Linear Complexity Algorithms for Maximum Advance Deflection Routing in Some Networks

Saad Mneimneh* and Franck Quessette†

* saad@alum.mit.edu, Visiting Professor, Hunter College of CUNY, New York, NY 10021, USA

† PRiSM, Université De Versailles, Versailles Cedex, France

***Abstract*— We consider routing in a network with no buffers at intermediate nodes: packets must move in a synchronized manner in every time step until they reach their destinations. If contention prevents a packet from advancing, i.e. taking an outgoing link on a shortest path from its current node to its destination, it is deflected on a different link, hence the name deflection routing. One common strategy in the design of deflection routing algorithms is *maximum advance*, which advances a maximum number of packets at every node in every time step. We examine two settings: non capacitated networks and capacitated networks. We present linear complexity algorithms for maximum advance deflection routing in networks with topological properties as follows: When the network is non capacitated, we require that each packet can advance on at most two links from any intermediate node in the network. When the network is capacitated, we require a special condition on the links in addition to the one mentioned above. Metropolitan and wide area networks typically satisfy those conditions.**

***Index Terms*— Maximum advance, deflection routing, maximum matching, bipartite graphs.**

I. Introduction

DEFLECTION routing is an example of the family of routing algorithms that make no use of buffers at intermediate nodes, and hence provide an attractive solution for routing in bufferless (e.g. all-optical) networks. For some literature on deflection routing algorithms see [2], [3], [4], [1], and [6].

We assume that the time proceeds in discrete time steps synchronized throughout the network, and that a packet traverses one link per time step. If the current node is the final destination of the packet, the packet is absorbed and hence disappears from the network; otherwise, the packet is forwarded on an outgoing link (no buffering). However, if contention prevents a packet from advancing, i.e. taking an outgoing link on a shortest path from its current node to its destination, it is deflected on a different link, hence the name deflection routing. A common contention resolution rule applied locally at every node in every time step is known in the literature as *maximum advance*.

Maximum Advance deflection: a maximum number of packets must advance at every node in every time step.

Therefore, a maximum advance algorithm does not deflect packets unless the deflection cannot be avoided, and hence is expected to make good progress over time.

We assume that links are bidirectional, and that each link can carry per time step in each direction at most a number of packets equal to its capacity. In a non capacitated network, each link has a unit capacity. In a capacitated network, the capacities of all links are given (we assume here that they are integers) and need not be equal. The number of packets at a node at any given time step is at most equal to the sum of link capacities at that node [1]. Since links are bidirectional, the network capacity is sufficient for these packets to leave the node in the next time step [2].

We will denote by the advance set at a node in a given time step, the set of packet advances made by that node in that time step. More formally, we can represent the advance set using the following abstraction: For every node at a given time step, we construct a bipartite graph $G = (V_P, V_L, E)$ such that there is a vertex in V_P for every packet that is not destined to that particular node, there is a vertex in V_L for every outgoing link that can advance packets in V_P, i.e. that is on a shortest path to the destination of some packets in V_P, and finally there is an edge (p, l) in E where $p \in V_P$ and $l \in V_L$ (thus a bipartite graph) if packet p can advance on link l, i.e. if l is on a shortest path to p's destination. The bipartite graph G, therefore, represents the preferences of packets to links [3]. Consequently, an advance set at a node with a bipartite graph $G = (V_P, V_L, E)$ can be represented as a set $A \subseteq E$ where an edge $(p, l) \in A$ implies that packet p will advance on link l. For a non capacitated network, the advance set A is a matching. For a capacitated network, the advance set A satisfies that every vertex in V_P belongs to at most one edge in A and every vertex $l \in V_L$ belongs to at most $c(l)$ edges in A, where $c(l)$ is the capacity (an integer) of link l.

Despite the fact that maximum advance algorithms make good progress over time, they are difficult to implement [4], as they require finding at every node a maximum advance set (an advance set of maximum size). For a non capacitated network, this is the same as finding a maximum matching in the bipartite graph $G = (V_P, V_L, E)$, which can be solved in $O(|E|\sqrt{|V_P| + |V_L|})$ time in general [9]. For a capacitated network, the problem can be transformed into a matching problem by duplicating a node $l \in V_L$ $c(l)$ times. However, this makes the problem size depend on the capacities. Alternatively, one could model the problem as a maximum flow problem which

[1] Packets can reach the node only through those links and a new packet is not injected into the network at the node unless there is enough capacity to move that packet.

[2] The assumption of bidirectional links can be replaced by the more general assumption that at every node the total capacity of the inward links is equal to the total capacity of the outward links. For instance, in a Manhattan street network [8], every node has two inward and two outward links of unit capacity each.

[3] An edge in the bipartite graph $G = (V_P, V_L, E)$ is not to be confused with a link in the original network.

[4] Compare this strategy to other advance strategies like, for instance, advancing at least one packet, which is known in the literature as *minimum advance* [6].

0-7803-9569-7/06/$20.00

has a higher time complexity than the maximum matching problem (see [9] for a discussion on flows and matchings).

Note that a time of $O(|V_P| + |V_L| + |E|)$ is needed for constructing the bipartite graph. Note also that performing maximum advance deflection routing does not stop at the point of finding a maximum advance set, because deflected packets need to be routed as well, even if they will be routed arbitrarily on the remaining available links. For this reason, the algorithm needs to find an available link for each deflected packet. This operation can be done in a time at most linear in the number of outgoing links at the node.

In the rest of this paper, we only concentrate on finding a maximum advance set. Our goal is to examine some topological properties of the network that will make it possible to obtain algorithms that compute a maximum advance set in a time linear in the number of packets, i.e. in $O(|V_P|)$ time. Our topological properties will be as follows: When the network is non capacitated, we require that each packet can advance on at most two links from any intermediate node in the network. When the network is capacitated, we require a special condition on the links in addition to the one mentioned above. Details about these conditions and the intuition behind them will be explained throughout the paper. The rest of the paper is organized as follows: Section II discusses the case of non capacitated networks. Section III discusses the case of capacitated networks. Finally, we conclude in Section IV.

II. Non Capacitated Networks

For a non capacitated network, we will impose the following condition on the network.

Condition I: Let t be any node in the network. For every node s, there exist at most two outgoing links that are on a shortest path from s to t.

Therefore, each packet can advance on at most two links. This is true for instance in a 2-dimentional square grid network, in a $m \times n$ HR^4 network [10] (a toroidal network) where both m and n are odd, and in a Manhattan street network [8] (a toroidal network where each node has two inward and two outward links, see footnote 2). All these metropolitan network topologies provide a motivation for Condition I. In a general network, however, this condition does not necessarily hold. Nevertheless, if we look at other practical networks, especially wide area networks, we find that this condition is not a very restrictive one. For instance, Figure 1 illustrates examples of commercial backbone networks in the USA where Condition I is satisfied.

A direct consequence of Condition I is its impact on the bipartite graph $G = (V_P, V_L, E)$ obtained at every node. Each vertex in V_P will have a degree of at most two, because every packet in V_P can advance on at most two links in V_L. Therefore, $|V_L| \leq 2|V_P|$ and $|E| \leq 2|V_P|$, and hence both $|V_L|$ and $|E|$ are $O(|V_P|)$, which will make it possible to obtain an $O(|V_P|)$ time algorithm for computing a maximum advance set. Note that a general maximum matching algorithm will have a running time of $O(|V_P|^{1.5})$ under these conditions.

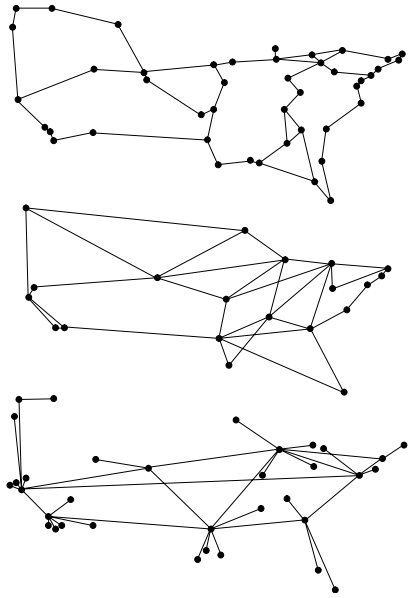

Fig. 1. Example backbone networks from www.caida.org

We formally capture the impact of Condition I on our bipartite graph $G = (V_P, V_L, E)$ in the following definition.

Definition 1: A bipartite graph $G = (V_P, V_L, E)$ is called a BG_2 graph iff for every $p \in V_P$, $deg(p) \leq 2$.

Based on the representation described in Section I of an advance set, our problem of finding a maximum advance set at a node in a non capacitated network satisfying Condition I will be equivalent to computing a maximum matching in a BG_2 graph.

A. The Algorithm

The following algorithm, labeled *Algorithm I* in Figure 2, computes a matching A in a graph with an edge set E.

Algorithm I

$A = \emptyset$
while $E \neq \emptyset$
  do if $\exists (p, l) \in E$ with $deg(l) = 1$
    then $A \leftarrow A \cup \{(p, l)\}$
    else pick any $(p, l) \in E$
      $A \leftarrow A \cup \{(p, l)\}$
    remove all edges in E that are
    incident to vertices p or l

Fig. 2. *Algorithm I*

Figure 2 gives a high level description of the algorithm which by no means is to be taken as the actual implementation. However, this description will provide a clearer and more intuitive understanding of the algorithm. We will discuss the implementation issues and the running time of *Algorithm I* at the end of section II. For now, it is enough to note that *Algorithm I* can be implemented to have a running time of $O(|V_P|)$ if the graph with the edge set E is a BG_2 graph.

Algorithm I picks a vertex $l \in V_L$ with degree 1 and adds the edge (p, l) to the matching and removes all edges incident to p or l. If no such vertex is found, the algorithm adds an arbitrary edge to the matching and updates the graph in the same way. The algorithm stops when no more edges can be added to the matching.

In terms of our routing problem, *Algorithm I* picks a link l that advances one packet only, say p, and advances p on l. Intuitively, this is a good choice since no other packets can advance on l. Then the algorithm updates the preferences by disregarding link l (no more packets can advance on l) and packet p (p will advance on link l). If no such link l is found, *Algorithm I* performs an arbitrary advance and updates the preferences in the same way. When no more advances can be done, the algorithm stops.

A similar but more restrictive algorithm appears in [10]. The algorithm of [10] works by repeatedly advancing a packet p on a link l with minimum $deg(l)$ and updating the graph accordingly. Therefore, the algorithm of [10] is a special implementation of *Algorithm I*.

Next we formally prove that *Algorithm I* computes a maximum matching in a BG_2 graph. Therefore, when $G = (V_P, V_L, E)$ is the bipartite graph obtained at a node in a network satisfying Condition I, the matching A computed by *Algorithm I* will also represent a maximum advance set.

B. Correctness of the Algorithm

As argued before, to prove that *Algorithm I* computes a maximum advance set at a node in a network satisfying Condition I, it is enough to show that it computes a maximum matching in a BG_2 graph. We start with the following simple lemma.

Lemma 1: If a graph G contains a vertex l with degree 1, then there exists a maximum matching that contains edge (p, l) which connects l in G.

Proof: Consider a maximum matching A that does not contain edge (p, l). Vertex l is not matched in A since otherwise $(p, l) \in A$ (vertex l has degree 1). Vertex p must be matched in A, otherwise A cannot be a maximum matching since $A \cup (p, l)$ is also a matching. Unmatching vertex p and adding edge (p, l) to A will result in a matching A' with the same size as A. Therefore, A' is a maximum matching that contains edge (p, l). ■

Note that Lemma 1 is true for any graph G, not necessarily a BG_2 or even a bipartite graph.

Lemma 2: Given a BG_2 graph $G = (V_P, V_L, E)$ where every vertex $l \in V_L$ has $deg(l) \geq 2$, let (p, l) be any edge in E. Then there exists a maximum matching A that contains edge (p, l).

Proof: Let A be a maximum matching that does not contain edge (p, l). If either vertex p or vertex l is not matched in A, then we can obtain a maximum matching that contains (p, l) as in the proof of Lemma 1.

So assume both vertices p and l are matched in A. Let p be matched to l_0 and l be matched to p_0. Consider an alternating path $p_0, l, p, l_0, \ldots$. Following this alternating path, we obtain a matching A' by removing edge (p_0, l) from A, adding edge (l, p) to A, removing edge (p, l_0) from A, etc...

A vertex in $l \in V_L$ is reached at most once from V_P on the alternating path, since vertices in V_L are reachable only through edges that are in the matching A. Since every vertex $l \in V_L$ has $deg(l) \geq 2$, when a vertex $l \in V_L$ is reached on the alternating path, an edge (l, p) for some $p \in V_p$ exists, and hence the alternating path continues along that edge (while adding (l, p) to A).

Consequently, a vertex $p \in V_P$ is reached at most once from V_L on the alternating path because $deg(p) \leq 2$ for all $p \in V_p$ (BG_2 graph). If that vertex is matched in A and different from p_0, the alternating path continues along an edge (p, l) in A (while removing (p, l) from A). Otherwise, it stops.

Since the graph is finite, the alternating path stops at a vertex in V_P that is not matched (possibly p_0). Therefore, we obtain a matching A' with the same size as A. Therefore, A' is a maximum matching that contains edge (p, l). ■

Using Lemma 1 and Lemma 2, we can now prove the following result.

Theorem 1: Algorithm I computes a maximum advance set at a node in a non capacitated network satisfying Condition I.

Proof: The bipartite graph $G = (V_P, V_L, E)$ constructed at the node is a BG_2 graph because of Condition I. Since the network is non capacitated, a maximum advance set corresponds to a maximum matching in the BG_2 graph G. By Lemma 1 and Lemma 2, *Algorithm I* always picks an edge in the BG_2 graph G that can be part of a maximum matching. Therefore, when *Algorithm I* stops, the computed matching is a maximum matching. ■

We proved that *Algorithm I* computes a maximum advance set at a node in a non capacitated network satisfying Condition I. We can further quantify this maximum, as stated in the following lemma in terms of the matching size.

Lemma 3: For a BG_2 graph with k connected components $G_1 = (V_{P1}, V_{L1}, E_1)$, ..., $G_k = (V_{Pk}, V_{Lk}, E_k)$, the size of the maximum matching is

$$\sum_{i=1}^{k} \min(|V_{Pi}|, |V_{Li}|)$$

Proof: Consider one connected component G_i. We will prove that the size of a vertex cover for G_i is at least $\min(|V_{Pi}|, |V_{Li}|)$. This will imply that the minimum size vertex cover is at least $\min(|V_{Pi}|, |V_{Li}|)$. By the König/Egerváry theorem, for a bipartite graph, the size of the maximum matching is equal to the size of the minimum vertex cover. But the maximum matching in G_i cannot be greater than $\min(|V_{Pi}|, |V_{Li}|)$, so it is exactly equal to $\min(|V_{Pi}|, |V_{Li}|)$. Summing over all connected components, we obtain the result of the lemma.

Note that G_i is a connected BG_2 graph. Consider a vertex cover C for G_i. Let S_1 be the set of all vertices in V_{Li} that are

in C (see Figure 3). If S_1 is empty, then C must contain all vertices in V_{Pi} ($|C| = |V_{Pi}|$) and we are done. So assume that S_1 is not empty.

Let S_2 be the set of vertices in V_{Pi} that are connected to $l_1, l_2 \in V_{Li}$ such that $l_1 \in S_1$ and $l_2 \notin S_1$. If S_2 is empty, then $|V_{Li}| = |S_1|$ (G_i is connected) and we are done.

Let S_3 be the set of vertices in V_{Li} that are not in S_1 but connected to a vertex $p \in S_2$. Since every vertex $p \in V_{Pi}$ has $deg(p) \leq 2$ (BG_2 graph), $|S_2| = |S_3|$. Since non of the vertices in S_3 are in C by definition of S_1, all vertices in S_2 must be in C.

Let S_4 be the set of vertices in V_{Li} that are not in S_1 nor in S_3. Without loss of generality, assume S_4 is not empty. We know that vertices in S_4 are not in C by definition of S_1. Since G_i is connected, the only way for a vertex in S_4 to be connected to the rest of the graph is by an edge going to a vertex p in V_{Pi} that is in C and that is in turn connected to S_3. Let the set of those vertices be S_5. Since every vertex in $p \in V_{Pi}$ has $deg(p) \leq 2$ (BG_2 graph), $|S_5| = |S_4|$. As a result $|C| \geq |S_1| + |S_2| + |S_5| = |S_1| + |S_3| + |S_4| = |V_{Li}|$ and we are done. ■

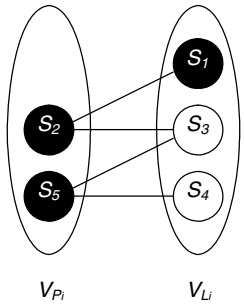

Fig. 3. A vertex cover for G_i

C. Time Complexity of the Algorithm

Algorithm I can be implemented using the adjacency list representation of the graph [5], where $v \in Adj[u]$ iff $(u, v) \in E$. Edges will not be removed; however, only the degrees of vertices in V_P and V_L will be updated, thus simulating edge removals. An edge will be considered removed when at least one of its vertices has a non positive degree.

We maintain a queue of vertices in V_L. Originally, the queue contains all vertices in V_L with degree 1. Each time we dequeue a vertex l with positive degree, we look for an existing edge (p, l). This can be done by looking for a positive degree for vertices in $Adj[l]$. When such a vertex p is found, (p, l) will be added to the advance set, the degrees of both p and l will be set to zero (signifying the removal of all edges incident to p or l), and finally, the degrees of all $p' \in Adj[l]$ and all $l' \in Adj[p]$ will be updated (decremented by 1). This of course might result in new vertices with degree 1, which will be added to the queue.

If the queue becomes empty (when no more edges (p, l) with $deg(l) = 1$ exist), then a single pass over all vertices in V_L is performed, and if a vertex with positive degree is found, it is enqueued, thus activating the queue again.

A detailed implementation of *Algorithm I* is found in Appendix A. Initializing the degrees of vertices takes $O(|V_P| + |V_L| + |E|)$ time. Initializing the queue can be done in $O(|V_L|)$ time. The total time spent on looking for edges in FIND_EDGE after a vertex is dequeued is $O(|E|)$. The total time spent on updating degrees in ADVANCE is $O(|2E|)$ (updates on both sides). The final pass over vertices in V_L takes $O(|V_L|)$ time. All queue operations take $O(1)$. Therefore, the total running time of *Algorithm I* is $O(|V_P|)$ in a BG_2 graph.

Theorem 2: The time complexity of *Algorithm I* on a BG_2 graph is $O(|V_P|)$ in the RAM model.

III. Capacitated Networks

In this section, we address a more general setting for maximum advance deflection routing, namely a capacitated network. In a capacitated network, each link l has a capacity $c(l)$ (assumed here to be an integer), and hence can carry up to $c(l)$ packets in one time step. The maximum advance set no more corresponds to a matching in our bipartite graph $G = (V_P, V_L, E)$ obtained at every node. Therefore, algorithms for computing a maximum matching in a bipartite graph will not work. One has to revert to a more general maximum flow algorithm for which the time complexity is higher than $O(|E|\sqrt{|V_P| + |V_L|})$ [9].

Motivated by the same set of metropolitan networks mentioned previously, and the network topologies of Figure 1, we further strengthen Condition I with the intention to obtain a simple $O(|V_P|)$ time algorithm for computing a maximum advance set at every node in the capacitated network.

Therefore, we will assume that our capacitated network satisfies a Condition II, which will be a stronger version of Condition I.

Definition 2: For a node s in the network, two outgoing links are adjacent iff there exists a node t such that both links are on shortest paths from s to t.

Condition II: Condition I is satisfied, and given a node s, every outgoing link has at most two adjacent links.

The impact of Condition II on the bipartite graph $G = (V_P, V_L, E)$ obtained at every node is that G is a BG_2 graph, since Condition II $\Rightarrow$ Condition I, and G satisfies that every vertex in V_L can reach at most two other vertices in V_L with paths of length 2. We formally capture the impact of Condition II on our bipartite graph $G = (V_P, V_L, E)$ in the following definition.

Definition 3: A bipartite graph $G = (V_P, V_L, E)$ is called an *adjacent* BG_2 graph iff it is a BG_2 graph and for every $l \in V_L$, l can reach at most two other vertices in V_L with paths of length 2.

As a result of the definition above, a connected component in an adjacent BG_2 graph can be represented as one of the two graphs in Figure 4, where S_i is the set of packets that can advance on link l_i only, and $T_{i,j}$ is the set of packets that can

advance on both links l_i and l_j. Note that in the graphs of Figure 4, each vertex in V_L (represented as a black dot), can reach at most two other vertices in V_L with paths of length 2.

Given the special representation in Figure 4 of a connected component in our bipartite graph $G = (V_P, V_L, E)$, we devise an algorithm for computing a maximum advance set in $O(|V_P|)$ time.

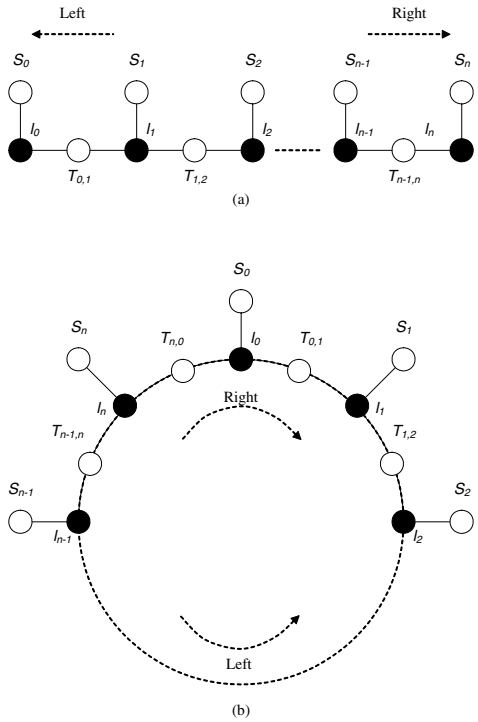

Fig. 4. A connected component in an adjacent BG_2 graph

A. The Algorithm

Consider *Algorithm II* in Figure 5 that operates on an adjacent BG_2 graph. As before, Figure 5 is just a high level description of the algorithm. It is possible to implement *Algorithm II* to have a running time of $O(|V_P|)$ on an adjacent BG_2 graph. We will discuss the implementation issues and running time of *Algorithm II* at the end of Section III.

Algorithm II advances packets in three stages. In each stage, if a packet advances on a link, the capacity of that link is decremented by 1. In the first stage, the algorithm makes advances for packets in the sets S_i on links l_i. In the second stage, the algorithm makes advances for packets in the sets $T_{i,j}$ on links l_j (to the right in Figure 4). In the third stage, the algorithm makes advances for packets in the sets $T_{i,j}$ on links l_i (to the left in Figure 4).

The first and third stages of the algorithm are straight forward because they basically entail advancing as many packets as possible in the sets S_i on links l_i and in the sets $T_{i,j}$ on links l_i, respectively. Therefore, the conditions of the first and third stages do not need to be checked explicitly, and we can immediately advance $min(|S_i|, c(l_i))$ packets in S_i on l_i in the first stage, and $min(|T_{i,j}|, c(l_i))$ packets in $T_{i,j}$ on l_i in the third stage. On the other hand, the condition of the second stage for a given set $T_{i,j}$ may change from true to false and vise-versa. Therefore, determining the conditions in the second stage is the most critical aspect of the algorithm. For a given set $T_{i,j}$, we need to determine if $|T_{i,j}| > c(l_i)$ and $c(l_j) > 0$, in which case, we advance a packet in $T_{i,j}$ on link l_j. If no set $T_{i,j}$ satisfies the condition, we go to the third stage.

Algorithm II

$A \leftarrow \emptyset$
while $\exists i$ such that $S_i \neq \emptyset$ and $c(l_i) > 0$
  do pick any $p \in S_i$
    $S_i \leftarrow S_i - \{p\}$
    $c(l_i) \leftarrow c(l_i) - 1$
    $A \leftarrow A \cup \{(p, l_i)\}$

▷ right advances
while $\exists i, j$ such that $|T_{i,j}| > c(l_i)$ and $c(l_j) > 0$
  do pick any $p \in T_{i,j}$
    $T_{i,j} \leftarrow T_{i,j} - \{p\}$
    $c(l_j) \leftarrow c(l_j) - 1$
    $A \leftarrow A \cup \{(p, l_j)\}$

▷ left advances
while $\exists i, j$ such that $|T_{i,j}| \neq 0$ and $c(l_i) > 0$
  do pick any $p \in T_{i,j}$
    $T_{i,j} \leftarrow T_{i,j} - \{p\}$
    $c(l_i) \leftarrow c(l_i) - 1$
    $A \leftarrow A \cup \{(p, l_i)\}$

Fig. 5. Algorithm II

By symmetry, the *Right* and *Left* conventions of Figure 4 can be reversed. Not only that, but it is also possible for *Algorithm II* to reverse the *Right* and *Left* conventions at any time during its operation, but this is a detail that is not of great importance here. In the next section we prove that *Algorithm II* computes a maximum advance set in an adjacent BG_2 graph.

B. Correctness of the Algorithm

We state and prove a simple lemma similar to Lemma 1.

Lemma 4: In an adjacent BG_2 graph, if $S_i \neq \emptyset$ and $c(l_i) > 0$, then there exists a maximum advance set that advances a packet in S_i on link l_i.

Proof: Consider a maximum advance set A that does not advance a packet in S_i on link l_i. This implies that all the packets in S_i are deflected (they can advance on l_i only). Let p be an arbitrary packet in S_i. We can advance p on l_i and deflect another packet that advances on l_i in A, thus not exceeding the capacity $c(l_i)$. We obtain an advance set A' that has the same size as A. Therefore, A' is a maximum advance set that advances a packet in S_i on link l_i. ■

Lemma 4 justifies the first stage of *Algorithm II*. Note that after the first stage in done, $c(l_i) > 0 \Rightarrow S_i = \emptyset$. Next we prove

another simple lemma which shows that the work done in the second stage does not violate the maximum advance strategy.

Lemma 5: In an adjacent BG_2 graph, if $|T_{i,j}| > c(l_i)$ and $c(l_j) > 0$, then there exists a maximum advance set that advances a packet in $T_{i,j}$ on link l_j.

Proof: Consider a maximum advance set A that does not advance a packet in $T_{i,j}$ on link l_j. Since $|T_{i,j}| > c(l_i)$, there must be a packet $p \in T_{i,j}$ that is deflected. We can advance p on l_j and deflect another packet that advances on l_j in A, thus not exceeding the capacity $c(l_j)$. We obtain an advance set A' that has the same size as A. Therefore, A' is a maximum advance set that advances a packet in $T_{i,j}$ on link l_j. ■

Note that Lemma 4 and Lemma 5 are independent, implying that the first and second stages of *Algorithm II* can be interleaved in any way until both stages are done. Note also that after the second stage is done, $c(l_j) > 0 \Rightarrow |T_{i,j}| \leq c(l_i)$. The next lemma provides a justification for the last stage of the algorithm.

Lemma 6: In an adjacent BG_2 graph, if $\forall\, T_{i,j}$, $c(l_i) > 0 \Rightarrow S_i = \emptyset$ and $c(l_j) > 0 \Rightarrow |T_{i,j}| \leq c(l_i)$, then advancing $\min(c(l_i), |T_{i,j}|)$ packets in $T_{i,j}$ on link l_i, for all $T_{i,j}$, results in a maximum advance set.

Proof: Consider an advance set A that advances $\min(c(l_i), |T_{i,j}|)$ packets in $T_{i,j}$ on link l_i, for all $T_{i,j}$ (this is what *Algorithm II* performs in the third stage). We will prove that A is a maximum advance set. Since $c(l_i) > 0 \Rightarrow S_i = \emptyset$, no packets in the sets S_i can advance. On the other hand, a maximum advance set cannot advance more than $\min(c(l_i), |T_{i,j}|)$ packets in $T_{i,j}$ if $c(l_j) = 0$, and in general, cannot advance more than $|T_{i,j}|$ packets in $T_{i,j}$. For all j such that $c(l_j) = 0$, A advances $\min(c(l_i), |T_{i,j}|)$ packets in $T_{i,j}$ on link l_i. For all j such that $c(l_j) > 0$, and since $|T_{i,j}| \leq c(l_i)$, A advances $\min(c(l_i), |T_{i,j}|) = |T_{i,j}|$ packets in $T_{i,j}$ on link l_i. Therefore, the number of packets that advance in A is equal to the number of packets that advance in a maximum advance set, hence A is a maximum advance set. ■

Using the above three lemmas, we can prove the following result:

Theorem 3: Algorithm II computes a maximum advance set at a node in a capacitated network satisfying Condition II.

Proof: Since the capacitated network satisfies Condition II, we know that the bipartite graph $G = (V_P, V_L, E)$ obtained at a node is an adjacent BG_2 graph. Therefore, Lemma 4, 5, and 6 apply. By Lemma 4 and 5, up to the end of both the first and the second stages, *Algorithm II* makes advances that are part of a maximum advance set. Since after the first and second stages of *Algorithm II* are done, $\forall\, T_{i,j}$, $c(l_i) > 0 \Rightarrow S_i = \emptyset$ and $c(l_j) > 0 \Rightarrow |T_{i,j}| \leq c(l_i)$, then starting from the third stage, *Algorithm II* computes a maximum advance set by Lemma 6. Therefore, *Algorithm II* computes a maximum advance set at every node in a capacitated network satisfying Condition II. ■

C. Time Complexity of the Algorithm

It is possible to implement *Algorithm II* to run in $O(|V_P|)$ time. A pseudocode for the implementation is shown in Appendix B. There are two important parts of the implementation. The first is to construct the representation in Figure 4 for the adjacent BG_2 graph. This can be done in $O(|V_P| + |V_L| + |E|) = O(|V_P|)$ time, since the graph is a BG_2 graph. By appropriate use of pointers, the representation will allow to determine S_i and $T_{i,j}$ for a given l_i in $O(1)$ time, and similarly, to determine l_i given S_i or $T_{i,j}$ in $O(1)$.

The second important part of the implementation involves the second stage, since as argued before, the first and third stages are trivial to implement once we have the appropriate representation of Figure 4. For the second stage we maintain a queue of $T_{i,j}$'s that satisfy the condition $|T_{i,j}| > c(l_i)$ and $c(l_j) > 0$. If the queue is not empty, we dequeue a $T_{i,j}$ and advance packets in $T_{i,j}$ on link l_j until $T_{i,j}$ no more satisfies the above condition. By updating $c(l_j)$, the condition for some $T_{j,k}$ if it exists (it will be unique) may change. So if $T_{j,k}$ now satisfies the condition, we enqueue it if it is not already in the queue. Checking whether $T_{j,k}$ is in the queue or not can be done by maintaining a Boolean for each $T_{i,j}$. Whenever the queue is empty, we go to the third stage.

The work of the first and third stages takes $O(|V_P|)$ time since they both advance a number of packets proportional to their work.

The work of the second stage takes also $O(|V_P|)$ time since every time a $T_{i,j}$ is dequeued, at least one packet advances. This is because once a $T_{i,j}$ is in the queue, the condition $|T_{i,j}| > c(l_i)$ and $c(l_j) > 0$ remains valid until $T_{i,j}$ is dequeued ($c(l_i)$ can only decrease and $c(l_j)$ is constant while $T_{i,j}$ is enqueued). After a $T_{i,j}$ is dequeued, all updates to the queue take $O(1)$ because the condition for at most one other $T_{j,k}$ is affected.

Therefore, the total running time of *Algorithm II* is $O(|V_P|)$ as stated in the following theorem:

Theorem 4: The time complexity of *Algorithm II* on an adjacent BG_2 graph is $O(|V_P|)$ in the RAM model.

IV. Conclusion

We presented two linear time algorithms for maximum advance deflection routing: *Algorithm I* for non capacitated networks satisfying that every packet can advance on at most two links from its current node, and *Algorithm II* for capacitated networks satisfying a special condition on the links in addition to the one mentioned above. Practical networks, like metropolitan and wide area networks, tend to satisfy those conditions. Future work will consider efficient approximation and probabilistic algorithms in the absence of such conditions. Our algorithms can also be used in buffered networks where routing tables can store up to two outgoing links for each destination. The routing will therefore be such that a maximum number of packets advance, and hence a minimum number of packets are buffered, without compromising the linear time complexity of the routing algorithm.

REFERENCES

[1] D. Barth, P. Berthomé, T. Czarchoski, J.M. Fourneau, C. Laforest, and S. Vial, *A mixed deflection and convergence routing algorithm: Design and performance.*

[2] F. Borgonovo, L. Fratta, and J. Bannister, *Unslotted deflection routing in all-optical networks.* IEEE Globecom, 1993, pp. 119-125.

[3] F. Borgonovo, L. Fratta, and J. Bannister, *On the design of optical deflection routing networks.* IEEE Infocom, 1994, pp. 120-129.

[4] T. Chich, J. Cohen, and O. Fraigniaud, *Unslotted deflection routing: A practical and efficient protocol for multihop optical networks.* IEEE/ACM Transactions on Networking, vol. 9, no. 1, February 2001.

[5] T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Introduction to Algorithms.* Second edition, MIT Press.

[6] U. Feige, R. Krauthgamer, *Networks on which hot-potato routing does not block.* Distributed Computing, vol. 13, pp. 53-58, 2000.

[7] D. König, *Graphen und matrices.* Mathematikai és Fizikai Lapok 39, pp. 116 - 119, 1931.

[8] N. F. Maxemchuck, *Routing in the Manhattan street network.* IEEE Transactions on Communications, vol. COM-35, No. 5, May 1987.

[9] R. Tarjan, *Data Structures and Network Algorithms.* Society of Industrial and Applied Mathematics, SIAM, 1983.

[10] J. Wong and Y. Kang, *Distributed and fail-safe routing algorithms in toroidal-based metropolitan area networks.* Journal of Computer Networks and ISDN Systems, vol. 18, pp. 379-391, 1989/90.

APPENDIX A

Algorithm I

FIND_EDGE(l)
for each $p \in Adj[l]$
  do if $deg[p] > 0$
    then return p

ADVANCE(p, l, Q)
for each $p' \in Adj[l]$
  do $deg[p'] \leftarrow deg[p'] - 1$ ▷ no need if $deg[p'] = 0$
for each $l' \in Adj[p]$
  do $deg[l'] \leftarrow deg[l'] - 1$ ▷ no need if $deg[l'] = 0$
    if $deg[l'] = 1$
      then ENQUEUE(Q, l')
$deg[p] \leftarrow 0$
$deg[l] \leftarrow 0$
$A \leftarrow A \cup \{(p, l)\}$

PROCESS_QUEUE(Q)
while $Q \neq \emptyset$
  do $l \leftarrow$ DEQUEUE(Q)
    if $deg[l] > 0$
      ▷ there still exists an edge (p, l) incident to l
      then $p \leftarrow$ FIND_EDGE(l)
        ADVANCE(p, l, Q)

$A \leftarrow \emptyset$
$Q \leftarrow \emptyset$
initialize degrees for all vertices in V_P and V_L
for each $l \in V_L$
  do if $deg[l] = 1$
    then ENQUEUE(Q, l)
PROCESS_QUEUE(Q)
for each $l \in V_L$
  do if $deg[l] > 0$
    then ENQUEUE(Q, l)
      PROCESS_QUEUE(Q)

APPENDIX B

Algorithm II

$A \leftarrow \emptyset$
$Q \leftarrow \emptyset$
construct the representation in Figure 4 of the adjacent BG_2 graph $G = (V_p, V_L, E)$

▷ first stage
for every S_i
  do let $P \subseteq S_i$ be of size $\min(|S_i|, c(l_i))$
    $c(l_i) \leftarrow c(l_i) - \min(|S_i|, c(l_i))$
    $A \leftarrow A \cup [\bigcup_{p \in P}\{(p, l_i)\}]$

▷ second stage
for every $T_{i,j}$
  do if $|T_{i,j}| > c(l_i)$ and $c(l_j) > 0$
    then ENQUEUE($Q, T_{i,j}$)
while $Q \neq \emptyset$
  do $T_{i,j} \leftarrow$ DEQUEUE(Q)
    let $P \subseteq T_{i,j}$ be of size $\min(|T_{i,j}| - c(l_i), c(l_j))$
    $c(l_j) \leftarrow c(l_j) - \min(|T_{i,j}| - c(l_i), c(l_j))$
    $A \leftarrow A \cup [\bigcup_{p \in P}\{(p, l_j)\}]$
    ▷ $T_{j,k}$ is unique if it exists
    if $|T_{j,k}| > c(l_j)$ and $c(l_k) > 0$ and $T_{j,k} \notin Q$
      then ENQUEUE($Q, T_{j,k}$)

▷ third stage
for every $T_{i,j}$
  do let $P \subseteq T_{i,j}$ be of size $\min(|T_{i,j}|, c(l_i))$
    $c(l_i) \leftarrow c(l_i) - \min(|T_{i,j}|, c(l_i))$
    $A \leftarrow A \cup [\bigcup_{p \in P}\{(p, l_i)\}]$