International Journal of Foundations of Computer Science Vol. 12 No. 4 (2001) 411–434

ON THE ROUTING NUMBER OF COMPLETE d-ARY TREES

ALAN ROBERTS
Basser Department of Computer Science, University of Sydney
Sydney, N.S.W. 2006, Australia

and

ANTONIOS SYMVONIS[1]
Basser Department of Computer Science, University of Sydney
Sydney, N.S.W. 2006, Australia

Received 22 October 1997
Revised 7 March 2000
Communicated by Frank Hsu

ABSTRACT

We consider the *routing number* of trees, denoted by $rt()$, with respect to the matching routing model. For an arbitrary n-node tree T, it is known that $rt(T) < 3n/2 + O(\log n)$. In this paper, by providing a recursive off-line permutation routing algorithm, we show that the routing number of an n-node complete d-ary tree of height $h(T) > 1$ is bounded from above by $n + o(n)$. This is near optimal since, for an n-node complete d-ary tree T of height $h(T) > 1$ it holds that $rt(T) \geq n$.

Keywords: Routing, trees, matching model, permutation.

1. Introduction

The *permutation packet routing problem* on a connected undirected graph is the following: We are given a graph $G = (V, E)$ and a permutation π of the vertices of G. Every vertex v of G contains a packet destined for $\pi(v)$. Our task is to route all packets to their destinations.

During the routing, the movement of the packets follows a set of rules. These rules specify the *routing model.* Routing models might differ on the way edges are treated (unidirectional, bidirectional), the number of packets a vertex can receive/transmit in a single step, the number of packets that are allowed to queue in a vertex (queue-size), etc. Usually, routing models are described informally.

Let $rt_M(G, \pi)$ be the number of steps required to route permutation π on graph G by using routing model M. The routing number of graph G with respect to

[1]Currently with Department of Mathematics, University of Ioannina, GR-45110 Ioannina, Greece. The work of Dr. Symvonis was supported by an ARC Institutional Grant.

routing model M, $rt_M(G)$, is defined to be

$$rt_M(G) = \max_{\pi} rt_M(G, \pi)$$

over all permutations π of the vertex set V of G.

The routing number of a graph was first defined by Alon et al. in [2]. In their routing model, the only operation allowed during the routing is the exchange of the packets at the endpoints of an edge of graph G. The exchange of the packets at the endpoints of a set of disjoint edges (a *matching* on G) can occur in a single routing step. During the routing, a packet might enter its destination node, however, it is not *consumed* (that is, removed from the routing) at that time. Instead, the routing terminates at the step where *all* packets are in their destination nodes simultaneously for the first time. We refer to this model as the *matching routing model* and we refer to the routing number of graph G with respect to the matching routing model simply as the *routing number* of G, denoted by $rt(G)$.

The matching routing model was introduced in [2] by Alon et al. Their work included an off-line algorithm for routing permutations on arbitrary trees of n nodes in at most $3n$ routing steps. Roberts, Symvonis, and L. Zhang [13] managed to reduce the required number of routing steps to $2.3n$ and, later on, L. Zhang [17] managed to route any permutation in at most $3n/2+\log n$ routing steps. This result is near optimal since it is known that there exist n-node trees for which there are permutations that require $\lfloor 3(n-1)/2 \rfloor$ routing steps under the matching model [2]. Note that, for an arbitrary n-node graph G it is implied that $rt(G) < 3n/2 + \log n$ since any permutation can be routed along the edges of a spanning tree of G. The matching routing model has been also used by Houle and Turnet in the study of load balancing for mesh connected networks [7]. Algorithms for routing permutations on trees under different routing models have been presented by Borodin, Rabani and Schieber [3] (hot-potato routing model) and Symvonis [15] (simplified routing model).

There has been some recent work on routing patterns different than permutations under an extension of the matching model that allows for the *consumption* of packets when they reach their destination. Krizanc and L. Zhang [9] studied the case where each node of the tree can be the destination of more that one packet and showed that the routing can be completed within $9n$ steps. Independently, Pantziou, Roberts and Symvonis [10, 11] provided an algorithm that handles *dynamic routing*, that is, allows the creation of packets while the routing of other packets takes place. A consequence of their analysis is that the same problem studied by Krizanc and L. Zhang can be solved in at most $3n$ routing steps.

In this paper we study the routing number of complete d-ary trees with respect to the matching routing model. We show that it is easier (that is, faster) to route a permutation on a complete d-ary tree of n-nodes than on an arbitrary tree with the same number of nodes. We present an off-line algorithm which routes any permutation in a complete d-ary tree within $n + o(n)$ steps. For clarity reasons, we have chosen to present our algorithm in the way it evolved through our research. Firstly we describe how to route a permutation on an n-node complete binary tree

in $\frac{4}{3}n + o(n)$ routing steps. Then, we extend the algorithm to route a permutation in an n-node complete d-ary tree in $(1 + \frac{1}{d^2-1})n + o(n)$, and finally we extend the later algorithm to achieve a routing time of $n + o(n)$ steps.

The paper is organised as follows: In Section 2 we present definitions that are used throughout the paper as well as some known results regarding routing and heap construction on rooted trees which are necessary for our analysis. Sections 3 and 4 present non-optimal recursive algorithms for routing on complete binary and complete d-ary trees, respectively. By stretching recursion to its limits, the non-optimal algorithms are extended in Section 5 to yield an algorithm that routes each permutation within $n + o(n)$ steps. We conclude in Section 6.

2. Preliminaries

A *tree* $T = (V, E)$ is an undirected acyclic graph with node set V and edge set E. Throughout the paper we assume n-node trees, i.e., $|V| = n$. We use the notation $V(T)$ and $E(T)$ to denote the node set and edge set of T, respectively. A tree T is *rooted* if one of its nodes, say r, is distinguished as its root. Consider a tree T which is rooted at node r. If node u appears in the simple path from r to node v, then we say that u is an *ancestor* of v or, equivalently, that v is a *descendant* of u. If (u, v) is an edge of the tree and u is an ancestor of v we say the u is the *parent* of v or, equivalently, that v is a *child* of u. Nodes that are children of the same parent are called *siblings*. Nodes with no children are called *leaves*. All non-leaf nodes are referred as *internal nodes*.

The *depth* $d_T(v)$ of node v is defined to be the distance (length of a shortest path) from the root r to v. The *height* of tree T, denoted by $h(T)$, is defined to be $h(T) = max_{v \in V(T)} d_T(v)$. We say that a node v is a *level-i node* (or, at *depth-level i*) if $d_T(v) = i$. The root of the tree is a level-0 node. We say that an edge e is a *level-i edge* if it connects a level-i node with a level-$(i+1)$ node. All edges connected to the root r are level-0 edges. The *lowest common ancestor* of nodes v and u, denoted $lca(v, u)$, is defined to be their common ancestor of largest depth-level.

To comply with the conventional drawing of rooted trees in which the root of a tree (or subtree) is the topmost node of its drawing, and in order to facilitate the description of our algorithms/proofs, we give some additional definitions. We say that a level-i node u is *below* a level-j node v, if v is an ancestor of u (it is also implied that $i \geq j + 1$). Equivalently, we say that node v is *above* node u.

A *subtree rooted at v*, denoted by T_v, consists of v, all descendants of v and the edges between them. A *partial tree* is a connected subgraph of a tree. (Note the difference between a subtree and a partial tree.) By T^i we denote the partial tree of T which is rooted at r and contains all level-j nodes, $0 \leq j \leq i$.

A *d-ary tree*, $d \geq 2$ is defined to be a rooted tree of which all internal nodes have exactly d children. A d-ary tree T is said to be *complete* if all of its leaves are level-$h(T)$ nodes. It is easy to verify that in a complete d-ary tree there are exactly d^i level-i nodes and that the tree is of height $h(T) = \lfloor \log_d n \rfloor$, where n is the number of nodes of the tree. For the purposes of this paper, we will use a special naming convention for the nodes of complete d-tree T. We provide a recursive definition for

this naming convention.

- The root r of the tree is denoted by $r_{(0,1)}$.
- The children of the internal node $r_{(i,j)}$, $0 \leq i < h(T)$, $1 \leq j \leq d^i$ are $\{r_{(i+1,k)} \mid d(j-1)+1 \leq k \leq dj\}$.

In the rest of the paper, when we draw a complete d-ary tree we position node $r_{(i,j)}$ above (higher than) node $r_{(k,l)}$ if $i < k$. We position node $r_{(i,j)}$ to the left of node $r_{(i,k)}$ if $j < k$. Figure 1 shows a complete ternary tree using the introduced naming and layout conventions.

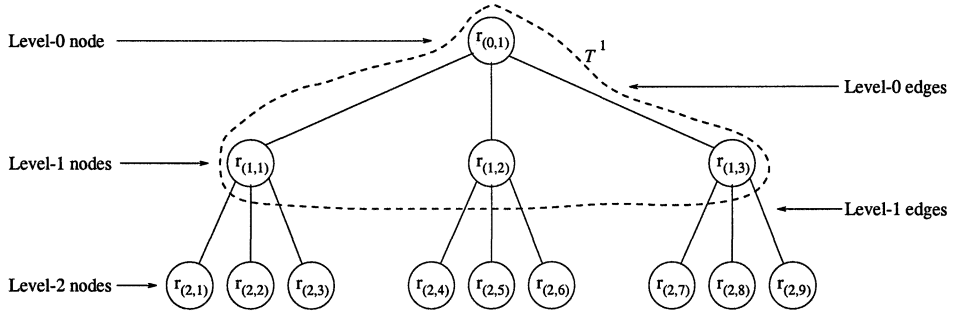

Fig. 1. A complete ternary tree.

Most of the work available on the routing number of trees (including this one) is based on off-line recursive routing algorithms. In order to be able to apply recursion, we must identify subtrees in which all packets destined for nodes in the subtree have arrived in it (and, as a consequence of the fact that we route permutations, no packet wants to leave the subtree). We say that a subtree is a *destination subtree* for a packet if it contains the node the packet is destined for. The following lemma considers the situation in which we want to route packets to their destination subtree.

Lemma 1 *Consider a tree T (rooted at r) and a permutation that has to be routed. Let the number of packets that have to cross the root r in order to reach their destination be m and assume that these packets form partial trees rooted at the children of r. Then these m packets can be routed into their destination subtrees (rooted at children of r) within at most $m+d-1$ steps, where d is the degree of r.*

Proof. An off-line routing algorithm which achieves the stated bound was originally given in [2]. Since we use this fact several times in the paper, we briefly describe how the routing of the packets is performed in order to guarantee the stated performance.

Call the packets that have already reached their destination subtree *proper* and all the other packets *improper*. According to the lemma, the improper packets form partial trees rooted at children of r. Without loss of generality assume that the packet p initially at r is also destined for r and thus, it can be classified as a proper packet (if not, we can save one routing step). The routing consists of at most $d-1$ *cycles*. A cycle starts with the packet p leaving the root r and ends with p returning

to it. We describe the routing actions that occur within such a cycle. We select a subtree rooted at a child of r which still contains improper packets (note that if the routing is not over yet, at least two such subtrees exist). As it is assumed by the lemma, there exists an improper packet, say q_1, at the root of the subtree. We swap q_1 with p and the cycle starts. After the first step, q_1 is at the root r. During the second step, we swap q_1 with the improper packet, say q_2, at the root of the subtree q_1 wants to enter. Note that because we route a permutation q_2 must exist. We proceed in the same manner until the packet p returns to the root. At that step the cycle ends. It is quite easy to see that during every step of a cycle but the first one, one improper packet enters its destination subtree. This, together with the facts that i) initially there are m improper packets and ii) at most $d-1$ cycles can occur, implies that routing terminates after $m+d-1$ steps. Some (obvious) details must still be specified in order to be able to route the packets as described. Firstly, the root of each subtree must be able to provide r with improper packets (provided they exist) at least every second step. This is achieved by routing packets that just entered the subtree "deep" into it, i.e., to a node which has no descendant holding improper packets. Secondly, in order to have at most $d-1$ cycles we must ensure that packet p is the last improper packet that leaves the subtree it is currently in. This is again achieved by routing p deep into the subtree. □

During the course of our off-line routing algorithm we do need to solve the problem of *heap construction.* Consider a rooted tree T and let each of its nodes have a *key-value* associated with it. We say that T is *heap ordered* if each non-leaf node satisfies the *heap invariant*: "the key-value of the node is not larger than the key-values of its children". When the key-value at each node is carried (or associated with) the packet currently in the node, the problem of heap construction is simply to route the packets on the tree in a way that guarantees that at the end of the routing the packets are heap-ordered based on the key-values they carry. Needless to say, we are interested in forming the heap in the smallest number of parallel routing steps when routing is performed according to the matching routing model.

The notion of the *heap* was introduced by Williams [16]. Floyd [5] described how to create a heap efficiently. More details can be found in Knuth's book [8]. Heaps are also discussed in the context of the Parallel Random Access Machine (PRAM) model [4, 12, 18]. Rao and W. Zhang [12] and W. Zhang and Korf [18] described how to construct a heap (implemented as a complete binary tree) of n elements in $2\log_2 n$ steps.

Let the height of the tree T be $h(T)$. It is not difficult to heap-order T within $O((h(T))^2)$ routing steps. This is achieved by establishing the heap property in a bottom-up manner. (If we assume that all subtrees rooted at children of node x are heap ordered then the subtree rooted at x, i.e., T_x, can be heap ordered in exactly $h(T_x)$ routing steps.) However, we can heap-order a tree substantially faster. We describe an algorithm that completes the task in at most $2h(T)$ routing steps. The algorithm for arbitrary rooted trees can be considered to be a generalisation of the odd-even transposition sorting method [1, 6, 8] and is similar to the algorithm proposed in [12, 18].

Algorithm *Odd-Even_Heap_Construction*(T)
/* W.l.o.g., we assume that all key-values associated with the packets are distinct. */

1. Assign label $h(T) - i$ to each level-i edge, $0 \leq i < h(T)$.
2. $t = 1$
3. **While** $t \leq 2h(T)$ **do**
   (a) For any node u with edges connecting to its children labelled congruent to t mod 2, select out of the children of u the child, say v, that contains the packet of the smallest key-value. Order for a comparison between the key-values of the packets at u and v to take place at time t. If the key-value of the packet at v is smaller than the key value of the packet at u, a swap of the packets takes place.
   (b) $t = t + 1$

The similarity of Algorithm *Odd-Even_Heap_Construction*() with the odd-even transposition method should be obvious. At odd (even) steps only odd (even) labelled edges can be *active*. However, because of the restrictions of the matching routing model, out of the potentially active edges that are connected to the same node, only one becomes active.

The proof originally reported in [12] appears to generalise to arbitrary trees, and thus, supports the following theorem. A different analysis of the algorithm based on potential function arguments that supports the same theorem was given by Roberts and Symvonis in [14].

Theorem 1 *Algorithm* Odd-Even_Heap_Construction() *heap-orders any tree T in at most $2h(T)$ steps, where $h(T)$ is the height of the tree T.*

3. Routing on Complete Binary Trees

Consider a complete binary tree T of n nodes. We assume the naming and drawing conventions for complete trees which were described in Section 2 (see Figure 2). Recall that a subtree rooted at node x is denoted by T_x and that by T^i we denote the complete partial tree of T of depth i (rooted at the root of T).

Algorithm *Route_on_Complete_Binary_Tree* /* *RCBT* for short */

1. Assign to every packet destined for a node in T^1 a class-value of 0. The remaining packets are assigned a class-value of 1.
2. Heap-order T with respect to the class-values of its packets.
3. Route the class-0 packets into the subtrees rooted at level-2 nodes such that each such subtree contains at most 1 class-0 packet.

4. Partition the packets into classes. A new class-value is assigned to every packet in the tree as follows:

Current node	Destination	Class-value	
$T_{r_{(1,1)}}$	$T_{r_{(2,3)}} \cup T_{r_{(2,4)}}$	0	
$T_{r_{(1,2)}}$	$T_{r_{(2,1)}} \cup T_{r_{(2,2)}}$	0	
r	$T_{r_{(2,1)}} \cup T_{r_{(2,2)}} \cup T_{r_{(2,3)}} \cup T_{r_{(2,4)}}$	0	
$T_{r_{(2,1)}}$	$T_{r_{(2,2)}}$	1	
$T_{r_{(2,2)}}$	$T_{r_{(2,1)}}$	1	
$T_{r_{(2,3)}}$	$T_{r_{(2,4)}}$	1	
$T_{r_{(2,4)}}$	$T_{r_{(2,3)}}$	1	
T	T^1	2	
$T_{r_{(2,i)}}$	$T_{r_{(2,i)}}$	3	$i = 1, 2, 3, 4$
$r_{(1,1)}$	$T_{r_{(2,1)}} \cup T_{r_{(2,2)}}$	2*	
$r_{(1,2)}$	$T_{r_{(2,3)}} \cup T_{r_{(2,4)}}$	2*	

Class-2* packets (if any) become either class-1 or class-3 packets during the next step of the routing.

5. Heap-order tree T with respect to the class-values of their packets. During the heap construction, update the class of packets as follows:
   - When a class-1 packet reaches the root of T, it immediately becomes a class-0 packet.
   - When a class-2* packet enters its destination subtree (rooted at a level-2 node), it immediately becomes a class-3 packet.
   - When a class-2* packet enters a subtree (rooted at a level-2 node) that does not contain its destination, it immediately becomes a class-1 packet.
   - If at the end of the heap construction a class-2* packet is still at a level-1 node, it becomes a class-1 packet.
6. Route the packets to their destination subtrees (T^1, and $T_{r_{(2,i)}}$, $i = 1, 2, 3, 4$).
7. Recursively route the packets in T^1, and $T_{r_{(2,i)}}$, $i = 1, 2, 3, 4$.

It should be obvious that Algorithm *Route_on_Complete_Binary_Tree* completes the routing correctly. No matter what the distribution of the packets at the end of Step 5 is, Steps 6 and 7 correctly complete the routing. Actually, the first 5 steps are executed in order to ensure that the distribution of the packets after Step 5 allows for the implementation of Step 6 in at most $n + 1$ routing steps.

Some explanations are necessary regarding Steps 4 and 5 and the class updates that happen for some packets. According to the algorithm, if during the heap construction a class-1 packet reaches the root of T, then that packet becomes a class-0 packet. We do this update in order to stop packets from crossing the root to the other side of the tree. The class update does not affect the analysis of the

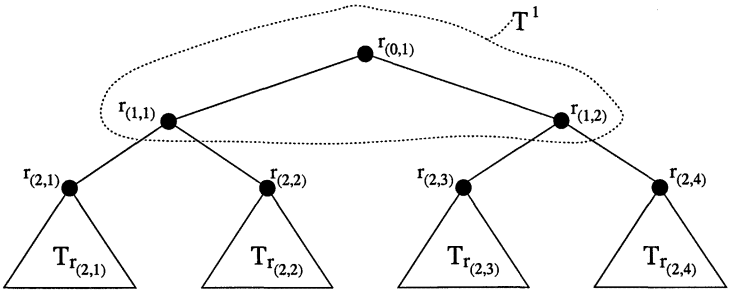

Fig. 2. A complete binary tree.

heap construction algorithm since the packet remains at the root till the end of the heap construction. Another possible class update is that of class-2* packets which become either class-1 or class-3 packets. Recall that a class-2* packet is initially located at a level-1 node and is destined for one of the subtrees rooted at its children. Firstly observe that during the heap-construction at Step 5, it is not possible for a class-2* packet to move to the root of T. If this was not the case, it would require the class-2* packet being swapped with a class-3 packet at the root of T, a situation which cannot occur. Thus, a class-2* packets either moves to a node towards the leaves of the tree or it remains in its current position (at the level-1 nodes of T). In the case where a class-2* packet enters its destination subtree, we update it to a class-3 packet. Note that this update does not cause any problems to the heap construction algorithm, because the packet never moved upwards in the tree. If the class-2* packet enters the subtree which does not contain its destination, it becomes a class-1 packet. Again this class change does not cause any problems to the heap construction algorithm. If during the heap construction a class-2* packet does not move, i.e., it remains at a level-1 node, it is updated to a class-1 packet. This update happens at the end of the heap construction algorithm and certainly does not affect its complexity.

A final comment regarding the first three steps of the algorithm. The purpose of these steps is to distribute the class-2 packets in a balanced way among subtrees $T_{r_{(1,1)}}$ and $T_{r_{(1,2)}}$. These steps are not really necessary. If we omit them, we can still achieve routing on a complete binary tree in $\frac{4}{3}n + o(n)$ with the algorithm of this section but the refinement of Step 6 that follows would be slightly more complicated. This becomes evident in the cases of d-ary trees (examined in the next 2 sections) where the terms hidden in the $o(n)$ notation are also affected. If our final goal was only to examine routing on complete binary trees instead of d-ary trees, the first 3 steps of Algorithm RCBT could be safely left out.

3.1. Refinement of Step 6

During the routing of Step 6, packets that enter their correct subtree continue moving towards the leaves of the subtree. Their movements stops when they reach a node such that all descendent nodes of it hold class-3 packets. This movement of

packets towards the leaves of their destination subtree guarantees that the packets which have to exit the subtree are always close to the root of the subtree.

Also note that during the movement of a packet towards the leaves of the tree no problems occur if the packet is swapped with other packets that want to move upwards. This is because both packets move towards their destination subtree and this movement can be easily accommodated. The problems start when a packet that moved upwards because of a swap with another packet, reaches the node (other than the root) where it has to switch direction and start moving towards the leaves of the tree. In Step 6, this can only happen at the 2 level-1 nodes. This is because we are only interested in routing the packets to their destination subtree, not to their destination node. The reason that we would like the packets to switch the direction of their movement is because we do not want them to cross from one side of the tree to the other. However, this change of direction requires that for consecutive steps level-1 edges adjacent to the same node are active. This has an effect on the flow of packets from $T_{r_{(1,1)}}$ to $T_{r_{(1,2)}}$ and vice versa.

In order to be able to bound the number of routing steps required by Step 6 of the algorithm, we assume for the moment that the flow of packets is uninterrupted. This means that packets which are moving towards the leaves of the tree proceed uninterrupted and as a result of their movement, some packets that have to move upwards during the trip to their destination subtrees do so (later on, these packets reach the root and they start their movement towards the leaves). What we described above does actually happen if we have to route a permutation in which all packets initially in $T_{r_{(1,1)}}$ are destined for $T_{r_{(1,2)}}$ and vice versa. In this case the routing is performed as described in the proof of Lemma 1 and its analysis is quite simple [2]. However, we have to route permutations that do not conform with the above pattern. In these permutations, packets have to change direction at level-1 nodes.

We deal with the problem of packets that want to turn at level-1 nodes by introducing (in the description of the refinement of Step 6) the *"freeze command"* which freezes any routing that is taking place at parts of the tree. During these *"frozen steps"* packets are able to change direction at a level-1 node. This is done by swapping the packet that wants to change direction of movement with the packet at the root of its destination subtree (which is a level-2 node). W.l.o.g, assume that the level-1 node v contains packet p that just arrived from its child u and wants to change direction of movement and enter the subtree T_w which is rooted at the level-2 node w (which is a child of v). During the *"frozen step"* packet p is swapped with the packet at node w while the routing in $T \backslash T_u$ is frozen. The routing continues in T_u because we want to load u with a packet that wants to exit T_u (if it exists). This is necessary if we want to guarantee the uninterrupted flow of the packets in the case where we might need to execute two or more consecutive freeze commands for packets that want to turn at node v.

The time spent on these *freeze* commands is time well spent. Assume that we have an initial estimate on how difficult (in terms of required routing steps) the routing problem at hand is. As we will see, whenever we freeze the routing for one

step, we are able to update our estimate regarding the required number of routing steps. More specifically, for each freeze command we execute, we can save at least one routing step from our initial estimate. Thus, the extra routing steps due to the *freeze* commands does not affect the worst case complexity.

After Step 5 (heap construction), the packets of class-0 form a partial subtree rooted at $r_{(0,1)}$. The routing of Step 6 mainly concentrates on packets of class-0 and moves them to their correct subtree $T_{r_{(2,i)}}$, $i = 1, 2, 3, 4$. We now describe how this is done. W.l.o.g., assume that the packet p at $r_{(0,1)}$ is destined for $T_{r_{(2,1)}}$. All other cases can be handled symmetrically. After the first swap, p finds itself at node $r_{(1,1)}$. Now we have four cases to consider:

1. $r_{(2,1)}$ *contains a class-0 packet.* Say that this packet is q. In this case, p swaps with q and it enters its destination subtree $T_{r_{(2,1)}}$. At this stage, it "becomes" a class-3 packet. (In the following steps, it continues moving towards the bottom (leaves) of the tree until it reaches a node of which all descendants are class-3 packets.) If at the same time that the swap of p and q is taking place, $r_{(0,1)}$ gets a new class-0 packet, q will be swapped with it in subsequent steps and thus, the flow of class-0 packets continues.

2. $r_{(2,1)}$ *contains a class-1 packet.* Let q be the packet at node $r_{(2,1)}$. We distinguish the following cases based on the class of the packet, say q', at node $r_{(2,2)}$.

   (a) $r_{(2,2)}$ *contains a class-0 packet.* We swap p with q and then we issue 1 *freeze* command. During this extra step, q (now at $r_{(1,1)}$) swaps with q'. After this swap, q' is able to move towards its destination subtree (through a subsequent swap with the packet at $r_{(0,1)}$) while q has reached its destination subtree $T_{r_{(2,2)}}$ and starts moving towards its leaves.

   (b) $r_{(2,2)}$ *contains a class-1 packet.* This is an interesting case. Firstly observe that the class-0 packets in $T_{r_{(1,1)}}$ are exhausted. We swap p with q and then we issue 1 *freeze* command. During this extra step, q (now at $r_{(1,1)}$) swaps with q' while p (now at $r_{(2,1)}$) swaps with the packet of smaller class immediately below it (one must exist since we route a permutation). Now, after the first *freeze* command, we have at node $r_{(1,1)}$ a class-1 packet and at node $r_{(2,1)}$ a class-1 or a class-2 packet[a]. In the case where there is a class-2 packet at node $r_{(2,1)}$, we issue a final *freeze* command and during the extra step we swap the packets at nodes $r_{(1,1)}$ and $r_{(2,1)}$. In the case where there is a class-1 packet at node $r_{(2,1)}$, we also execute a *freeze* command and make the same swap but this *freeze* command is not the last one. More specifically, we continue issuing *freeze* commands and making swaps that have as a result to route class-1 packets to their destination subtree. This sequence of *freeze* commands ends when a class-2 packet arrives at node $r_{(1,1)}$. Such a class-2 packet must

[a]This packet arrived at node $r_{(2,1)}$ during the frozen step. This is why it is very important that during the frozen step the routing in $T_{r_{(2,1)}}$ continues uninterrupted.

exist since we assumed that the packet p that triggered the sequence was a class-0 packet.

(c) $r_{(2,2)}$ *contains a class-2 packet.* We swap p with q and then we execute a *freeze* command. During the extra step q (now at $r_{(1,1)}$) swaps with q'. This case can be considered as a special case of 2b where the sequence of class-1 packets consists of only 1 packet.

(d) $r_{(2,2)}$ *contains a class-3 packet.* Impossible (because of the initial sorting and the fact that we are routing a permutation).

3. $r_{(2,1)}$ *contains a class-2 packet.* Let q be the packet at node $r_{(2,1)}$. We distinguish the following cases based on the class of the packet, say q', at node $r_{(2,2)}$.

(a) $r_{(2,2)}$ *contains a class-0 packet.* Swap p with q and then issue a *freeze* command. During the extra step swap q with q'. This keeps q from being trapped into $T_{r_{(1,2)}}$ (there might be many class-0 packets that want to reach $T_{r_{(1,1)}}$). In subsequent steps q moves towards the leaves of $T_{r_{(2,2)}}$ but it always stays above class-2 or class-3 packets in any leaf-to-root path[b].

(b) $r_{(2,2)}$ *contains a class-1 packet.* We simply swap p with q. Note that because of the initial balancing of the class-2 packets, there is no chance that q will be trapped into $T_{r_{(1,2)}}$. No *freeze* command is needed.

(c) $r_{(2,2)}$ *contains a class-2 packet.* As in Case 3b.

(d) $r_{(2,2)}$ *contains a class-3 packet.* As in Case 3b.

4. $r_{(2,1)}$ *contains a class-3 packet.* This is impossible. The fact that the packets were heap-ordered together with the fact that $r_{(2,1)}$ contains a class-3 packet imply that all packets in $T_{r_{(2,1)}}$ have destinations inside it. Then, p must have a destination outside $T_{r_{(2,1)}}$

Notice that when the routing of class-0 packets finishes, we must have at node $r_{(1,1)}$ a class-2 packet. Thus, all packets at the subtrees below it must contain only class-3 packets which in turn implies that the routing of Step 6 is over (for $T_{r_{(1,1)}}$).

Lemma 2 *The routing that occurs during Step 6 of Algorithm RCBT terminates after at most $n+1$ routing steps.*

Proof. Without loss of generality, consider the number of packets that have to cross the root of the tree in order to enter their destination subtree rooted at a level-2 node. These packets are all class-0 packets. Let their number be denoted by m. Moreover, because of the initial sorting, if our only concern was to get them into their correct subtrees rooted at level-1 nodes, we could do it in $m+1$ steps (Lemma 1). However, since our plans are more ambitious and we want to get each

[b]Note that this case can occur at most 1 time. This is because of the balanced distribution of the class-2 packets during the first 3 steps of the algorithm.

packet into its correct destination subtree which is rooted at a level-2 node, we have to employ the *freeze* commands.

Disregard for the moment the freeze commands due to Case 3a. During the *frozen* steps due to all other cases, one class-1 packet moves into its correct subtree. Let the total number of class-1 packets be k. This implies that k *freeze* commands were executed, each requiring 1 routing step. Thus, if we ignore the *freeze* commands due to case 3a, we have that the routing will terminate after $(m+1)+k$ steps. Case 3a can contribute at most 2 extra routing steps. This is because it can occur at most once for each subtree rooted at a level 1 node (a property due to the first three "balancing" steps of the algorithm). Finally we might also need one extra routing step for the last packet to enter its destination subtree (recall that the routing described in Lemma 1 only routes the packets into their destination subtrees that are rooted at level-1 nodes). Thus, the total number of required routing steps is at most $(m+1)+k+2+1 = m+k+4$. Since $m+k \leq n-3$, we conclude that the routing of Step 6 terminates after at most $n+1$ steps. □

3.2. Analysis of Algorithm RCBT

Theorem 2 *Algorithm RCBT routes in an off-line fashion any permutation in an n-node complete binary tree in at most $\frac{4}{3}n + o(n)$ routing steps.*

Proof. Let $T(n)$ denote the number of routing steps required by the algorithm for routing a permutation on a complete binary tree of n nodes. Steps 1 and 4 do not require any routing. From Theorem 1 we know that Step 2 of Algorithm RCBT finishes after at most $2\log n$ routing steps. Step 3 of the algorithm requires at most 3 routing steps. Steps 5 and 6 require at most $2\log n$ and $n+1$ routing steps, respectively. The recursive part of the routing requires $T(\frac{n-3}{4})$ steps since the routing in all 5 trees is done in parallel.

Thus, we can write the following recurrence relation for the required number of routing steps:

$$T(n) \leq T\left(\frac{n-3}{4}\right) + n + 4\log n + 4$$

For the base of our recursion we have that:

$T(1) = 0$ (Only the identical permutation can be defined on an 1-node tree.)

$T(3) = 3$ (A chain of 3 nodes. Use odd-even transposition.)

By solving the recurrence relation we get that $T(n) \leq \frac{4}{3}n + o(n)$. □

4. Routing on Complete d-ary Trees

We use a generalisation of algorithm *Route_on_Complete_Binary_Tree* for routing on complete d-ary trees. However, Step 4 of the algorithm for d-ary trees is described differently. This is done in order to facilitate the description of the asymptotically optimal algorithm of the next section.

Algorithm *Route_on_Complete_d-ary_Tree* /* *RCdT* for short */

1. Assign to every packet destined for a node in T^1 a class-value of 0. The remaining packets are assigned a class-value of 1.
2. Heap-order T with respect to the class-values of its packets.
3. Route the class-0 packets into the subtrees rooted at level-2 nodes such that each subtree rooted at a level-1 node contains at least 1 (and at most 2) class-0 packets.
4. Partition the packets into classes. A new class-value is assigned to packet p, currently at node *curr* and destined for node *dest*, as follows:
   - Let l be the level of the lowest common ancestor of nodes *curr* and *dest*, i.e., $l = d_T(lca(curr, dest))$.
   - **If** *dest* is in T^1 **then** p is a class-2 packet
     else if *curr* is a level-1 node and *dest* is in $T_{curr} \backslash \{curr\}$ **then** p is a class-2* packet
     else if $l > 1$ **then** p is a class-3 packet
     else p is a class-l packet[c].

   Class-2* packets (if any) become either class-1 or class-3 packets during the next step of the routing.
5. Heap-order tree T with respect to the class-values of their packets. During the heap construction, update the class of packets as follows:
   - When a class-1 packet reaches the root of T, it immediately becomes a class-0 packet.
   - When a class-2* packet enters its destination subtree (rooted at a level-2 node), it immediately becomes a class-3 packet.
   - When a class-2* packet enters a subtree (rooted at a level-2 node) that does not contain its destination, it immediately becomes a class-1 packet.
   - If at the end of the heap construction a class-2* packet is still at a level-1 node, it becomes a class-1 packet.
6. Route the packets to their destination subtrees (T^1, and $T_{r_{(2,i)}}$, $1 \leq i \leq d^2$).
7. Recursively route the packets in T^1, and $T_{r_{(2,i)}}$, $1 \leq i \leq d^2$.

Lemma 3 *The routing that occurs during Step 6 of Algorithm RCdT terminates after at most* $n + d^2 - 1$ *routing steps.*

[c]Note that for complete binary trees the class assigned to each packet is identical with the class assigned by algorithm *Route_on_Complete_Binary_Tree*.

Proof. To prove the lemma we have to describe the details of the routing in Step 6. We can distinguish cases as we did for algorithm *RCBT* but this would be a tedious repetition. Thus, we only describe what are the differences between the routing that takes place at Step 6 of algorithm *RCBT* and that of algorithm *RCdT*.

Since in algorithm *RCBT* we were concerned with binary trees, in all sub-cases of parts 2 and 3 of the refinement of Step 6 there was only one sibling of node $T_{r_{(2,1)}}$ to consider. When routing in complete d-ary trees with algorithm *RCdT*, all sub-cases must be updated to account for the fact that each level-2 node has exactly $d-1$ siblings. So, in the refinement of Step 6 of algorithm *RCdT* case 2(a) is changed to: *"RBdT* : 6.2(a) *There exists a sibling node of* $T_{r_{(2,1)}}$ *which contains a class-0 packet"*, case 2(b) is changed to: *"RBdT* : 6.2(b) *There exists a sibling node of* $T_{r_{(2,1)}}$ *which contains a class-1 packet"*, and so on. Most importantly, these sub-cases are organised in an **"if ... then ... else if ... else ..."** statement. We proceed to case 2(b) only if we fail to locate a node satisfying case 2(a).

As in the analysis of algorithm *RCBT* (Lemma 2), we are able to account for all frozen steps which involve class-1 packets. However, we have to pay the extra cost for the cases that a class-2 packet (located at a level-1 node) is swapped with a class-0 packet (located at a level-2 node). In this case, no class-2 packet enters the same subtree (rooted at a level-2 node) twice. Because of the initial balancing at most 2 class-2 packets are in any subtree rooted at a level-1 node (and thus, at a level-1 node). This implies that the extra number of routing steps for a given subtree rooted at a level-1 node is at most $2(d-1)$. This situation can occur at every subtree rooted at a level-1 node. Because of the initial balancing, in one subtree we can have at most $2(d-1)$ extra steps while in all of the remaining ones we can have a total of at most $(d-1)^2$ extra steps ($d-1$ extra steps for each of the remaining $d-1$ subtrees rooted at level-1 nodes). We conclude that the total number of extra routing steps due to class-2 packets is d^2-1. Thus, the routing of Step 6 will terminate within $n+d^2-2$ steps. □

Note 1. If we had omitted the first three "balancing" steps of the algorithm, $O(d^3)$ extra steps might be needed to account for the *freeze* commands due to class-2 packets (located at level-1 nodes) swapped with class-0 packet (located at level-2 nodes). This could happen in the case where all (or most of) the class-2 packets end up (after the heap-ordering of step 5) in the same subtree (rooted at a level-2 node) and then the routing proceeds in such an unfortunate way that these class-2 packets have to enter every subtree rooted at a level-2 node.

Note 2. A more careful refinement of Step 6 of algorithm RCdT that requires $n+2d$ steps is possible. In this refinement, we initially avoid swaps between class-1 and class-2 packets. Instead, we swap the class-1 packet with a class-0 packet (provided it exists). After the class-0 packets are exhausted, we proceed as usual. This simple modification guarantees that the swaps which involve class-2 packets in subtrees rooted at different level-1 nodes do not interfere with each other. Since only lower order terms are affected by this improvement, we omit a detailed description. However, we use the improvement described here when we deal (in the next section)

with d-ary trees with $d \geq \frac{\sqrt{n}}{\log n}$.

Theorem 3 *Algorithm RCdT routes in an off-line fashion any permutation on an n-node complete binary tree in $(1 + \frac{1}{d^2-1})n + o(n)$ steps.*

Proof. Let $T(n)$ denote the number of routing steps required by the algorithm for routing a permutation on a complete d-ary tree of n nodes. Step 1 does not involve any routing. Step 2 can be completed within $2\log_d n$ steps while Step 3 can be completed within $2d$ steps (there are exactly $d+1$ packets destined for T^1). Step 4 does not require any routing while Step 5 terminates after at most $2\log_d n$ steps. By Lemma 3, Step 6 requires $n + d^2 - 2$ steps. Step 7 requires $T(\frac{n-(d+1)}{d^2})$ steps. Thus, we can write the recurrence relation

$$T(n) \leq T\left(\frac{n-(d+1)}{d^2}\right) + n + 4\log n + 2d + d^2.$$

This recurrence relation together with the facts that $T(1) = 0$ and $T(d+1) \leq \frac{3d}{2}$ (routing on a "star" [2]) imply that

$$T(n) \leq (1 + \frac{1}{d^2-1})n + o(n).$$

This completes the proof. □

5. Stretching the Method to its Limits

In this section, we describe how to route a permutation on a d-ary tree in $n+o(n)$ routing steps. We achieve this by generalising Algorithm *RCdT*. In Algorithm *RCdT* we routed the packets to their destination subtrees rooted at level-2 nodes and then we proceeded recursively. We extend this idea by routing the packets to their destination subtrees rooted at nodes deeper in the tree. More specifically, we restrict the recursive routing to trees of $O(\sqrt{n})$ nodes. Consider a complete d-ary tree of n nodes. Let $h = \lfloor \log_d n \rfloor$ be the height of the tree. For simplicity, in our description of the routing algorithm we assume that h is even. The case where h is odd is treated in a similar way. Consider the partial subtree $T^{\frac{h}{2}}$ that is rooted at r and call its nodes *top nodes*. Tree $T^{\frac{h}{2}}$ has exactly $d^{\frac{h}{2}} \leq d^{\frac{\log_d n}{2}} = \sqrt{n}$ top nodes as leaves. Moreover, $T^{\frac{h}{2}}$ has less than $2\sqrt{n}$ nodes. The children of the level-$\frac{h}{2}$ top nodes of $T^{\frac{h}{2}}$ are the roots of $d^{\frac{h}{2}+1}$ subtrees $T_{r_{(\frac{h}{2}+1,i)}}$, $1 \leq i \leq d^{\frac{h}{2}+1}$, each of height $\frac{h}{2} - 1$ (and thus, of at most $\sqrt{n}$ nodes). Our algorithm routes the packets to their correct subtrees, i.e., $T^{\frac{h}{2}}$ and $T_{r_{(\frac{h}{2}+1,i)}}$, $1 \leq i \leq d^{\frac{h}{2}+1}$. Then the routing is completed either recursively, or by executing Algorithm *RCdT*, or by executing the algorithm described in [2]. Again, it is critical for the algorithm to route the packets to their correct subtree fast.

For clarity, we describe the algorithm by specifying the postconditions of each of its steps. In the refinement of the steps we show how to achieve these postconditions.

Algorithm *Fast_Route_on_Complete_d-ary_Trees* /* *Fast_RCdT* for short */

1. Let $h = \lfloor \log_d n \rfloor$ be the height of the tree. W.l.o.g., assume that h is even.
2. *[Balancing]* Call the packets destined for $T^{\frac{h}{2}}$ *top packets.* Route the packets in the tree such that at the end of this step:
   (a) All top packets are in subtrees rooted at level-$(\frac{h}{2}+1)$ nodes.
   (b) All subtrees rooted at a level-i top node contain at least $\sum_{j=0}^{\frac{h}{2}-i} d^j$ (and at most $\sum_{j=0}^{\frac{h}{2}-i} d^j + i$) top packets.
3. Perform routing and assign new class numbers to the packets such that at the end of this step the following conditions are satisfied:
   (a) The balancing of top packets (as described in the previous step) is maintained.
   (b) Let p be an arbitrary packet currently at node *curr* and destined for node *dest* and let l be the level of the lowest common ancestor of nodes *curr* and *dest*, i.e., $l = d_T(lca(curr, dest))$.
   If *dest* is in $T^{\frac{h}{2}}$ **then** p is a class-$(\frac{h}{2}+1)$ packet
   else if $l > \frac{h}{2}$ **then** p is a class-$(\frac{h}{2}+2)$ packet
   else p is a class-l packet.
   (c) The tree is heap-ordered with respect to the class numbers assigned to its packets.
4. Route the packets to their destination subtrees ($T^{\frac{h}{2}}$, and $T_{r_{(\frac{h}{2}+1,i)}}$, $1 \le i \le d^{\frac{h}{2}+1}$).
5. Recursively, or by using Algorithm *RCdT*, route the packets in $T^{\frac{h}{2}}$, and $T_{r_{(\frac{h}{2}+1,i)}}$, $1 \le i \le d^{\frac{h}{2}+1}$.

5.1. Refinement of Step 2 of Algorithm Fast_RCdT

The purpose of Step 2 is to distribute the top packets (referred as class-$(\frac{h}{2}+1)$ packets in Step 3) into subtrees rooted at level-$(\frac{h}{2}+1)$ nodes. This minimises the overhead due to their movement in the refinement of Step 4. Consider postcondition 2(b). Note that a tree rooted at a level-i top node contains exactly $\sum_{j=0}^{\frac{h}{2}-i} d^j$ top nodes. Thus, exactly $\sum_{j=0}^{\frac{h}{2}-i} d^j$ top packets are destined for it. Since at most $\sum_{j=0}^{\frac{h}{2}-i} d^j + i$ top packets can be in it after completion of the step, we only allow as many top packets as the number of nodes in the path from the root of the subtree (rooted at a level-i node) to the root r of the d-ary tree. (This fact will play an important role in bounding the number of steps required to realize Step 4 of the algorithm.) Also observe that, because of postcondition 2(a), no top packet is located at a top node at the beginning of Step 4.

Step 2 corresponds to the first three steps of algorithms *RCBT* and *RCdT* and is implemented in a similar way. First we assign a class value of 0 to each top packet and a class value of 1 to the remaining packets. Then we heap-order the tree. After the heap-ordering, all top packets form a partial tree rooted at r. Finally, we complete the step by computing how many packets have to enter/leave each subtree and then routing the packets according to our computations. The fact that we do not care which specific top packet enters each subtree makes routing easy.

Lemma 4 *The postconditions of Step 2 of algorithm* Fast_RCdT *can be satisfied in at most* $\log_d n + 4\sqrt{n}$ *routing steps.*

Proof. We satisfy the postconditions as described in the refinement of Step 2 above. Let μ be the exact number of the top packets, $\mu < 2\sqrt{n}$. Assigning class values does not require any routing. By Theorem 1 we know that the tree can be heap-ordered in $2\log_d n$ routing steps. For the routing that follows the worst case occurs when, after the heap-ordering, the top packets form a "broomstick" pattern where top packets are positioned at nodes on the path from the root r of the tree to a level-$\frac{h}{2}$ node, say m, and the remaining top packets are in subtree T_m. During the routing, at most $\mu - h$ top packets have to exit T_m and exactly 1 of them can exit every 2 steps. Thus, they exit T_m after at most $2(\mu - h)$. An additional number of h routing steps may be required for the last packet to reach the root and then move to an appropriate subtree. The total number of required routing steps is at most $2\log_d n + 2(\mu - h) + h = 2\log_d n + 2\mu - h < \log_d n + 4\sqrt{n}$ since $h \leq \log_d n$ and $\mu < 2\sqrt{n}$. □

5.2. Refinement of Step 3 of Algorithm Fast_RCdT

Step 3 of Algorithm *Fast_RCdT* corresponds to Steps 4 and 5 of algorithms *RCBT* and *RCdT*. While it is possible to design a dynamic version of the heap-ordering algorithm in which class values change during the heap-ordering (as in Step 5 of algorithms *RCBT* and *RCdT*), we choose to present a simpler to analyse, but slightly slower, algorithm that guarantees the postconditions of Step 3.

The implementation of Step 3 consists of a bottom-up fashion heap construction. We initially assign to all packets class values as specified by postcondition 3(b). Note that postcondition 3(a) is initially satisfied since it was also the postcondition of Step 2. We show how to build a heap such that 3(a) and 3(b) are invariants throughout the construction (will be referred to as such) and 3(c) is also true at the end.

Assume that 3(a) and 3(b) are currently true and that all subtrees rooted at a level-i node, $i > 0$, are heap-ordered with respect to their current class values. (All subtrees rooted at leaf nodes are trivially heap-ordered.) Consider any packet p at a level-$(i-1)$ node u and let $class(p)$ be its class value. Furthermore, let q be the packet of the smallest class value ($class(q)$) located at a level-i node, say v, that is a child of u. By our (inductive) assumption, T_v is heap-ordered. We consider two cases:

1. $class(p) \leq class(q)$. No swap of the packets is necessary. Subtree T_u is

heap-ordered and since no packet updated its class value, 3(a) and 3(b) still hold.

2. $class(p) > class(q)$. The two packets must be swapped in order to establish the heap invariant at node u. We consider the following cases:

   (a) *p is a class-$(\frac{h}{2}+2)$ packet.* Since invariant 3(b) is satisfied, it is implied that $i > \frac{h}{2}+1$. After the swap, both p and q maintain their original class values and q is located at a level-$(i-1)$ node, $i-1 > \frac{h}{2}$. Thus, invariants 3(a) and 3(b) are still valid.

   (b) *p is a class-$(\frac{h}{2}+1)$ packet.* Since invariant 3(a) is satisfied, it is implied that $i > \frac{h}{2}+1$. After the swap, both p and q maintain their original class values and since p, the class-$(\frac{h}{2}+1)$ packet, moves towards the leaves of the d-ary tree, invariant 3(a) is maintained. Since q moves to a level-$(i-1)$ node, $(i-1) > \frac{h}{2}$, invariant 3(b) is also satisfied.

   (c) $class(p) \leq \frac{h}{2}$. After the swap, and if necessary, we update the class values of p and q such that invariant 3(b) is re-established. Since the destination of p and q remains the same, it is impossible that during the update one of them becomes a class-$(\frac{h}{2}+1)$ packet. Thus, invariant 3(a) is maintained.

Note that packet p has to keep moving towards the leaves of the tree until it reaches a node for which the heap invariant is true. At the worst case, this happens when p reaches the leaves of the tree, i.e., after $h-i$ routing steps. When we establish the heap invariant at the root r of the d-ary tree, all the postconditions of Step 3 are satisfied. It is trivial to see that this happens in less than h^2 routing steps. Since $h = \log_d n$, we can state the following lemma:

Lemma 5 *The postconditions of Step 3 of algorithm* Fast_RCdT *can be satisfied in less than* $\log_d^2 n$ *routing steps.*

5.3. Refinement of Step 4 of Algorithm Fast_RCdT

The routing of Step 4 of Algorithm *Fast_RCdT* is performed in a fashion that can be described as an extension of the odd-even transposition method. In order to demonstrate the method, assume for the moment that all packets want to cross the root of the tree, i.e., they all are class-0 packets. Consider the packet that is initially at the root and assume that it is destined for subtree $T_{r_{(1,i)}}$, $1 \leq i \leq d$. Then, during the first step of the algorithm, edge $(r_{(0,1)}, r_{(1,i)})$ is active while all the even-level edges of $T_{r_{(1,i)}}$ (with respect to $r_{(0,1)}$) and all the odd-level edges of $T_{r_{(1,i')}}$, $i \neq i'$, (with respect to $r_{(0,1)}$) are *potentially active*[d]. After the swap of the packets over edge $(r_{(0,1)}, r_{(1,i)})$, assume that the new packet at the root is destined for subtree $T_{r_{(1,j)}}$, $1 \leq j \leq d$, $i \neq j$. Then, at the second step of the routing, edge $(r_{(0,1)}, r_{(1,j)})$ is active while the even-level edges of $T_{r_{(1,j)}}$ (w.r.t. $r_{(0,1)}$) and the

[d]We use the term *potentially active* to indicate that only one of the edges that connect a node with its children can be active at any time.

odd-level edges of $T_{r_{(1,i)}}$ (w.r.t. $r_{(0,1)}$) are potentially active. The odd-level edges of $T_{r_{(1,i)}}$ are active in order to allow for a new class-0 packet to reach node $r_{(1,i)}$. The routing continues in the same fashion where the potentially active edges during step i are determined by the destination of the packet at the root of the tree at the end of step $i-1$.

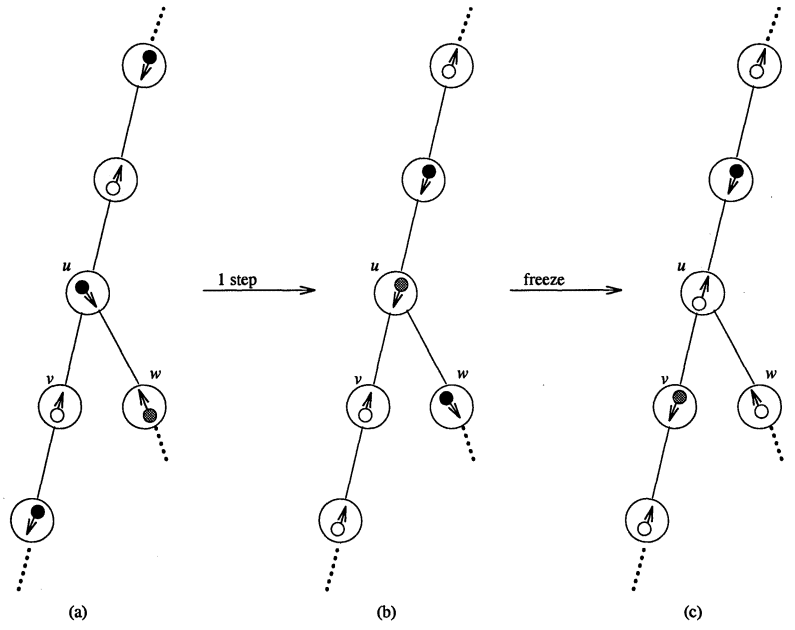

Fig. 3. The actions taking place during a "frozen" step due to a packet changing direction.

However, not all packets are class-0 packets. We now see how to deal with class-i packets, $i \neq 0$. During the routing, and at a given step, we can distinguish two types of packets. Packets that want to move towards the root of the tree in order to reach their destination subtree, and those packets that want to move towards the leaves of the tree. Assume that no two packets that want to move towards the leaves of the tree are at adjacent nodes. (This will obviously be the case if all packets are initially class-0 packets.) All of these packets can move one step towards the leaves of the tree at the same time, provided that the edges they have to use are active. We want to keep this property, i.e., no two packets that want to move towards the leaves of the tree are at adjacent nodes, as an invariant. This will guarantee that a class-0 packet crosses the root of the tree at each step, provided it exists. Consider the packets that move closer to the leaves during the current step. If the packets they were swapped with want to move (after the swap) towards the root, the invariant is obviously maintained. However, this is not the situation in the case where one of these packets reached the node where it has to turn. In this case, we might have two consecutive packets that want to move towards the leaves. Consider the situation described in Figure 3(a). White packets want to move towards the root and black packets want to move towards the leaves (the arrows also indicate

the desired direction of movement). We focus on the grey packet located at node w that wants to change direction of movement at node u, the parent of w. While the invariant is maintained in Figure 3(a), after one step of routing it is not valid anymore. This can be seen in Figure 3(b) where the grey packet (now located at node u), the packet located at the parent of u, and the packet located at w, want to move towards the leaves of the tree and are at adjacent nodes. We re-establish the invariant by executing a *freeze* command. During the frozen step we swap the packet that just changed direction (the grey packet in Figure 3(b)) with the packet at the root of the subtree it wants to move into (the white packet at node v in Figure 3(b)). This re-establishes the invariant. During the frozen step we also make swaps along the even-level edges of T_w (with respect to w). This results with reloading node w with a packet that wants to move towards the root (Figure 3(c)). This is necessary for the case where we need to execute two consecutive *freeze* commands. In the case where more than two packets want to turn during the same step, in order to avoid possible conflicts we first execute the freeze command for the packets that are closer to the leaves of the tree.

What remains to be specified is how we treat the case where we have packets that, at the beginning of Step 4 of Algorithm *Fast_RCdT*, want to move towards the leaves of the tree and occupy adjacent nodes. This situation clearly violates the invariant. However, it is easy to establish the invariant by starting routing these packets towards the leaves (the ones closest to the leaves first). This will re-establish the invariant after at most $\frac{\log_d n}{2}$ steps (without counting the frozen steps that we might have to interleave). The above procedure guarantees that as long as class-0 packets exist the root is going to hold one or, in the case that it does not, at least one of its children will do so.

Consider now the case where the root does not hold a class-0 packet. In this case it has to hold a class-$(\frac{h}{2}+1)$ packet. Our aim will be to make this situation infrequent. We now describe how to achieve this. Recall the balancing of the class-$(\frac{h}{2}+1)$ packets that was performed during Step 2 of Algorithm *Fast_RCdT*. Every subtree rooted at a level-i top node contains between $\sum_{j=0}^{\frac{h}{2}-i} d^j$ and $\sum_{j=0}^{\frac{h}{2}-i} d^j + i$ top (i.e., class-$(\frac{h}{2}+1)$) packets. Consider such a level-i, node, say u, where $i > 0$. During the routing we allow a class-$(\frac{h}{2}+1)$ packet to move from u to its parent, say v, only if the last i packets that have subtree T_u as their destination have started arriving in it. In this case, the packet (located at v) that is going to enter T_u is swapped with the class-$(\frac{h}{2}+1)$ packet located at u. If there are $i-1$ or more packets that are still expected to arrive in T_v, the class-$(\frac{h}{2}+1)$ packet starts moving towards the leaves of the tree in such a way that the balancing invariant 2(b) is never violated and the packet's final position is below packets of lower class. To initiate the movement of the class-$(\frac{h}{2}+1)$ packet we use a *freeze* command. During the frozen step we perform a swap. The swap of the class-$(\frac{h}{2}+1)$ packet that was just described corresponds with the swaps of the class-2 packets in Algorithm *RCdT* the cost of which had to be added in the number of required routing steps but was, fortunately, absorbed in the $o(n)$ term.

Lemma 6 *The routing of Step 4 of algorithm* Fast_RCdT *can be completed after*

$n + O(d\sqrt{n}\log_d n)$ *routing steps.*

Proof. Assume that there are m class-0 packets. There always is one of them next to the root (if we ignore the frozen steps) and thus they all start their movement towards the leaves of the tree within $m+d-1$ steps (see Theorem 1). Consider the remaining, say k, class-i packets, $0 < i \leq \frac{h}{2}$. They start their movement towards the leaves after executing a *freeze* command. Since each packet is responsible for exactly one *freeze* command, k additional steps are required. Finally, the last packet that changed the direction of its movement might need $\frac{\log_d n}{2}$ extra steps in order to enter its destination subtree which is rooted at a level-$(\frac{h}{2}+1)$ node.

We still have to pay the price for the frozen steps due to swaps initiated by class-$(\frac{h}{2}+1)$ packets. Consider an arbitrary level-i top node u, $0 < i \leq \frac{h}{2}$. For each class-$(\frac{h}{2}+1)$ packet that arrives at node u from its children we might have to issue a *freeze* command. Node u can receive at most $i+1$ class-$(\frac{h}{2}+1)$ packets from each of its children. This is because the subtree rooted at a level-$(i+1)$ node can find itself with at most $i+1$ more class-$(\frac{h}{2}+1)$ packets than the number of top nodes in it. As long as the other children of u have space for these packets, i.e., the children of u contain class-j packets, $0 \leq j \leq \frac{h}{2}$, that want to exit from the subtrees rooted at them, they are forwarded to these subtrees. This results in having the same packet causing more than one freeze command at the same node, each time arriving from a different child of that node.

Since there are $O(\sqrt{n})$ top nodes and since each of them can receive at most $\frac{\log_d n}{2}$ class-$(\frac{h}{2}+1)$ packets from each of its $d-1$ children, $O(d\sqrt{n}\log_d n)$ *freeze* commands might be executed. Within $m+k+d-1$ steps (ignoring the frozen steps) all other packets start their movement towards the leaves and after at most $\frac{\log_d n}{2}$ additional steps the last one enters its destination subtree. Since $m+k<n$, we conclude that the routing of Step 4 of algorithm *Fast_RCdT* is completed after $n + O(d\sqrt{n}\log_d n)$ routing steps. □

Theorem 4 *Algorithm* Fast_RCdT *routes in an off-line fashion any permutation on an n-node complete d-ary tree in* $n + o(n)$ *steps, provided that* $\frac{d}{\log d} = o(\frac{\sqrt{n}}{\log n})$.

Proof. Let $T(n)$ denote the number of routing steps required by the algorithm for routing any permutation on a complete d-ary tree of n nodes. Step 1 does not involve any routing. By Lemma 4, Step 2 can be completed within $\log_d n + 4\sqrt{n}$ routing steps. By Lemma 5, Step 3 can be completed within $\log_d^2 n$ routing steps. By Lemma 6, Step 4 can be completed within $n + O(d\sqrt{n}\log_d n)$ routing steps. In Step 5, we apply the routing recursively on a tree of at most $2\sqrt{n}$ nodes.

Thus, we can write the recurrence relation

$$T(n) \leq T(2\sqrt{n}) + n + O(d\sqrt{n}\log_d n + \log_d^2 n).$$

Instead of solving the recurrence relation (and thus applying the algorithm recursively) we use any routing algorithm for trees (that terminates in a number of steps that is linear to the number of nodes of the tree, e.g., algorithm *RCdT* or the algorithm of [2]) to perform the routing that corresponds to the $T(2\sqrt{n})$ term. By doing so, we get that $T(n) = n + o(n)$, provided that $\frac{d}{\log d} = o(\frac{\sqrt{n}}{\log n})$. □

As Theorem 4 indicates, in $n+o(n)$ routing steps we can route any permutation on a complete d-ary tree provided that $\frac{d}{\log d} = o(\frac{\sqrt{n}}{\log n})$. However, our objective is to complete the routing on d-ary trees of n nodes in $n+o(n)$ steps for any possible d. We achieve this as follows: In the case where $d < \frac{\sqrt{n}}{\log n}$ we use algorithm *Fast_RCdT* to perform the routing and thus, Theorem 4 guarantees that the routing finishes within $n+o(n)$ steps. In the case where $d \geq \frac{\sqrt{n}}{\log n}$ we use algorithm *RCdT* that was described in Section 4.

Consider an n-node complete d-ary tree T of height $h(T) > 1$, $d \geq \frac{\sqrt{n}}{\log n}$. Let x be the number of nodes in each subtree of T rooted at level-2 nodes. Since there exist one level-0 node, d level-1 nodes and d^2 level-2 nodes, it holds that: $d^2x+d+1 = n$ and thus,

$$x = \frac{n-d-1}{d^2} < \frac{n}{d^2} \leq \frac{n}{(\frac{\sqrt{n}}{\log n})^2} = \log^2 n$$

Thus, if we use algorithm *RCdT* to do the routing, the recursive routing of T^1 and of the subtrees rooted at level-2 nodes is completed in at most $3\max(d+1, log^2 n)$ steps if the algorithm of [2] is used. Thus, the cost of the routing at the second (and last) level of the recursion is absorbed in the lower order term. If we also use the refinement that was briefly described in Note 2 of page 14, all packets reach their destination subtree rooted at a level-2 node in about $n+2d$ steps. The above, together with the fact that for all n-node complete d-ary trees of height $h(T) > 1$ it holds that $d < \sqrt{n}$, allow us to state our final theorem regarding routing on complete d-ary trees.

Theorem 5 *Any permutation on an n-node complete d-ary tree can be routed in an off-line fashion in $n+o(n)$ routing steps.*

The result does not hold for the marginal case where the height of the d-ary tree is 1. In that case, $d = n-1$ and the tree is identical to the star graph that was used in [2] for the proof of the $\lfloor 3(n-1)/2 \rfloor$ lower bound on the routing number of arbitrary trees.

6. Conclusions

We presented an off-line algorithm that routes a permutations (with respect to the matching routing model) on a complete d-ary tree of n nodes in at most $n+o(n)$ routing steps. The routing performance of the algorithm is near optimal (within a lower order term) since it is easy to construct an instance of the problem that requires n steps for its solution (a binary rooted tree where all packets in its left subtree are destined for nodes in its right subtree, and vice versa). As a result of our work, the routing number for complete d-ary trees was substantially reduced. However, there are several questions raised in this paper. We list some of them below.

(i) What is the exact routing number of complete d-ary trees? Can we eliminate the $o(n)$ term from Theorem 5?

(ii) All the algorithms developed in this paper are off-line algorithms. Design and analyse on-line algorithms for routing on trees using the matching model.

(iii) Our result holds for complete d-ary trees of height greater than 1. When the height of the tree is 1, the lower bound of $\lfloor 3(n-1)/2 \rfloor$ steps applies. However, in this case the tree is identical to the star graph. Can we incorporate the "shape" of the tree in the derived lower bound? It appears that trees of very large degree "similar" to the star graph have large routing number.

(iv) This paper demonstrates the need for non-recursive algorithms. We managed to prove that the routing number of n-node complete d-ary trees is at most $n + o(n)$ by employing a complicated recursive algorithm and an elaborate analysis and proof of correctness. In a sense, we stretched recursion to its limits. Non-recursive routing algorithms need to be examined. Can we extend the potential-function based analysis of the non-recursive heap construction algorithm given in [10, 11] to cover general routing algorithms?

References

1. S. Akl, *Parallel Sorting Algorithms*, Academic Press, 1985.
2. N. Alon, F.R.K. Chung, R.L. Graham, "Routing permutations on graphs via matchings", *SIAM Journal on Discrete Mathematics*, 7(3):513–530, 1994.
3. A. Borodin, Y. Rabani, B. Schieber, "Deterministic many-to-many hot potato routing", *IEEE Transactions on Parallel and Distributed Systems*, 8(6):587–596, 1997.
4. N. Deo, S. Prasad, "Parallel heap: An optimal parallel priority queue", *The Journal of Supercomputing*, 6(1):87–98, March 1992.
5. R.W. Floyd, "Algorithm 245: Treesort 3", *Communications of ACM*, 7:701, 1964.
6. N. Haberman, "Parallel neighbor-sort (or the glory of the induction principle)", Technical Report AD-759 248, National Technical Information Service, US Department of Commerce, 5285 Port Royal Road, Springfieldn VA 22151, 1972.
7. M. Houle, G. Turner, "Dimension-exchange token distribution on the mesh and the torus", *Parallel Computing*, 24(2):247–265, 1998.
8. D.E. Knuth, *The Art of Computer Programming. Volume 3: Sorting and Searching*, Addison-Wesley, 1973.
9. D. Krizanc, L. Zhang, "Many-to-one packet routing via matchings", in *Proceedings of the Third Annual International Computing and Combinatorics Conference, COCOON '97*, Shanghai, China, August 1997, eds. Tao Jiang, D. T. Lee, LNCS 1276, (Springer-Verlag), pp. 11–17.
10. G. Pantziou, A. Roberts, A. Symvonis, "Dynamic tree routing under the 'matching-with-consumption' model", in *Proceedings of the 7th International Symposium on Algorithms and Computation ISAAC '96*, Osaka, Japan, December 1996, eds. T. Asano, Y. Igarashi, H. Nagamochi, S. Miyano, S. Suri LNCS 1178, (Springer-Verlag), pp. 275–284.
11. G. Pantziou, A. Roberts, A. Symvonis, "Many-to-many routing on trees via matchings", *Theoretical Computer Science*, 185(2):347–377, 1997.
12. N. Rao, W. Zhang, "Building heaps in parallel", *Information Processing Letters*, 37:355–358, March 1991.
13. A. Roberts, A. Symvonis, L. Zhang, "Routing on trees via matchings", in *Proceedings of the Fourth Workshop on Algorithms and Data Structures (WADS'95)*, Kingston, Ontario, Canada, August 1997, LNCS 955, (Springer-Verlag), pp. 251–262.

14. A. Roberts, A. Symvonis, "Potential-functionbased Analysis of an off-line Heap Construction Algorithm", *Journal of Universal Computer Science*, 6(2):240–255, 2000.
15. A. Symvonis, "Routing on trees", *Information Processing Letters*, 29(4):215–223, 1996.
16. J.W.J Williams, "Algorithm 232: Heapsort", *Communications of ACM*, 7:347–348, 1964.
17. L. Zhang, "Optimal bounds for matching routing on trees", *SIAM Journal on Discrete Mathematics*, 12(1):64–77, 1999.
18. W. Zhang, R.E. Korf. "Parallel heap operations on an EREW PRAM", *Journal of Parallel and Distributed Computing*, 20(2):248–255, February 1994.