implementation that is nearly as fast as a priority encoder. Switch scheduling can be done by one such grant arbiter for every output port (to arbitrate requests) and one accept arbiter for every input port (to arbitrate between grants). Priorities and multicast are retrofitted into the basic structure by adding a filter on the inputs before it reaches the arbiter; for example, a priority filter zeroes out all requests except those at the highest-priority level.

Although in principle the unicast schedulers can be designed using a separate chip per port, the state is sufficiently small to be handled by a single scheduler chip with control wires coming in from and going to each of the ports. Also, the multicast algorithm requires a shared multicast pointer per priority level, which also implies a centralized scheduler. Centralization, however, implies a delay, or latency, to send requests and decisions from the port line cards to and from the central scheduler.

To tolerate this latency, the scheduler [**GM99a**] works on a pipeline of m cells (8 in Tiny Tera) from each VOQ and n cells (5 in Tiny Tera) from each multicast queue. This in turn implies that each line card in the Tiny Tera must communicate 3 bits per unicast VOQ denoting the size of the VOQ, up to a maximum of 8. With 32 outputs and four priority levels, each input port has to send 384 bits of unicast information. Each line card also communicates the fanout (32 bits per fanout) for each of five multicast packets in each of four priority levels, leading to 640 bits. The $32 * 1024$ total bits of input information is stored in on-chip SRAM. However, for speed the information about the heads of each queue (smaller state, for example, only 1 bit per unicast VOQ) is stored in faster but less dense flip-flops.

Now consider handling multiple iterations. Note that the request phase occurs only on the first iteration and needs to be modified only on each iteration by masking off matched inputs. Thus K iterations appear to take at least $2K$ time steps, because the grant and accept steps of each iteration take one time step. At first glance, the architecture appears to specify that the *grant* phase of iteration $k + 1$ be started after the accept phase of iteration k. This is because one needs to know whether an input port I has been accepted in iteration k so as to avoid doing a grant for such an input in iteration $k + 1$.

What makes partial pipelining possible is a simple observation [**GM99a**]: If input I receives *any* grant in iteration k, then I must accept exactly one and so be unavailable in iteration $k + 1$. Thus the implementation specification can be relaxed (**P3**) to allow the grant phase of iteration $k+1$ to start immediately after the grant phase of iteration k, thus overlapping with the accept phase of iteration k. To do so, we simply use the OR of all the grants to input I (at the end of iteration k) to mask out all of I's requests (in iteration $k + 1$).

This reduces the overall completion time by nearly a factor of 2 time steps for k iterations — from $2k$ to $k + 1$. For example, the Tiny Tera iSLIP implementation [**GM99a**] does three iterations of iSLIP in 51 nsec (roughly OC-192 speeds) using a clock speed of 175 MHz; given that each clock cycle is roughly 5.7 nsec, iSLIP has roughly nine clock cycles to complete. Since each grant and accept step takes two clock cycles, the pipelining is crucial in being able to handle three iterations in 9 clock cycles; the naive iteration technique would have taken at least 12 clock cycles.

1.11 COMPUTING NEAR-OPTIMAL BIPARTITE MATCHINGS VIA LEARNING

Invoke the principle of incremental computation in this section

Recall from Section 1.8 that a heavy matching, such as maximum weighted matching (MWM), is generally a good matching in terms of throughput and delay performances. In most bipartite matching algorithms for switching, such as PIM and iSLIP described earlier, the computation of the crossbar schedule (matching) for each switching cycle is done from scratch, or in other words oblivious of earlier computations. This should sound computationally wasteful (**P1**). For example, a heavy matching computed in the previous cycle typically remains heavy in the current cycle, since in the previous cycle there can be at most N departures, but possibly close to N arrivals, under a heavy load. Intuitively, if we can somehow learn the heavy edges from the matching used in the previous time slot, we may be able to arrive at a new heavy matching using less (additional) computation. Indeed we can do just that with a family of adaptive algorithms that we will describe in the rest of this section.

Besides the bipartite matching used in the previous cycle, there are other things an adaptive algorithm can learn from earlier computations to help compute a good new matching faster. For example, an intelligent adaptive algorithm can develop and maintain "situational awareness" of the weights of all N^2 edges (VOQs) gradually over many past switching cycles, which can help the algorithm make quick yet wise matching decisions, as will be shown in Section 1.11.3. In the following, we describe three such adaptive algorithms. Throughout this section, we assume there is an equal number N of input ports and output ports, and that, an edge is allowed to have weight 0 (i.e., with the corresponding VOQ being empty). Under both assumptions, a maximum matching (defined in Section 1.8) always contains N edges and is necessarily a perfect matching. Hence we use the term perfect matching instead throughout this section to avoid causing any confusion.

1.11.1 Sample-and-Compare: A Stunningly Simple Adaptive Algorithm

We start with the adaptive algorithm proposed in [**Tas98**], which we call sample-and-compare. This algorithm is stunningly simple: At each cycle t, sample a (uniform) random matching denoted as $R(t)$ (**invoke the principle of randomization, to be added as P15b**), compare its weight to that of the matching $S(t-1)$ used in the previous cycle, and use the heavier matching as $S(t)$ (i.e., for the current cycle t). Compared to the MWM algorithm, the sample-and-compare algorithm has a much lower computational complexity of $O(N)$. Amazingly, just like MWM, sample-and-compare can provably attain 100% throughput under all traffic patterns. However, its delay performance is poor under heavy load, as explained by the following queueing dynamics of it.

It can be shown that the current (i.e., at time t) throughput of a switching algorithm roughly corresponds to the weight of $S(t)$ as a fraction of that of MWM with respect to the edge weights at time t. In particular, for any switching algorithm to attain 100% throughput, the weight of $S(t)$ has to eventually be very close to that of WMW. This is actually the case with sample-and-compare. However, since only with a tiny probability can a (uniform) random matching $R(t)$ have a large enough weight to exceed that of $S(t-1)$ (that is already decently large), the rate at which $R(t)$ can "gain enough weight" under sample-and-compare to approach that of MWM is extremely slow. Hence the weight of $S(t)$ can be much smaller than that of MWM for a long time, during which the throughput of the switch is much smaller than 100% and as a consequence the weights of the edges (VOQ lengths) become very large under heavy load. This explains the poor delay performance of sample-and-compare. When (almost) all edge weights become gigantic, however, the weight of a random matching $R(t)$ starts to have a decent probability of beating that of $S(t-1)$, and the weight of $S(t)$ starts to approach that of MWM much more rapidly. This explains why sample-and-compare can attain 100% throughput.

1.11.2 SERENA: An Improved Adaptive Algorithm

SERENA is the best of the several adaptive algorithms proposed in [**SGP02**, **GPS03**]. It has the same algorithmic framework and computational complexity as sample-and-compare, yet delivers much better delay performance. Just like sample-and-compare, SERENA also derives $S(t)$ from $S(t-1)$ and a random matching $R(t)$. However, there are two key differences that make it work much better than sample-and-compare. The first difference is that in SERENA, $R(t)$ is not in general a uniform random matching as in sample-and-compare, but is derived from the set of packet arrivals at time t. This difference results in a statistically heavier $R(t)$ than a uniform random matching. The second difference is that, through a MERGE procedure, SERENA picks heavy edges for $S(t)$ from both $R(t)$ and $S(t-1)$, so that the weight of $S(t)$ can be larger than those of both $R(t)$ and $S(t-1)$. This difference allows $S(t)$ to "gain weight" much faster in SERENA than in sample-and-compare. Next, we describe how $R(t)$ is derived from the packet arrivals and how the MERGE procedure works.

Derive R(t) from the Arrival Graph

In SERENA, the random matching $R(t)$ is derived from the *arrival graph* $A(t)$ defined as follows: An edge (I_i, O_j) belongs to $A(t)$ if and only if there is a packet arrival to the corresponding VOQ during cycle t. Note that $A(t)$ is not necessarily a matching, because more than one input ports could have a packet arrival (*i.e.,* edge) destined for the same output port at time slot t. Hence in this case, each output port prunes all such edges incident upon it except the one with the heaviest weight (with ties broken randomly). The pruned graph, denoted as $A'(t)$, is now a matching.

This matching $A'(t)$, which is typically partial, is then randomly populated into a full matching $R(t)$ by pairing the yet unmatched input ports with the yet unmatched output ports in a round-robin manner. This pairing operation clearly has $O(N)$ computational complexity. Deriving $R(t)$ from $A(t)$ in SERENA is better than generating $R(t)$ from scratch in sample-and-compare in two different ways. First, it allows SERENA to tap into the randomness naturally contained in the packet arrival process, making it cheaper to implement since "manmade" randomness incurs a computational cost, as explained in Section 1.10. Second, as explained in [**SGP02**, **GPS03**], some packet arrivals go to heavily backlogged VOQs, and hence $R(t)$ derived from $A(t)$ often has a larger weight than a uniform random matching.

Merge $R(t)$ with $S(t-1)$

We now describe the MERGE procedure that allows SERENA to pick heavy edges for $S(t)$ from both $R(t)$ and $S(t-1)$. To do so with best clarity, we need to color-code and orient edges of $R(t)$ and $S(t-1)$, like in [**GPS03**], as follows. We color all edges in $R(t)$ red and all edges in $S(t-1)$ green, and hence in the sequel, rename $R(t)$ to S_r ("r" for red) and $S(t-1)$ to S_g ("g" for green) to emphasize the coloring. *We drop the henceforth unnecessary term t here with the implicit understanding that the focus is on the MERGE procedure at time slot t.* We also orient all edges in S_r as pointing from input ports (*i.e.,* I) to output port (*i.e.,* O) and all edges in S_g as pointing from output ports to input ports. We use notations $S_r(I \to O)$ and $S_g(O \to I)$ to emphasize this orientation when necessary in the sequel. Finally, we drop the term t from $S(t)$ and denote the final outcome of the MERGE procedure as S. An example pair of thus oriented full matchings $S_r(I \rightarrow O)$ and $S_g(O \rightarrow I)$, over an 8×8 crossbar, are shown in Figure 1.12.

We now describe how the two color-coded oriented full matchings $S_r(I \rightarrow O)$ and $S_g(O \rightarrow I)$ are merged to produce the final full matching S. The MERGE procedure consists of two steps. The first step is to simply union the two full matchings, viewed as two subgraphs of the complete bipartite graph $G(I \bigcup O)$, into one that we call the *union graph* and denote as $S_r(I \rightarrow O) \bigcup S_g(O \rightarrow I)$ (or $S_r \bigcup S_g$ in short). In other words, the union graph $S_r(I \rightarrow O) \bigcup S_g(O \rightarrow I)$ contains the directed edges in both $S_r(I \rightarrow O)$ and $S_g(O \rightarrow I)$.

It is a mathematical fact that any such union graph can be decomposed into disjoint directed cycles [**GPS03**]. Furthermore, each directed cycle, starting from an input port I_i and going back to itself, is an alternating path between a red edge in S_r and a green edge in S_g, and hence contains equal numbers of red edges and green edges. In other words, this cycle consists of a red sub-matching of S_r and a green sub-matching of S_g. Then in the second step, for each directed cycle, the MERGE procedure compares the weight of the red sub-matching (*i.e.,* the total weight of the red edges in the cycle), with that of the green sub-matching, and includes the heavier sub-matching in the final merged matching S.

To illustrate the MERGE procedure by an example, Figure 1.13 shows the union graph of the two full matchings shown in Figure 1.12(a) and Figure 1.12(b) respectively. The union graph contains three disjoint directed cycles that consist of 2, 6, and 8 edges respectively. In each cycle, we pick the heavier sub-matching and the resulting merged matching is shown in Figure 1.12(c). For example, in the third cycle, the sub-matching $\{(I_3, O_1), (I_5, O_4), (I_4, O_7), (I_1, O_6)\}$, which has total weight $8 + 5 + 7 + 5 = 25$, is heavier than the other sub-matching, which has total weight $7 + 3 + 1 + 3 = 14$. Hence it is chosen to be a part of the final matching. The standard centralized algorithm for implementing the MERGE procedure is to linearly traverse every cycle once, by following the directed edges in the cycle, to obtain the weights of the green and the red sub-matchings that comprise the cycle [**GPS03**]. Clearly, this algorithm has a computational complexity of $O(N)$.

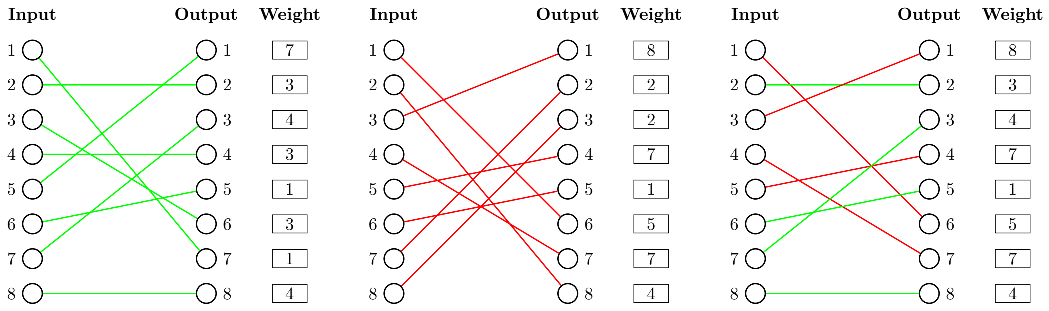

FIGURE 1.12 MERGE procedure.

1.11.3 The Queue-Proportional Sampling Strategy

In SERENA, the way a (typically partial) starter matching $A'(t)$ is derived from the arrival graph $A(t)$ can be viewed as a *proposing process*: Each input port i "proposes" to an output port j, if and only if the edge (i, j) belongs to $A(t)$, by sending the output port j a message containing the length

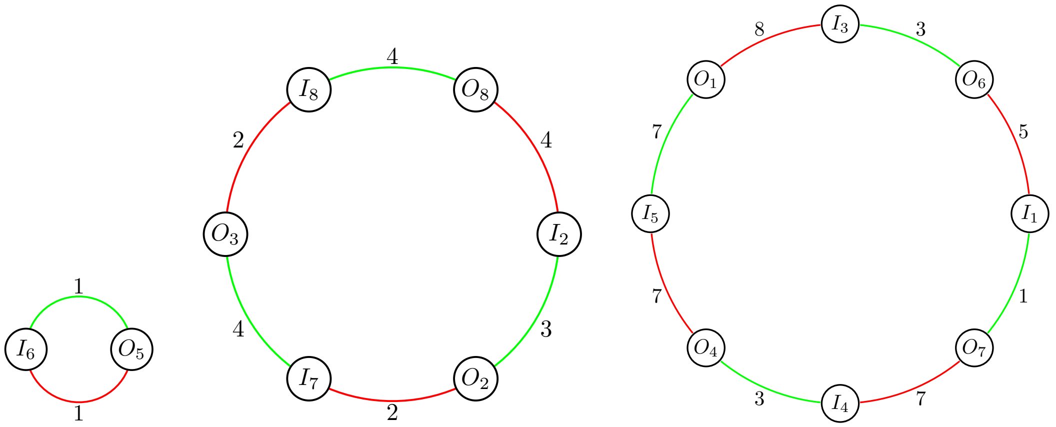

FIGURE 1.13 Cycles.

of the corresponding VOQ; an output port, upon receiving proposals from one or more input ports, accepts the one whose corresponding VOQ is the longest. As explained earlier, this starter matching $A'(t)$, which is typically partial, is then populated into a full matching $R(t)$, and finally refined into the final matching $S(t)$ via merging with $S(t-1)$.

This $A(t)$-generated proposing strategy is quite sensible because, at each input port, an output port is always proposed to with a probability proportional to the packet arrival rate of the corresponding VOQ. However, this proposing strategy has a subtle shortcoming: It is oblivious to the current lengths of N VOQs at each input port, so not enough attention is devoted to reducing the lengths of longest VOQs. For example, a VOQ with many packets but without recent arrivals, which could happen under bursty traffic, will mostly be denied service until it has new arrivals.

In [**GTL$^+$17**], researchers proposed a data structure that allows the switch to constantly maintain keen "situational awareness" concerning the lengths of its N^2 VOQs. This data structure supports, in constant (i.e., $O(1)$) time, an operation called *Queue-Proportional Sampling (QPS)* that generates an excellent starter matching. It was shown in [**GTL$^+$17**], that SERENA, when using the QPS-generated starter matching instead (called QPS-SERENA), has better delay performances than using the $A(t)$-generated starter matching. In addition, just like SERENA, QPS-SERENA can also attain 100% throughput under all traffic patterns.

Furthermore, scheduling algorithms that start from "scratch" (*i.e.,* an empty matching), such as iSLIP, may also benefit significantly from QPS, by instead starting from a QPS-generated starter matching. It was shown in [**GTL$^+$17**] that iSLIP, when using a QPS-generated starter matching instead (called QPS-iSLIP), attain higher throughputs and better delay performances than iSLIP under various traffic patterns.

The QPS proposing strategy, at any input port, is extremely simple to state: The input port proposes to an output port with a probability proportional to the length of the corresponding VOQ. QPS's name comes from the fact that the output port proposed to by any input port is sampled, out of all N output ports, using the queue-proportional distribution at the input port.

It was shown in [**GTL$^+$17**] that the aforementioned data structure that can perform a QPS operation (at an input port) in $O(1)$ time is comprised entirely of "rudimentary" components such as arrays and linked lists. We skip the details of the data structure here, which can be found in [**GTL$^+$17**]. The constant time complexity of the QPS operation may be surprising to readers, since even to "read" the lengths of all N VOQs at an input port takes $O(N)$ time. However, a simple explanation is that, properly designed, the QPS data structure really only needs to track the small number of changes in the lengths of these N VOQs, caused by packet arrivals to and departures from the input port during a switching cycle. This is exactly what the data structure does: gain constant situational awareness via learning gradually and steadily over time rather than on the spot.

1.12 SCALING TO LARGER AND FASTER SWITCHES

As various hurdles exist for existing switching schemes to scale to a large number of ports and higher per-port speeds, Section 1.12 describes two approaches towards that. Both approaches achieve the two scalability objectives by reducing one or both of the two major switching costs, namely, the overall size of the switch circuitry and the time complexity of the bipartite matching computation. The first approach, based on the principle of divide and conquer (**P15**), reduces the overall size of the switch circuitry through the use of more space-efficient switch fabrics than a monolithic crossbar, such as the Clos fabric (Section 1.12.2) and the Benes fabric (Section 1.12.3). The other, called Load-Balanced Switching (LBS), reduces the algorithmic cost of matching computation to virtually zero. However, it does so by using two large crossbars instead of one.

So far this chapter has concentrated on fairly small switches that suffice to build up to a 32-port router. Most Internet routers deployed up to the point of writing have been in this category, sometimes for good reasons. For instance, building wiring codes tend to limit the number of offices that can be served from a wiring closet. Thus switches for local area networks [**SP94**] located in wiring closets tend to be well served with small port sizes.

However, the telephone network has generally employed a few very large switches that can switch 1000–10,000 lines. Employing larger switches tends to eliminate switch-to-switch links, reducing overall latency and increasing the number of switch ports available for users (as opposed to being used to connect to other switches). Thus, while a number of researchers (e.g., Refs. Tur97 and CFFT97) have argued for such large switches, there was little large-scale industrial support for such large switches until recently.

There are three recent trends that favor the design of large switches.

1. **DWDM:** The use of dense wavelength-division multiplexing (DWDM) to effectively bundle multiple wavelengths on optical links in the core will effectively increase the number of logical links that must be switched by core routers.
2. **Fiber to the home:** There is a good chance that in the near future even homes and offices will be wired directly with fiber that goes to a large central office–type switch.
3. **Modular, multichassis routers:** There is increasing interest in deploying router clusters, which consist of a set of routers interconnected by a high-speed network. For example, many network access points connect up routers via an FDDI link or by a Gigaswitch (see Section 1.4). Router clusters, or *multichassis* routers as they are sometimes called, are becoming increasingly interesting because they allow *incremental growth*, as explained later.

Maybe to add one more bullet to the list above? Data center networks?