1.7 Packet Routing

All of the algorithms described thus far in the chapter have the property that the right data always manages to get to the right place at the right time. Although making sure this happened was sometimes tricky, the flow of data almost always followed a regular pattern. For some applications (e.g., circuit simulation), this will not be the case, and we may need to route the data in a very nonregular fashion. In general, we may have to solve several packet routing problems just to implement a single algorithm.

A *packet routing problem* consists of a set of M packets, each with a desired destination address p_i. Initially, the packets are stored individually among the N nodes of the network, and for the most part we will assume that the desired destinations are all different (i.e., that $p_i \neq p_j$ for $1 \leq i < j \leq M$). (Such problems are called *one-to-one* routing problems.) The problem is to route the packets to their desired destinations using local control in as few steps as possible.

We will describe a variety of algorithms for packet routing in this section. We start with greedy algorithms in Subsection 1.7.1. Greedy algorithms run optimally on a linear array, but do not work as well on a two-dimensional array. For example, a greedy algorithm can be made to run in $2\sqrt{N} - 2$ steps (the fewest possible, in general) on a $\sqrt{N} \times \sqrt{N}$ array, but only if we allow queues of packets to build up at some processors. In the worst case, some queues can grow to contain as many as $\Theta(\sqrt{N})$ packets. For random (i.e., average case) routing problems, however, the situation is better. For example, we show in Subsection 1.7.2 that for random routing problems, the maximum queue size needed is a small constant with probability close to 1. We also analyze a dynamic model of routing in which packets are generated at random over a long period of time. The material in Subsection 1.7.2 provides our first probabilistic analysis of an algorithm. The probabilistic methods developed in this subsection will be used quite heavily in later chapters.

In Subsection 1.7.3, we describe and analyze a simple randomized algorithm for packet routing. The randomized algorithm solves any one-to-one routing problem on a $\sqrt{N} \times \sqrt{N}$ array in $2\sqrt{N} + o(\sqrt{N})$ steps using queues of size $O(\log N)$ with high probability. Randomized algorithms are often better than algorithms that work well on average because the probability that a randomized algorithm fails is independent of the problem being solved. This is not true of algorithms that work well for random problems.

(In other words, there is no worst-case input for a randomized algorithm.) The material in Subsection 1.7.3 is quite useful and will be developed further in Chapter 3.

In Subsection 1.7.4, we describe deterministic algorithms for packet routing that precondition the packets using the sorting algorithms from Section 1.6. The best of these algorithms run in nearly $2\sqrt{N}$ steps and have constant size queues for all routing problems. Unfortunately, the algorithms become more complicated as the running time improves.

In Subsection 1.7.5, we describe a very simple algorithm for off-line packet routing. The off-line routing problem is the same as the on-line routing problem studied in Subsections 1.7.1–1.7.4, except that we are allowed to perform some global precomputation before the routing begins. Off-line algorithms are generally less useful than on-line algorithms since the routing problem must be known (and, in some sense, solved) in advance. The algorithm described in Subsection 1.7.5 is so simple, however, that we will make use of it at several points later in the text.

For the most part, we concentrate on one-to-one routing problems in the word model in this section. There are many other routing models of interest, however, and we conclude in Subsection 1.7.6 with a brief discussion of different routing models and algorithms. In particular, we mention some of the issues involved in many-to-one routing problems (such as combining), and in bit-serial routing models. Much of this material will be developed more fully in Chapter 3, when we discuss message routing algorithms at length.

1.7.1 Greedy Algorithms

Any algorithm that routes every packet along a shortest path to its destination can be considered to be a *greedy algorithm*. For example, consider the following algorithm for routing packets on a linear array. At each step, each packet that still needs to move rightward or leftward does so. The algorithm terminates when all packets have reached their destination. We will refer to this algorithm as *the* greedy algorithm for linear arrays.

The first thing to notice about the greedy algorithm for linear arrays is that it is well defined, at least if (as we will often assume) each processor contains just one packet at the beginning and the end of the routing. In particular, two packets will never be contending for use of the same edge (in the same direction) at the same time. Hence, whenever a packet is supposed to move according to the algorithm, it is able to do so. This is

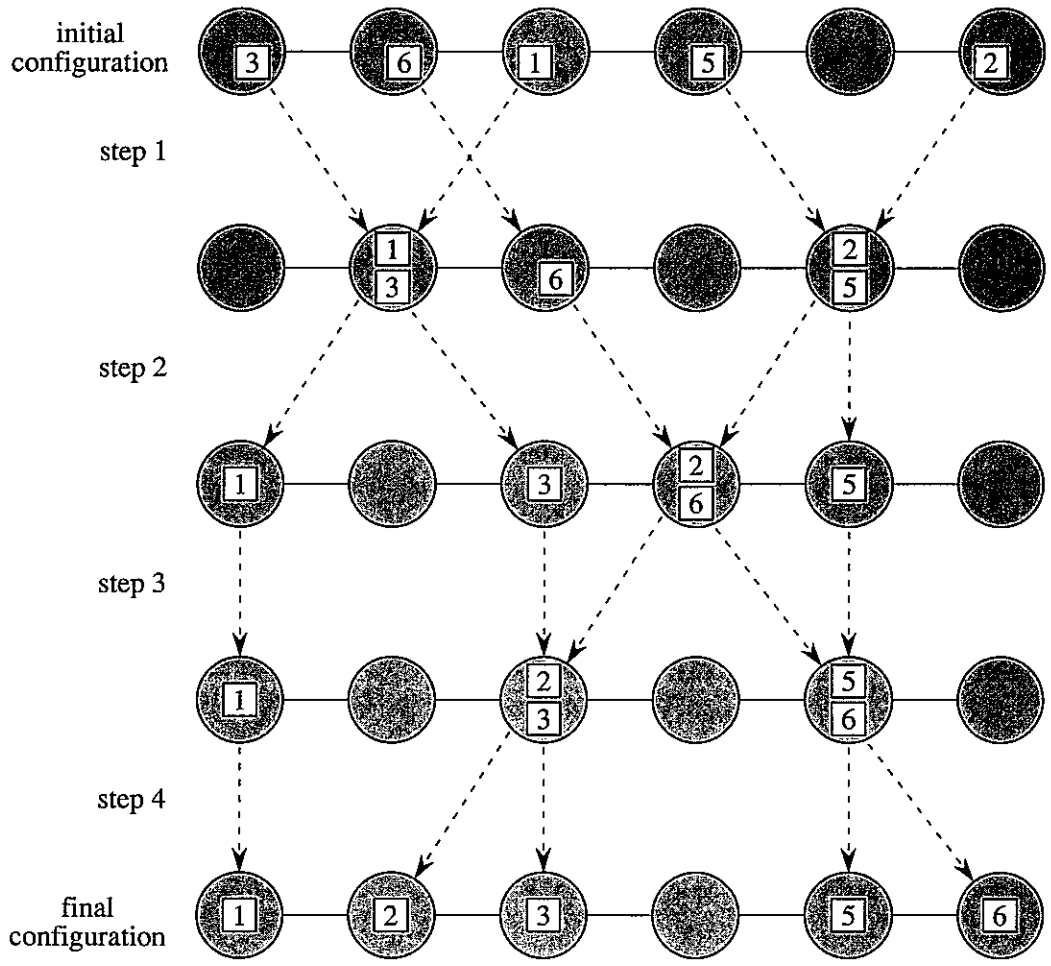

Figure 1-89 *Routing five packets with the greedy routing algorithm on a 6-cell linear array. Each packet keeps moving until it reaches its destination.*

not to say that two packets will never reside in the same processor at the same time (this can happen), but no two packets which are travelling in the same direction will ever reside in the same processor at the same time.

Another important fact about the greedy algorithm on a linear array is that each packet reaches its destination in d steps, where d is the distance that the packet needs to travel. This is because the distance between each packet and its destination decreases by one during each step of the algorithm. Hence, the algorithm always terminates in at most $N-1$ steps. For example, see Figure 1-89.

Unfortunately, matters aren't nearly as simple when we run a greedy algorithm on most other networks. One of the biggest problems that arises with other networks is that two or more packets might be contending for

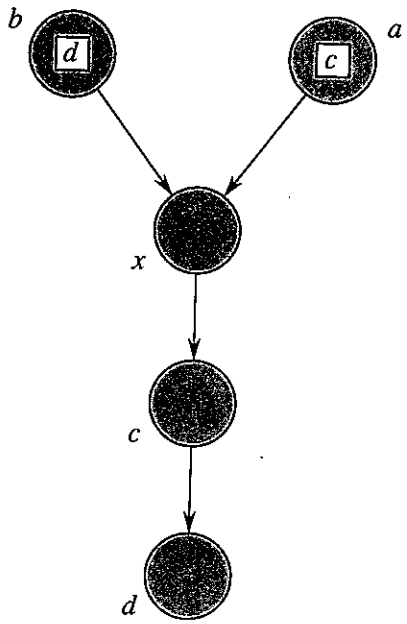

Figure 1-90 *A packet routing problem that leads to contention for the edge leading out of node x at step 2.*

use of the same edge (in the same direction) at the same time. For example, consider the situation illustrated in Figure 1-90. In this example, the packets destined for processors c and d move to node x on the first step. At the second step, both packets contend for the edge leading from node x into node c. Since only one packet can advance, the other must be queued, and wait until later before it can proceed.

It is not difficult to see that the example illustrated in Figure 1-90 can be made much worse by having two streams of packets merge into a single stream at node x. For example, see Figure 1-91. In such a case, the queue at node x might grow to contain as many as N packets, unless we constrain the algorithm not to advance packets into a processor that has a large queue.

There are many strategies for arbitrating between packets that are contending for the same edge (e.g., the packet that needs to go farthest goes first), and for preventing the buildup of large queues (e.g., we could preset a maximum threshold q, and simply not advance a packet forward into a processor with a queue that is at or near the threshold). Not surprisingly, the choice of queueing protocol can have a substantial impact on the performance of the resulting algorithm. In what follows, we will

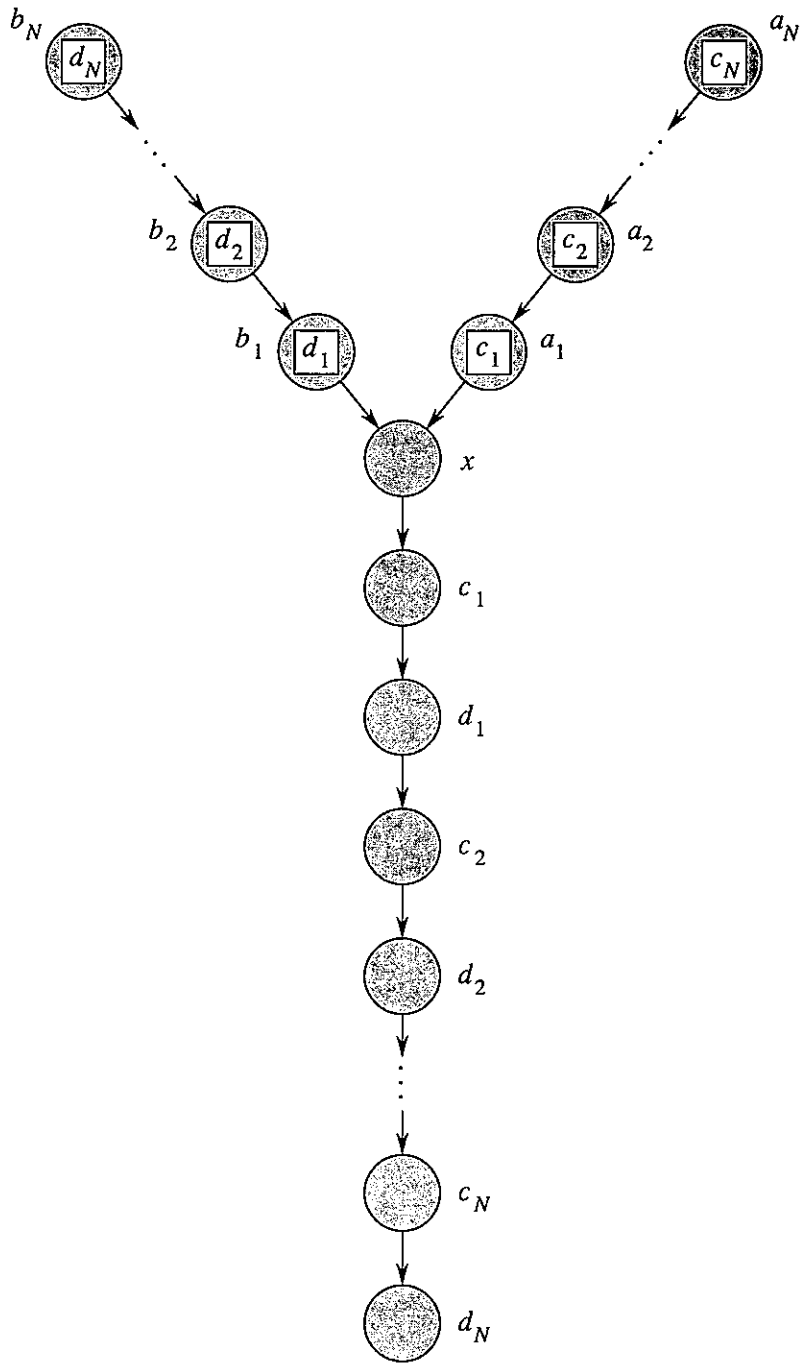

Figure 1-91 *A packet routing problem which leads to a queue of size N in node x, unless the packets in nodes $a_1, \ldots, a_N, b_1, \ldots, b_N$ are restricted from advancing because of a large queue ahead.*

consider the scenario in which queues are allowed to grow arbitrarily and edge contention is resolved by giving priority to the packet that needs to travel farthest in that direction. (We call this the *farthest-first* contention resolution protocol.) We will show that this simple protocol results in a $(2\sqrt{N} - 2)$-step routing algorithm for a $\sqrt{N} \times \sqrt{N}$ array if each packet is first routed to its correct column, and then on to its destination within that column. For simplicity, we will sometimes refer to this algorithm as the *basic greedy algorithm* for two-dimensional arrays. For example, we have illustrated the operation of the algorithm for a 3×3 routing problem in Figure 1-92. Variations of the greedy algorithm derived by altering the queueing and contention resolution protocols are considered in the exercises.

Analysis of the Basic Greedy Algorithm on an Array

The analysis of the running time of the basic greedy algorithm on a $\sqrt{N} \times \sqrt{N}$ array is divided into two stages. The first stage focuses on the routing activity that takes place in the rows during the first $\sqrt{N} - 1$ steps.

The first fact to observe about the basic greedy algorithm is that every packet reaches the correct column during the first $\sqrt{N} - 1$ steps. This is because there is never any contention for row edges. Each row acts like a linear array—packets needing to move rightward or leftward do so in lockstep fashion. Of course, it is possible for packets to pile up at a node, but the buildup does not affect the analysis of the running time during this stage of the algorithm.

After the first $\sqrt{N} - 1$ steps, therefore, every packet is in the correct column. In addition, some packets may have even initiated movement within a column toward their destination, although we will not need to make use of this fact in our analysis. Unfortunately, however, several of the packets within a column might be piled up in large queues, and so we cannot naively apply the same analysis that we used for linear arrays to argue that the column routing is completed in $\sqrt{N} - 1$ steps. In fact, if we use the wrong protocol to arbitrate edge contention, we might need many more than $\sqrt{N} - 1$ steps to finish up. By giving priority to the packets that need to go farthest, however, all of the column routing can be accomplished in $\sqrt{N} - 1$ steps, resulting in a total of $2\sqrt{N} - 2$ steps overall. This fact is proved in the following lemma.

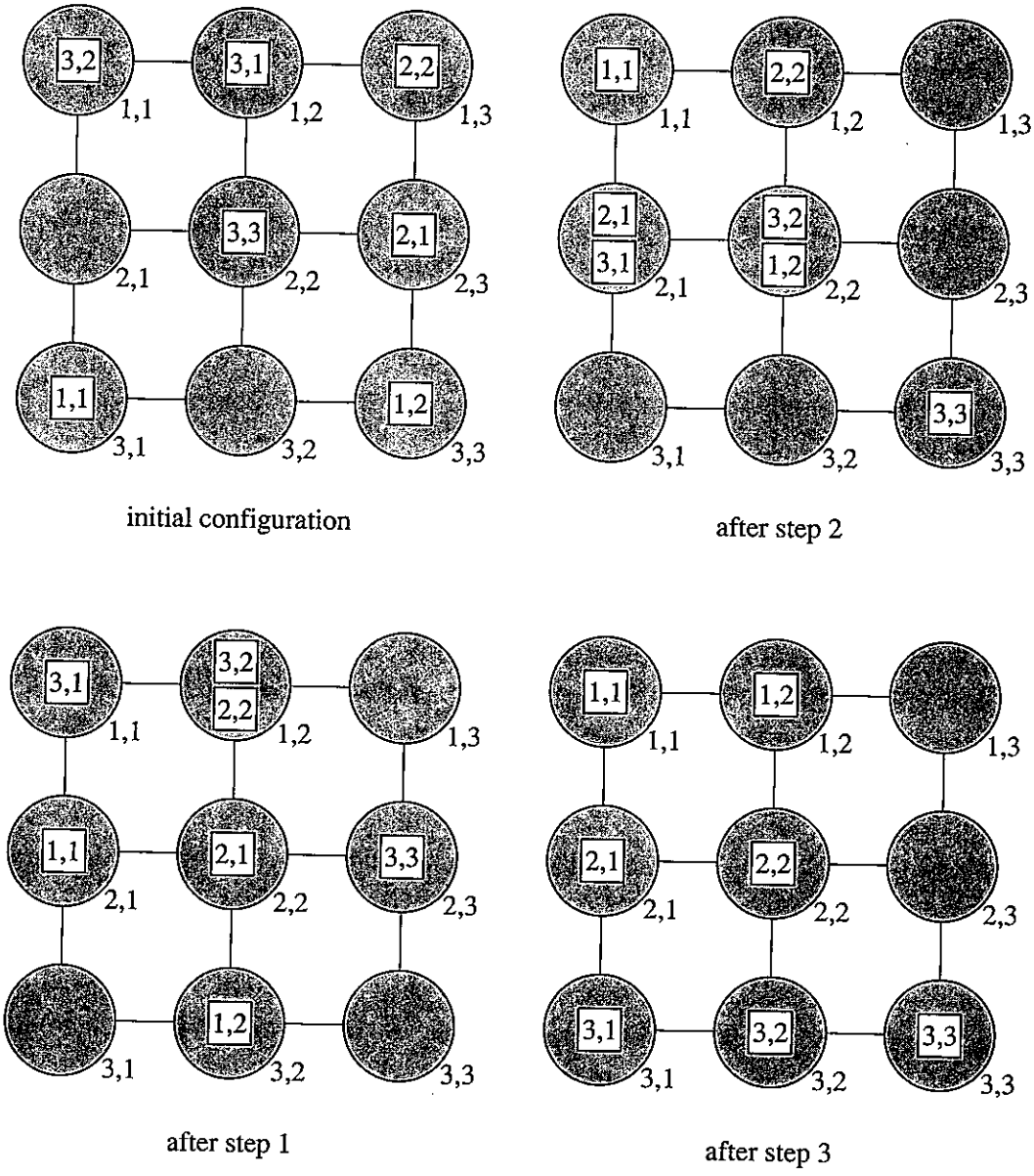

Figure 1-92 *Operation of the basic greedy algorithm on a* 3×3 *array. Each packet moves left or right until it reaches the correct column, and then moves up or down until it reaches the correct row (i.e., its destination). Edge contention is resolved by giving priority to the packet that needs to go farthest in that direction (e.g., the packet destined for node* $3,2$ *moves ahead of the packet destined for node* $2,2$ *during step 2). No constraint is placed on queue size.*

LEMMA 1.5 *Consider an N-node linear array in which each node contains an arbitrary number of packets, but for which there is at most one packet destined for each node. If edge contention is resolved by giving priority to the packet that needs to go farthest, then the greedy algorithm routes all the packets in $N-1$ steps.*

Proof. Since the leftward and rightward moving packets never interfere with each other, we can restrict our attention to the rightward moving packets without loss of generality. To show that $N-1$ steps are sufficient to route the rightward moving packets, we will use an argument that is reminiscent of the analysis of odd-even transposition sort presented in Subsection 1.6.1. In particular, for each i, we will show that each of the packets that is destined for one of the rightmost i nodes has reached one of the rightmost i nodes within $N-1$ steps. Note that if this condition holds for all i $(1 \leq i \leq N)$ simultaneously, then the routing of rightward moving packets is completed in $N-1$ steps, thereby proving the lemma.

Fix i $(1 \leq i \leq N)$ and consider the packets destined for the rightmost i nodes. For ease of reference, we will call these packets *priority packets.* The first point to notice about priority packets is that they can never be delayed by a nonpriority packet. This is because if a priority packet and a nonpriority packet are contending for an edge, then the priority packet moves ahead since (by definition) it has farther to go. Hence, we can ignore the nonpriority packets entirely when analyzing the movement of the priority packets.

Consider the rightmost priority packet at the start of the algorithm. (If there is a tie, then pick the packet that will move on the first step.) Since this packet cannot be delayed by nonpriority packets, it moves rightward at each step until it reaches the rightmost i nodes. This happens within $N-i$ steps. More importantly, the packet is never located in the same node with another priority packet after the first step (until it reaches its destination, that is). This is because the other priority packets can't catch up. Hence, this packet can't delay any of the other priority packets after the first step.

Next, consider the second rightmost priority packet after the first step. (Break ties by choosing the packet that will move at the second step.) Although this packet might have been delayed during the first step, it cannot be delayed on subsequent steps since it never encounters another priority packet after the second step. This is because following

packets cannot catch up, and the preceding packet is not slowed down. Hence, the second priority packet reaches the rightmost i cells within $N-i+1$ steps, and neither of the first two packets can delay any of the other priority packets after step 2.

By continuing to argue in this fashion, we find that the ith rightmost priority packet after step $i-1$ (if there is one) cannot be delayed after step $i-1$, because all of the other $i-1$ (at most) priority packets are already on their way rightward and they cannot be slowed down after step $i-2$. At worst, the ith packet is still in the first node after step $i-1$, and has $N-i$ edges to traverse before reaching the last i nodes. Hence, the last priority packet reaches the rightmost i nodes within $N-1$ steps, thereby completing the proof of the lemma. ■

By Lemma 1.5, we know that the column routing is completed within $\sqrt{N}-1$ steps after the last packet reaches its correct column. Hence, all the packets reach their destination within $2\sqrt{N}-2$ steps. In general, this is the best that we can hope for since a packet might have to travel from processor (1,1) to processor $(\sqrt{N},\sqrt{N})$, and this will always take at least $2\sqrt{N}-2$ steps.

Although the basic greedy algorithm is optimal in terms of worst-case running time, the maximum queue size can be as large as $\frac{2}{3}\sqrt{N}-1$ in the worst case. To see why, consider the routing problem where the packets in processors $(1,2), (1,3), \ldots, (1,\frac{\sqrt{N}}{3})$, and $(2,1), (2,2), \ldots, (2,\frac{2\sqrt{N}}{3}-1)$ are destined for processors $(3,\frac{\sqrt{N}}{3}), (4,\frac{\sqrt{N}}{3}), \ldots, (\sqrt{N},\frac{\sqrt{N}}{3})$. All of these $\sqrt{N}-2$ packets arrive in processor $(2,\frac{\sqrt{N}}{3})$ within $\frac{\sqrt{N}}{3}-1$ steps, but only $\frac{\sqrt{N}}{3}-1$ of them can be passed across the edge from processor $(2,\frac{\sqrt{N}}{3})$ to processor $(3,\frac{\sqrt{N}}{3})$ during this time. Hence, the queue of packets waiting to go from processor $(2,\frac{\sqrt{N}}{3})$ to processor $(3,\frac{\sqrt{N}}{3})$ will eventually become as large as $\sqrt{N}-2-(\frac{\sqrt{N}}{3}-1)=\frac{2}{3}\sqrt{N}-1$.

Fortunately, things aren't nearly so bad on average. For example, we will show in the next subsection that if each packet is headed to a random destination, then at most $O(1)$ packets are ever contained in the same queue at the same time, with probability very close to 1. We will also show that packets are very rarely delayed, on average. In fact, we will show that the expected number of times a packet is delayed on the way to its destination is a constant, independent of N or the distance travelled by the packet.

3.1 The Hypercube

In this section, we define the hypercube and explain why it is such a powerful network for parallel computation. Among other things, we will show how the hypercube can be used to simulate all of the networks discussed in Chapters 1 and 2. In fact, we will find that the hypercube contains (or nearly contains) all of these networks as subgraphs. This material is both surprising and important because it demonstrates how all of the parallel algorithms discussed thus far can be directly implemented on the hypercube without significantly affecting the number of processors or the running time. Hence, we will quickly understand one of the main reasons why the hypercube is so powerful.

We begin the section with some definitions and a brief discussion of the hypercube's simplest properties in Subsection 3.1.1. In Subsection 3.1.2, we show that the hypercube is Hamiltonian, and we explain the correspondence between Hamiltonian cycles in the hypercube and Gray codes. We also prove that any N-node array (of any dimensionality) is a subgraph of the N-node hypercube (assuming that N is a power of 2).

In Subsection 3.1.3, we describe several embeddings of an $(N-1)$-node complete binary tree in an N-node hypercube. Although the $(N-1)$-node complete binary tree is not a subgraph of the N-node hypercube, we will find that a complete binary tree can be simulated very efficiently on the hypercube.

More generally, we will show that the N-node hypercube can efficiently simulate any binary tree in Subsection 3.1.4. In particular, we will show how to grow an arbitrary M-node binary tree in an on-line fashion in an N-node hypercube so that neighboring nodes in the tree are nearby in the hypercube and so that at most $O(M/N+1)$ tree nodes are mapped to each hypercube node with high probability. The analysis of the tree-growing algorithm involves an interesting relationship between one-error-correcting codes and hypercubes that has numerous applications. We also define the hypercube of cliques in Subsection 3.1.4 and prove that it is computationally equivalent to the hypercube.

In Subsection 3.1.5, we show how to efficiently simulate a mesh of trees on the hypercube. As a consequence, we will find that all of the algorithms described in Chapter 2 can be implemented without significant slowdown on a hypercube of approximately the same size.

We conclude in Subsection 3.1.6 with a brief survey of some related

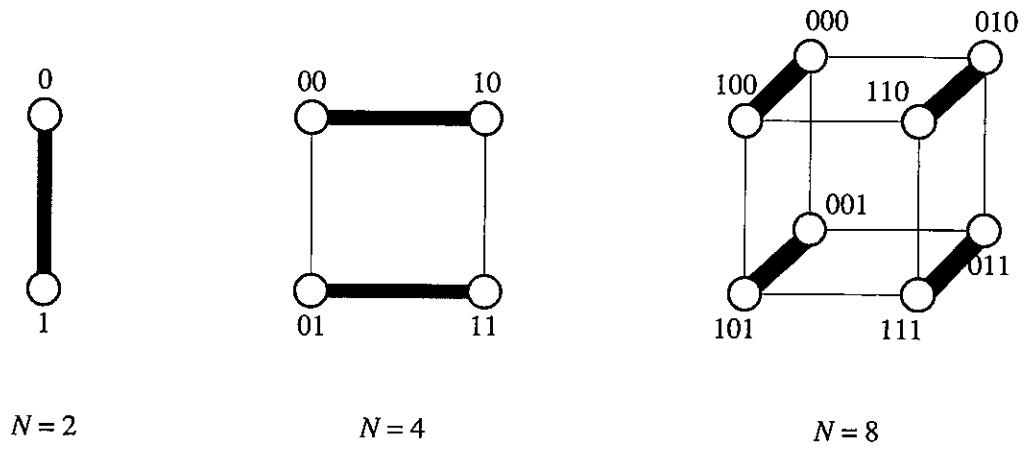

Figure 3-1 *The N-node hypercube for $N = 2$, 4, and 8. Two nodes are linked with an edge if and only if their strings differ in precisely one bit position. Dimension 1 edges are shown in boldface.*

network containment and simulation results for the hypercube.

3.1.1 Definitions and Properties

The *r-dimensional hypercube* has $N = 2^r$ nodes and $r2^{r-1}$ edges. Each node corresponds to an r-bit binary string, and two nodes are linked with an edge if and only if their binary strings differ in precisely one bit. As a consequence, each node is incident to $r = \log N$ other nodes, one for each bit position. For example, we have drawn the hypercubes with 2, 4, and 8 nodes in Figure 3-1.

The edges of the hypercube can be naturally partitioned according to the dimensions that they traverse. In particular, an edge is called a *dimension k edge* if it links two nodes that differ in the kth bit position. We will use the notation u^k to denote the neighbor of node u across dimension k in the hypercube. In particular, given any binary string $u = u_1 \cdots u_{\log N}$, the string u^k is the same as u except that the kth bit is complemented. More generally, we will use the notation $u^{\{k_1,k_2,\ldots,k_s\}}$ to denote the string formed by complementing the k_1th, k_2th, ..., and k_sth bits of u. For example, $0011010^2 = 0111010$ and $0011010^{\{3,4\}} = 0000010$ in a 128-node hypercube.

The dimension k edges in a hypercube form a perfect matching for each k, $1 \leq k \leq \log N$. (Recall that a perfect matching for an N-node graph is a set of $N/2$ edges that do not share any nodes.) Moreover, removal of the dimension k edges for any $k \leq \log N$ leaves two disjoint copies of an

$\frac{N}{2}$-node hypercube. Conversely, an N-node hypercube can be constructed from two $\frac{N}{2}$-node hypercubes by simply connecting the ith node of one $\frac{N}{2}$-node hypercube to the ith node of the other for $0 \leq i < \frac{N}{2}$. For example, see Figure 3-2.

In addition to a simple recursive structure, the hypercube also has many of the other nice properties that we would like a network to have. In particular, it has low diameter ($\log N$) and high bisection width ($N/2$). The bound on the diameter is easily proved by observing that any two nodes $u = u_1 u_2 \cdots u_{\log N}$ and $v = v_1 v_2 \cdots v_{\log N}$ are connected by the path

$$u_1 u_2 \cdots u_{\log N} \rightarrow v_1 u_2 \cdots u_{\log N} \rightarrow v_1 v_2 u_3 \cdots u_{\log N}$$
$$\rightarrow \cdots \rightarrow v_1 v_2 \cdots v_{\log N - 1} u_{\log N} \rightarrow v_1 v_2 \cdots v_{\log N}.$$

The bound on bisection width is established by showing that the smallest bisection consists of the edges in a single dimension. The proof follows as a special case of Theorem 1.21 from Section 1.9.

As an interesting aside, it is worth noting that a hypercube can be bisected by removing far fewer than $\frac{N}{2}$ nodes, even though $\frac{N}{2}$ edges are required to bisect the N-node hypercube. For example, consider the partition formed by removing all nodes with size $\left\lfloor \frac{\log N}{2} \right\rfloor$ and $\left\lceil \frac{\log N}{2} \right\rceil$. (The *size*, or *weight*, of a node in the hypercube is the number of 1s contained in its binary string.) A simple calculation reveals that removal of these nodes forms a bisection with $\Theta(N/\sqrt{\log N})$ nodes, which is the best possible. The details of these and some related results are left to the exercises (see Problems 3.3–3.7).

It is also worth noting that the hypercube possesses many symmetries. For example, it is *node* and *edge symmetric*. In other words, by just relabelling nodes, we can map any node onto any other node, and any edge onto any other edge. More precisely, for any pair of edges (u, v) and (u', v') in an N-node hypercube H, there is an automorphism σ of H such that $\sigma(u) = u'$ and $\sigma(v) = v'$. (An *automorphism* of a graph is a one-to-one mapping of the nodes to the nodes such that edges are mapped to edges.) In fact, there are many such automorphisms. For example, let $u = u_1 u_2 \cdots u_{\log N}$, $u' = u'_1 u'_2 \cdots u'_{\log N}$, k be the dimension of (u, v), and k' be the dimension of (u', v'). Then for any permutation π on $\{1, 2, \ldots, \log N\}$ such that $\pi(k') = k$, we can define an automorphism σ with the desired property by setting

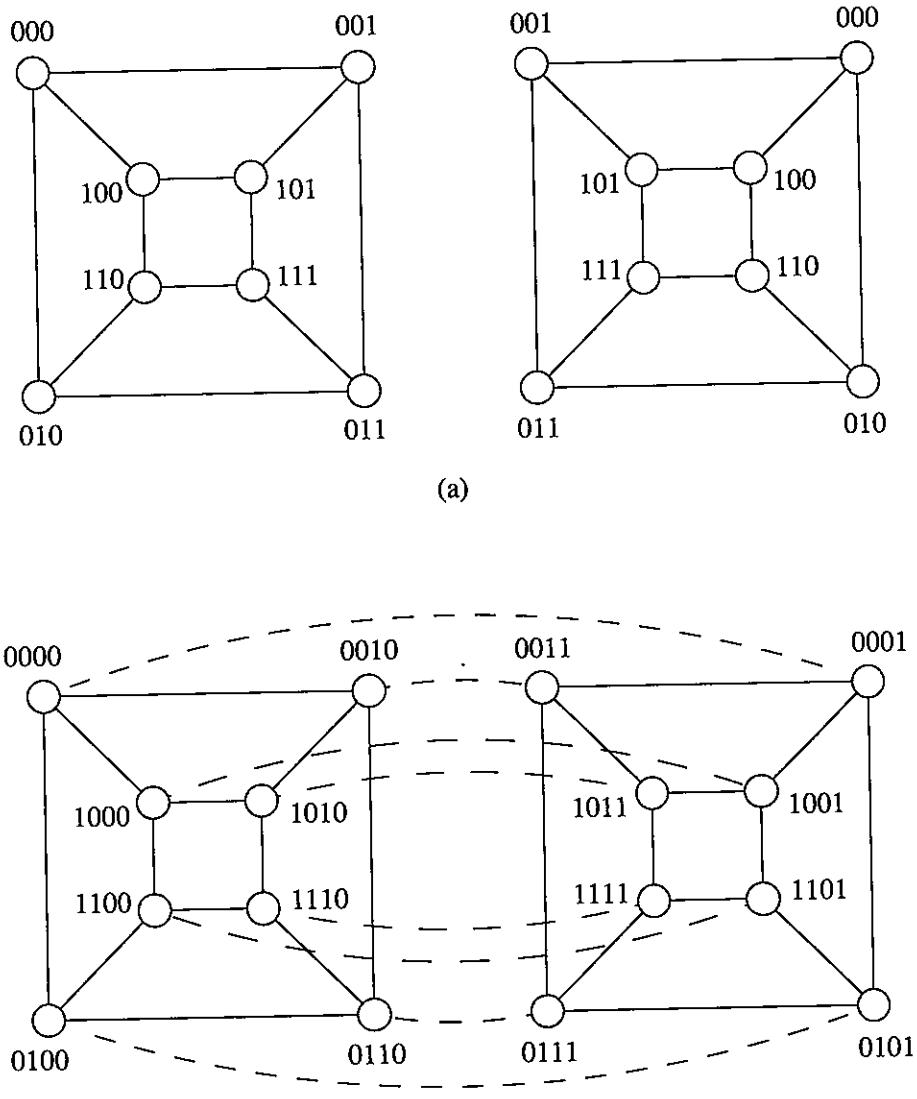

Figure 3-2 *Construction of a four-dimensional hypercube (b) from two three-dimensional hypercubes (a). Dashed edges form a matching between the two three-dimensional cubes.*

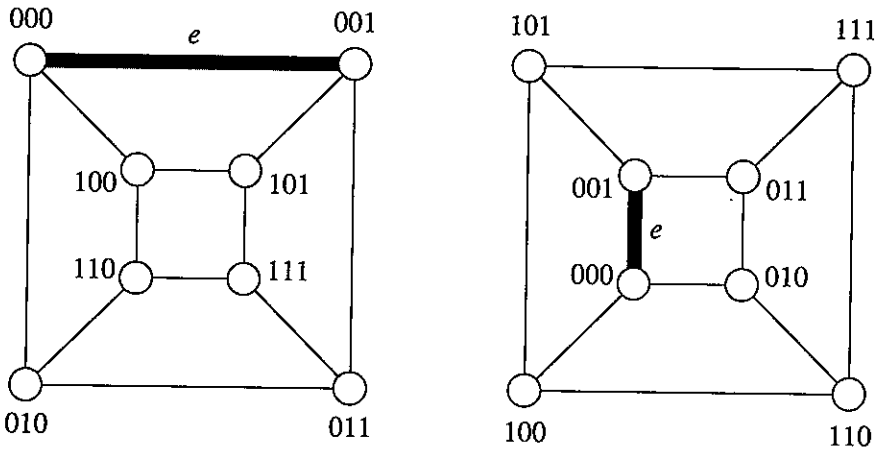

Figure 3-3 *Two labellings of the 8-node hypercube. By relabelling appropriately, we could have mapped edge $e = (000, 001)$ to any position in the network.*

$$\sigma(x_1 x_2 \cdots x_{\log N}) = (x_{\pi(1)} \oplus u_{\pi(1)} \oplus u'_1) \mid (x_{\pi(2)} \oplus u_{\pi(2)} \oplus u'_2) \mid \cdots \mid (x_{\pi(\log N)} \oplus u_{\pi(\log N)} \oplus u'_{\log N}). \quad (3.1)$$

(Here and throughout Chapter 3, we use the notation $\alpha \mid \beta$ to denote the concatenation of α and β.) It is a simple exercise to check that σ is an automorphism of the hypercube with the desired properties. (See Problem 3.10.)

As an example, we have illustrated two labellings of the 8-node hypercube in Figure 3-3. In the example, we have mapped the edge $(000, 001)$ to edge $(110, 100)$ using the automorphism

$$\sigma(x_1 x_2 x_3) = (x_1 \oplus 1) \mid (x_3 \oplus 1) \mid x_2.$$

In general, we can rearrange the dimensions of the edges in any order that we want (by varying π) without altering the network. (See Problems 3.11–3.13.) We will use such symmetries routinely in the chapter to simplify explanations.

3.1.2 Containment of Arrays

One of the most interesting properties of the N-node hypercube is that it contains every N-node array as a subgraph. This result holds true even for high-dimensional arrays and even if wraparound edges are allowed. For example, the embedding of a 4×4 array in a 16-node hypercube is shown

3.2 The Butterfly, Cube-Connected-Cycles, and Beneš Network

Although the hypercube is quite powerful from a computational point of view, there are some disadvantages to its use as an architecture for parallel computation. One of the most obvious disadvantages is that the node degree of the hypercube grows with its size. This means, for example, that processors designed for an N-node hypercube cannot later be used in a $2N$-node hypercube. Moreover, the complexity of the communications portion of a node can become fairly large as N increases. For example, every node in a 1024-processor hypercube has 10 neighbors, and every node in a one million-processor hypercube has 20 neighbors.

In order to circumvent the difficulties associated with node degrees in hypercubes, several variations of the hypercube have been devised that have similar computational properties but bounded degree (usually 3 or 4). In this section, we discuss three such variations: the butterfly, the cube-connected-cycles, and the Beneš network. Each is a simple variation of the hypercube, and each is very similar to the others in structure.

The section is divided into four subsections. In Subsection 3.2.1, we define the networks and mention some of their most important properties, including their relationships to each other as well as to the hypercube. In particular, we show that the butterfly, cube-connected-cycles, and Beneš network are virtually identical from a computational standpoint. We also show how to partition the edges of a Beneš network into paths that connect the nodes at the first level to the last level in any desired pattern. This powerful property of the Beneš network will be used at several points later in the chapter and should not be overlooked. For example, we apply the result in Subsection 3.2.2 to show that an N-node butterfly, cube-connected-cycles or Beneš network can simulate any other bounded-degree N-node network with an $O(\log N)$-factor slowdown, the least possible in general. Similar results hold for the hypercube, shuffle-exchange, and de Bruijn graphs. Hence, we will find that the hypercubic networks are *universal* in the sense that they can simulate any other network with a comparable number of processors and wires with only a logarithmic factor slowdown.

The relationship between the butterfly-related networks and the hypercube is explored further in Subsection 3.2.3, where we show how to simulate any normal hypercube algorithm on the butterfly, cube-connected-cycles, and Beneš network with only constant slowdown. This material is par-

ticularly important since it means that all of the algorithms described in Chapter 2 can be implemented on these networks with only a constant factor loss in efficiency.

We conclude in Subsection 3.2.4 with a review of network containment results analogous to those proved for the hypercube in Subsections 3.1.2–3.1.6. Among other things, we find that the butterfly, cube-connected-cycles, and Beneš network contain linear arrays and complete binary trees with constant dilation, but that any embedding of higher-dimensional arrays requires logarithmic dilation, which is the worst possible. We also prove a general result that every N-node connected network contains an N-node linear array with dilation 3.

3.2.1 Definitions and Properties

In what follows, we describe the butterfly, a variant of the butterfly called the wrapped butterfly, the cube-connected-cycles, and the Beneš network. All four networks have a similar structure, and all four are computationally equivalent.

The Butterfly

The r-dimensional butterfly has $(r+1)2^r$ nodes and $r2^{r+1}$ edges. The nodes correspond to pairs $\langle w, i\rangle$ where i is the *level* or *dimension* of the node ($0 \leq i \leq r$) and w is an r-bit binary number that denotes the *row* of the node. Two nodes $\langle w, i\rangle$ and $\langle w', i'\rangle$ are linked by an edge if and only if $i' = i + 1$ and either:

1) w and w' are identical, or

2) w and w' differ in precisely the i'th bit.

If w and w' are identical, the edge is said to be a *straight edge*. Otherwise, the edge is a *cross edge*. For example, see Figure 3-19. In addition, edges connecting nodes on levels i and $i+1$ are said to be *level $i+1$ edges*.

The butterfly and hypercube are quite similar in structure. In particular, the ith node of the r-dimensional hypercube corresponds naturally to the ith row of the r-dimensional butterfly, and an ith dimension edge (u, v) of the hypercube corresponds to cross edges $(\langle u, i-1\rangle, \langle v, i\rangle)$ and $(\langle v, i-1\rangle, \langle u, i\rangle)$ in level i of the butterfly. In effect, the hypercube is just a folded up butterfly (i.e., we can obtain a hypercube from a butterfly by merging all butterfly nodes that are in the same row and then removing the extra copy of each edge). Hence, any single step of N-node hypercube

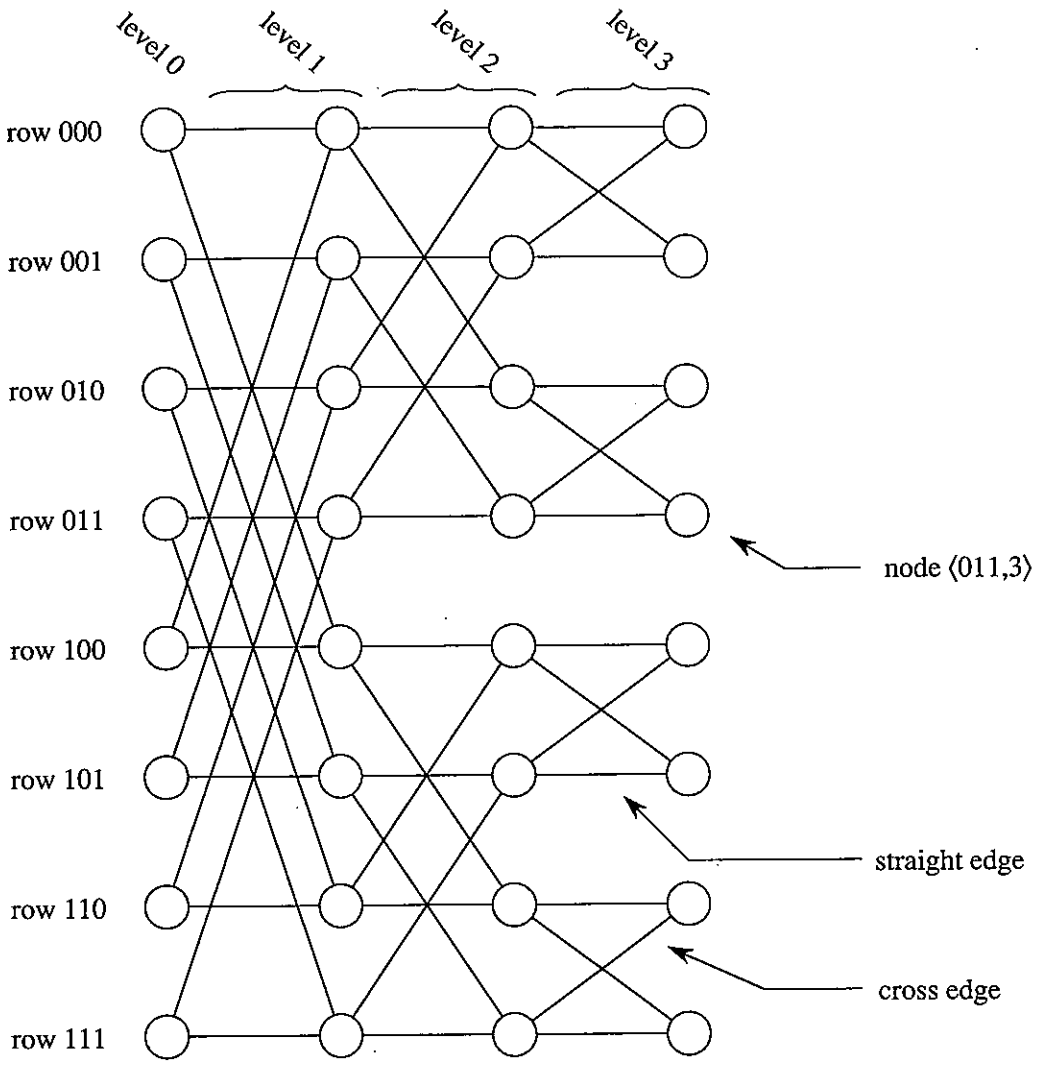

Figure 3-19 *The three-dimensional butterfly. Level i straight edges link nodes in the same row for $0 < i \leq r$. Level i cross edges link nodes in rows that differ in the ith bit.*

calculation can be simulated in $\log N$ steps on an $N(\log N + 1)$-node butterfly by having the ith row of the butterfly simulate the operation of the ith node of the hypercube for each i.

Because of the great similarity between the butterfly and the hypercube, the butterfly has several nice properties. First, it has a simple recursive structure. For example, it can be seen from Figure 3-19 that one r-dimensional butterfly contains two $(r-1)$-dimensional butterflies as subgraphs. (Just remove the level 0 nodes of the r-dimensional butterfly. Alternatively, we could remove the level r nodes, as is done in Figure 3-20, although it takes a little longer to realize that the resulting graph is simply two $(r-1)$-dimensional butterflies.)

Another useful property of the r-dimensional butterfly is that the level 0 node in any row w is linked to the level r node in any row w' by a unique path of length r. The path traverses each level exactly once, using the cross edge from level i to level $i+1$ if and only if w and w' differ in the $(i+1)$st bit. For example, see Figure 3-21. As a simple consequence of this fact, we can see that the N-node butterfly has diameter $O(\log N)$.

Like the hypercube, the butterfly also has a large bisection width. In particular, the bisection width of the N-node butterfly is $\Theta(N/\log N)$. To construct a bisection of this size, simply remove the cross edges from a single level. To show that $\Omega(N/\log N)$ is a lower bound on the bisection width of the network, we can apply the same technique used to prove Theorem 1.21 in Section 1.9. (For example, see Problem 3.89.)

The Wrapped Butterfly

For computational purposes, the first and last levels of the butterfly are sometimes merged into a single level. In particular, node $\langle w, 0\rangle$ is merged into node $\langle w, r\rangle$ for each w. The result is an r-level graph with $r2^r$ nodes, each of degree 4. Two nodes $\langle w, i\rangle$ and $\langle w', i'\rangle$ are linked by an edge if and only if $i' \equiv i+1 \bmod r$ and either $w = w'$ or w and w' differ in the i'th bit. Such edges are called *level i'* edges. To distinguish between this structure and the unmerged butterfly of Figure 3-19, we will refer to the former as a *wrapped butterfly* and the latter as an *ordinary butterfly*. For example, a three-dimensional wrapped butterfly is illustrated in Figure 3-22.

At first glance, it might seem that the wraparound edges could make the wrapped butterfly more powerful than the ordinary butterfly from a computational point of view. This turns out not to be the case, however. In fact, the relationship between the butterfly and wrapped butterfly is

3.4 Packet-Routing Algorithms

One of the most important components of any large-scale general-purpose parallel computer is its packet-routing algorithm. This is because most large-scale general-purpose parallel machines spend a large portion of their resources making sure that the right data gets to the right place within a reasonable amount of time.

We already studied packet-routing algorithms for arrays in Chapter 1 and meshes of trees in Chapter 2. Although the algorithms described in these chapters are optimal for arrays and meshes of trees, they are not especially efficient in a general setting. For example, the routing algorithms for arrays use few processors but are relatively slow. The routing algorithms for meshes of trees, on the other hand, are fast but use an excessive number of processors.

We have also studied the packet-routing problem for hypercubic networks in Sections 3.2 and 3.3. In particular, we showed how to solve any fixed N-packet permutation routing problem in $O(\log N)$ steps on an N-processor butterfly or shuffle-exchange graph in Theorems 3.12 and 3.16. The solution to a routing problem found by this approach is fast and efficient, but suffers from the limitation that there is no $O(\log N)$-step algorithm known for finding the routing paths on-line. In other words, we proved in Theorems 3.12 and 3.16 that there is a fast and efficient solution to any permutation routing problem on a hypercubic network, but we don't know how to find the solution quickly in parallel. For some applications, this constraint doesn't matter, since we can afford to precompute the solution (off-line) and then store the routing information in the network. For many applications, however, the limitation is crucial, since we may not know the routing problem ahead of time. For such applications, we will need to develop *on-line* routing algorithms (i.e., algorithms for which the local routing decisions are made without precomputation and without knowledge of the global routing problem).

In this section, we describe several on-line algorithms for routing on hypercubic networks. For the most part, the algorithms will perform quickly (taking $\Theta(\log N)$ steps) and efficiently (using N processors to route N packets), although all of the algorithms described in this section can perform very badly in the worst case. In Section 3.5, we will describe algorithms for sorting that can be used to construct routing algorithms that are guaranteed to always perform well, but the sorting-based algorithms are quite

complicated and are often not as useful in practice.

We begin our discussion of packet-routing algorithms with some definitions and a description of some of the most common routing models in Subsection 3.4.1. We then define the greedy routing algorithm, and analyze its worst-case performance in Subsection 3.4.2. Unfortunately, we will find that the worst-case performance of the greedy algorithm is very poor and that several important routing problems exhibit worst-case performance.

On the other hand, there are also large classes of important problems for which the greedy algorithm performs very well. For example, we will show in Subsection 3.4.3 that the greedy algorithm performs optimally for packing, spreading, and monotone routing problems. These special classes of routing problems arise in many applications, and we will use them frequently throughout the remainder of Chapter 3. For example, we show in Subsection 3.4.3 how to decompose an arbitrary routing problem into a sorting problem and a monotone routing problem. Since any monotone routing problem can be solved in $O(\log N)$ steps using the greedy algorithm, this gives us an automatic way to convert any sorting algorithm into a packet-routing algorithm. Even though sorting N items quickly on a hypercubic network is a challenging task, this means that all of the sorting algorithms that are described in Section 3.5 can be converted into packet-routing algorithms with very little additional effort.

Greedy algorithms also perform well for average-case routing problems. In fact, we will show in Subsection 3.4.4 that almost all N-packet-routing problems can be solved in $O(\log N)$ steps by using the greedy algorithm on an N-processor hypercubic network. Hence, we will find that the greedy algorithm is optimal (up to constant factors) for random routing problems in a hypercubic network. This fact is quite important, since so many parallel machines use variations of the greedy algorithm to solve routing problems. In addition, we can use this fact to design a randomized algorithm for solving worst-case problems. In particular, we will show in Subsection 3.4.5 how to use randomness to convert any worst-case one-to-one routing problem into two average-case problems, thereby solving *any* one-to-one routing problem in $O(\log N)$ steps with high probability.

One problem with the naive greedy algorithm is that it allows packets to pile up at certain nodes in the network, resulting in queues which (for most routing problems) can grow as large as $\Theta(\log N)$ in size. In Subsection 3.4.6, we show how to modify the naive greedy algorithm so that all the queues stay small, and so that the overall performance remains good.

We also show how to generalize the result to apply to a much larger class of networks (including arrays).

Another problem with the naive greedy algorithm is that it doesn't always work well for some many-to-one routing problems, even if randomization is used. In Subsection 3.4.7, we show how to modify the naive greedy algorithm to handle many-to-one routing problems. We also describe an effective strategy for combining packets that are headed for the same destination, if that is desired.

In Subsection 3.4.8, we describe a variation of the greedy routing algorithm known as the information dispersal algorithm. The information dispersal approach to routing makes use of coding theory to partition a packet into many subpackets, only some fraction of which need to be successfully routed in order for the contents of the packet to be reconstructed at the destination. As a consequence, some packet components that get stuck in a congested area or that encounter a faulty component can be discarded without harm. As we will see in Section 3.6, information dispersal is also a useful tool when it comes to organizing data in a distributed memory.

We conclude our study of packet routing with a discussion of circuit-switching and bit-serial routing algorithms in Subsection 3.4.9. The algorithms described in this subsection differ from those discussed in Subsections 3.4.4–3.4.8 in that each packet needs to have a dedicated, uncongested path through the network from its source to its destination in order for the message to be transmitted. Even in this more restricted routing model, however, we find that the greedy algorithm performs fairly well for most (i.e., random) routing problems.

3.4.1 Definitions and Routing Models

As was mentioned in Section 1.7, there are many different types of routing models. For the most part, we will focus our attention on the *store-and-forward* model (also known as the *packet-switching* model) of packet routing in Section 3.4. In the store-and-forward model, each packet is maintained as an entity that is passed from node to node as it moves through the network and a single packet can cross each edge during each step of the routing. Depending on the algorithm, we may or may not allow packets to pile up in queues located at each node. When queues are allowed, we will generally make efforts to keep them from getting very large.

In Subsection 3.4.9, we consider the *circuit-switching* (or *path-lockdown*)

model of routing. In the circuit-switching model of routing, the entire path from the source of a packet to its destination must be dedicated to the packet in order for the packet's data to be transmitted.

For the most part, we will focus our attention on *static* routing problems (i.e., those for which the packets to be routed are present in the network when the routing commences) in Section 3.4. Many of the results that we obtain can also be applied to *dynamic* routing problems (in which packets arrive at the network at arbitrary times and the routing proceeds in a continuous fashion), although we will only specifically discuss the case of dynamic routing problems in Subsection 3.4.4.

There are many different types of static routing problems. Generally, we will assume that each processor starts with at most one packet, and, for the most part, we will focus our attention on the simplest case of one-to-one routing problems. A routing problem is said to be *one-to-one* if at most one packet is destined for any processor and if each packet has precisely one destination. We will also consider many-to-one and one-to-many routing problems. A routing problem is said to be *many-to-one* if more than one packet can have the same destination. It is said to be *one-to-many* if a single packet can have multiple destinations (i.e., if copies of one packet need to be sent to more than one destination).

When many packets are headed for the same destination, the usual problems with congestion in the network can become even more severe. For example, if at most one packet can be delivered to its destination during each step, then most of the packets that are headed for a common destination will experience significant delays due to (if for no other reason) the bottleneck at the destination. Such bottlenecks are often referred to as *hot spots*. Needless to say, hot spots can be a serious problem since they can also cause packets which are headed for other destinations to become delayed.

We will describe many methods for overcoming or minimizing the effects of hot spots and bottlenecks resulting from multiple packets having the same destination. One approach to dealing with such problems is to allow packets that are headed for the same destination to be combined. When *combining* is allowed, we can merge two packets P_1 and P_2 into a single (possibly larger) packet provided that P_1 and P_2 are headed for the same destination and that P_1 and P_2 are contained in the same node at the same time. Packet-routing algorithms that make use of combining will be described in Subsections 3.4.3 and 3.4.7.

Throughout Section 3.4, we will insist that our algorithms be *on-line*. This means that each processor (or switch) must decide what to do with the packets that pass through it based only on its local control and the information carried with the packets. In particular, we will not allow a global controller to precompute routing paths as was done in the proofs of Theorems 3.12 and 3.16. As a consequence, our algorithms will be able to handle any packet-routing problem immediately using only local control.

As was mentioned previously in the text, the development of an efficient routing algorithm for a network enables that network to efficiently emulate any other network. More generally, it will enable us to get the right data to the right place at the right time. As a consequence of the on-line feature of the routing algorithm, we will also be able to emulate abstract parallel machines such as a parallel random access machine (PRAM). Methods for simulating PRAMs on hypercubic networks will be studied extensively in Section 3.6.

3.4.2 Greedy Routing Algorithms and Worst-Case Problems

We begin our study of packet routing algorithms on hypercubic networks by considering the problem of routing N packets from level 0 to level $\log N$ in a $\log N$-dimensional butterfly. In particular, we assume that each node $\langle u, 0\rangle$ on level 0 of the butterfly contains a packet that is destined for node $\langle \pi(u), \log N\rangle$ on level $\log N$ where $\pi : [1, N] \rightarrow [1, N]$ is a permutation. For example, we have illustrated an 8-packet routing problem in Figure 3-48. In this example, we have selected π to be the *bit-reversal permutation* (i.e., $\pi(u_1 \cdots u_{\log N}) = u_{\log N} \cdots u_1$, where $u_1 \cdots u_{\log N}$ denotes the binary representation of u).

At first glance, the routing problem shown in Figure 3-48 does not seem particularly difficult. Indeed, any of the packets in the problem can be easily routed to its destination simply by sending the packet along the unique path of length $\log N$ through the butterfly to its destination. For example, we have illustrated this path for the packet destined for node $\langle 001, 3\rangle$ in Figure 3-48.

In general, the unique path of length $\log N$ from a level 0 node $\langle u, 0\rangle$ to a level $\log N$ node $\langle v, \log N\rangle$ in the butterfly is known as the *greedy path* from $\langle u, 0\rangle$ to $\langle v, \log N\rangle$. In the *greedy routing algorithm*, each packet is constrained to follow its greedy path. When there is only one packet to route, it is easy to see that the greedy algorithm performs very well. Trouble can arise when many packets have to be routed in parallel, however.

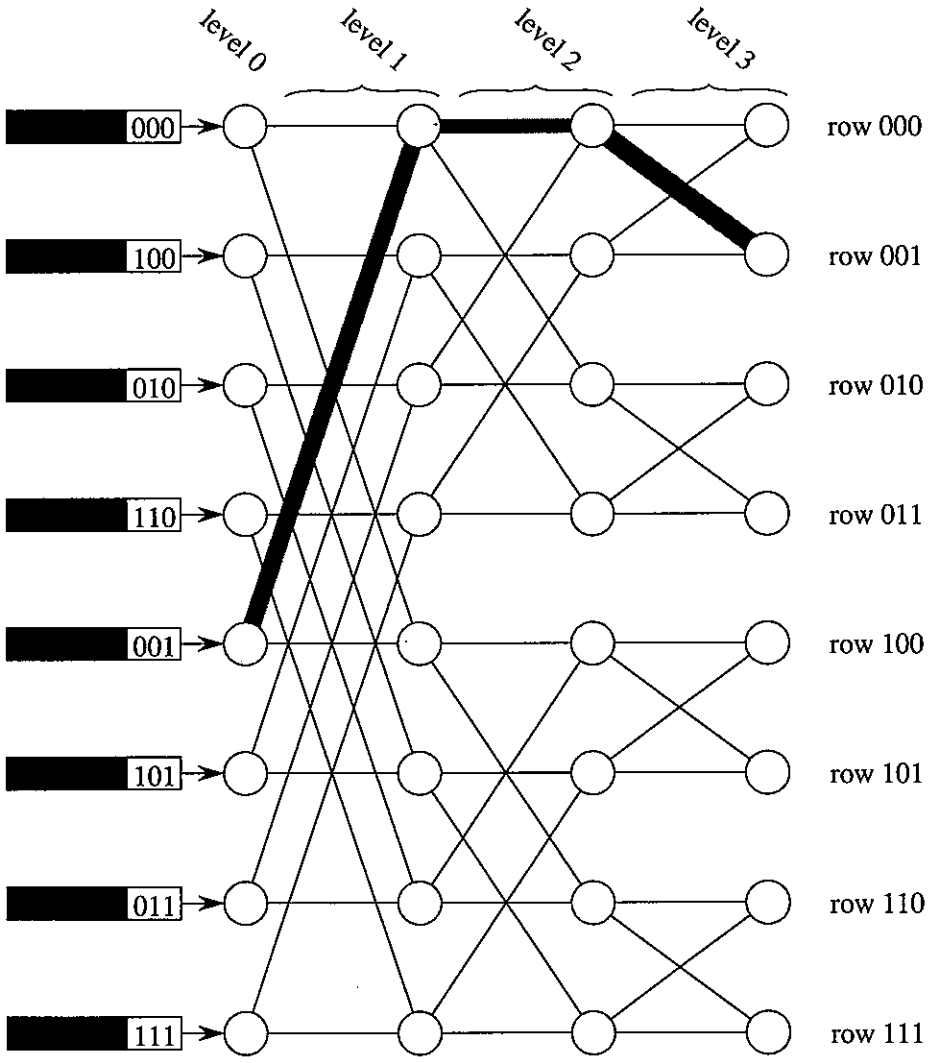

Figure 3-48 *An 8-packet routing problem on the three-dimensional butterfly. In this problem, the packet starting at node* $\langle u_1u_2u_3, 0\rangle$ *wants to go to node* $\langle u_3u_2u_1, 3\rangle$ *for each* $u_1u_2u_3$*. The greedy path for the packet starting at node* $\langle 100, 0\rangle$ *is shown in boldface.*

The problem is that many greedy paths might pass through a single node or edge. For example, the packets starting at nodes $\langle 000, 0\rangle$ and $\langle 100, 0\rangle$ in Figure 3-48 both must pass through edge $(\langle 000, 1\rangle, \langle 000, 2\rangle)$ on the way to their destinations. Since only one of these packets can use the edge at a time, one of them must be delayed before crossing the edge.

It is not difficult to check that the congestion problem arising in the example illustrated in Figure 3-48 is not overly serious. At most two packets will ever contend for a middle-level edge, and every packet can reach its destination in a total of 4 steps using the greedy algorithm.

When N is larger, however, the problem can be much more serious. In fact, a total of

$$2^{\frac{\log N-1}{2}} = \sqrt{N/2}$$

greedy paths will use the edge $\left(\langle 0\cdots 0, \frac{\log N-1}{2}\rangle, \langle 0\cdots 0, \frac{\log N+1}{2}\rangle\right)$ in a $\log N$-dimensional butterfly when the greedy algorithm is used to route N packets according to the bit-reversal permutation. (Here, we have assumed for simplicity that $\log N$ is odd. A similar result holds when $\log N$ is even. For example, see Problem 3.179.) The reason is that the packet which starts at node $\langle u_1 \cdots u_{\frac{\log N-1}{2}} 00\cdots 0, 0\rangle$ must travel to node $\langle 0\cdots 00 u_{\frac{\log N-1}{2}} \cdots u_1, \log N\rangle$ along the path

$$\begin{aligned}
\langle u_1 \cdots u_{\frac{\log N-1}{2}} 00\cdots 0, 0\rangle &\rightarrow \left\langle 0u_2 \cdots u_{\frac{\log N-1}{2}} 00\cdots 0, 1\right\rangle \\
&\rightarrow \cdots \\
&\rightarrow \left\langle 0\cdots 0 u_{\frac{\log N-1}{2}} 00\cdots 0, \frac{\log N-3}{2}\right\rangle \\
&\rightarrow \left\langle 0\cdots 000\cdots 0, \frac{\log N-1}{2}\right\rangle \\
&\rightarrow \left\langle 0\cdots 000\cdots 0, \frac{\log N+1}{2}\right\rangle \\
&\rightarrow \left\langle 0\cdots 00 u_{\frac{\log N-1}{2}} 0\cdots 0, \frac{\log N+3}{2}\right\rangle \\
&\rightarrow \cdots \\
&\rightarrow \langle 0\cdots 00 u_{\frac{\log N-1}{2}} \cdots u_1, \log N\rangle.
\end{aligned}$$

Note that this path contains the edge

$$e = \left\langle 0\cdots 000\cdots 0, \frac{\log N-1}{2}\right\rangle \rightarrow \left\langle 0\cdots 000\cdots 0, \frac{\log N+1}{2}\right\rangle,$$

since the middle bit of both $u_1 \cdots u_{\frac{\log N-1}{2}} 00 \cdots 0$ and $0 \cdots 00 u_{\frac{\log N-1}{2}} \cdots u_1$ is 0. There are $2^{\frac{\log N-1}{2}} = \sqrt{N/2}$ possible values of $u_1 \cdots u_{\frac{\log N-1}{2}}$, and thus $\sqrt{N/2}$ packets must traverse e when the greedy algorithm is used to route the packets. This means that at least one of the packets will be delayed by $\sqrt{N/2}-1$ steps, and that the greedy algorithm will take at least $\sqrt{N/2}+\log N-1$ steps to route all of the packets to their destinations. In fact, the greedy algorithm takes precisely $\sqrt{N/2}+\log N-1$ steps to route the bit-reversal permutation when $\log N$ is odd. (See Problem 3.181.)

Unfortunately, the bit-reversal permutation is not the only permutation that requires $\Theta(\sqrt{N})$ steps to route using the greedy algorithm. Indeed, many natural permutations exhibit poor performance with the greedy algorithm. For example, the commonly used *transpose permutation*

$$\pi\left(u_1 \cdots u_{\frac{\log N}{2}} u_{\frac{\log N}{2}+1} \cdots u_{\log N}\right) = u_{\frac{\log N}{2}+1} \cdots u_{\log N} u_1 \cdots u_{\frac{\log N}{2}}$$

also requires $\Theta(\sqrt{N})$ steps using the greedy algorithm. (See Problem 3.182.)

In fact, the bit-reversal permutation and the transpose permutation are (up to constant factors) worst-case permutations for greedy routing on the butterfly. This is because every one-to-one routing problem can be solved in $O(\sqrt{N})$ steps on a $\log N$-dimensional butterfly. We will prove this fact in the following theorem.

THEOREM 3.22 *Given any routing problem on a $\log N$-dimensional butterfly for which at most one packet starts at each level 0 node and at most one packet is destined for each level $\log N$ node, the greedy algorithm will route all the packets to their destinations in $O(\sqrt{N})$ steps.*

Proof. For simplicity, we will assume that $\log N$ is odd. The case when $\log N$ is even is handled in a similar fashion.

Let e be any edge in level i of the $\log N$-dimensional butterfly ($0 < i \leq \log N$), and define n_i to be the number of greedy paths that traverse e. We first observe that $n_i \leq 2^{i-1}$ for every i. This is because there are at most 2^{i-1} nodes at level 0 which can reach e using a path through levels $1, 2, \ldots, i-1$. For example, only the packets starting at nodes $\langle 000, 0\rangle$ and $\langle 100, 0\rangle$ can use the edge $(\langle 000, 1\rangle, \langle 000, 2\rangle)$ in a three-dimensional butterfly, no matter where each packet is destined.

Similarly, $n_i \leq 2^{\log N - i}$ for every i. This is because there are at most $2^{\log N - i}$ nodes at level $\log N$ which can be reached from e using a path through levels $i+1, i+2, \ldots, \log N$. For example, only the packets

ending at nodes $\langle 000, 3\rangle$ and $\langle 001, 3\rangle$ can use the edge $(\langle 000, 1\rangle, \langle 000, 2\rangle)$, no matter where the packets originate.

Since any packet crossing e can only be delayed by the other $n_i - 1$ packets that want to cross the edge, the total delay encountered by any packet as it traverses levels 1, 2, ..., $\log N$ can be at most

$$\begin{aligned}\sum_{i=1}^{\log N}(n_i - 1) &\leq \sum_{i=1}^{\frac{\log N+1}{2}} 2^{i-1} + \sum_{i=\frac{\log N+3}{2}}^{\log N} 2^{\log N - i} - \log N \\ &= 2^{\frac{\log N+1}{2}} + 2^{\frac{\log N-1}{2}} - \log N - 2 \\ &= \frac{3\sqrt{N}}{\sqrt{2}} - \log N - 2.\end{aligned}$$

Hence, the total time to complete any one-to-one routing problem is at most $O(\sqrt{N})$, as claimed. ■

The preceding analysis does not specifically deal with the problem of packets piling up in queues. Indeed, the queues at nodes in the middle levels of the butterfly might grow to be as large as $\Theta(\sqrt{N})$ if we do not limit their size. (See Problem 3.183.) We can restrict the growth of the queues by not allowing any packet to move forward across an edge if there are too many packets (say q) in the queue at the other end of the edge. The problem with limiting the queue sizes, however, is that packets can be delayed even further. In fact, if we restrict queue sizes to be $O(1)$ in the butterfly, then the greedy algorithm can be forced to use $\Theta(N)$ steps to route some permutations. (See Problem 3.185.)

For small values of N, the worst-case performance of the greedy routing algorithm is not so bad. This is because $\sqrt{N}$ and $\log N$ are not all that different when N is small (say, less than 100). For large N, the worst-case performance of the greedy algorithm becomes more of a problem, however, particularly since so many of the natural permutations (such as bit-reversal and transpose) exhibit worst-case performance for the greedy algorithm.

Of course, we know from Theorem 3.11 that every permutation can be routed in $2 \log N$ steps on the butterfly with queues of size 1, provided that we are allowed to use off-line precomputation and that we can make two passes through the butterfly. Hence, it makes sense to use a special set of precomputed routing paths (instead of the greedy algorithm) whenever we encounter one of the known worst-case permutations. As a consequence, we really don't have to worry about the worst-case performance of the

bit-reversal and transpose permutations since we don't have to route them using the greedy algorithm.

Unfortunately, there are many bad permutations for the greedy algorithm, and it is not feasible to precompute special routing paths for all of them using Theorem 3.11. In an attempt to overcome this problem, special-purpose routing algorithms have been developed that work well for large classes of permutations that are not handled efficiently by the greedy algorithm. For example, see Problems 3.188–3.191. While such special-purpose algorithms can efficiently handle several natural permutations such as the transpose and bit-reversal permutations, they are still not sufficient to efficiently handle all of the permutations for which the greedy algorithm performs poorly. Indeed, we will have to cover a lot more material before we are ready to describe routing algorithms that perform well for all permutations.

In the preceding discussion, we concentrated on the problem of routing packets from one end of the butterfly to the other. (Such routing problems are sometimes called *end-to-end* routing problems.) In practice, the butterfly is often used to route packets in exactly this fashion. It is also sometimes used to route packets between all of the nodes of the network. When each node of the $\log N$-dimensional wrapped butterfly starts and finishes with one packet, and each of the $\log N$ packets is greedily routed (in the same direction) first to the correct row and then to the correct node, then each of the $N \log N$ packets will be routed to the correct destination within $\Theta(\sqrt{N \log N})$ steps in the worst case. (See Problems 3.192–3.193.)

For simplicity, we will continue to focus our study of hypercubic routing algorithms on the problem of routing packets from one end of the butterfly to the other. The results that we obtain for this particular problem can usually be extended to hold for most other hypercubic routing problems of interest. For example, results for end-to-end routing on a $\log N$-dimensional butterfly can be immediately extended to hold for arbitrary routing problems (i.e., routing problems where every node may start with a packet) on an N-node hypercube, since there is such a close relationship between the edges of the $\log N$-dimensional butterfly and the edges of the N-node hypercube. Moreover, if the packets move through the hypercube in a normal fashion, then the results can also be extended to hold for arbitrary routing problems on any N-node hypercubic network.

Results for end-to-end routing on a butterfly can also be extended to hold for arbitrary butterfly routing problems by first routing each packet

to the level 0 node in its row, and then routing the packet to the level $\log N$ node in its destination row, before routing the packet to its correct destination. The hard part of the routing is the end-to-end routing, since routing packets within their rows can usually be accomplished in $O(\log N)$ additional steps. In other words, solving an arbitrary routing problem on a $\log N$-dimensional butterfly is often not much harder than solving $\log N$ end-to-end routing problems on the butterfly. In addition, by using pipelining, we will often find that solving $\log N$ end-to-end routing problems on a butterfly is not much harder than solving a single end-to-end routing problem on the butterfly.

Despite the fact that the greedy routing algorithm performs poorly in the worst case, the greedy algorithm is very useful. In fact, we will show that the greedy algorithm often performs exceptionally well. For example, for many useful classes of permutations, the greedy algorithm runs in $\log N$ steps, which is optimal. And, for most permutations, the greedy algorithm runs in $\log N + o(\log N)$ steps. (We prove these important facts in Subsections 3.4.3 and 3.4.4, respectively.) As a consequence, the greedy algorithm is widely used in practice.

In what follows, we digress briefly from our study of greedy routing algorithms on hypercubic networks in order to prove a general lower bound on the time required for any greedylike algorithm to route a worst-case permutation on an arbitrary network.

A General Lower Bound for Oblivious Routing ⋆

A routing algorithm is said to be *oblivious* if the path travelled by each packet depends only on the origin and destination of the packet (and not on the origins and destinations of the other packets nor on congestion encountered during the routing). For example, the greedy routing algorithm on the butterfly is oblivious, since each packet follows the greedy path to its destination.

In what follows, we will show that for any N-node, degree-d network and any oblivious routing algorithm, there is an N-packet one-to-one routing problem for which the algorithm will take $\Omega(\sqrt{N}/d)$ steps to complete the routing. This means that the worst-case running time of any oblivious or greedy routing algorithm on the butterfly will be $\Omega(\sqrt{N})$, a far cry from the desired bound of $O(\log N)$. In fact, this means that the worst-case running time of the greedy algorithm on any N-node bounded-degree network is $\Omega(\sqrt{N})$. For the hypercube, the worst-case bound will be $\Omega(\sqrt{N}/\log N)$.

Hence, we will have to resort to nonoblivious algorithms if we want to be able to route on-line every one-to-one routing problem in $O(\log N)$ steps.

THEOREM 3.23 *Let $G = (V, E)$ be any N-node degree-d network, and consider any oblivious algorithm $\mathcal{A}$ for routing packets in G. Then there is a one-to-one packet-routing problem for which $\mathcal{A}$ will take at least $\sqrt{N}/2d$ steps to complete.*

Proof. Since $\mathcal{A}$ is oblivious, the path followed by a packet starting at a node u and ending at a node v (call the path $P_{u,v}$) depends only on u and v, and not on any of the other packet origin/destination pairs. Hence, $\mathcal{A}$ can be specified by the N^2 paths $P_{u,v}$ that are used to route packets (along with some timing information that is not of concern in the present argument).

Our objective is to find a large subset of nodes u_1, v_1, u_2, v_2, ..., u_k, v_k for which $u_i \neq u_j$ for $i \neq j$, $v_i \neq v_j$ for $i \neq j$, and for which P_{u_1,v_1}, P_{u_2,v_2}, ..., and P_{u_k,v_k} all contain the same edge e. Then we can prove that $\mathcal{A}$ takes at least $\frac{k}{2}$ steps to complete any routing for which there is a packet starting at u_i and destined for v_i for $1 \leq i \leq k$. The reason is that all of these packets must eventually pass through edge e, but only 2 packets can do so at any step (one in each direction). In what follows, we will show how to construct such a set of paths with $k = \sqrt{N}/d$, thereby establishing the theorem.

For any node v, consider the $N-1$ paths $P_{u,v}$ that end at v. For any integer k, let $S_k(v)$ denote the set of edges in G which have k or more of the paths ending at v passing through them. In addition, we define $S_k^*(v)$ to be the set of nodes that are incident to an edge in $S_k(v)$.

Note that for all k and v, $|S_k^*(v)| \leq 2|S_k(v)|$, since there are two nodes incident to each edge. In addition, if $k \leq \frac{N-1}{d}$, then $v \in S_k^*(v)$. This is because there are $N-1$ paths coming into v, and thus at least $\frac{N-1}{d}$ of the paths must include the same edge incident to v.

We next show that for $k \leq \frac{N-1}{d}$,

$$|V - S_k^*(v)| \leq (k-1)(d-1)|S_k^*(v)|.$$

This is because every node u not in $S_k^*(v)$ is at the start of a path $P_{u,v}$ that enters $S_k^*(v)$ (since $v \in S_k^*(v)$), and there are a limited number of paths that can enter $S_k^*(v)$ from outside. In particular, for any node $u \notin S_k^*(v)$, there must be consecutive nodes w and w' in $P_{u,v}$ such that $w \notin S_k^*(v)$ and $w' \in S_k^*(v)$. Since $w \notin S_k^*(v)$, we know that $(w, w') \notin$

$S_k(v)$, and thus that there are at most $k-1$ nodes u for which $P_{u,v}$ enters $S_k^*(v)$ on edge (w, w'). In addition, for each of the $|S_k^*(v)|$ choices for w', there are at most $d-1$ choices for w such that w is adjacent to w' and $(w, w') \notin S_k(v)$. (This is because w' has at most d neighbors, and it is linked to at least one of its neighbors by an edge in $S_k(v)$.) Hence, there are at most $(k-1)(d-1)|S_k^*(v)|$ nodes u for which $P_{u,v}$ enters $S_k^*(v)$ from the outside. This means that

$$|V - S_k^*(v)| \leq (d-1)(k-1)|S_k^*(v)|,$$

as claimed.

As a consequence of the preceding analysis, we can also conclude that

$$\begin{aligned} N &= |V - S_k^*(v)| + |S_k^*(v)| \\ &\leq (k-1)(d-1)|S_k^*(v)| + |S_k^*(v)| \\ &\leq 2[1 + (k-1)(d-1)]|S_k(v)| \\ &\leq 2kd|S_k(v)|, \end{aligned}$$

and thus that

$$|S_k(v)| \geq \frac{N}{2kd}$$

for any $k \leq \frac{N-1}{d}$. Setting $k = \sqrt{N}/d$, and summing over all N nodes v, we find that

$$\sum^{v \in V} |S_k(v)| \geq \frac{N^2}{2kd} = \frac{N^{3/2}}{2}.$$

Since there are at most $Nd/2$ edges in G, this means that there is some edge e for which $e \in S_k(v)$ for at least

$$\frac{N^{3/2}/2}{Nd/2} = \frac{\sqrt{N}}{d} = k$$

different values of v.

Select e and $v_1, v_2, \ldots, v_k$ such that $e \in S_k(v_i)$ for $1 \leq i \leq k$. Let u_1 be one of the k nodes for which P_{u_1,v_1} passes through e. Let $u_2 \neq u_1$ be one of the (at least) $k-1$ other nodes for which P_{u_2,v_2} passes through e. In general, let $u_i \notin \{u_1, u_2, \ldots, u_{i-1}\}$ be one of the (at least) $k-(i-1)$ previously unused nodes for which P_{u_i,v_i} passes through e. For $i \leq k$, we can always find such a u_i since there are at least k choices of u_i for

which P_{u_i,v_i} passes through e, at most $i-1 \leq k-1$ of which have been used previously. Hence, we can construct a collection of nodes u_1, v_1, u_2, v_2, ..., u_k, v_k with $k = \sqrt{N}/d$ for which $u_i \neq u_j$ for $i \neq j$, $v_i \neq v_j$ for $i \neq j$, and for which P_{u_1,v_1}, P_{u_2,v_2}, ..., and P_{u_k,v_k} all pass-through some edge e, as claimed. ■

It is not difficult to extend the proof of Theorem 3.23 to show that there are many bad routing problems for any oblivious routing algorithm. In fact, there are at least $(\sqrt{N}/d)!$ problems which will require $\sqrt{N}/2d$ steps to route on an N-node network with maximum degree d. (See Problem 3.196.) The proof can also be easily extended to hold for N-node networks with fewer than N inputs and outputs. In particular, given any oblivious algorithm for routing on an M-input/output, N-node, degree-d network, there is an M-packet one-to-one routing problem for which the algorithm will take $\Omega\left(\frac{M}{d\sqrt{N}}\right)$ steps to route all the packets. (See Problem 3.197.)

Although we won't discuss randomized routing algorithms or bit-serial routing for some time, it is worth pointing out here that Theorem 3.23 can also be extended to lower bound the performance of any randomized oblivious algorithm. In particular, it can be shown that any randomized oblivious algorithm (where the path for a packet is chosen at random from a distribution that depends only on the origin and destination of the packet) must use

$$\Omega\left(\frac{\left(L+\frac{\log N}{\log d}\right)\log N}{\log d + \log\log N}\right)$$

bit steps with high probability in order to route N packets of length L in an N-node degree-d network. For example, if we are routing N packets of length $O(\log N)$ on an N-node hypercube, then a randomized oblivious algorithm will use $\Omega(\log^2 N/\log\log N)$ bit steps with high probability. In Subsection 3.5.4, we will describe randomized nonoblivious algorithms for routing that can solve such problems in $O(\log N)$ bit steps with high probability.

Now that our digression is complete, we will return to our study of the greedy routing algorithm on the butterfly.

3.4.3 Packing, Spreading, and Monotone Routing Problems

Although the greedy routing algorithm performs poorly in the worst case, it performs exceptionally well in the best case. In fact, there are many