

# Scheduling VOQ Switches under Non-Uniform Traffic

**Adisak Mekkittikul**

**Nick McKeown**

Departments of Electrical Engineering and Computer Science  
Stanford University, Stanford, CA 94305-4070

**Abstract** — *Input queueing is becoming increasingly important for high-bandwidth switches and routers. In previous work, we proved that it is possible to achieve 100% throughput for input-queued switches using a cell-by-cell scheduling algorithm called LQF. However, LQF is too complex to implement in hardware. In this paper we introduce a new algorithm called Longest Port First (LPF), which is designed to overcome the complexity problem of LQF and can be implemented in hardware at high speed. By giving preferential service based on queue lengths, we prove that LPF can achieve 100% throughput.*

## 1 Introduction

Congestion in today's high-speed backbone networks, particularly the Internet, has spurred interest in the development of very high-speed, high aggregate bandwidth switches and routers. However, high-bandwidth switch design requires a different approach from today's commercial medium-speed switches. Fast line rates, coupled with limited memory bandwidth, increasingly require switches to use input queueing [11], instead of output queueing. Output queueing is well known for its high efficiency and ability to guarantee quality-of-service (QoS) [3][14][19][20]. But output queueing requires buffer memories that run  $N$  times faster than the line rate, where  $N$  is the number of switch ports. Input queueing, on the other hand, does not have such a requirement [6]. Therefore, switches with input queueing can run faster than switches with output queueing.

But input queueing has two problems: potentially low efficiency due to head-of-line (HOL) blocking [5] and the difficulty to control cell delay. In this paper, we address the first problem. Although HOL blocking is well known to limit the throughput of input-queued switches with a single FIFO at each input to approximately 58% [7], it can be completely eliminated by a queueing technique known as virtual output queueing (VOQ) [1][10][12][13][16]. Input-queued switches using VOQ need a scheduling algorithm to schedule cell transmission across the switching fabric, and their efficiency

depends on the scheduling algorithm used. It has been shown that a scheduling algorithm that uses a maximum *size* bipartite matching algorithm can increase the throughput from 58% to 100% when traffic is uniform and independent [10]. Intuitively, it is not difficult to explain. By its definition, a maximum size matching algorithm offers the maximum instantaneous throughput, in other words, forwards a maximum number of cells in any cell-time. Moreover, a maximum size matching algorithm is not too complex to implement. Most practical algorithms that are fast and simple to implement in hardware approximate a maximum size matching algorithm [1][10][17].

Unfortunately, a maximum size matching algorithm is known to perform poorly when traffic is not uniform [12]. In reality, of course, traffic is not uniform. When traffic is non-uniform, the occupancies of the input queues can differ greatly. Queues with heavy traffic can overflow while ones with light traffic are empty most of the time. This is not only undesirable but leads to low throughput. A maximum size matching algorithm cannot prevent queue overflow because it does not consider queue lengths when servicing the input queues.

In earlier work [12][13], we overcame this problem with a new algorithm: the longest queue first (LQF). LQF gives preferential service to long queues by using a maximum *weight* matching algorithm with each weight set to the corresponding queue length. LQF can achieve 100% throughput for both uniform and non-uniform traffic. But LQF is not a practical algorithm;<sup>1</sup> it is very hard to implement in hardware and cannot operate at high speed. So are its approximations [10]; they involve many multiple-bit comparators, which limit the speed of the algorithm.

We propose a new algorithm that overcomes the impracticalities of LQF and its approximations. We avoid using a maximum weight matching algorithm by asking a different question:

---

1. For an  $N \times N$  switch, the complexity of the most efficient algorithm [18] known to date is  $O(N^3 \log N)$ .

Can we make a maximum size matching algorithm consider queue lengths?

The idea behind this approach is as follows. Among choices of maximum size matches of equal size, we can choose one that has the largest total weight, where each weight is a function of queue lengths. This would allow us to take advantage of both the high instantaneous throughput of a maximum size matching algorithm and the ability to prevent queues from overflow of a maximum weight matching algorithm. But we need to do this without greatly increasing the complexity.

Our new algorithm, Longest Port First (LPF) provides a simple mechanism to select such a match. By carefully weighting requests, a maximum size matching algorithm only requires a simple modification so that it can choose a maximum size match that has the largest weight.

Consequently, LPF can be easily approximated and implemented using a variety of existing algorithms [1][9][10][16] which already approximate a maximum size matching algorithm. Furthermore, considering that the approximating algorithm can take advantage of pipeline implementation to speedup scheduling decisions, we analyze the effect of a pipeline delay on LPF and find that LPF can tolerate a pipeline delay well without any loss in throughput.

The paper is organized as follows. In Section 2, we give an overview of input-queued switches using VOQ and introduce necessary definitions. In Section 3, we describe LPF and its properties, and present our performance analysis.

## 2 Our Switch Model

Figure 1 shows an  $M \times N$  input-queued switch consisting of  $M$  input and  $N$  output ports, a nonblocking switching fabric and a scheduler. To eliminate head-of-line (HOL) blocking, each input maintains  $N$  FIFO queues, one for each output. We call these queues

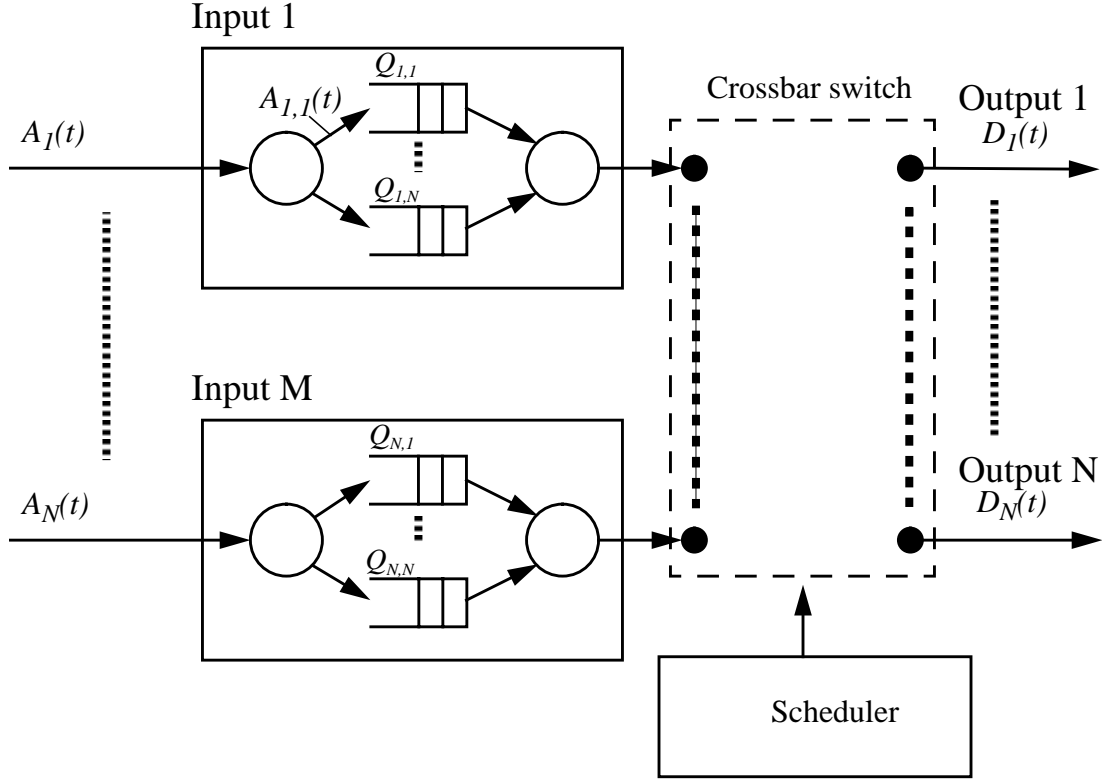


Figure 1: A Simple Model of VOQ Switches

“virtual output queues” (VOQs).  $Q_{i,j}$  denotes the queue at input  $i$  for cells destined to output  $j$ . Arrivals are fixed size packets or cells. Time is slotted into cell-times that we call slots. During any given slot, there is at most one arrival to and departure from each input, and similarly for each output.  $A_{i,j}(n)$  is the arrival process of cells to input  $i$  destined to output  $j$  at rate  $\lambda_{i,j}$ . Consequently,  $A_i(n)$  is the aggregate process of all arrivals to input  $i$

$$\text{at rate } \lambda_i = \sum_{j=1}^N \lambda_{i,j}.$$

**Definition:** An arrival process is said to be admissible when no input or output is oversub-

$$\text{scribed, i.e., when } \sum_{i=1}^M \lambda_{i,j} < 1, \sum_{j=1}^N \lambda_{i,j} < 1, \lambda_{i,j} \geq 0.$$

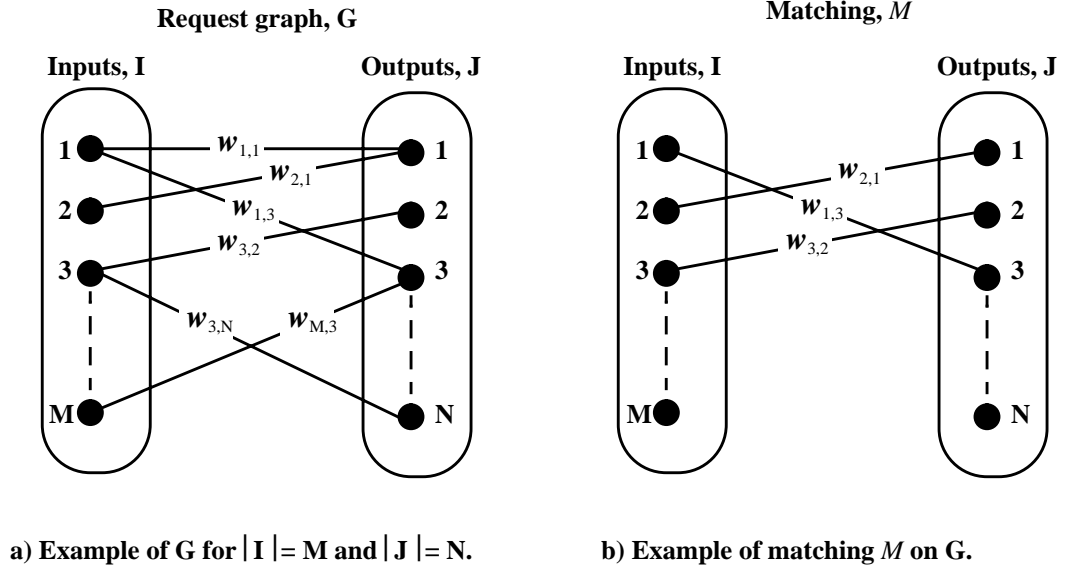
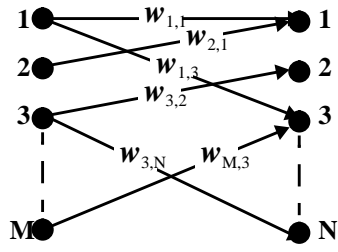


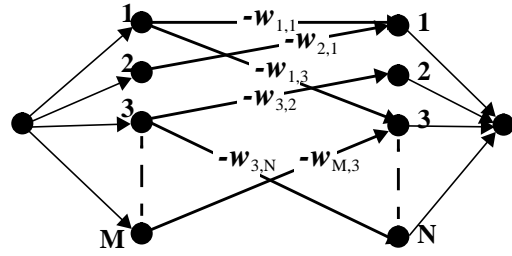
Figure 2: A request graph and a matching graph of an  $M \times N$  switch. Define  $G = [V, E]$  as an undirected graph connecting the set of vertices  $V$  with the set of edges  $E$ . The edge connecting vertices  $i$ ,  $1 \leq i \leq M$  and  $j$ ,  $1 \leq j \leq N$  has an associated weight denoted  $w_{i,j}$ . Graph  $G$  is bipartite if the set of inputs  $I = \{i: 1 \leq i \leq M\}$  and outputs  $J = \{j: 1 \leq j \leq N\}$  partition  $V$  such that every edge has one end in  $I$  and one end in  $J$ . Matching  $M$  on  $G$  is any subset of  $E$  such that no two edges in  $M$  have a common vertex.

**Definition:** The traffic is uniform if all arrival processes have the same arrival rate, and destinations are uniformly distributed over all outputs. Otherwise the traffic is non-uniform.

While VOQ eliminates HOL blocking [6], our switch needs a scheduler to determine which inputs and outputs are connected during each slot. Typically, a centralized scheduler chip is used — each switch port is connected to the scheduler [11][15]. The scheduling problem can be viewed as a bipartite graph matching problem [2][18], an example of which is shown in Figure 3. Each input makes a request to every output for which it has cells waiting. An edge in the graph represents a request from  $Q_{i,j}$  with weight  $w_{i,j}(n)$  (denoted in Figure 3 and Figure 3 as  $w_{i,j}$ ). Weighting requests provides a mechanism for a scheduler to give preferential service to the input queues, for example, based on a function of queue lengths. Doing so enables a scheduler to adapt to different traffic conditions in order to maximize the throughput.



a) Weighted request graph.



b) A corresponding flow network.

Figure 3: Transformation of a request graph into a flow network. (a) A weighted request graph. (b) The corresponding flow network,  $G$ , whose all edges are of unity capacity. A source  $s$  and a target  $t$  are added. The cost of every edge from  $s$  and to  $t$  is set to zero. The cost of all other edges are equal to the negated value of the corresponding weight.

Let  $S_{i,j}(n)$  be a service indicator such that  $\sum_{i=1}^M S_{i,j}(n) \leq 1$  and  $\sum_{j=1}^N S_{i,j}(n) \leq 1$ ; a value

of one indicates that input  $i$  is matched to output  $j$ , i.e.,  $Q_{i,j}$  is allowed to forward one cell to its output.

**Definition:** A maximum size match is one that maximizes  $\sum_{i,j} S_{i,j}(n)$ , i.e., the number of connections.

**Definition:** A maximum weight match is one that maximizes  $\sum_{i,j} S_{i,j}(n)w_{i,j}(n)$ , i.e., the total weight.

Alternatively, a bipartite graph matching problem can be easily solved and understood by transforming it into a flow network [2][18], as illustrated in Figure 3. A maximum size match is found by solving for a maximum flow while a maximum weight match is obtained by solving for a minimum cost flow.

### 3 LPF Algorithm

Although in practice, LPF can be thought of as an extension of a maximum size matching algorithm; in theory, it is easier to consider LPF as a maximum weight matching algorithm. LPF is, by definition, a maximum weight matching algorithm. Each LPF weight is a function of queue occupancies. More precisely, a request weight,  $w_{i,j}(n)$ , for a request from input  $i$  to output  $j$  is defined as follows.

$$w_{i,j}(n) = \begin{cases} R_i(n) + C_j(n), & L_{i,j}(n) > 0 \\ 0, & \text{otherwise,} \end{cases} \quad (1)$$

where  $L_{i,j}(n)$  is an occupancy of an input queue  $Q_{i,j}$  at slot  $n$ ,  $R_i(n) = \sum_j^N L_{i,j}(n)$  and

$$C_j(n) = \sum_i^N L_{i,j}(n).$$

The rationale for weighting requests this way is the following.  $R_i(n)$ , we call an input occupancy, is the total number of cells that are currently waiting at input  $i$  to be forwarded one by one in each slot to their outputs. Similarly,  $C_j(n)$ , an output occupancy, is the total number of cells at all inputs to be forwarded to output  $j$ , which can receive only one cell in each slot. Together, an input occupancy and output occupancy represent a work load or congestion that a cell will encounter as it competes for transmission to its output destination. Therefore, LPF favors a set of queues with high input and output occupancies in order to prevent them from overflowing.

**Property 1:** *The total weight of an LPF match is equal to the occupancy sum of all matched inputs and outputs, i.e.,  $\sum_{i,j} S_{i,j}(n) w_{i,j}(n) = \sum_{i \in I} R_i + \sum_{j \in J} C_j$ , where  $I$  and  $J$  are the set of matched inputs and matched outputs respectively.*

We now show that LPF is actually an extension of a maximum size matching algorithm.

**Theorem 1:** *A match that LPF finds is both a maximum size and maximum weight match.*

**Proof:** see Appendix A.  $\square$

Since an LPF match is actually a maximum size match, we can use a maximum size matching algorithm to find an LPF match. But we need to make sure that among all possible maximum size matches we choose one with the largest total weight. The following describes our modified maximum size matching algorithm and our throughput analysis of LPF.

### 3.1 Finding an LPF Match Using a Maximum Size Matching Algorithm

Existing maximum size matching algorithms cannot be used to implement LPF because they do not have any mechanism to select a maximum size match that has the largest weight. A simple modification is needed. In order to find an LPF match, a maxi-

#### Modified Edmonds-Karp algorithm

```

1   for each edge  $(u, v) \in E[G]$ 
2       do  $f[u, v] = 0$ 
3          $f[v, u] = 0$ 
4   while LPFS finds a path  $p$  from  $s$  to  $t$  in the residual network  $G_f$ 
5       for each edge  $(u, v)$  in  $p$ 
6           do if  $f[u, v] = 0$  then  $f[u, v] = c[u, v]$ 
7             else  $f[v, u] \leftarrow 0$ 
8              $f[u, v] = -f[v, u]$ 

```

Figure 4: Modified Edmonds-Karp algorithm [2].  $G$  is a flow network or graph constructed as described in Figure 3.  $E[G]$  is the set of all edges in  $G$ ;  $u$  or  $v$  is a vertex in  $G$  representing an input or output;  $(u, v)$  is an edge from  $u$  to  $v$ ;  $f$  is the total flow through the network;  $f[u, v]$  denotes a flow from  $u$  to  $v$ .  $G_f$  is a residual network [2] [18], also called a residual graph. LPFS is a largest unmatched port first search.

### LPFS(G)

```
1  for each vertex  $u \in V[G]$ 
2      do  $color[u] \leftarrow white$ 
3       $\pi[u] \leftarrow nil$ 
4  LPFS-Visit( $t$ )
```

### LPFS-Visit( $u$ )

```
1   $color[u] \leftarrow gray$ 
2  for each  $v \in Adjacent[u]$  , starting the largest to the smallest.
3      do if  $color[v] = white$ 
4          then  $\pi[v] \leftarrow u$ 
5          LPFS-Visit( $v$ )
6   $color[u] = black$ 
```

Figure 5: A largest-unmatched-port first search (LPFS). First, LPFS builds a tree with  $t$  as its root. Initially every input and output is colored white — undiscovered, then is grayed when it is discovered, and finally is blackened when it is finished.  $\pi[v]$  is the predecessor of  $v$ . From the tree, an augmenting path from  $s$  to  $t$  which must go through an unmatched input can be found by walking the predecessor list which begins at a selected unmatched input.

---

imum size matching algorithm must take request weights into consideration when selecting a match. Described in pseudocode in Figure 4 is a modified Edmonds-Karp algorithm [2][18]. A simple modification is made to the original algorithm so that the new algorithm can find an LPF match. A breadth first search (BFS) in the original algorithm is replaced by a largest-unmatched-port first search (LPFS) described in Figure 3. LPFS enables the modified algorithm to search for a maximum weight match while performing path augmentation [18] to find a maximum size match. In order to keep the algorithm free of complex magnitude comparison, all inputs and outputs are pre-ordered prior to running the algorithm. As a result, line 2 of the LPFS-Visit does not involve any magnitude comparison. It is proved in Appendix B that the modified algorithm, in fact, finds a match that is an LPF match — a match that is both a maximum size and weight match.

**Theorem 2:** *A maximum size match found by the modified Edmonds-Karp algorithm is also a maximum weight match whose weights are as defined in Equation 1.*

**Proof:** see Appendix A.  $\square$

### 3.2 A Practical Approximation to LPF

Similar to other theoretical scheduling algorithms, LPF can be adapted to run at higher speed using simpler approximations. Shown in Figure 6 is a simple iterative algorithm approximating LPF called *i*LPF. All weight processing is done in step 1. The second step consists of a double for-loop used to find a *maximal* size match. Since the requests have already been ordered in the first step, the maximal size matching in the second step does not need to compare request weights. Shown in Figure 7 is a hardware implementation of *i*LPF. Our exploratory design work suggests that the second step can be implemented using simple hardware; for a  $32 \times 32$  switch, our synthesized design can make a scheduling decision in just 10 ns using Texas Instruments' TSC5000 0.25  $\mu\text{m}$  CMOS ASIC technology. The first step, which requires simple integer arithmetic, can also run in 10 ns, allowing the switch to run at a line rate of 20 Gbps<sup>1</sup>.

### 3.3 Stability

We now prove that LPF can achieve 100% throughput for all traffic patterns with independent arrivals. We do this by using the notion of *stability* [8]. We define a switch to be stable for a particular arrival process if the expected length of the input queues does not grow without bound, i.e.,

$$E\left[\sum_{i,j} L_{i,j}(n)\right] < \infty, \forall n. \quad (2)$$

**Definition:** *A switch can achieve 100% throughput if it is stable for all independent and admissible arrivals.*

---

1. Calculated based on the size of an ATM cell.



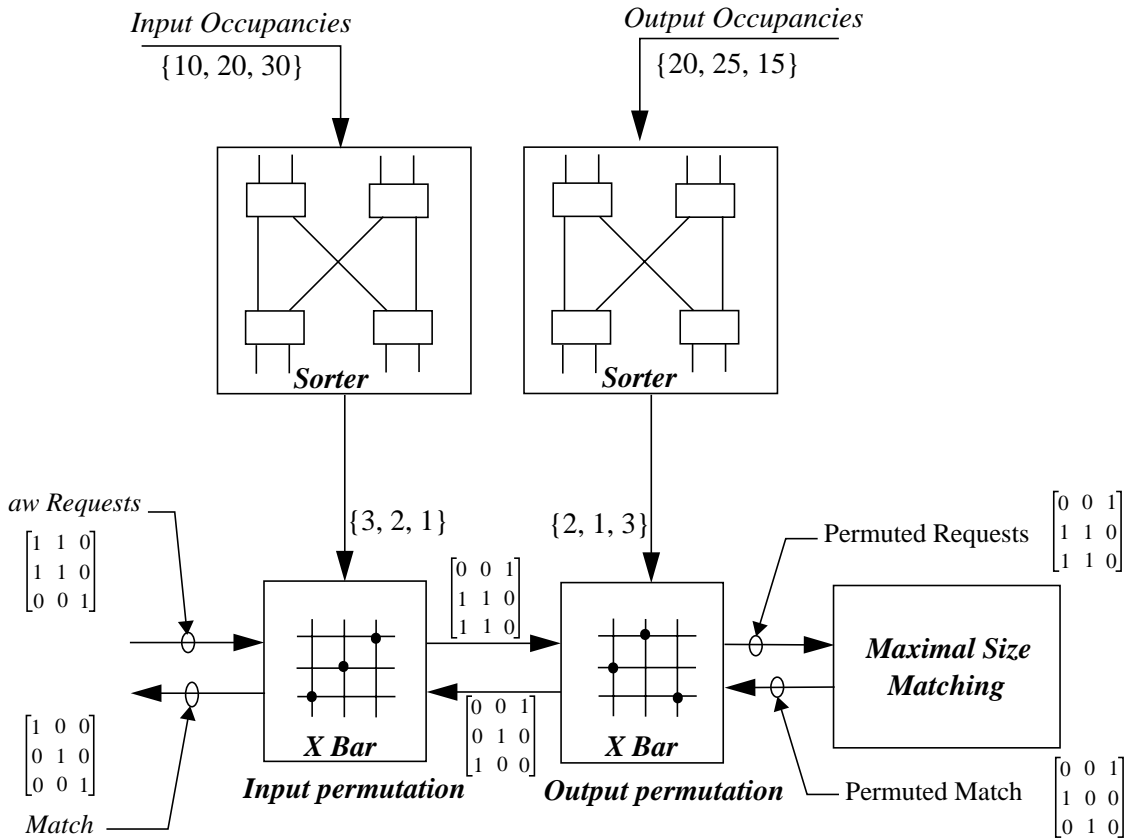


Figure 7: A block diagram of *iLPF*. Referring to the algorithm in Figure 6, inputs and outputs are pre-sorted by the two sorter networks. Raw requests (requests with weights removed) is given in a matrix form. Request reordering is done by the two crossbars which are configured by the sorting results. The maximal size matching block, which implements the double for-loop, finds a maximal size match that approximates an LPF match. The match needs to be permuted back to its natural order.

the performance of LPF. The following analysis includes the stability of LPF with a pipeline delay.

Pipelining allows each sorter to increase its throughput so that it does not become a speed bottleneck. But the consequence of breaking up the sorters into several stages of pipeline is that the sorter outputs represent old information, and therefore misconfigures the request stages. In fact, a  $k$  slot pipeline delay is simply equivalent to non-pipelined LPF but with  $k$  slot old weights,  $w_{i,j}(n-k)$ . Hence, it finds the match based on the wrong

weights, i.e., maximizes  $\sum_{i,j} S_{i,j}(n)w_{i,j}(n-k)$ . Perhaps surprisingly, we can verify the following:

**Theorem 4:** *Using  $k$  slot old weights, the LPF algorithm is stable for all admissible independent arrival processes,  $0 < k < \infty$ .*

**Proof:** see Appendix B.  $\square$

### 3.5 Conclusion

Growth in traffic and congestion indicate a need for new high-bandwidth switches and routers. While output-queued switches are limited by a speedup requirement, it has been shown that input-queued switches can offer as high as one terabit-per-second aggregate bandwidth [11] using only standard technologies. To achieve high throughput, we recommend that input-queued switches employ virtual output queueing (VOQ) and a centralized scheduling algorithm. Most existing algorithms are either inefficient or too complex to run at high speed. Our new scheduling algorithm, LPF, is both efficient and practical, and can achieve 100% throughput for all non-uniform traffic with independent arrivals. Because LPF uses a maximum *size* matching algorithm, it can be well approximated by many existing algorithms; algorithms that are fast and simple to implement in hardware. We also find that LPF is tolerant of a pipeline delay, lending itself well to fast pipelined implementation.

## 4 References

- [1] Anderson, T.; Owicki, S.; Saxe, J.; and Thacker, C. "High speed switch scheduling for local area networks," *ACM Trans. on Computer Systems*. Nov 1993 pp. 319-352.
- [2] Cormen, T.; Leiserson C.E.; Rivest R.L. "Introduction to algorithms," The MIT Press, Cambridge, Massachusetts, March 1990.
- [3] Demers, A., et al. "Analysis and simulation of a fair queueing algorithm," *Internet-working: Research and Experience*, Sept. 1990, vol.1, no.1, pp. 3-26.
- [4] Hopcroft, J.E.; Karp, R.M. "An  $n^{5/2}$  algorithm for maximum matching in bipartite graphs," *Society for Industrial and Applied Mathematics J. Comput.*, 1973, pp.225-231.

- [5] Karol, M.; Eng, K.; Obara, H. "Improving the performance of input-queued ATM packet switches," *INFOCOM '92*, pp.110-115.
- [6] Karol, M.; Hluchyj, M. "Queueing in high-performance packet-switching," *IEEE J. Selected Area Communications*, Vol.6, Dec. 1988, pp.1587-1597.
- [7] Karol, M.; Hluchyj, M.; and Morgan, S. "Input versus output queueing on a space division switch," *IEEE Trans. Communications*, 35(12) (1987) pp.1347-1356.
- [8] Kumar, P.R.; Meyn, S.P.; "Stability of queueing networks and scheduling policies," *IEEE Transactions on Automatic Control*, Vol.40, No.2, Feb. 1995, pp. 251-260.
- [9] Liu, C. L.; "Topics in combinatorial mathematics," Mathematical Association of America, Massachusetts, 1972.
- [10] Lund, C.; Phillips, S.; Reingold, N. "Fair prioritized scheduling in an input-buffered switch," *Proceedings of the International IFIP-IEEE Conference on Broadband Communications*, Montreal, Que., Canada April 1996, pp. 358-69.
- [11] McKeown, N. "Scheduling Algorithms for Input-Queued Cell Switches," PhD Thesis. University of California at Berkeley, 1995.
- [12] McKeown, N.; Izzard M.; Mekkittikul, A.; Ellersick W.; Horowitz M. "The Tiny Tera: a packet switch core," *Proceedings of Hot Interconnects IV*, Stanford, Aug 1996, pp. 161-173.
- [13] McKeown, N.; Anantharam, V.; and Walrand, J. "Achieving 100% throughput in an input-queued switch," *Proceedings of INFOCOM, 1996*, pp. 296-302.
- [14] Mekkittikul, A.; McKeown, N "A Starvation-free Algorithm for Achieving 100% Throughput in an Input-Queued Switch," *Proceedings of ICCCN'96*, October, 1996, pp. 226-231.
- [15] Parekh, A.K.; Gallager, R. "A generalized processor sharing approach to flow control in integrated services networks: the multiple node case," *IEEE/ACM Transactions on Networking*, vol.2, no.2, April 1994, pp. 137-50.
- [16] Partridge, C.; et al. "A fifty gigabit per second IP router," Submitted to *IEEE/ACM Transactions on Networking*.
- [17] Tamir, Y.; Frazier, G. "High performance multi-queue buffers for VLSI communication switches," *Proc. of 15th Ann. Symp. on Comp. Arch.*, June 1988, pp.343-354.
- [18] Tamir, Y. et al. "Symmetric crossbar arbiters for VLSI communication switches" *IEEE Transactions on Parallel and Distributed Systems*, Jan 1993. vol.4, no.1, pp. 13-27.
- [19] Tarjan, R.E. "Data structures and network algorithms," *Society for Industrial and Applied Mathematics*, Pennsylvania, Nov 1983.
- [20] Zhang, H. "Service disciplines for guaranteed performance service in packet-switching networks" *Proceedings of the IEEE*, vol.83, no.10, Oct, 1995, pp. 1374-96.
- [21] Zhang, L. "VirtualClock: a new traffic control algorithm for packet-switched networks," *ACM Transactions on Computer Systems*, vol.9, no.2, May 1991, pp. 101-24.

## Appendix A: An LPF Match Property

**Theorem 1:** *A match that LPF finds is both a maximum size and maximum weight match.*

**Proof:** We prove the theorem by contradiction. Let  $M$  be a maximum weight match but not a maximum size match found by LPF; that is, there exists another larger size match  $M'$  that can be found by any maximum size matching algorithm.

Consider the Ford-Fulkerson method [2][18]. With this method, if  $M$  was not a maximum size match, a larger size match  $M'$  could be found by augmenting the flow produced by  $M$  on the corresponding flow network similar to the one shown in Figure 3. Since this augmentation process does not remove any matched input or output of the previous flow (match) [2][18],  $M'$  would contain all matched inputs and outputs in  $M$  plus some newly matched inputs and outputs. As a result of the set of inputs and outputs in  $M$  being a subset of that in  $M'$ , Property 1 implies that, had it existed,  $M'$  would be a larger weight match than  $M$ . Hence,  $M$  could not be the match found by LPF because it was not a maximum weight match. This contradicts the assumption above. Therefore,  $M'$  does not exist, and an LPF match must be both a maximum size and maximum weight match.  $\square$

## 5 Modified Edmonds-Karp algorithm is equivalent to LPF

**Theorem 2:** *A match found by the modified Edmonds-Karp algorithm is a maximum weight match whose weights are as defined in Equation 1.*

**Proof:** Before we prove the theorem, we first prove the following lemma needed for proving the theorem.

Let  $M$  be a match found by the modified Edmonds-Karp algorithm,  $F$  be the associated flow of  $M$  and  $R$  be a residual graph produced by  $F$ . Based on the principle that a maximum weight match can be found by solving for a flow of a minimum cost, the strategy for the proof is therefore to show that  $F$  is a minimum cost flow on the associated flow network. To show that, we need the following theorem [18].

**Theorem:** *A flow  $F$  is minimum cost if and only if its residual graph  $R$  has no negative cost cycle.*

Based on this theorem, the following proves that  $F$  is a minimum cost flow by showing that  $R$  contains no negative cycle. As an example, Figure 8 shows is a residual graph of a match which is of only maximum size, not a maximum weight, because the residual graph contains one negative cost cycle (shown by the dashed lines). In principle, the cycle in this figure indicates that the total cost of the flow can be lowered by replacing the matched edges (input and output pairs) in the cycle with the unmatched edges [52]. Before we proceed with the proof, it is necessary to state the following properties, which are required for the proof and can be easily verified.

**Property 2:** *Flow augmentation<sup>1</sup> does not remove any input or output from the matched sets but adds one or more new inputs and outputs to the sets. Once matched, every input or output remains matched throughout the augmentation process.*

**Property 3:** *In  $R$ , an input is never adjacent to another input, and an output is never adjacent to another output; i.e., no edge exists between any two inputs or between any two outputs.*

**Property 4:**  *$F$  cannot be increased since it is already a maximum flow ( $M$  is a maximum size match).*

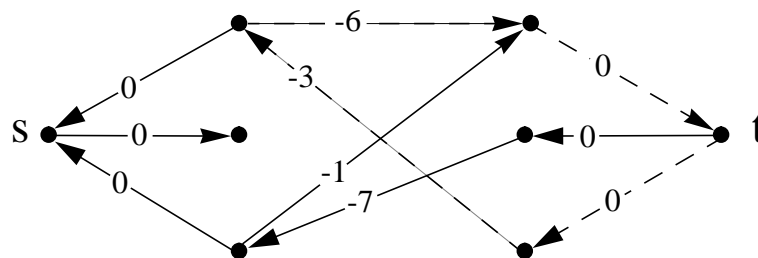


Figure 8: A residual network containing a negative cost cycle shown by the dotted edges.

---

<sup>1</sup>The Ford-Fulkerson method.

**Property 5:** *In  $R$ , all edges from unmatched inputs are to only matched outputs; otherwise,  $F$  is not a maximum flow (an additional flow can be trivially added).*

**Property 6:** *In  $R$ , all edges to unmatched outputs must be from matched inputs; otherwise,  $F$  is not a maximum flow (an additional flow can be trivially added).*

We are now ready to begin the proof by considering the following three cases in which a negative cost cycle can occur. Shown in Figure 10, Figure 11 and Figure 12 are residual graphs containing three types of negative cost cycles: an input cycle, an output cycle and a compound cycle, respectively. These residual graphs are constructed from the corresponding flow network by reversing the directions of all matched edges including the ones connecting  $s$  to matched inputs and  $t$  with matched outputs. All matched edges are represented by the shaded lines, and all unmatched edges are represented by the solid lines. In these graphs, the inputs are rearranged and divided into two non-overlapping sets: matched and unmatched, and similarly, the outputs are also rearranged and divided. The set of matched inputs and the set of matched outputs are shown by the shaded dots and are connected by two sets of edges: matched and unmatched. These sets are shown by the thicker lines.

Through these sets of edges, there exists at least one path which begins at a matched output and ends at a matched input connecting a subset of matched inputs and matched outputs. Each of the paths is formed by matched and unmatched edges lying alternately along the path. On such paths, the inputs and outputs also lie alternately [18]. Moreover, every matched input and matched output must belong to one and only one path. Figure 9 shows examples of such paths. We shall refer to these paths as *reverse flow* paths.

When considering the three cases, we assume that there exists an unmatched edge from some matched input  $i$  to some unmatched output  $l$ , and that there exists an unmatched edge from some unmatched input  $k$  to some matched output  $j$ . However, according to Property 5 and 6, we cannot assume the existence of any edge connecting an unmatched input to an unmatched output in these graphs.

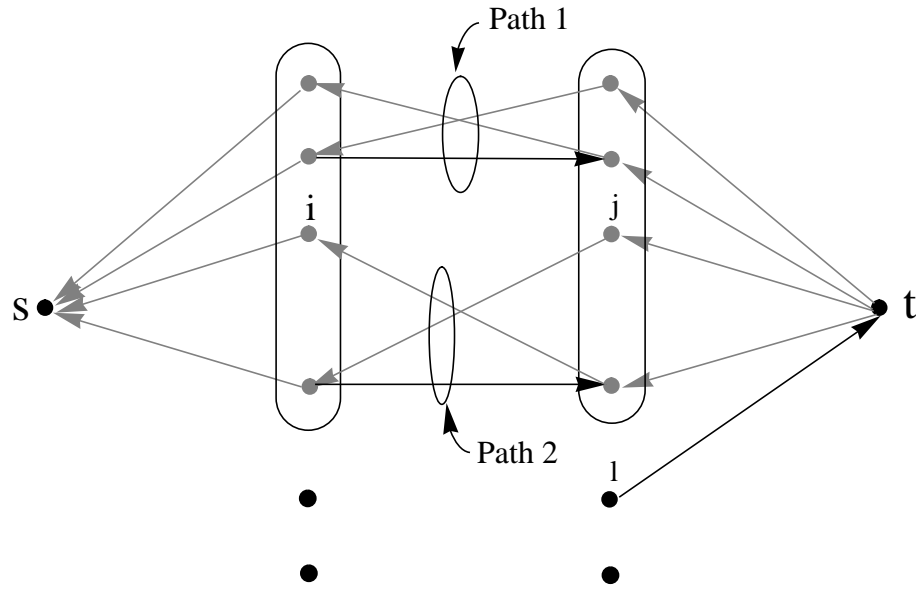


Figure 9: A residual graph showing two reverse flow paths between two pairs of inputs and outputs.

**Case 1: Output cycle:** Shown in Figure 10, the cycle starts from  $t$  to some matched output  $j$ , goes through a reverse flow path connecting  $j$  to some input  $i$ , and finally returns from  $i$  to  $t$  without visiting  $s$  via some unmatched output  $l$ . When the occupancy of  $l$  is greater than that of  $j$ , property 3.1 infers that the cycle has a negative cost.

**Case 2: Input cycle:** Shown in Figure 11, the cycle begins at  $s$  and goes to some unmatched input  $k$  that connects to some matched output  $j$ . From  $j$  the cycle reaches matched input  $i$  from where it can return to  $s$ . Likewise, property 3.1 infers that the cycle is a negative cost cycle if the occupancy of  $i$  less than that of  $k$ .

**Case 3: Compound cycle:** Unlike an input or an output cycle, a compound cycle visits both  $s$  and  $t$ . As shown in Figure 12, the cycle starts from  $s$  and goes to some unmatched input  $k$  which has a request for some matched output  $j$ . The request leads the cycle to  $j$ . From  $j$  to some matched input  $i$ , there exists a reverse flow path which leads the cycle to  $i$ . From  $i$ , the cycle reaches  $t$  via some unmatched output  $l$ . The cycle then

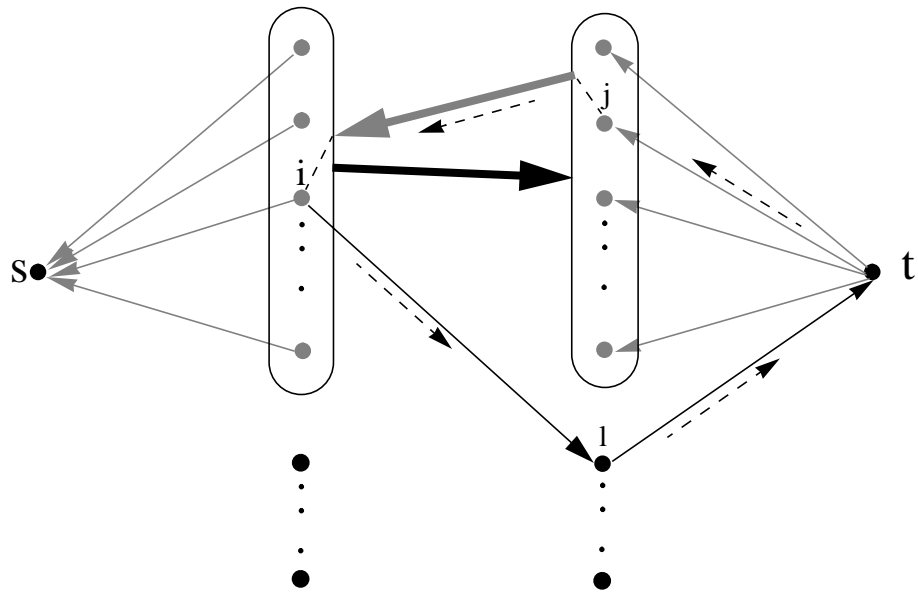


Figure 10: A residual graph showing an output cycle as indicated by the dotted lines.

---

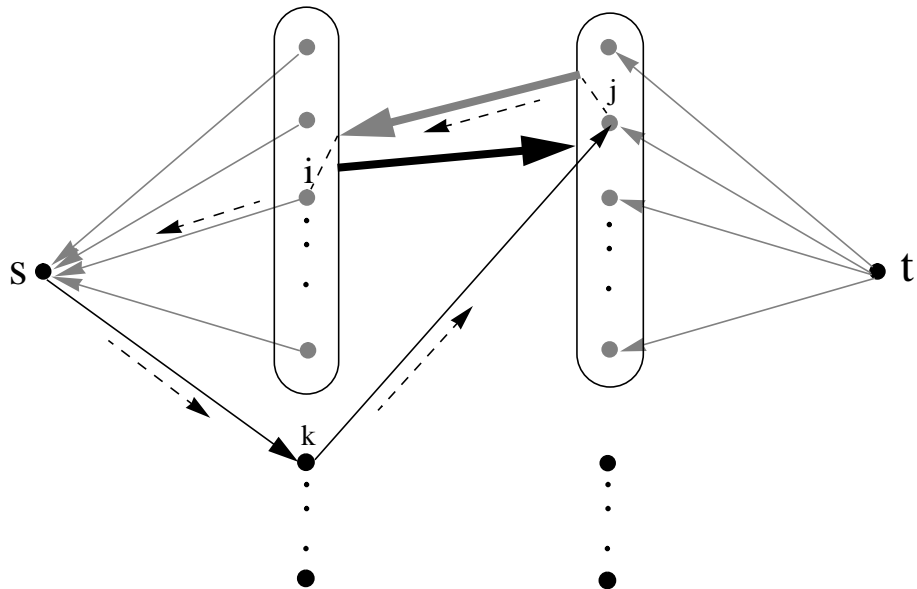


Figure 11: A residual graph showing an input cycle as indicated by the dotted lines.

---

input  $q$  by some reverse flow path. From  $q$ , the cycle reaches  $s$  where it began. Accord-

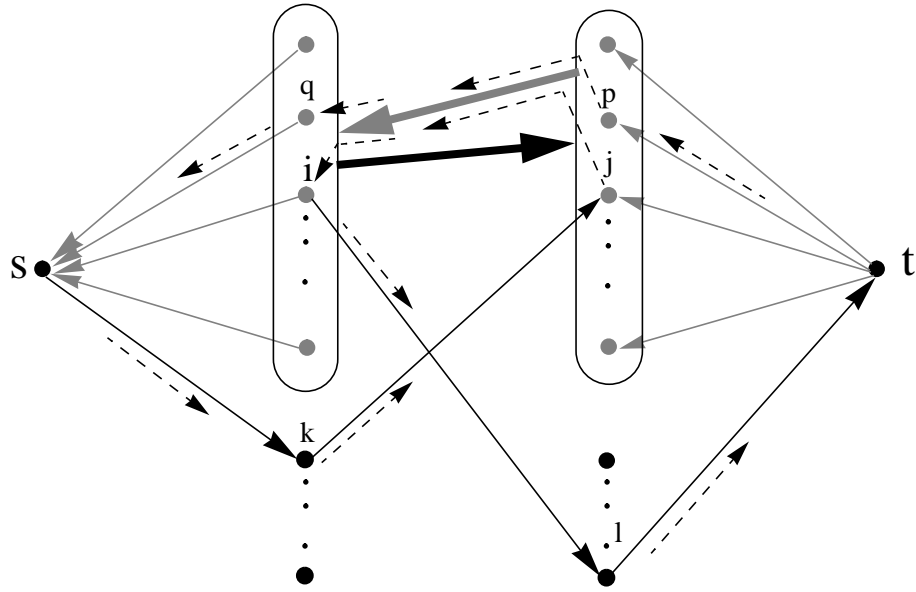


Figure 12: A residual graph showing a compound cycle as indicated by the dotted lines.

---

ing to Property 1, the cycle has a negative cost if the combined occupancy of input  $i$  and output  $j$  is less than the combined occupancy of input  $k$  and output  $l$ .

With the above definitions of negative cycles, we are ready to prove that no negative cycle exists in a residual graph of a match found by the modified algorithm.

**Lemma 1:** *No negative cost output cycle exists in  $R$ .*

**Proof:**

We prove this Lemma by contradiction. First, we assume that such a cycle, as shown in Figure 13, exists. This implies that somehow a reverse flow path  $P_o$  exists in  $R$  from some output  $j$  to input  $i$  that directly leads to output  $l$  by an unmatched edge. Of all outputs on the path,  $j$  must be the smallest; otherwise, we can form a more negative cost cycle by replacing  $j$  with a smaller output on the path.

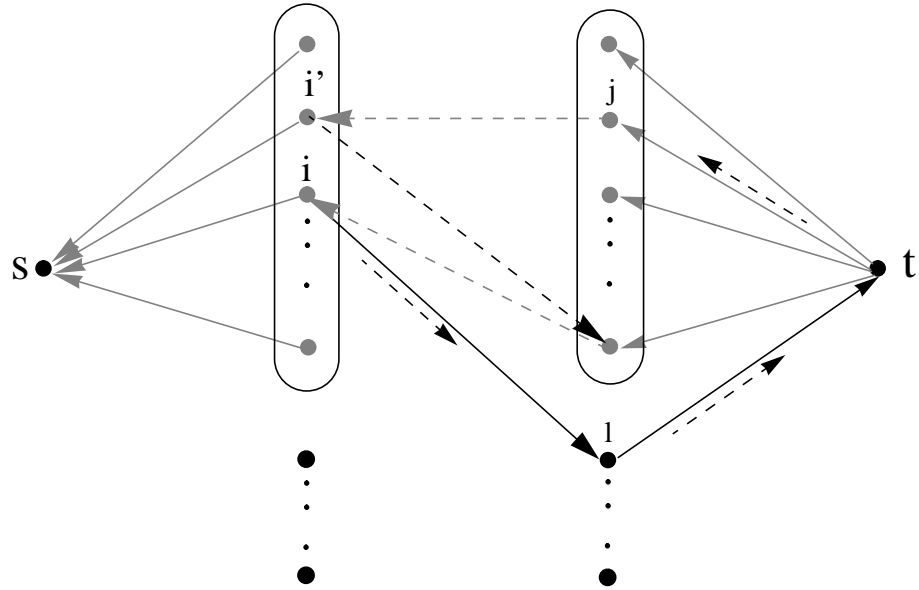


Figure 13: A residual graph for the proof of Lemma 1, depicting an assumed reverse flow path  $P_o$  from  $j$  to  $i$  by the dotted edges.

---

To show that such a path cannot possibly exist, consider the formation of  $P_o$  in Figure 13 when the modified algorithm matches input  $i'$  to  $j$ . When it was the turn of input  $i'$  to be matched, the modified algorithm performed LPFS to find the largest unmatched output for  $i'$ . For  $j$  to be appended to  $P_o$ ,  $i'$  must have at least two requests: one to  $j$  which is subsequently matched and another one to some output on  $P_o$  which is later never unmatched. Hence, there would have been at least two augmenting paths through  $i'$ , leading  $i'$  to more than one unmatched output: (a)  $j$ , (b) some output on  $P_o$  or  $l$  that has not been matched then. Since  $j$  is smaller than  $l$  and all outputs on the path, this creates a contradiction because LPFS would have chosen to match  $i'$  not with  $j$  but with the largest unmatched outputs on the path or  $l$  then.

Given the fact that  $j$  is matched to  $i'$ ,  $P_o$  and hence the cycle do not exist.  $\square$

**Lemma 2:** *No negative cost input cycle exists in  $R$ .*

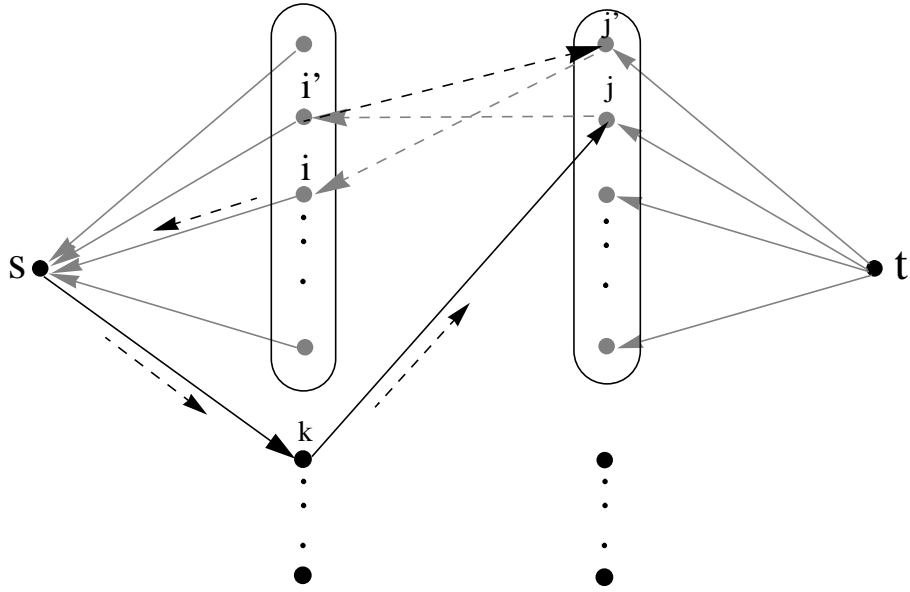


Figure 14: A residual graph for the proof of Lemma 2, depicting an assumed reverse flow path  $P_i$  from  $j$  to  $i$  by the dotted edges.

---

**Proof:**

We prove this lemma by employing the same strategy as that used for the previous lemma. We first assume that the cycle exists and then contradict the assumption by using the property of LPFS. As shown in Figure 14, the assumed cycle visits some matched input  $i$ , some matched output  $j$  and unmatched input  $k$ . Based on LPF request weighting, the occupancy of  $k$  must be greater than that of  $i$  for the cycle to be of a negative cost. But most important of all, in order for the cycle to exist, there must be a reverse flow path,  $P_i$ , from  $j$  to  $i$  in  $R$ .

The following, however, contradicts the existence of the path. Consider that the modified algorithm would attempt to match  $k$  before  $i$  because  $k$  is larger than  $i$ . With the existence of the path, there would be two possibilities to match  $k$ . One possibility was that  $k$  discovered that output  $j$  had not been matched at that time, and thus  $k$  would have been matched to  $j$ . This possibility contradicts the fact that  $k$  is never matched.

The other possibility is that  $j$  is already matched to  $i'$ , which must be larger than  $k$  when it is the turn of  $k$  to be matched. Consequently, the matching of  $j$  appends  $j$  to  $P_i$ . However, from  $j$ ,  $P_i$  would have led  $k$  some unmatched output on  $P_i$  which had not been matched at that time. In the worst case,  $P_i$  would have led  $k$  to  $i'$  and consequently to output  $j'$ , which is supposed be matched to  $i$  but remained unmatched at that time since it was not the turn of  $i$  to be matched yet. In any case, had  $P_i$  existed,  $k$  would be matched.

Given the fact that  $k$  is unmatched,  $P_i$  and hence the cycle do not exist.  $\square$

**Lemma 3:** *No negative cost compound cycle exists in  $R$ .*

**Proof:**

As defined and as illustrated in Figure 15, in order for such a cycle to exist, there must be a path from  $s$  to  $t$  via  $k$  and  $l$ . However, this path is exactly an augmenting path on which more flow can be carried. The existence of the path contradicts Property 4, which

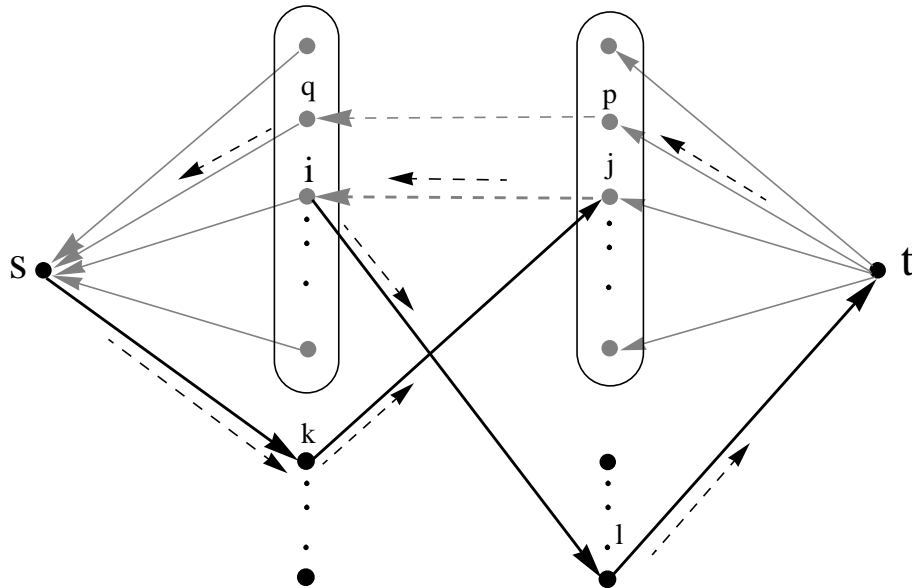


Figure 15: A residual graph for the proof of Lemma 3, showing a forward flow path from  $s$  to  $t$  which can increase the total flow from  $s$  to  $t$ .

says that the flow is already a maximum flow. Therefore, such a path from  $s$  to  $t$  and hence a negative cost compound cycle cannot exist.  $\square$

**Lemma 4:** *No negative cost cycle of any kind exists in  $R$ .*

**Proof:** By Lemma 1, Lemma 2 and Lemma 3.  $\square$

With Lemma 4, we can prove Theorem 2. According to Theorem Theorem:,  $F$  must be a minimum cost flow because, as proved by Lemma 4, its residual graph contains no negative cost cycle. Therefore,  $M$  is a maximum weight match.  $\square$

## Appendix B: LPF Stability

### B.1 Definitions

In addition to the definitions defined in Section 2, in this appendix we also use the following definitions for an  $N \times N$  switch:

1. The rate matrix of the stationary arrival processes:

$$\Lambda \equiv [\lambda_{i,j}], \quad \text{where: } \sum_{i=1}^N \lambda_{i,j} \leq 1, \quad \sum_{j=1}^N \lambda_{i,j} \leq 1, \quad \lambda_{i,j} \geq 0$$

and associated rate vector:

$$\underline{\lambda} \equiv (\lambda_{1,1}, \dots, \lambda_{1,N}, \dots, \lambda_{N,1}, \dots, \lambda_{N,N})^T.$$

2. The arrival matrix, representing the sequence of arrivals into each queue:

$$\mathbf{A}(n) \equiv [A_{i,j}(n)],$$

where:

$$A_{i,j}(n) \equiv \begin{cases} 1 & \text{if an arrival occurs at } Q_{i,j} \text{ at slot } n \\ 0 & \text{else,} \end{cases}$$

and associated arrival vector:

$$\underline{A}(n) \equiv (A_{1,1}(n), \dots, A_{1,N}(n), \dots, A_{N,N}(n))^T.$$

3. The service matrix, indicating which queues are served at time  $n$ :

$\mathbf{S}(n) \equiv [S_{i,j}(n)]$ , where:

$$S_{i,j}(n) \equiv \begin{cases} 1 & \text{if } Q_{i,j} \text{ is served at time } n \\ 0 & \text{else,} \end{cases}$$

and  $\mathbf{S}(n) \in \mathbf{S}$ , the set of service matrices.

Note that:  $\sum_{i=1}^N S_{i,j}(n) = \sum_{j=1}^N S_{i,j}(n) = 1$ , and  $\mathbf{S}(n) \in \mathbf{S}$  is a *permutation matrix*. We define the associated service vector:

$$\underline{S}(n) \equiv (S_{1,1}(n), \dots, S_{1,N}(n), \dots, S_{N,N}(n))^T.$$

4. A positive-definite transformation matrix,  $T_{lpf}$ .

For an  $N \times N$  switch,  $T_{lpf}$  is an  $N^2 \times N^2$  matrix whose elements are defined as follows:

$$T_{lpf(i,j)} = \begin{cases} 2, & i = j \\ 1, & \left\lfloor \frac{i}{N} \right\rfloor = \left\lfloor \frac{j}{N} \right\rfloor \\ 1, & i \bmod N = j \bmod N \\ 0, & \text{otherwise.} \end{cases}$$

As defined,  $T_{lpf}$  is positive definite as well as symmetric.

## B.2 Stability without a Pipeline Delay

### B.2.1 Main Theorem

**Theorem 3:** *Under the LPF algorithm, the queue occupancies are stable for all admissible and independent arrival processes, i.e.,  $E[\|L(n)\|] \leq C < \infty$ .*

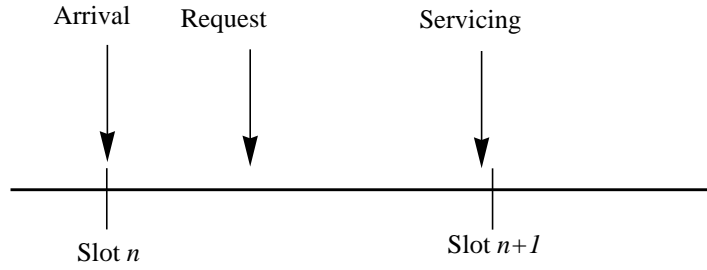


Figure 16: A time line showing events at a VOQ. During every slot, an arrival to the queue, if any, takes place at the beginning of the slot. Shortly after the beginning of the slot, every non-empty queue makes a request. LPF then takes all requests into consideration and arrives at its scheduling decision before the end of the slot. Each selected queue is then serviced before a new arrival occurs.

---

### 5.0.1 Proof

Consider a quadratic Lyapunov function  $V(\underline{L}(n)) = \underline{L}(n)T_{lpf}\underline{L}(n)$ , the next state occupancy vector:<sup>1</sup>

$$\underline{L}(n+1) \equiv \underline{L}(n) - \underline{S}(n) + \underline{A}(n) \quad (3)$$

and an LPF request vector whose each element is a function of queue occupancies defined as:

$$L'_{i,j}(n) = \begin{cases} R_i + C_j, & L_{i,j}(n) > 0 \\ 0, & \text{otherwise,} \end{cases} \quad (4)$$

where  $R_i = \sum_j L_{i,j}(n)$  and  $C_j = \sum_i L_{i,j}(n)$ .

We can directly consider the actual next state occupancy vector because Fact 4 below guarantees that the vector contains no negative element. As shown by Figure 16 that LPF

---

<sup>1</sup>.See [10] for detailed definition.

considers current arrivals when making a scheduling decision, it is trivial to verify the following fact, which is critical for the proofs that follow.

**Fact 4:** A queue with zero occupancy and no arrival cannot make a request. Hence, a queue which makes a request must have either non-zero occupancy, an arrival or both.

**Lemma 5:** Under the LPF algorithm,  $E[\underline{L}'^T(n)(\underline{A}(n) - \underline{S}^*(n)) | \underline{L}(n)] \leq 0$  for

$$\sum_{i=1}^N \lambda_{i,j} \leq 1, \sum_{j=1}^N \lambda_{i,j} \leq 1, \lambda_{i,j} \geq 0, \text{ where } \underline{S}^*(n) \text{ is an LPF service matrix such that}$$

$$\underline{L}'^T(n)\underline{S}^*(n) = \max(\underline{L}'^T(n)\underline{S}(n)).$$

**Proof:** Consider the following two cases. The first case is when the match size is  $N$ ,  $|\underline{S}^*(n)| = N$ .

For this case, Property 1 implies that

$$\underline{L}'^T(n)\underline{S}^*(n) = 2 \sum_{i,j} L_{i,j}(n). \quad (5)$$

This is because all inputs and all outputs are selected (matched). Also, for the match size of  $N$ ,

$$E[\underline{L}'^T(n)\underline{A}(n) | \underline{L}(n), |\underline{S}^*(n)| = N] = \underline{L}'^T(n)T_{lpf}\underline{\lambda}. \quad (6)$$

For the admissibility constraint:  $\sum_{i=1}^N \lambda_{i,j} \leq 1, \sum_{j=1}^N \lambda_{i,j} \leq 1, \lambda_{i,j} \geq 0,$

$$T_{lpf}\underline{\lambda} \leq \underline{2}. \quad (7)$$

Substituting equation 7 into equation 6, we obtain the following inequality,

$$E[\underline{L}'^T(n)\underline{A}(n) | \underline{L}(n), |\underline{S}^*(n)| = N] \leq 2 \sum_{i,j} L_{i,j}(n). \quad (8)$$

By equations 5 and 8, we prove the first case.

$$E [L^T(n) (\underline{A}(n) - \underline{S}^*(n)) | L(n), |\underline{S}^*(n)| = N] \leq 0. \quad (9)$$

For the second case when the match size is  $k < N$ , we use Konig-Egervary's theorem, which states that *the size of a maximum size match is equal to the minimum number of rows plus columns which can contain all requests*<sup>1</sup> [9]. In this case, rows and columns are inputs and outputs. Consider any non-weighted request matrix,  $R(n)$ , from which LPF finds a match of size  $k$ . Consequently, as a result of Theorem 1 (which says that a match found by the LPF is also a maximum size match), there exists a minimum set of  $k$  rows or columns or rows and columns combined which contains all requests in the matrix.

These rows and/or columns may not necessarily be consecutive. In order for the ease of visualizing the proofs, we can rearrange the request matrix so that the columns are consecutively moved to the left of the matrix, and so that the rows are also consecutively moved to the top of the matrix. By permuting the rows and the columns, we obtain such a matrix,  $R'(n)$ , whose first  $l$  rows and  $k - l$  columns belong to the minimum set and therefore contain all requests, where  $0 \leq l \leq k$ , as shown in Figure 17. The shaded area, which is not covered by the first  $l$  rows or  $k - l$  columns, contains no request.

For a given set  $I$  containing the original indices of the first  $l$  rows and a given set  $J$  containing the original indices of the first  $k - l$  columns, let:

$$\tilde{\lambda} \equiv E [\underline{A}(n) | I, J] \quad (10)$$

be a conditional arriving rate vector. Since the shaded area has no arrival,

$$\tilde{\lambda}_{i,j} = \begin{cases} \lambda_{i,j}, & i \in I, i \in J \\ 0, & \text{otherwise.} \end{cases} \quad (11)$$

---

1. "Containing," in this context, means that all requests belong to the interested set of either rows or columns.

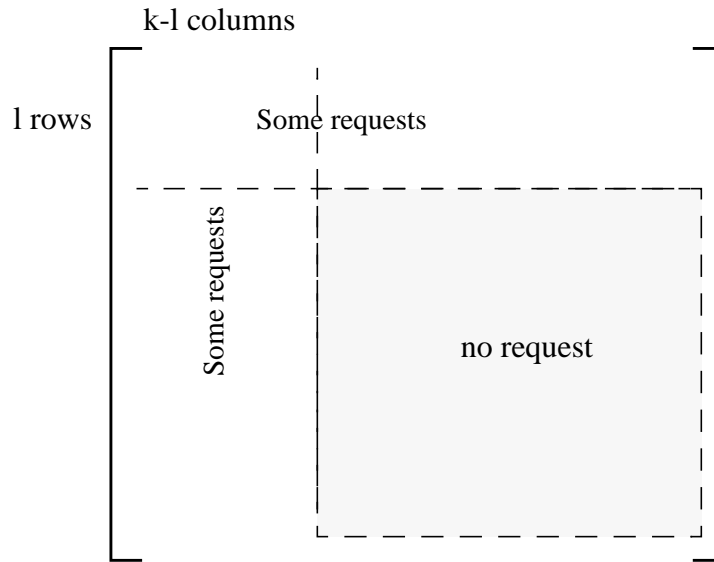


Figure 17: A rearranged request matrix. From the original request matrix, row permutation was performed to move all  $l$  rows of the minimum set to the top, and column permutation was performed to move all  $k - l$  columns of the same set to the left. Together these rows and columns contain all requests, leaving the shaded area of the new request matrix with no request.

---

With the traffic admissibility constraint:  $\sum_{i=1}^N \lambda_{i,j} \leq 1, \sum_{j=1}^N \lambda_{i,j} \leq 1, \lambda_{i,j} \geq 0,$

$$\sum_{i,j} \tilde{\lambda}_{i,j} \leq k. \quad (12)$$

Now, consider a linear programming problem:

$$\begin{aligned}
& \max(\underline{L}^T(n)\tilde{\underline{\lambda}}) \\
& \text{s.t. } \sum_{i=1}^N \tilde{\lambda}_{i,j} \leq 1, \sum_{j=1}^N \tilde{\lambda}_{i,j} \leq 1, \tilde{\lambda}_{i,j} \geq 0 \\
& \sum_{i,j} \tilde{\lambda}_{i,j} \leq k.
\end{aligned} \tag{13}$$

Because  $\Lambda$  is a doubly stochastic matrix and forms a convex set with a set of  $S(n)$  that satisfies the following constraint as its set of extreme points [11],

$$\begin{aligned}
& \text{s.t. } \sum_{i=1}^N S_{i,j}(n) = 1, \sum_{j=1}^N S_{i,j}(n) = 1, S_{i,j}(n) = 0, 1 \\
& \sum_{i,j} S_{i,j}(n) = k
\end{aligned} \tag{14}$$

hence,

$$E[\underline{L}^T(n)\tilde{\underline{\lambda}}] \leq \max[\underline{L}^T(n)\underline{S}(n)], \tag{15}$$

$$E[\underline{L}^T(n)\underline{A}(n)|I, J] \leq \max[\underline{L}^T(n)\underline{S}(n)]. \tag{16}$$

Since the above is true for all  $I$  and  $J$  satisfying  $|I| + |J| = k$ , according to Konig-Egervary's theorem, it follows that:

$$E[\underline{L}^T(n)(\underline{A}(n) - \underline{S}^*(n)) | \underline{L}(n), |\underline{S}^*(n)| = k] \leq 0, \tag{17}$$

where  $\underline{S}^*(n)$  is such that  $\underline{L}^T(n)\underline{S}^*(n) = \max[\underline{L}^T(n)\underline{S}(n)]$ . This proves the second case when  $|\underline{S}^*(n)| < N$ .

From both cases, it can be concluded that:

$$E[\underline{L}^T(n)(\underline{A}(n) - \underline{S}^*(n)) | \underline{L}(n)] \leq 0. \square \tag{18}$$

Note: whenever  $S_{i,j}(n)$  cannot be one because there is no request, Fact 4 implies that

$A_{i,j}(n)$  must be zero, and hence,  $\tilde{\lambda}_{i,j} = 0$ .

**Lemma 6:** Under the LPF algorithm,  $E [\underline{L}^T(n) (\underline{A}(n) - \underline{S}^*(n)) | \underline{L}(n)] \leq -2\beta \sum_{i,j} L_{i,j}(n)$   
 $\forall \underline{\lambda} \leq (1 - \beta) \underline{\lambda}_m, 0 < \beta < 1$ , where  $\underline{\lambda}_m$  is any admissible rate vector such that  
 $\sum_{i,j} \lambda_{m_{i,j}} = N$ .

**Proof:** Following the same steps as done in the proof of Lemma 5 and using the fact that  $T\underline{\lambda}_m = \underline{2}$ , it is can be shown that for every  $|\underline{S}^*(n)| = k$ :

$$E [\underline{L}^T(n) (\underline{A}(n) - \underline{S}^*(n)) | \underline{L}(n), |\underline{S}^*(n)| = k] \leq -2\beta \sum_{i,j} L_{i,j}(n). \quad \square \quad (19)$$

**Lemma 7:** Under the LPF algorithm,

$E [\underline{L}^T(n+1) T_{lpf} \underline{L}(n+1) - \underline{L}^T(n) T_{lpf} \underline{L}(n) | \underline{L}(n)] \leq -\varepsilon \sum_{i,j} L_{i,j}(n) + N^2$   
 $\forall \underline{\lambda} \leq (1 - \beta) \underline{\lambda}_m, 0 < \beta < 1$ , where  $\underline{\lambda}_m$  is any rate vector such that  $\sum_{i,j} \lambda_{m_{i,j}} = N$  and  $\varepsilon > 0$ .

**Proof:** Since the LPF algorithm always finds a maximum weight match,  $\underline{S}^*(n)$ , using equation 3, we obtain the following equality by expansion.

$$\begin{aligned} & \underline{L}^T(n+1) T_{lpf} \underline{L}(n+1) - \underline{L}^T(n) T_{lpf} \underline{L}(n) \\ &= 2\underline{L}^T(n) T_{lpf} (\underline{A}(n) - \underline{S}^*(n)) + (\underline{S}^*(n) - \underline{A}(n))^T T_{lpf} (\underline{S}^*(n) - \underline{A}(n)). \end{aligned} \quad (20)$$

Since  $|\underline{S}^*(n) - \underline{A}(n)| \leq N$ ,  $(\underline{S}^*(n) - \underline{A}(n))^T T_{lpf} (\underline{S}^*(n) - \underline{A}(n)) \leq 2N^2$ . Using Lemma 6,

$$\begin{aligned} & E [\underline{L}^T(n+1) T_{lpf} \underline{L}(n+1) - \underline{L}^T(n) T_{lpf} \underline{L}(n) | \underline{L}(n)] \leq -\varepsilon \sum_{i,j} L_{i,j}(n) + 2N^2, \text{ where} \\ & \varepsilon = 4\beta. \quad \square \end{aligned} \quad (21)$$

Now, we are ready to prove the main theorem.

**Proof:**

There exists a *quadratic Lyapunov function*,  $V(\underline{L}(n)) = \underline{L}(n) T_{lpf} \underline{L}(n)$ , such that:

$$E [V(\underline{L}(n+1)) - V(\underline{L}(n)) | \underline{L}(n)] \leq -\varepsilon \sum_{i,j} L_{i,j}(n) + 2N^2. \quad (22)$$

According to Kumar [8], the sum of all queue occupancies is stable-in-the-mean, i.e.,

$$\frac{1}{N+1} \sum_{n=0}^N \sum_{i,j} E [L_{i,j}(n)] < \infty, \forall N. \quad (23)$$

Furthermore, if the arrivals are independent, the queue occupancy vector forms a Markov chain, which equation 23 guarantees to be positive recurrence. Hence,

$$E [\|\underline{L}(n)\|] \leq C < \infty. \quad (24)$$

## 6 Stability with the Pipeline Delay

### 6.1 Main Theorem

**Theorem 4:** *Using  $k$  slot old weights, the LPF algorithm is stable for all admissible independent arrival processes,  $0 < k < \infty$ , i.e.,  $E [\|\underline{L}(n)\|] \leq C < \infty$*

### 6.2 Proof

The proof requires the subsequent lemma and is given after the proof of the lemma. The following lemma states the existence of an upper bound on the difference between the total weight of the optimum match found by non-pipelined LPF and the total weight of the actual match found by pipelined LPF.

**Lemma 8:**  $\underline{L}'(n)\underline{S}^*(n) - \underline{L}'(n)\tilde{\underline{S}}(n) \leq (N^2 + 3N)k$  where  $\underline{S}^*(n)$  is the optimum service vector if LPF had been given the correct weights, and  $\tilde{\underline{S}}(n)$  is the actual service vector which optimizes on the incorrect weight.

**Proof:** Let  $\tilde{\underline{L}}'(n)$  be the incorrect weight vector given to LPF as a result of the pipeline delay in the sorters, and by definition:

$$\tilde{\underline{L}}'(n) = \underline{L}'(n-k). \quad (25)$$

Furthermore, we can establish an upper bound and a lower bound of  $\tilde{L}'(n)$  with respect to the correct vector as follows. Consider the fact that there can be up to  $k$  arrivals to any input and  $Nk$  arrivals for any output, and that there can be no departure from any input or for any output during a period of  $k$  slots. This fact implies that a correct LPF weight can increase by at most  $(N + 1)k$  from its previous  $k$  slot value:

$$\underline{L}'(n) \leq \underline{L}'(n - k) + \underline{(N + 1)k}. \quad (26)$$

From the above equation and equation 25, a lower bound of  $\tilde{L}'(n)$  is:

$$\underline{L}'(n) - \underline{(N + 1)k} \leq \tilde{L}'(n). \quad (27)$$

For an upper bound, consider the case when there is no arrival but one departure from every input and to every output during every slot in a period of  $k$  slots. An upper bound of  $\tilde{L}'(n)$  with respect to  $\underline{L}'(n)$  is therefore:

$$\underline{L}'(n) \geq \underline{L}'(n - k) - \underline{2k}, \quad (28)$$

$$\tilde{L}'(n) \leq \underline{L}'(n) + 2k. \quad (29)$$

With the two bounds, we are now ready to derive an upper bound on the difference between the two total weights. For the following derivation, it is worth noting that  $\underline{*}(n)$  maximizes the total weight on  $\underline{L}'(n)$  while  $\tilde{\underline{S}}(n)$  maximizes the total weight on  $\tilde{L}'(n)$ , and that the numbers of connections made by  $\underline{*}(n)$  and  $\tilde{\underline{S}}(n)$  are equal, i.e.,  $|\tilde{\underline{S}}(n)| = |\underline{\mathcal{S}}^*(n)|$ .

Using equation 29, consider  $\tilde{\underline{S}}(n)$ .

$$\underline{L}'(n)\tilde{\underline{S}}(n) \geq \tilde{L}'(n)\tilde{\underline{S}}(n) - \underline{2k}\tilde{\underline{S}}(n). \quad (30)$$

But,

$$\underline{2k}\tilde{\underline{S}}(n) \leq 2kN. \quad (31)$$

Hence,

$$L'(n)\tilde{\underline{S}}(n) \geq \tilde{L}'(n)\tilde{\underline{S}}(n) - 2kN. \quad (32)$$

Now consider  $\underline{S}^*(n)$  using equation 27.

$$L'(n)\underline{S}^*(n) - \underline{(N+1)k\underline{S}^*(n)} \leq \tilde{L}'(n)\underline{S}^*(n). \quad (33)$$

Since  $\underline{(N+1)k\underline{S}^*(n)} \leq (N+1)kN$ ,

$$L'(n)\underline{S}^*(n) - (N+1)kN \leq \tilde{L}'(n)\underline{S}^*(n). \quad (34)$$

Furthermore,  $\tilde{L}'(n)\underline{S}^*(n) \leq \tilde{L}'(n)\tilde{\underline{S}}(n)$  because  $\tilde{\underline{S}}(n)$  is the optimum match with respect to  $\tilde{L}'(n)$ . Thus,

$$L'(n)\underline{S}^*(n) - (N+1)kN \leq \tilde{L}'(n)\tilde{\underline{S}}(n). \quad (35)$$

Substituting equation 35 back into equation 32 yields:

$$L'(n)\tilde{\underline{S}}(n) \geq L'(n)\underline{S}^*(n) - (N+1)kN - 2kN. \quad (36)$$

Finally, an upper bound on the difference between the two total weights is:

$$L'(n)\underline{S}^*(n) - L'(n)\tilde{\underline{S}}(n) \leq \left( N^2 + 3N \right) k. \quad \square \quad (37)$$

With the lemma proved, we can now prove the main theorem.

**Proof:**

Similarly to the proof of Theorem 3, by taking the same steps as done in Lemma 5, Lemma 6 and Lemma 7 and by using the relationship stated by Lemma 8, we can show that there exists a *quadratic Lyapunov function*,  $V(\underline{L}(n)) = \underline{L}(n)T_{lpf}\underline{L}(n)$ , such that:

$$E [ V(\underline{L}(n+1)) - V(\underline{L}(n)) | \underline{L}(n) ] \leq -\epsilon \sum_{i,j} L_{i,j}(n) + 2N^2 + \left( N^2 + 3N \right) k. \quad (38)$$

And likewise, according to Kumar [8], the sum of all queue occupancies is stable-in-the-mean, i.e.,

$$\frac{1}{N+1} \sum_{n=0}^N \sum_{i,j} E [L_{i,j}(n)] < \infty, \forall N. \quad (39)$$

Furthermore, if the arrivals are independent, the queue occupancy vector forms a Markov chain, which equation 39 guarantees to be positive recurrence. Hence,

$$E [\|L(n)\|] \leq C < \infty. \quad (40)$$