 2001.

[54] Poompat Saengudomlert, *Architectural study of high speed networks with optical bypassing*, Ph.D. thesis, MIT, Jul 2002.

[55] Laxman H. Sahasrabuddhe and Biswanath Mukherjee, *Light-trees: optical multicasting for improved performance in wavelength-routed networks*, IEEE Commun. Mag. (1999), 67–73. 1.3

[56] Petra Scheffler, *A practical linear time algorithm for disjoint paths in graphs with bounded tree-width.* 1.5

[57] P.D. Seymour, *Disjoint paths in graphs*, Discrete mathematics **29** (1980), 293–309.

[58] Y. Shiloach, *The two paths problem is polynomial*, Tech. Report STAN-CS-78-654, Computer Science Department, Stanford University, Stanford University, CA., July 1978, Technical Report.

[59] Y. Shiloach and Y. Perl, *Finding two disjoint paths between two pairs of vertices in a graph*, Journal of the ACM **25** (1978), no. 1, 1–9.

[60] Jan Späth, *Dynamic routing and resource allocation in WDM transport network*, Computer Networks **32** (2000), 519–538.

[61] T.E. Stern and K. Bala, *Multiwavelength optical networks - a layered approach*, Addison-Wesley, 1999.

[62] S. Subramaniam, M. Azizoglu, and A. K. Somani, *On optimal converter placement in wavelength-routed networks*, IEEE/ACM Trans. Networking **7** (1999), no. 5, 754–766. 1.5

[63] Andrew S. TanenBaum, *Computer networks*, 4 ed., Prentice-Hall PTR, 2002.

[64] Jan Arne Telle and Andzej Proskurowski, *Algorithms for vertex partitioning problems on partial k-trees*, SIAM Journal on Discrete Mathematics **10** (1997), no. 4, 529–550. 1.5

[65] S. Thiagarajan and A. K. Somani, *An efficient algorithm for optimal wavelength converter placement on wavelength-routed networks with arbitrary topologies*, Proc. INFOCOM, Mar. 1999, pp. 916–923. 1.5

[66] Vijay V. Vazirani, *Approximation algorithms*, Springer-Verlag, Berlin, 2001.

[67] K.R. Venugopal, M. Sivakumar, and P. Srenivasakumar, *A heuristic for placement of limited range wavelength converters in all-optical networks*, Computer Networks (2001), no. 35, 143–163. 1.5

[68] Douglas B. West, *Introduction to graph theory*, 2 ed., Prentice-Hall, 2001.

[69] Gaoxi Xiao and Yiu-Wing Leung, *Algorithms for allocating wavelength converters in all-optical network*, IEEE/ACM Trans. Networking **7** (1999), no. 4, 545–557.

[70] Hui Zang, Jason P. Jue, and Biswanath Mukherjee, *A review of routing and wavelength assignment approaches for wavelength-routed optical WDM networks*, SPIE/Baltzer Optical Networks Magazine (ONM) **1** (2000), no. 1, 47–60.

[71] Zhensheng Zhang and Anthony S. Acampora, *A heuristic wavelength assignment algorithm for multihop WDM networks with wavelength routing and wavelength re-use*, IEEE/ACM Trans. Networking **3** (1995), no. 3, 281–288.

[72] Xiao Zhou, Shin−Ichi Nakano, and Takao Nishizeki, *Edge-coloring partial k-trees*, Journal of Algorithms **21** (1996), no. 3, 598–617.

Appendix: Code of btw.c

```
// The main data structure used is "bTable",
// means the table for a block, that is,
// a subset of graph nodes that form a tree node.
// For example, nodes c, d, and e.
//
//                     CPRS
//====================================================
//                      PRS
//      ._._._._._._._._._._._._._._._._._._._._._
//      !                                         !
//      !  provWl      The dotted line box is RS  !
// +--+ !  +-+-+   .............................. !
// |00|--->|^| |   .                            . !
// +--+ !  +-+-+   .   parSol                   . !
//      !     |    . +-+-+        +-+-+         . !
//      !      ----->| |^|------->| |^|         . !
//      !          . +-+-+        +-+-+         . !
//      !          .  |  +-+  +-+  |  +-+  +-+  . !
//      !          .  +->|1|->|3|  +->|2|->|3|  . !
//      !          .     +-+  +-+     +-+  +-+  . !
//      !          .............................. !
//      ._._._._._._._._._._._._._._._._._._._._._!
//
// +--+    +-+-+
// |01|--->| | |
// +--+    +-+-+
//          | |             +-+-+
//          | +------------>| | |
//          |               +-+-+
//          |  +-+  +-+      | |   +-+
//          +->|1|->|2|      | +-->|c|
//             +-+  +-+      |     +-+
//                           | +-+
//                           +-|3|
//                             +-+
//
// +--+    +-+-+
// |10|--->| | |
// +--+    +-+-+
```

```
//          | |            +-+-+
//          | +----------->|^| |
//          |              +-+-+
//          |  +-+  +-+       |   +-+
//          +->|2|->|3|       +-->|e|
//             +-+  +-+           +-+
//
//
// +--+    +-+-+
// |11|--->| | |
// +--+    +-+-+
//          | |               +-+-+
//          | +-------------->|^| |
//          |                 +-+-+
//          |  +-+  +-+  +-+     |   +-+  +-+
//          +->|1|->|2|->|3|     +-->|c|->|e|
//             +-+  +-+  +-+         +-+  +-+
//
//=====================================================
//

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

// Boolean
#define NEWLINE             1
#define TRUE                1
#define FALSE               0
#define COVERED             1
#define DELETED             1

#define PRINT_DELETED       1

#define DATA_FILE          "graphFile/btw.dat"

typedef struct WlPtr {
        int data;
        // We do not physically move the pointer;
        // we simply mark it as "deleted".
        int deleted; // list item is not present (not used always)
        struct WlPtr *next;
```

```
} WlPtr;

typedef struct TNode{
        int covered;
        int numRemWls;
        WlPtr *wlPtr;
        struct TNode *next;
} TNode;

typedef struct {
        TNode *first, *last;
} TNodeDPtr;

typedef struct IntNode {
        int data;
        struct IntNode *next;
} IntNode;

// edges not listed, but implicitly covered by the tree decomposition.
typedef struct GraphNode {
        int hasConv;
        IntNode *inciWl;
} GraphNode;

typedef struct AdjNode {
        int nodeId;
        // "isParent" indicates whether current adjNode
        // is a parent or not,
        // if isParent = 1, then adjNode is a parent;
        // otherwise, it is a child.
        int isParent;
        struct AdjNode *next;
} AdjNode;

typedef struct TreeNode {
        int level;   // Used for the rooted tree.
        int *map;    // Map a tree node to a subset of graph nodes.
        int mapNum;  // how many graphNodes are in this treeNode.
        int numConv; // the number of converters in current treeNode
        int *conv;
        AdjNode *parent; // Used for the rooted tree,
        AdjNode *firstAdjNode, *lastAdjNode;
```

```
} TreeNode;

// We associate the request wavelength with partial solution.
typedef struct RS {
        IntNode *reqWl;    // request wavelength
        IntNode *parSol;   // partial solution for a list of converters
        struct RS *next;
} RS;

// We associate the provide wavelength, the request wavelength,
// and parSol.
typedef struct PRS {
        IntNode *provWl; // list
        RS *rs;
        struct PRS *next;
} PRS;

// We associate the coverters in use, the provide wavelength,
// the request wavelength, and parSol.
typedef struct CPRS {
        int convInUse;
        PRS *prs;
} CPRS;

typedef struct ReqWlStru{
    IntNode *reqWl;
struct ReqWlStru *next;
} ReqWlStru;

// Global variables
int maxWlId, numGraphNodes, numTreeNodes, widthPlusOne,
    maxLevel, numCombs;
FILE *fpInp;
IntNode **levels;
GraphNode *graphNodes;
ReqWlStru *reqWlStru;
TreeNode *treeNodes;

// cprsForCombinedTable is to hold the intermediate result
// when we combine a parent's table with a child's table.
CPRS *cprsForCombinedTable;
CPRS **bTable;
```

```
int ListSize(IntNode *intNode) {
 int i;

 for (i = 0; intNode != NULL;
      intNode = intNode->next, i++);

 return i;
}

// If two ordered integer-list have the different size,
// find which one is smaller;
// otherwise, compare these two lists in dictionary order
// when their size are the same.
int CompIntNode(IntNode *intNode, IntNode *intNode2) {

 if (ListSize(intNode) < ListSize(intNode2))  // 1->3 < 1->2->3
    return -1;
 else if (ListSize(intNode) > ListSize(intNode2))
         return 1;

 for (; ((intNode != NULL) && (intNode2 != NULL)); )
     if (intNode->data < intNode2->data)
        // intNode < intNode2, example: 1->2 < 1->3->4
        return -1;
     else if (intNode->data == intNode2->data) {
             intNode = intNode->next;
             intNode2 = intNode2->next;
          }
          else return 1; // intNode > intNode2

 return 0;
}

int power (int base, int exp) {
 int i, result;

 for (result = 1, i = 0; i < exp; i++, result *= base);
 return result;
}

/////////////////////////////////////////////////////////////////
```

```
// USAGE Example:
// Declare  IntNode *inciWl;
//   inciWl = InsToIntNode(inciWl, 3);
// then, if 3 is already in the inciWl list, then do nothing;
// otherwise, 3 will be inserted to the list in increasing order.
/////////////////////////////////////////////////////////////////
IntNode *InsToIntNode(IntNode *intNode, int data) {
 IntNode *currIntNode, *insertBefore, *insertAfter, *newIntNode;

 newIntNode = (IntNode *)calloc(1, sizeof(IntNode));
 if (newIntNode == NULL) {
    printf("Out of memory when applying for newIntNode"
           "in InsToIntNode.\n");
    exit(1);
 }

 newIntNode->data = data;
 newIntNode->next = NULL;

 // Insert when the list is empty.
 if (intNode == NULL) {
    intNode = newIntNode;
    return intNode;
 }

 // Find a position to insert an item
 // that is not in the list.
 for (currIntNode = intNode; currIntNode != NULL;)
     if (data > currIntNode->data) {
        insertAfter = currIntNode;
        insertBefore = currIntNode->next;
        currIntNode = currIntNode->next;
     }
     else if (data == currIntNode->data) return intNode;
          else {// if (data < currIntNode->data)
                insertBefore = currIntNode;
                break;
   }
 if (insertBefore == intNode) {
    // Insert before the first item of a non-empty list.
    newIntNode->next = intNode;
    intNode = newIntNode;
```

```
 }
 else {newIntNode->next = insertAfter->next;
       insertAfter->next = newIntNode;
      }
 return intNode;
}

ReqWlStru *InsToReqWlStru(ReqWlStru *reqWlStruHead, IntNode *reqWl) {
 ReqWlStru *currReqWlStru, *insertBefore, *insertAfter, *newReqWlStru;

 newReqWlStru = (ReqWlStru *)calloc(1, sizeof(ReqWlStru));
 if (newReqWlStru == NULL) {
    printf("Out of memory when applying for newReqWlStru"
           "in InsToReqWlStru.\n");
    exit(1);
 }

 newReqWlStru->reqWl = reqWl;
 newReqWlStru->next = NULL;

 // Insert when the list is empty.
 if (reqWlStruHead == NULL) {
    reqWlStruHead = newReqWlStru;
    return reqWlStruHead;
 }

 // Find a position to insert an item
 // that is not in the list.
 for (currReqWlStru = reqWlStruHead; currReqWlStru != NULL;)
     if (reqWl > currReqWlStru->reqWl) {
        insertAfter = currReqWlStru;
        insertBefore = currReqWlStru->next;
        currReqWlStru = currReqWlStru->next;
     }
     else if (reqWl == currReqWlStru->reqWl)
             return reqWlStruHead;
          else {// if (reqWl < currReqWlStru->reqWl)
                insertBefore = currReqWlStru;
                break;
               }
 if (insertBefore == reqWlStruHead) {
    // Insert before the first item of a non-empty list.
```

```
    newReqWlStru->next = reqWlStruHead;
    reqWlStruHead = newReqWlStru;
 }
 else {newReqWlStru->next = insertAfter->next;
       insertAfter->next = newReqWlStru;
      }
 return reqWlStruHead;
}

// Insert to RS in increasing dictionary order
RS *InsToRs(RS *rsHead, IntNode *reqWl, IntNode *parSol) {
 RS *currRs, *insertBefore, *insertAfter, *newRs;

 newRs = (RS *)calloc(1, sizeof(RS));
 if (newRs == NULL) {
    printf("Out of memory when applying for newRs in InsToRes.\n");
    exit(1);
 }

 newRs->reqWl = reqWl;
 newRs->parSol = parSol;
 newRs->next = NULL;

 // Insert when the list is empty.
 if (rsHead == NULL) {
    rsHead = newRs;
    return rsHead;
 }

 // Find a position to insert an item
 // that is not in the list.
 for (currRs = rsHead; currRs != NULL;) {
     if (CompIntNode(reqWl, currRs->reqWl) == 1) {
        // Either reqWl has a larger size than that of currRs->reqWl
        // or, when reqWl and currRs->reqWl has the same size,
        // reqWl is behind currRs->reqWl in the dictionary order.
        //
        // Ex:         reqWl = 1->2->3
        //     currRs->reqWl = 1->3
        // or
        //             reqWl = 1->3
        //     currRs->reqWl = 1->2
```

```
        //
        insertAfter = currRs;
        insertBefore = currRs->next;
        currRs = currRs->next;
     }
     else if (CompIntNode(reqWl, currRs->reqWl) == 0) {
             // reqWl and currRs->reqWl are the same.
             if ( ListSize(parSol) < ListSize(currRs->parSol) )
                // The incoming parSol is smaller than currRs->parSol.
                currRs->parSol = parSol;

             // The incoming parSol is at least the same size as
             // currRs->parSol, so we do not update.
             return rsHead;
          }
          else {// if (reqWl < currRs->reqWl)
                insertBefore = currRs;
                break;
               }
 }

 if (insertBefore == rsHead) {
    // Insert before the first item of a non-empty list.
    newRs->next = rsHead;
    rsHead = newRs;
 }
 else {newRs->next = insertAfter->next;
       insertAfter->next = newRs;
      }

 return rsHead;
}

// Insert to PRS in increasing size;
// if of the same size, then use dictionary order.
// Example: {3} {1 2} {1 3}
PRS *InsToPrs(PRS *prsHead, IntNode *provWl, RS *rs) {
 PRS *currPrs, *insertBefore, *insertAfter, *newPrs;

 newPrs = (PRS *)calloc(1, sizeof(PRS));
 if (newPrs == NULL) {
    printf("Out of memory when applying for newPrs in InsToPrs.\n");
```

```
    exit(1);
 }

 newPrs->provWl = provWl;
 newPrs->rs = rs;
 newPrs->next = NULL;

 // Insert when the list is empty.
 if (prsHead == NULL) {
    prsHead = newPrs;
    return prsHead;
 }

 // Find a position to insert an item that is not in the list.
 for (currPrs = prsHead; currPrs != NULL;)
     //if (provWl > currPrs->provWl)
     if (CompIntNode(provWl, currPrs->provWl) == 1) {
        insertAfter = currPrs;
        insertBefore = currPrs->next;
        currPrs = currPrs->next;
     }
     else if (CompIntNode(provWl, currPrs->provWl) == 0)
             return prsHead;
          else {// if (data < currPrs->data)
                insertBefore = currPrs;
                break;
               }

 if (insertBefore == prsHead) {
    // Insert before the first item of a non-empty list.
    newPrs->next = prsHead;
    prsHead = newPrs;
 }
 else {newPrs->next = insertAfter->next;
       insertAfter->next = newPrs;
      }

 return prsHead;
}

PRS *InsToPrs2(PRS *prsHead, IntNode *provWl, IntNode *reqWl,
               IntNode *parSol) {
```

```
PRS *currPrs, *insertBefore, *insertAfter, *newPrs;

newPrs = (PRS *)calloc(1, sizeof(PRS));
if (newPrs == NULL) {
   printf("Out of memory when applying for newPrs in InsToPrs.\n");
   exit(1);
}

newPrs->provWl = provWl;
newPrs->rs = (RS *)calloc(1, sizeof(RS));
newPrs->rs->reqWl = reqWl;
newPrs->rs->parSol = parSol;
newPrs->next = NULL;

// Insert when the list is empty.
if (prsHead == NULL) {
   prsHead = newPrs;
   return prsHead;
}

// Find a position to insert an item that is not in the list.
for (currPrs = prsHead; currPrs != NULL;)
    //if (provWl > currPrs->provWl)
    if (CompIntNode(provWl, currPrs->provWl) == 1) {
       insertAfter = currPrs;
       insertBefore = currPrs->next;
       currPrs = currPrs->next;
    }
    else if (CompIntNode(provWl, currPrs->provWl) == 0) {
            currPrs->rs = InsToRs(currPrs->rs, reqWl, parSol);
            return prsHead;
         }
         else {// if (provWl < currPrs->provWl)
               insertBefore = currPrs;
               break;
              }

if (insertBefore == prsHead) {
   // Insert before the first item of a non-empty list.
   newPrs->next = prsHead;
   prsHead = newPrs;
}
```

```
 else {newPrs->next = insertAfter->next;
       insertAfter->next = newPrs;
      }

 return prsHead;
}

void SkipRestofLine()
{char c;
 fscanf(fpInp, "%c", &c);
 while (c != '\n') fscanf(fpInp, "%c", &c);
}

void SkipComments()
{char c;
 for (; ;) {
     fscanf(fpInp, "%c", &c);
     // any line beginning with / will be omitted;
     if (c == '/') SkipRestofLine(fpInp);
     else if ( ( (c >= '0') && (c <= '9') ) ||
           ( (c >= 'a') && (c <= 'z') ) ) {
             ungetc(c, fpInp);
             break;
          }
 }
}


////////////////////////////////////////////////////////////////////////
// ReadGraphNodes
// For example,
// for the diamond graph in
//     a
//     | 1
//     b
//  1 / \ 3
//   c   d
//  2 \ / 3
//     e
//  where b, c, and e are C-nodes,
//  we have the following set of data:
//     // for a [0]: 1
```

```
//     // [0] besides $a$ indicates that $a$ does not have a converter,
//     // 1 means the wavelengths incident at node $a$
//     5 // number of graph nodes
//     a [0]: 1
//     b [1]: 1 3
//     c [1]: 1 2
//     d [0]: 3
//     e [1]: 2 3
//  Output
//     graphNodes[0].hasConv = 0
//     graphNodes[0].inciWl = 1
//     graphNodes[1].hasConv = 1
//     graphNodes[1].inciWl = 1->3
//     graphNodes[2].hasConv = 1
//     graphNodes[2].inciWl = 1->2
//     graphNodes[3].hasConv = 0
//     graphNodes[3].inciWl = 3
//     graphNodes[4].hasConv = 1
//     graphNodes[4].inciWl = 2->3
////////////////////////////////////////////////////////////////////////
int ReadGraphNodes(void) {
 char c;
 int i, j, wl;
 IntNode *inciWl;

 fpInp = fopen(DATA_FILE, "r");

 if (fpInp == NULL) {
    printf("File %s cannot be opened.\n", DATA_FILE);
    return -1;
 }

 SkipComments();

 fscanf(fpInp, "%d", &numGraphNodes); SkipRestofLine();

 SkipComments();

 graphNodes = (GraphNode *)calloc(numGraphNodes, sizeof(GraphNode));
 if (graphNodes == NULL) {
    printf("Out of memory when applying for graphNodes"
           "in ReadGraphNodes.\n");
```

```
    exit(1);
  }

  maxWlId = 0;
  for (i = 0; i < numGraphNodes; ) { // i++ is done inside.
      // Skip spaces or tabs until we find the first integer,
      // that is where the first wavelength starts.
      for (; ;) {
          fscanf(fpInp, "%c", &c);
          if (c == '\n') break;
          if (c == EOF) return -1;
          if ( (c >= 'a') && (c <= 'z') ) {
             break;
          }
      }

      j = c - 'a';

      fscanf(fpInp, " [%d]:", &graphNodes[j].hasConv);

      inciWl = NULL;

      // Read wavelength information.
      for (; ;) {
          fscanf(fpInp, "%c", &c);
          if (c == '\n') break;
          if (c == EOF)  return -1;
          if ( (c >= '0') && (c <= '9') ) {
             ungetc(c, fpInp);
             fscanf(fpInp, "%d", &wl);
             if (wl > maxWlId) maxWlId = wl;
             inciWl = InsToIntNode(inciWl, wl);
          }
      }

      graphNodes[j].inciWl = inciWl;

      i++;
  }
  // do not close file yet, we need to read treeNodes.
  return numGraphNodes;
}
```

```
// Add "adjNodeId" to the adjacent list of nodeId.
// We can use add, instead of insert to,
// because in the input file, adjacent nodes are sorted already.
int AddToAdjList(int nodeId, int adjNodeId) {
 AdjNode *adjNodeToBeAdded;

 adjNodeToBeAdded = (AdjNode *)calloc(1, sizeof(AdjNode));
 if (adjNodeToBeAdded == NULL) {
    printf("Out of memory when applying adjNodeToBeAdded"
           "in AddToAdjList.\n");
    exit(1);
 }

 adjNodeToBeAdded->nodeId = adjNodeId;
 adjNodeToBeAdded->isParent = FALSE;
 adjNodeToBeAdded->next = NULL;

 if (!(treeNodes[nodeId].firstAdjNode)) // no adjNode has been added yet
    treeNodes[nodeId].firstAdjNode = treeNodes[nodeId].lastAdjNode
                                  = adjNodeToBeAdded;
 else treeNodes[nodeId].lastAdjNode = (treeNodes[nodeId].lastAdjNode)->next
                                   = adjNodeToBeAdded;
 return 0;
}

int ReadTreeNodes(void) {
 char c, chr;
 int i, j, k, adjNodeId, sizeATreeNode;

 SkipComments();

 fscanf(fpInp, "%d", &numTreeNodes); SkipRestofLine();

 fscanf(fpInp, "%d", &widthPlusOne); SkipRestofLine();

 SkipComments();

 treeNodes = (TreeNode *)calloc(numTreeNodes, sizeof(TreeNode));
 if (treeNodes == NULL) {
    printf("Out of memory when applying for treeNodes in ReadTreeNodes.\n");
    exit(1);
```

```
}

for (i = 0; i < numTreeNodes; ) { // i++ is done inside.
    // Skip spaces or tabs until we find the first integer,
    // that is where the first treeNode begins.
    for (; ;) {
        fscanf(fpInp, "%c", &c);
        if (c == '\n') break;
        if (c == EOF) {
           fclose(fpInp);
           return 0;
        }
        if ( (c >= '0') && (c <= '9') ) {
           break;
        }
    }

    treeNodes[i].map = (int *)calloc(widthPlusOne, sizeof(int));
    if (treeNodes[i].map == NULL) {
       printf("Out of memory when applying for treeNodes[i].map"
              "in ReadTreeNodes.\n");
       exit(1);
    }

    fscanf(fpInp, " ("); // Specific to the format of input file.
    sizeATreeNode = 0;
    for (j = 0; j < widthPlusOne;) {
        fscanf(fpInp, "%c", &chr); // read char 'b'
        if ((chr >= 'a') && (chr <= 'z')) {
           treeNodes[i].map[j] = chr - 'a';
           j++;
           sizeATreeNode++;
        }
        else if (chr == ')') {
                treeNodes[i].map[j] = -1;
                break;
              }
    }
    treeNodes[i].mapNum = j;
    if (j == widthPlusOne)
       fscanf(fpInp, "):");
    else fscanf(fpInp, ":");
```

```
        // Find the number of converters in each tree node.
        treeNodes[i].numConv = 0;
        for (j = 0; j < sizeATreeNode; j++)
            if (graphNodes[treeNodes[i].map[j]].hasConv)
               treeNodes[i].numConv++;
        treeNodes[i].conv = (int *)calloc(treeNodes[i].numConv, sizeof(int));
        if (treeNodes[i].conv == NULL) {
           printf("Out of memory when applying for treeNodes[i].conv"
                  "in ReadTreeNodes.\n");
           exit(1);
        }

        // Store the indice of converters of current treeNode in conv
        j = 0;
        for (k = 0; k < widthPlusOne; k++)
            if (graphNodes[treeNodes[i].map[k]].hasConv)
               treeNodes[i].conv[j++] = treeNodes[i].map[k];

        treeNodes[i].firstAdjNode = treeNodes[i].lastAdjNode = NULL;
        // Read adjacent tree node information of tree node i.
        for (; ;) {
            fscanf(fpInp, "%c", &c);
            if (c == '\n') break;
            if (c == EOF)  {
               fclose(fpInp);
               return;
            }
            if ( (c >= '0') && (c <= '9') ) {
               ungetc(c, fpInp);
               fscanf(fpInp, "%d", &adjNodeId);
               AddToAdjList(i, adjNodeId); // Variable i is the current nodeId.
            }
        }

        i++;
    }
    fclose(fpInp);
    return numTreeNodes;
}

int PrintTreeNodes(void) {
```

```
 int i, j;
 AdjNode *currAdjNode;

 printf("%-12s", "Tree Node");
 printf("%-25s", "Subset of Graph Nodes");
 printf("Adajacent Nodes\n");
 for (i = 0; i < numTreeNodes; i++) {
     printf("%-12d", i);
     // Print mapping between a tree node and a subset of graph nodes.
     printf("(");
     for (j = 0; j < widthPlusOne; j++)
         if (j < widthPlusOne -1)
            printf("%c, ", 'a' + treeNodes[i].map[j]);
         else printf("%c", 'a' + treeNodes[i].map[j]);

     printf(")");

     for (j = 0; j < 24 - 3 * widthPlusOne; j++)
         printf(" ");

     for (currAdjNode = treeNodes[i].firstAdjNode;
          currAdjNode != NULL;
          currAdjNode = currAdjNode->next)
         printf("%2d  ", currAdjNode->nodeId);
     printf("\n");
 }

 return 0;
}

// Based on the built-up adjacent link list nodes,
// mark who is the parent, mark the other ajdNodes as children.
// Build IntList levels
// where levels[i] is the list of all the nodes of the ith layer.
void BuildRootedTree(int currNodeId, int parentId) {
 int i, currLevel;
 AdjNode *adjNode;

 if (parentId == -1) {
    treeNodes[currNodeId].parent = NULL;
    currLevel = treeNodes[currNodeId].level = 0;
 }
```

```
 else currLevel = treeNodes[currNodeId].level = treeNodes[parentId].level +1;

 if (maxLevel < currLevel) maxLevel = currLevel;
 levels[currLevel] = InsToIntNode(levels[currLevel], currNodeId);

 // For each adjacent node (neighbor) of currNodeId,
 // decide which one is the parent, and which one is a child.
 // We use variable 'isParent' for this purpose.
 // Note that we do not physically update the adjacent list,
 // we just mark each adjacent node of currNodeId.
 for (adjNode = treeNodes[currNodeId].firstAdjNode; adjNode != NULL;
      adjNode = adjNode->next)
     if (adjNode->nodeId == parentId) {
        //adjNode is the parent.
        adjNode->isParent = 1;
        treeNodes[currNodeId].parent = adjNode;
     }
     else // adjNode is a child.
          {adjNode->isParent = 0;
           // Build the parent-child relationship.
           BuildRootedTree(adjNode->nodeId, currNodeId);
          }
 return;
}

// In an adjacent list of a rooted tree,
// print each node v and v's children.
void PrintRootedTree() {
 int i;
 AdjNode *currAdjNode;
 for (i = 0; i < numTreeNodes; i++) {
     printf("%2d: ", i);
     for (currAdjNode = treeNodes[i].firstAdjNode; currAdjNode != NULL;
          currAdjNode = currAdjNode->next)
         printf("%2d ", currAdjNode->nodeId);
     printf("\n");
 }
}

// Print the subset of graphNodes that treeNodeId is mapped to.
int PrintMap(int treeNodeId, int newLine) {
 int i;
```

```
 for (i = 0; i < treeNodes[treeNodeId].mapNum; i++)
     if (i < treeNodes[treeNodeId].mapNum -1)
        printf("%c, ", treeNodes[treeNodeId].map[i] + 'a');
     else printf("%c}", treeNodes[treeNodeId].map[i] + 'a');

 if (newLine == NEWLINE) printf("\n");
 return 0;
}

int PrintIntNode(IntNode *intNode, int newLine) {
 int numItems;
 IntNode *currIntNode;

 numItems = 0;
 for (currIntNode = intNode; currIntNode != NULL;
      currIntNode = currIntNode->next, numItems++) {
     if (currIntNode->next != NULL)
        printf("%d, ", currIntNode->data);
     else printf("%d", currIntNode->data);
 }
 if (newLine == NEWLINE) printf("\n");

 if (numItems == 0) {
    printf("NULL");
    if (newLine == NEWLINE) printf("\n");
    return 0;
 }
 else return numItems;
}

// The same as that of PrintIntNode,
// except that we print the data in char, not in integer.
int PrintIntNodeInChar(IntNode *intNode, int newLine) {
 int numItems;
 IntNode *currIntNode;

 numItems = 0;
 for (currIntNode = intNode; currIntNode != NULL;
      currIntNode = currIntNode->next, numItems++) {
     if (currIntNode->next != NULL)
        printf("%c, ", (char)(currIntNode->data));
```

```
     else printf("%c", (char)(currIntNode->data));
 }
 if (newLine == NEWLINE) printf("\n");

 if (numItems == 0) {
    printf("NULL");
    if (newLine == NEWLINE) printf("\n");
    return 0;
 }
 else return numItems;
}

int PrintGraphNodes(void) {
 int i;

 printf("Node    Has converter?     Incident wavelengths\n");
 for (i = 0; i < numGraphNodes; i++) {
     printf("%4c    ", (char)(i + 'a'));
     printf("%d                   ", graphNodes[i].hasConv);
     PrintIntNode(graphNodes[i].inciWl, NEWLINE);
 }

 return 0;
}

///////////////////////////////////////////////////////////
// Print wavelengths pointed by wlPtr.
// --------------------------------------------------
// USAGE Example: PrintWls(wlPtr, PRINT_DELETED, NEWLINE);
//////////////////////////////////////////////////////////

int PrintWls(WlPtr *wlPtr, int printDeleted, int newLine) {
 int numItems;
 WlPtr *currWlPtr;

 numItems = 0;
 for (currWlPtr = wlPtr; currWlPtr != NULL;
      currWlPtr = currWlPtr->next, numItems++) {
     if (currWlPtr->deleted) {
        if (printDeleted)
           printf("%2d (*)", currWlPtr->data);
     }
```

```
    else printf("%2d    ", currWlPtr->data);
 }
 if (newLine == NEWLINE) printf("\n");

 if (numItems == 0) {
    printf("  NULL    ");
    if (newLine == NEWLINE) printf("\n");
    return 0;
 }
 else return numItems;
}

//////////////////////////////////////////////////////////
// Print TNode
// --------------------------------------------------
// USAGE Example:
//       PrintTNodes(tNodePtr, PRINT_DELETED, NEWLINE);
//       PrintTNodes(tNodePtr, !(PRINT_DELETED), NEWLINE);
//////////////////////////////////////////////////////////

int PrintTNodes(TNode *tNodePtr, int printDeleted, int newLine) {
 int numItems;
 TNode *currTNodePtr;

 numItems = 0;

 if (printDeleted)
    printf("The table of TNodes (* means deleted)\n");
 else printf("       The table of TNodes\n");
 printf("------------------------------------\n");
 printf("Covered  numRemWls  Wavelengths\n");
 printf("------------------------------------\n");
 for (currTNodePtr = tNodePtr; currTNodePtr != NULL;
      currTNodePtr = currTNodePtr->next, numItems++) {
     printf("%2d       ", currTNodePtr->covered);
     printf("%2d         ", currTNodePtr->numRemWls);
     PrintWls(currTNodePtr->wlPtr, printDeleted, !(NEWLINE));
     if (newLine == NEWLINE) printf("\n");
 }
 printf("------------------------------------\n\n");

 if (numItems == 0) {
```

```
    printf("No items in the tNode list.\n\n");
    return 0;
 }
 else return numItems;
}

// After the build-up of a rooted tree,
// print out which node is in which level.
void PrintLevels() {
 int i;
 printf("\n\n");
 for (i = 0; i <= maxLevel; i++) {
     printf("level %2d:  ", i);
     PrintIntNode(levels[i], NEWLINE);
 }
 printf("\n");
}

////////////////////////////////////////////////
// If wl is not in wlPtr,
// then insert wl to wlPtr in increasing order.
// -------------------------------------------
// USAGE Example:
// wlPtr = InsToWlPtr(wlPtr, 3, !DELETED);
////////////////////////////////////////////////
WlPtr *InsToWlPtr(WlPtr *wlPtr, int wl, int delSgn) {
 WlPtr *currWlPtr, *insertBefore, *insertAfter, *newWlPtr;

 newWlPtr = (WlPtr *)calloc(1, sizeof(WlPtr));
 if (newWlPtr == NULL) {
    printf("Out of memory when applying for newWlPtr in InsToWlPtr.\n");
    exit(1);
 }

 newWlPtr->data = wl;
 newWlPtr->deleted = delSgn;

 // Insert when the list is empty.
 if (wlPtr == NULL) {
    newWlPtr->next = NULL;
    wlPtr = newWlPtr;
    return wlPtr;
```

```
 }

 // Find a position to insert an item
 // that is not in the list.
 for (currWlPtr = wlPtr; currWlPtr != NULL;)
     if (wl > currWlPtr->data) {
        insertAfter = currWlPtr;
        insertBefore = currWlPtr->next;
        currWlPtr = currWlPtr->next;
     }
     else if (wl == currWlPtr->data) return;
          else if (wl < currWlPtr->data) {
                  insertBefore = currWlPtr;
                  break;
               }

 if (insertBefore == wlPtr) {
    // Insert before the first item of a non-empty list.
    newWlPtr->next = wlPtr;
    wlPtr = newWlPtr;
 }
 else {newWlPtr->next = insertAfter->next;
       insertAfter->next = newWlPtr;
      }
 return wlPtr;
}

// Copy the content of wlPtr, except those deleted wavelengths,
// to a return wlPtr2.
// For example, call in the following way:
// wlPtr2 = CopyWls(wlPtr);
WlPtr *CopyWls(WlPtr *sourceWlPtr) {
 WlPtr *targetWlPtr, *currWlPtr;

 targetWlPtr = NULL;
 for (currWlPtr = sourceWlPtr; currWlPtr != NULL;
      currWlPtr = currWlPtr->next)
     if (!(currWlPtr->deleted))
        targetWlPtr = InsToWlPtr(targetWlPtr, currWlPtr->data,
                                 currWlPtr->deleted);

 return targetWlPtr;
```

```
}

////////////////////////////////////////////////////////////////
// Copy the wl part of wlPtr, except those deleted wavelengths,
// to a return an IntNode;
// For example, call in the following way:
// wlNode = CopyWls(wlPtr);
// If wlPtr points to
// +-+-+  +-+-+
// |1|0|->|2|1|
// +-+-+  +-+-+
// where 1 0 means 1 is not a wavelength marked as deleted,
//       2 1 means 2 is a wavelength marked as deleted.
// After the calling of the following function,
// we will get
// +-+
// |1|
// +-+
////////////////////////////////////////////////////////////////

IntNode *CopyWlsNoDelSgn(WlPtr *sourceWlPtr) {
 IntNode *targetWlNode;
 WlPtr *currWlPtr;

 targetWlNode = NULL;
 for (currWlPtr = sourceWlPtr; currWlPtr != NULL;
      currWlPtr = currWlPtr->next)
     if (!(currWlPtr->deleted))
        targetWlNode = InsToIntNode(targetWlNode, currWlPtr->data);
 return targetWlNode;
}

TNodeDPtr AddToTNodeList
  (TNodeDPtr tNodeDPtr, int covered, int numRemWls, WlPtr *wlPtr) {
 TNode *tNodeToBeAdded;

 tNodeToBeAdded = (TNode *)calloc(1, sizeof(TNode));
 if (tNodeToBeAdded == NULL) {
    printf("Out of memory when applying for tNodeToBeAdded"
           "in AddToTNodeList.\n");
    exit(1);
 }
```

```
 tNodeToBeAdded->covered = covered;
 tNodeToBeAdded->numRemWls = numRemWls;
 tNodeToBeAdded->wlPtr = CopyWls(wlPtr);

 if (tNodeDPtr.first == NULL) // no tNode has been added yet
    tNodeDPtr.first = tNodeDPtr.last = tNodeToBeAdded;
 else tNodeDPtr.last = (tNodeDPtr.last)->next
                    = tNodeToBeAdded;

 (tNodeDPtr.last)->next = NULL;
 return tNodeDPtr;
}

////////////////////////////////////////////////////////////////
// Make a list tNodePtr2 that has the same content
// as that of tNodePtr,
// except the omission of those nodes marked as "covered"
// and those wavelengths marked as "deleted".
////////////////////////////////////////////////////////////////

TNodeDPtr CopyTNodes(TNodeDPtr tNodeDPtr) {
 TNode *currTNodePtr;
 TNodeDPtr tNodeDPtr2;
 WlPtr *wlPtr, *currWl;

 // Initialize tNodeDPtr2.first and tNodeDPtr2.last to be empty.
 tNodeDPtr2.first = tNodeDPtr2.last = NULL;

 for (currTNodePtr = tNodeDPtr.first; currTNodePtr != NULL;
      currTNodePtr = currTNodePtr->next)
     if (!(currTNodePtr->covered)) // not covered yet
        tNodeDPtr2 = AddToTNodeList(tNodeDPtr2, !(COVERED), //
          currTNodePtr->numRemWls, currTNodePtr->wlPtr);

 return tNodeDPtr2;
}

/////////////////////////////////////////////////////////////////////
// Include()
//   mark all the nodes having the target wavelength as "covered".
/////////////////////////////////////////////////////////////////////
```

```
TNodeDPtr Include(TNodeDPtr tNodeDPtr, int wl) {
 TNode *currTNodePtr;
 WlPtr *currWl;

 for (currTNodePtr = tNodeDPtr.first; currTNodePtr != NULL;
      currTNodePtr = currTNodePtr->next)
     for (currWl = currTNodePtr->wlPtr; currWl != NULL;
          currWl = currWl->next)
         if (!(currWl->deleted) && (currWl->data == wl)) {
            currTNodePtr->covered = TRUE;
            break;
         }
 return tNodeDPtr;
}


/////////////////////////////////////////////////////////////
// Exclude()
// .........................................................
// Goal: mark all the target wavelength as "deleted",
//       and decrease variable "numRemWls" from all the nodes
//       having the target wavelength by 1.
/////////////////////////////////////////////////////////////

TNodeDPtr Exclude(TNodeDPtr tNodeDPtr, int wl) {
 TNode *currTNodePtr;
 WlPtr *currWl;

 for (currTNodePtr = tNodeDPtr.first; currTNodePtr != NULL;
      currTNodePtr = currTNodePtr->next)
     for (currWl = currTNodePtr->wlPtr; currWl != NULL;
          currWl = currWl->next)
         if (currWl->data == wl) {
            currWl->deleted = TRUE;
            (currTNodePtr->numRemWls)--;
            break;
         }
 return tNodeDPtr;
}

// Given "convInUse", a number whose bit with value 1
```

```
// indicates the corresponding converter is put in use,
// find the set of acivated wavelengths
// provided by all the converters put in use.
// numConv is needed, since we need to know how many bits
// are in convInUse.
// For example, if convInUse = 8, then numConv = 4.
//              if convInUse = 7, then numConv = 3.
// Variable conv gives the real index of graphNodes.
// For example, if conv[0] = 2 (means 'a'+2 = 'c')
// and conv[1] = 4 (means 'a'+4 = 'e').
// Then, if convInUse = 1, then we are discussing node c,
// if convInUse = 2, then we are discussing node $e$

IntNode *ProvWl(int convInUse, int *conv, int numConv) {
 int i, *useConvB;
 IntNode *provWl, *currIntNode;

 useConvB = (int *)calloc(numConv, sizeof(int));
 if (useConvB == NULL) {
    printf("Out of memory when applying for useConvB in ProvWl.\n");
    exit(1);
 }

 provWl = NULL;
 for (i = 0; i < numConv; i++) {
     if (i == 0)
        useConvB[i] = convInUse & 1;
     else useConvB[i] = ( convInUse & power(2, i) ) >> i;
     // For example,
 // if convInUse = 5; numConv = 3 (2^3 >= 5)
     // useConvB[0] = 5 & 1 = 1;
     // useConvB[1] = (5 & 2^1) >> 1 = 0;
     // useConvB[2] = (5 & 2^2) >> 2 = 1;
     if (useConvB[i]) {
        for (currIntNode = graphNodes[conv[i]].inciWl; currIntNode != NULL;
             currIntNode = currIntNode->next)
            provWl = InsToIntNode(provWl, currIntNode->data);
     }
 }
 free(useConvB);
 return provWl;
}
```

```
// To see whether wlPtr2 has some wavelength that is in wlPtr.
int IsIntersIntNode(IntNode *wlPtr, IntNode *wlPtr2) {
 int i, *wlCount;
 IntNode *currWlPtr, *currWlPtr2;

 wlCount = (int *)calloc(maxWlId, sizeof(int));
 if (wlCount == NULL) {
    printf("Out of memory when applying for wlCount in IsIntersIntNode.\n");
    exit(1);
 }

 for (i = 0; i < maxWlId; i++)
     wlCount[i] = 0;

 for (currWlPtr = wlPtr; currWlPtr != NULL; currWlPtr = currWlPtr->next)
     wlCount[currWlPtr->data -1]++;

 for (currWlPtr2 = wlPtr2; currWlPtr2 != NULL;
     currWlPtr2 = currWlPtr2->next) {
     wlCount[currWlPtr2->data -1]++;
     if (wlCount[currWlPtr2->data -1] == 2) {
        free(wlCount);
        return TRUE;
     }
 }

 free(wlCount);

 return FALSE;
}

void FreeWls(WlPtr *wlPtr) {
 WlPtr *tempWlPtr, *currWlPtr;

 for (currWlPtr = wlPtr; currWlPtr != NULL;) {
     tempWlPtr = currWlPtr;
     currWlPtr = currWlPtr->next;
     free(tempWlPtr);
 }
 return;
}
```

```
void FreeIntNode(IntNode *intNode) {
 IntNode *currIntNode, *tempIntNode;

 for (currIntNode = intNode; currIntNode != NULL;) {
     tempIntNode = currIntNode;
     currIntNode = currIntNode->next;
     free(tempIntNode);
 }
 return;
}

void FreeRS(RS *rs) {
 RS *currRs, *tempRs;

 for (currRs = rs; currRs != NULL; ) {
     tempRs = currRs;
     currRs = currRs->next;
     FreeIntNode(tempRs->reqWl);
     FreeIntNode(tempRs->parSol);
     free(tempRs);
 }
}

void FreePRS(PRS *prs) {
 PRS *currPrs, *tempPrs;

 for (currPrs = prs; currPrs != NULL; ) {
     tempPrs = currPrs;
     currPrs = currPrs->next;
     FreeIntNode(tempPrs->provWl);
     FreeRS(tempPrs->rs);
     free(tempPrs);
 }
 return;
}

void FreeTableItem(int id) {
 int i, numConvSetting;
 PRS *tempPrs, *nextPrsToBeFree;
 CPRS *currCprs, *tempCprs;
```

```
 numConvSetting = power(2, treeNodes[id].numConv);

 for (i = 0; i < numConvSetting; i++) {
     FreePRS(bTable[id][i].prs);
 }

 tempCprs = bTable[id];
 free(tempCprs);
 bTable[id] = NULL;
 return;
}

void FreeCprs(CPRS *cprs, int numConvSetting) {
 int i;

 for (i = 0; i < numConvSetting; i++)
     FreePRS(cprs[i].prs);
 return;
}

void FreeTNodes(TNodeDPtr tNodeDPtr) {
 TNode *tempTNodePtr, *currTNodePtr;

 for (currTNodePtr = tNodeDPtr.first; currTNodePtr != NULL;) {
     FreeWls(currTNodePtr->wlPtr);
     tempTNodePtr = currTNodePtr;
     currTNodePtr = currTNodePtr->next;
     free(tempTNodePtr);
 }
 return;
}

// Find a wavelength that appears most often
// among all the undeleted wavelengths
// incident at all uncovered nodes.
int FindTag(TNode *tNodePtr) {
 int i, wlAppearOften, maxNumAppear, *numAppear;
 TNode *currTNodePtr;
 WlPtr *wlPtr;

 numAppear = (int *)calloc(maxWlId, sizeof(int));
 if (numAppear == NULL) {
```

```
    printf("Out of memory when applying for numAppear in FindTag.\n");
    exit(1);
 }

 for (i = 0; i < maxWlId; i++)
     numAppear[i] = 0;

 maxNumAppear = 0;
 for (currTNodePtr = tNodePtr; currTNodePtr != NULL;
      currTNodePtr = currTNodePtr->next)
     if (!(currTNodePtr->covered))
       for (wlPtr = currTNodePtr->wlPtr; wlPtr != NULL;
            wlPtr = wlPtr->next)
           if (!(wlPtr->deleted)) {
              // Note that the index of numAppear starts from 0.
              numAppear[wlPtr->data -1]++;
              if (numAppear[wlPtr->data -1] > maxNumAppear) {
                 maxNumAppear = numAppear[wlPtr->data -1];
                 wlAppearOften = wlPtr->data;
              }
           }
 free(numAppear);
 return wlAppearOften;
}


///////////////////////////////////////////////////////////////
// Find all the wavelength sets
// that covers each node in tNodeDPtr,
// when a set of wavelengths pointed by "inclWl" are provided.
///////////////////////////////////////////////////////////////

void FindReqWl(WlPtr *inclWl, TNodeDPtr tNodeDPtr) {
 int aNodeAWl, wl, allCovered;
 TNode *currTNodePtr;
 TNodeDPtr tNodeDPtr2, tNodeDPtr3;
 WlPtr *tempWlPtr, *wlPtr, *inclWl2;
 IntNode *retWl;

 // There are four types of exits for FindReqWl.

 // Exit 1: If (tNodeDPtr.first == NULL) return;
```

```
if (tNodeDPtr.first == NULL) return;


// Exit 2: If every link in tNodePtr is covered, return inclWl;
//
// The difference between the code with "allCovered" and
// ------------------------------------------------------
//  if (inclWl != NULL)
//     if (IsRemNodesCovered(inclWl, tNodeDPtr) == TRUE) {
//        PrintWls(inclWl, !(PRINT_DELETED), NEWLINE);
//        return;
//     }
// ------------------------------------------------------
// is that we just check the "covered" signal of
// each node in TNodePtr.

if (inclWl != NULL) {
   allCovered = TRUE;

   for (currTNodePtr = tNodeDPtr.first; currTNodePtr != NULL;
         currTNodePtr = currTNodePtr->next)
       if (!(currTNodePtr->covered)) {
          allCovered = FALSE;
          break;
       }

   if (allCovered) {// return inclWl;
      retWl = CopyWlsNoDelSgn(inclWl);
     reqWlStru = InsToReqWlStru(reqWlStru, retWl);
      return;
   }
}

aNodeAWl = TRUE;
for (currTNodePtr = tNodeDPtr.first; currTNodePtr != NULL;
     currTNodePtr = currTNodePtr->next) {
    if (!(currTNodePtr->covered))
       // Exit 3: If there is some node that cannot be covered,
       //         say, no remaining wavelengths can cover this node,
       //         then we end this procedure earlier.
       if (currTNodePtr->numRemWls == 0) // return NULL;
          return;
```

```
       else if (currTNodePtr->numRemWls > 1)
               aNodeAWl = FALSE;
}

// Exit 4: If there is exactly one wavelength
//        (may not be distinct though)
//        for each uncovered node,
//        then link the unique wavelengths up,
//        combine them with inclWl,
//        return the combined wavelength.
if (aNodeAWl) {
   retWl = CopyWlsNoDelSgn(inclWl);
   for (currTNodePtr = tNodeDPtr.first; currTNodePtr != NULL;
        currTNodePtr = currTNodePtr->next) {
       if (!(currTNodePtr->covered))
          for (wlPtr = currTNodePtr->wlPtr; wlPtr != NULL;
               wlPtr = wlPtr->next)
              if (!(wlPtr->deleted))
                 retWl = InsToIntNode(retWl, wlPtr->data);
   }
   reqWlStru = InsToReqWlStru(reqWlStru, retWl);
   return;
}

// Now every node has at least one wavelength to cover;
// furthermore, at least one node has two or more wavelengths to cover.
wl = FindTag(tNodeDPtr.first);

tNodeDPtr2 = CopyTNodes(tNodeDPtr);
// Include: mark all the nodes having the target wavelength as "covered"
//          do not change those nodes do not contain the target wavelength.
tNodeDPtr2 = Include(tNodeDPtr2, wl);

inclWl2 = CopyWls(inclWl);
inclWl2 = InsToWlPtr(inclWl2, wl, !(DELETED)); //wl is the tag wavelength.
FindReqWl(inclWl2, tNodeDPtr2);
FreeWls(inclWl2);

tNodeDPtr3 = CopyTNodes(tNodeDPtr);
// Exclude: mark the target wavelength from all the nodes
//          that contain it as "deleted";
//          decrease the corresponding numRemWls by 1.
```

```
 tNodeDPtr3 = Exclude(tNodeDPtr3, wl);
 FindReqWl(inclWl, tNodeDPtr3);

 FreeTNodes(tNodeDPtr2);
 FreeTNodes(tNodeDPtr3);

 return;
}

// Build TNodeDPtr from aTreeNode,
// numItems specifies how many elements in aTreeNode.
TNodeDPtr BuildTNodeList2(IntNode *provWl, int *aTreeNode, int numItems)
{
 int i, j, numNodes, wl, numRemWls;
 IntNode *inciWl;
 WlPtr *wlPtr, *currWlListPtr;
 TNodeDPtr tNodeDPtr;

 tNodeDPtr.first = tNodeDPtr.last = NULL;

 for (i = 0; i < numItems; i++) {
     if ( IsIntersIntNode(provWl, graphNodes[aTreeNode[i]].inciWl) )
        continue;
     // Only when the provWl cannot cover inciWl,
     // or, the intersection of provWl and inciWl is not empty,
     // do we need to store the corresponding inciWl
     // in tNodeDPtr to find all the wavelengths to cover it.

     numRemWls = 0;
     wlPtr = NULL;
     for (inciWl = graphNodes[aTreeNode[i]].inciWl; inciWl != NULL;
         inciWl = inciWl->next) {
         wlPtr = InsToWlPtr(wlPtr, inciWl->data, !DELETED);
         numRemWls++;
     }

     tNodeDPtr = AddToTNodeList(tNodeDPtr, !COVERED, numRemWls, wlPtr);
 }
 return tNodeDPtr;
}
```

```
// Given a number which represents for the subset of converters put
// in use in a tree node,
// find the ids of corresponding graph nodes.
// For example, if conv, the array of converters corresponding to
// this number has the following values
//
//    conv[0] = 'c' - 'a';
//    conv[1] = 'e' - 'a';
//
// where numConv tells us how big array conv is.
// if convInUseNum is 3,
// then we return ('c'-'a') -> ('e'-'a').
// Since we want the convInUseId to be ordered,
// we return the value in IntNode *, not simply in int *.
// Compare FindConvInUseId with FindConvInUseNum.
//
IntNode *FindConvInUseId(int convInUseNum, int *conv, int numConv) {
 int i, *useConvB;
 IntNode *convInUseList;

 useConvB = (int *)calloc(numConv, sizeof(int));
 if (useConvB == NULL) {
    printf("Out of memory when applying for useConvB"
           "in FindConvInUseId.\n");
    exit(1);
 }

 convInUseList = NULL;
 for (i = 0; i < numConv; i++) {
     if (i == 0)
        useConvB[i] = convInUseNum & 1;
     else useConvB[i] = ( convInUseNum & power(2, i) ) >> i;
   // For example,
     // if convInUseNum = 5; numConv = 3 (2^3 >= 5)
     // useConvB[0] = 5 & 1 = 1;
     // useConvB[1] = (5 & 2^1) >> 1 = 0;
     // useConvB[2] = (5 & 2^2) >> 2 = 1;
     if (useConvB[i])
        convInUseList = InsToIntNode(convInUseList, (int)('a' + conv[i]));
 }
 free(useConvB);
 return convInUseList;
```

```
}

// Given a list of integers, that is, IntNode *,
// which represents for the subset of converters put in use in a tree node,
// find the number.
// For example, if conv, the array of converters corresponding to
// this number has the following values
//
//   conv[0] = 'c' - 'a'
//   conv[1] = 'e' - 'a'
//
// where numConv tells us how big array conv is.
//
// If convInUseId is ('c' - 'a') -> ('e' - 'a'),
// then we return 3 = 2^0 + 2^1, where 0 and 1 are the index of
// 'c' and 'e' in conv respectively.
//
// If convInUseId is 'c' - 'a',
// then we return 1 = 2^0.
//
// Compare FindConvInUseNum with FindConvInUseId.
//
// NEEDED: 1. convInUseId is in increasing order, say 0->2
//         2. conv is also in increasing order. Say
//                  conv[0] = 'a' - 'a'.
//                  conv[1] = 'b' - 'a'.
//                  conv[2] = 'c' - 'a'.
int FindConvInUseNum(IntNode *convInUseId, int *conv, int numConv) {
 int i, convInUseNum;
 IntNode *currConv;

 convInUseNum = 0;
 for (currConv = convInUseId; currConv != NULL; currConv = currConv->next)
     for (i = 0; i < numConv; i++)
         if (conv[i] == currConv->data) {
            convInUseNum += power(2, i);
            break;
     }

 return convInUseNum;
}
```

```
int PrintBaseTable(CPRS *bTable, int treeNodeId) {
 int i, j, k, numConvSetting, currTableEntries;
 IntNode *convInUse;
 PRS *currPrs;
 RS *currRs;

 printf("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
        "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
 printf("Converters   Provide Wavelength   "
        "Request Wavelength   Partial Solution\n");
 printf("----------------------------------------------"
        "------------------------------\n");

 currTableEntries = 0;
 numConvSetting = power(2, treeNodes[treeNodeId].numConv);
 for (i = 0; i < numConvSetting; i++)
     for (currPrs = bTable[i].prs; currPrs != NULL;
          currPrs = currPrs->next)
         for (currRs = currPrs->rs; currRs != NULL;
              currRs = currRs->next) {
             currTableEntries++;
             if (i == 0)
                printf("%-13s", "NULL");
             else {//
                   convInUse = FindConvInUseId //
                     (i, treeNodes[treeNodeId].conv,
                      treeNodes[treeNodeId].numConv);
                   printf("{");
                   j = PrintIntNodeInChar(convInUse, !NEWLINE);
                   printf("}");
                   for (k = 12 - j*3; k >= 0; k--) printf(" ");
                   free(convInUse);
                  }

             if (currPrs->provWl != NULL) {
                printf("{");
                j = PrintIntNode(currPrs->provWl, !NEWLINE);
                printf("}");
                for (k = 20 - j*3; k >= 0; k--) printf(" ");
             }
             else printf("%-21s", "NULL");
```

```
            if (currRs->reqWl != NULL) {
               printf("{");
               j = PrintIntNode(currRs->reqWl, !NEWLINE);
               printf("}");
               for (k = 20 - j*3; k >= 0; k--) printf(" ");
            }
            else printf("%-21s", "NULL");

            if (currRs->parSol != NULL) {
               printf("%d, {", ListSize(currRs->parSol));
               PrintIntNodeInChar(currRs->parSol, !NEWLINE);
               printf("}\n");
            }
            else printf("0\n");
         }
 printf("---------------------------------------------------"
        "--------------------------\n");
 return currTableEntries;
}


// BuildBaseTable is to build a data structure
// to store the information for f(t, d, convInUse, provWl, reqWl);
CPRS *BuildBaseTable(int treeNodeId) {
 int i, j, numConvSetting;
 IntNode *provWl, *parSol;
 RS *rsHead;
 CPRS *cprs;
 TNodeDPtr tNodeDPtr;
 ReqWlStru *currReqWlStru;

 numConvSetting = power(2, treeNodes[treeNodeId].numConv);
 cprs = (CPRS *)calloc(numConvSetting, sizeof(CPRS));
 if (cprs == NULL) {
    printf("Out of memory when applying for cprs in buildBaseTable.\n");
    exit(1);
 }

 for (i = 0; i < numConvSetting; i++) {
     cprs[i].convInUse = i;
     cprs[i].prs = NULL;
     provWl = ProvWl(i, treeNodes[treeNodeId].conv,
```

```
                    treeNodes[treeNodeId].numConv);

     tNodeDPtr = BuildTNodeList2(provWl, treeNodes[treeNodeId].map,
                                treeNodes[treeNodeId].mapNum);

     if (tNodeDPtr.first == NULL) {
        parSol = FindConvInUseId //
          (i, treeNodes[treeNodeId].conv, treeNodes[treeNodeId].numConv);
        rsHead = NULL;
        rsHead = InsToRs(rsHead, NULL, parSol);
        cprs[i].prs = InsToPrs(cprs[i].prs, provWl, rsHead);
     }
     else {reqWlStru = NULL;
           FindReqWl(NULL, tNodeDPtr);
           parSol = FindConvInUseId //
             (i, treeNodes[treeNodeId].conv, treeNodes[treeNodeId].numConv);
           rsHead = NULL;
           for (currReqWlStru = reqWlStru; currReqWlStru != NULL;
                currReqWlStru = currReqWlStru->next)
               rsHead = InsToRs(rsHead, currReqWlStru->reqWl, parSol);
           cprs[i].prs = InsToPrs(cprs[i].prs, provWl, rsHead);
          }
 }
 return cprs;
}

// Find the intersection of intNode and intNode2 in increasing order.
// This function is related with Union
// For example,
//     if intNode = 2 -> 3 and
//        intNode2 = 1 -> 2 -> 3 -> 4,
//     then Intersect(intNode, intNode2) will return
//        2 -> 3.
// NEEDED: intNode and intNode2 are sorted in increasing order.

IntNode *Intersect(IntNode *intNode, IntNode *intNode2) {
 IntNode *currIntNode, *currIntNode2, *resIntNode;
   // resIntNode means the result intNode.

 resIntNode = NULL;
 for (currIntNode = intNode, currIntNode2 = intNode2;
      ( (currIntNode != NULL) && (currIntNode2 != NULL) ); )
```

```
    if (currIntNode->data < currIntNode2->data)
       currIntNode = currIntNode->next;
    else if (currIntNode->data == currIntNode2->data) {
            resIntNode = InsToIntNode(resIntNode, currIntNode->data);
            currIntNode = currIntNode->next;
            currIntNode2 = currIntNode2->next;
         }
         else // currIntNode->data > currIntNode2->data
              currIntNode2 = currIntNode2->next;

 return resIntNode;
}

// Generate an IntNode from a list of integers
IntNode *GenIntNode(int *conv, int numConv) {
 int i;
 IntNode *resIntNode;

 resIntNode = NULL;
 for (i = 0; i < numConv; i++)
     resIntNode = InsToIntNode(resIntNode, conv[i]+'a');

 return resIntNode;
}

// Join two integer lists together in increasing order.
// For example,
//     if intNode = 2 -> 3 and
//        intNode2 = 1 -> 2 -> 4,
//     then Union(intNode, intNode2) will return
//        1 -> 2 -> 3 -> 4.
// NEEDED: intNode and intNode2 are sorted in increasing order.

IntNode *Union(IntNode *intNode, IntNode *intNode2) {
 IntNode *currIntNode, *currIntNode2, *resIntNode;

 resIntNode = NULL;
 for (currIntNode = intNode, currIntNode2 = intNode2;
      ( (currIntNode != NULL) && (currIntNode2 != NULL) ); )
     if (currIntNode->data < currIntNode2->data) {
        resIntNode = InsToIntNode(resIntNode, currIntNode->data);
        currIntNode = currIntNode->next;
```

```
     }
     else if (currIntNode->data == currIntNode2->data) {
             resIntNode = InsToIntNode(resIntNode, currIntNode->data);
             currIntNode = currIntNode->next;
             currIntNode2 = currIntNode2->next;
          }
          else {// currIntNode->data > currIntNode2->data
                resIntNode = InsToIntNode(resIntNode, currIntNode2->data);
                currIntNode2 = currIntNode2->next;
               }

 if (currIntNode != NULL)
    for (; currIntNode != NULL; currIntNode = currIntNode->next)
         resIntNode = InsToIntNode(resIntNode, currIntNode->data);

 if (currIntNode2 != NULL)
    for (; currIntNode2 != NULL; currIntNode2 = currIntNode2->next)
         resIntNode = InsToIntNode(resIntNode, currIntNode2->data);

 return resIntNode;
}

// Perform an operation similar to set minus operation A - B.
// From the first integer list, keep only the elements
// that do not appear in the second one,
// the remaining integers should be linked in increasing order.
// For example,
//     if intNode = 2 -> 3 and
//        intNode2 = 1 -> 2 -> 4,
//     then Union(intNode, intNode2) will return
//        3
// NEEDED: intNode and intNode2 are sorted in increasing order.
//
IntNode *Minus(IntNode *intNode, IntNode *intNode2) {
 IntNode *currIntNode, *currIntNode2, *resIntNode;

 resIntNode = NULL;
 for (currIntNode = intNode, currIntNode2 = intNode2;
      ( (currIntNode != NULL) && (currIntNode2 != NULL) ); )
     if (currIntNode->data < currIntNode2->data) {
        resIntNode = InsToIntNode(resIntNode, currIntNode->data);
        currIntNode = currIntNode->next;
```

```
    }
    else if (currIntNode->data == currIntNode2->data) {
            currIntNode = currIntNode->next;
            currIntNode2 = currIntNode2->next;
         }
         else // currIntNode->data > currIntNode2->data
              currIntNode2 = currIntNode2->next;

 if (currIntNode != NULL)
    for (; currIntNode != NULL; currIntNode = currIntNode->next)
         resIntNode = InsToIntNode(resIntNode, currIntNode->data);

 return resIntNode;
}

// Decide whether the set consisting of all elements in intNode
// is a subset of the one consisting of those in intNode2.
// If intNode is a subset of or equals intNode2,
// then return TRUE;
// else return FALSE;
// Either intNode or intNode2 or both can be empty.

int IsSubset(IntNode *intNode, IntNode *intNode2) {
 int i, isSubset, maxElm;
 int *count;
 IntNode *currIntNode, *currIntNode2;

 // The following two NULL testing statements are needed,
 // otherwise, currIntNode->data when currIntNode == NULL
 // or currIntNode2->data when currIntNode2 == NULL will cause errors.

 if (intNode == NULL) return TRUE;

 if (intNode2 == NULL) return FALSE;

 // maxElm of intNode and intNode2, be aware that intNode != NULL.
 maxElm = intNode->data;

 for (currIntNode = intNode; currIntNode != NULL;
      currIntNode = currIntNode->next)
     if (maxElm < currIntNode->data)
        maxElm = currIntNode->data;
```

```
 for (currIntNode2 = intNode2; currIntNode2 != NULL;
      currIntNode2 = currIntNode2->next)
     if (maxElm < currIntNode2->data)
        maxElm = currIntNode2->data;

 count = (int *)calloc(maxElm, sizeof(int));

 if (count == NULL) {
    printf("There is no space for count in function IsSubset.\n");
    return -1;
 }

 // Step 1: initialize each element in intNode to be zero.
 for (currIntNode = intNode; currIntNode != NULL;
      currIntNode = currIntNode->next)
     count[currIntNode->data] = 0;

 // Step 2: for each element in intNode2, increase the count.
 // Hence, if an element in intNode appears in intNode2,
 // the corresponding count is 1.
 for (currIntNode2 = intNode2; currIntNode2 != NULL;
      currIntNode2 = currIntNode2->next)
     count[currIntNode2->data]++;

 // Step 3: if each element in intNode has count of 1,
 // then, by Step 2, each element in intNode appears in intNode2,
 // hence intNode is a subset of intNode2.
 // In another word, if any element in intNode does not
 // have count 1, then intNode is not a subset of intNode2.
 for (currIntNode = intNode; currIntNode != NULL;
      currIntNode = currIntNode->next)
     if (count[currIntNode->data] != 1) {
        free(count);
        return FALSE;
     }

 free(count);
 return TRUE;
}
```

```
IntNode *FindConvUseInParentChild(IntNode *convInParent, int childId) {
 int i;
 IntNode *currConv, *convUseInParentChild;

 convUseInParentChild = NULL;
 for (currConv = convInParent; currConv != NULL;
      currConv = currConv->next)
     for (i = 0; i < treeNodes[childId].numConv; i++) {
         // note that treeNodes[childId].conv[i] == char - 'a',
         if ((int)(treeNodes[childId].conv[i] +'a') == currConv->data) {
            convUseInParentChild = InsToIntNode(convUseInParentChild,
                         treeNodes[childId].conv[i]);
            break;
         }
     }

 return convUseInParentChild;
}

IntNode *FindAllWlOfParent(int parentId) {
  int i, wl;
  IntNode *resWl, *wlANode;

  resWl = NULL;

  for (i = 0; i < treeNodes[parentId].mapNum; i++)
      for (wlANode = graphNodes[treeNodes[parentId].map[i]].inciWl;
           wlANode != NULL;
           wlANode = wlANode->next)
          resWl = InsToIntNode(resWl, wlANode->data);

  return resWl;
}

// Given the appropriate convUseInParentNum and convUseInChildNum,
// combine all the appropriate items and return the prs for the
// corresponding bTable[convUseInParentNum].
// The parentPrs combined with the
// bTable[childId][convUseInChildNum].prs.
void CombineParentChild(int parentId, int convUseInParentNum,
       int childId, int convUseInChildNum) {
 IntNode *allWlOfParent, *intersectRes, *provWl, *reqWl, *parSol;
```

```
PRS *parentPrs, *childPrs, *resPrs;
RS *rsHead, *parentRs, *childRs;

allWlOfParent = FindAllWlOfParent(parentId);

for (parentPrs = bTable[parentId][convUseInParentNum].prs;
     parentPrs != NULL;
     parentPrs = parentPrs->next)
    for (childPrs = bTable[childId][convUseInChildNum].prs;
         childPrs != NULL;
        childPrs = childPrs->next) {

        provWl = Union(parentPrs->provWl, childPrs->provWl);
        provWl = Intersect(provWl, allWlOfParent);

        for (parentRs = parentPrs->rs; parentRs != NULL;
            parentRs = parentRs->next)
            for (childRs = childPrs->rs; childRs != NULL;
                childRs = childRs->next) {

              reqWl = Union(parentRs->reqWl, childRs->reqWl);

                reqWl = Minus(reqWl, provWl);

                // reqWl may not be a subset of allWlOfParent.
                // Then the request cannot be fulfilled anyhow.
                // Said differently, if reqWl cannot be satisfied,
                // then we do not add this entry to the table;
                // This is under the assumption that all the links of
                // the same wavelength form a connected subgraph.

                if ( !IsSubset(reqWl, allWlOfParent) ) continue;

                parSol = Union(parentRs->parSol, childRs->parSol);

                numCombs++;

                cprsForCombinedTable[convUseInParentNum].prs =
                  InsToPrs2(cprsForCombinedTable[convUseInParentNum].prs,
                            provWl, reqWl, parSol);
            }
    }
```

```
 return;
}

void CombineTables(int parentId, int childId) {
 int i, j, numConvSettingInParent, numConvSettingInChild;
 IntNode *convUseInParent, *convUseInChild, *parentConvIntNode,
        *childConvIntNode;

 printf("\n       parent = {");
 PrintMap(parentId, !NEWLINE);
 printf(",  ");

 printf("child = {");
 PrintMap(childId, NEWLINE);

 numConvSettingInParent = power(2, treeNodes[parentId].numConv);
 numConvSettingInChild = power(2, treeNodes[childId].numConv);

 cprsForCombinedTable = (CPRS *)calloc(numConvSettingInParent,
                                       sizeof(CPRS));
 if (cprsForCombinedTable == NULL) {
    printf("Out of memory when applying for cprsForCombinedTable"
           "in CombineTables.\n");
    exit(1);
 }

 parentConvIntNode = GenIntNode(treeNodes[parentId].conv,
                                treeNodes[parentId].numConv);
 childConvIntNode = GenIntNode(treeNodes[childId].conv,
                               treeNodes[childId].numConv);

 for (i = 0; i < numConvSettingInParent; i++) {
     cprsForCombinedTable[i].convInUse = i;
     cprsForCombinedTable[i].prs = NULL;
     // Find the converters used by the parent.
     convUseInParent = FindConvInUseId //...
       (i, treeNodes[parentId].conv, treeNodes[parentId].numConv);

     for (j = 0; j < numConvSettingInChild; j++) {
         convUseInChild = FindConvInUseId //...
          (j, treeNodes[childId].conv, treeNodes[childId].numConv);
```

```
        if (CompIntNode(Intersect(convUseInChild, parentConvIntNode),
             Intersect(convUseInParent, childConvIntNode) ) == 0)
           CombineParentChild(parentId, i, childId, j);
    }
 }
 FreeIntNode(parentConvIntNode);
 FreeIntNode(childConvIntNode);
 return;
}

int main(void) {
 int i, tableEntries;
 IntNode *currLevelPtr;
 AdjNode *currAdjNode;
 CPRS *parentCprs;

 ReadGraphNodes();

 ReadTreeNodes();

 bTable = (CPRS **)calloc(numTreeNodes, sizeof(CPRS *));
 if (bTable == NULL) {
    printf("Out of memory when applying for bTable in main.\n");
    exit(1);
 }

 levels = (IntNode **)calloc(numTreeNodes, sizeof(IntNode *));
 if (levels == NULL) {
    printf("Out of memory when applying for levels in main.\n");
    exit(1);
 }

 BuildRootedTree(0, -1); //  0 is the nodeId of the root,
                         // -1 is the parent of nodeId.
                         // Decide which node is in which level.
 numCombs = 0;
 tableEntries = 0;
 // maxLevel starts counting from level 0.
 // Each layer consists of the tree nodeId that are in this layer.
 for (i = maxLevel; i >= 0; i--)
     for (currLevelPtr = levels[i]; currLevelPtr != NULL;
```

```
          currLevelPtr = currLevelPtr->next) {
         bTable[currLevelPtr->data] = BuildBaseTable(currLevelPtr->data);

         printf("\n       parent = {");
         PrintMap(currLevelPtr->data, !NEWLINE);
         printf(",  child = none\n");

         tableEntries += PrintBaseTable(bTable[currLevelPtr->data],
                                       currLevelPtr->data);

         for (currAdjNode = treeNodes[currLevelPtr->data].firstAdjNode;
              currAdjNode != NULL;
              currAdjNode = currAdjNode->next)
              if ( !(currAdjNode->isParent) ) {
                 CombineTables(currLevelPtr->data, currAdjNode->nodeId);
                 parentCprs = bTable[currLevelPtr->data];
                 bTable[currLevelPtr->data] = cprsForCombinedTable;
                 tableEntries += PrintBaseTable(bTable[currLevelPtr->data],
                                               currLevelPtr->data);
             }
     }

 printf("Total number of table entries = %d.\n", tableEntries);
 printf("Total number of combinations = %d.\n", numCombs);
 return 0;
}
```

Vita

Tong Yi was born on December 22, 1972 in Yulin, China. She earned her bachelor and master degrees, both in computer science, at Huazhong University of Science and Technology in May 1993 and May 1998, respectively. From May 1993 to August 1995, she worked in Foshan Scientific and Teaching Equipment Company, China. In January 1999 she came to Louisiana State University (LSU) to pursue graduate studies in computer science. She earned a master degree in system science from LSU in December 2000. She is currently a candidate for the degree of Doctor of Philosophy in computer science, which will be awarded in December 2005.