Oblivious Routing in General Embedded Networks \label{sec:routing}

# Background \label{sec:routing:background}

Routing algorithms are responsible for finding a path from point A to point B in a given network. There are a variety of competing criteria that go into designing a routing algorithm. Should it be simple and compact? Should it be sensitive to current traffic loads? Should it be determined at compile-time or run-time? Should the path be determined in advance or a node at a time? These questions and more have led to a plethora of routing algorithms designed for all sorts of networks. This chapter will consider the basic types of routing and then present a relatively new method of routing that has never (to the author’s knowledge) been implemented in a working system before.

## Types of Routing Algorithms \label{sec:routing:background:comparison\_routing\_algorithms}

Routing algorithms can be divided and sub-divided into a number of groups. The basic organization ad heirarchy of unicast routing algorithms is shown in Figure \ref{fig:routing:taxonomy}.

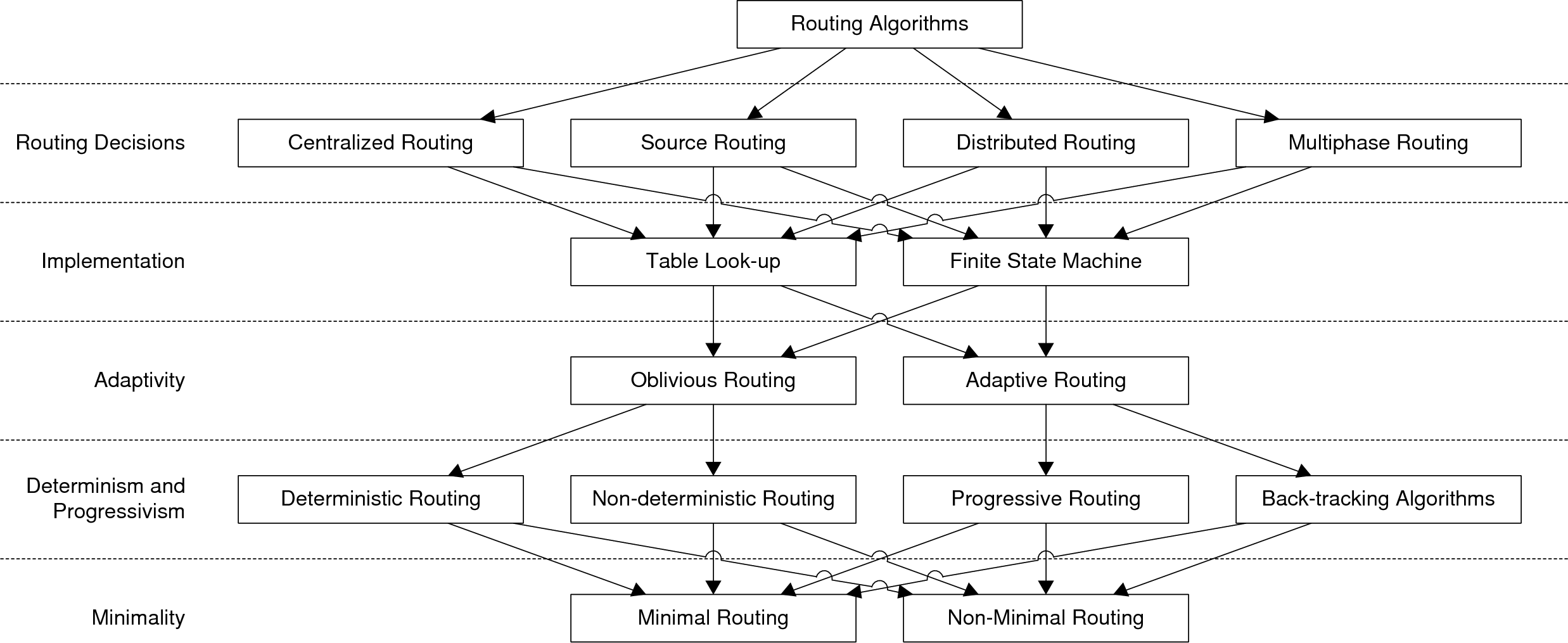


Figure x: Taxonomy of Routing Algorithms \cite{ref:1997-duato-interconnection\_networks}  
\label{fig:routing:taxonomy}

Routing decisions can generally be classified as centralized, source, distributed, or multiphase. In centralized routing, all routing decisions are made by a centralized controller. In source routing, the routing decisions are completely made by the source of the transmission. In distributed routing, each node in the path decides the next node to take. Combinations of the above can be used, and are called multi-phase routing decisions. For example, the source node could determine a few ``waypoints’’ but leave the path from waypoint to waypoint up to the nodes in the path. \cite{ref:1997-duato-interconnection\_networks}

Routing algorithms are usually implemented in one of two methods: with a routing table or with a finite state machine. Routing table implementations use a table that is generated either at compile-time or at run-time, and each entry in the table contains information on how to route to a given node. Finite state machines generate a path using a pre-determined state machine to generate a path for each request. \cite{ref:1997-duato-interconnection\_networks}

Routing algorithms are generally classified as either oblivious or adaptive. An oblivious routing scheme does not look at the state of the network (e.g. traffic load) when making a routing decision, although it can consider link cost. An adaptive routing algorithm does take the state of the network into consideration when making routing decisions. Adaptive algorithms generally use this information to attempt to lower the congestion in overworked nodes. Adaptive algorithms are more complex than oblivious algorithms, but are historically better at lowering congestion. \cite{ref:1997-duato-interconnection\_networks}

Oblivious routing algorithms can be deterministic or non-deterministic. Deterministic routing algorithms, such as Dijkstra’s algorithm, always provide the same shortest path between nodes. Non-deterministic oblivious routing algorithms can supply multiple, non-minimal paths, and select between them using randomization techniques. As a result, traffic is spread around the network, thus reducing localized congestion. \cite{ref:2003-racke-oblivious\_routing} Adaptive algorithms can be progressive, or can utilize backtracking. A progressive algorithm always sends the packet towards the destination, whereas backtracking allows a packet to travel backwards (usually to attempt a new path). \cite{ref:1997-duato-interconnection\_networks}

Routing algorithms can be either minimal or non-minimal. A minimal routing algorithm always supplies the shortest path, or potentially paths if there is more than one path of the same shortest length. A non-minimal path can supply paths that are not the shortest. One implication is that minimal algorithms always send a packet closer to the destination, and thus backtracking algorithms cannot be minimal. \cite{ref:1997-duato-interconnection\_networks}

## Oblivious Routing in General Networks \label{sec:routing:background:oblivious\_routing}

While it may seem that adaptive routing is clearly superior to oblivious routing, in reality adaptive routing is more complex than oblivious routing. Adaptive routing also requires nodes to send data to each other so that nodes know what traffic is like at other nodes, which increases overall traffic. Researchers have recently begun focusing on oblivious routing for special applications where the overhead and complexity of adaptive routing are undesirable. Oblivious routing has been under study for a long time, but oblivious routing had only been applied to specific types of networks and was not suitable for arbitrary networks until relatively recently. In 2002, a mechanism was finally devised that allowed oblivious routing to be applied to any arbitrary network. The rest of this section is devoted to explaining that mechanism as described by Harald Räcke in \cite{ref:2003-racke-oblivious\_routing}. Räcke's method is a source routing, table-lookup based non-minimal, non-deterministic oblivious routing scheme.

### Preliminaries \cite{ref:2003-racke-oblivious\_routing} \label{sec:routing:background:oblivious\_routing:preliminaries}

The network is modeled as an undirected graph  with nodes  and edges . The number of nodes is  and the weight of an edge (i.e. link capacity) is . It is assumed that the weights are normalized such that the minimum connected weight is 1 and the maximum weight is . Note that if  then the nodes  and  are not connected. The function  represents the aggregate capacity between the two sets of nodes  and is defined as:



The capacity leaving a set  is defined as , where .

### The Bisimulation Technique \cite{ref:2003-racke-oblivious\_routing} \label{sec:routing:background:oblivious\_routing:bisimulation\_technique}

Räcke's method works by transforming general networks into a tree network, finding the desired path, and then transforming the path back to the general network. This process is called the bisimulation technique. Trees are used because finding paths in tree networks is trivial.

The first step in the routing algorithm is to transform a given problem instance  (i.e. a routing request between nodes ) on  to a problem instance  on .  is translated onto  using a previously determined mapping . When a message is sent between two nodes , as defined in , a corresponding message is sent between two nodes .

The second step involves finding the path between nodes , which is the solution to . To find a path in a tree network, one simply needs to find the two paths to the root node from nodes , concatenating the two paths at the root, and then removing any common intermediate nodes.

The third step is to transform the solution to  back onto . This is done using the randomized mapping , where . This mapping is randomized (i.e. selects one of many paths at random) because there are typically more nodes in  than in , which means that one node in  is represented by multiple nodes in  and thus there are multiple paths in  given a single path in .

### The Virtual Tree Network \cite{ref:2003-racke-oblivious\_routing} \label{sec:routing:background:oblivious\_routing:virtual\_tree\_network}

The virtual tree network is constructed using a hierarchical decomposition .  starts with graph  and partitions it into smaller sub-graphs. These sub-graphs are recursively partitioned until the last sub-graphs contain a single node in . A graph is partitioned such that the cut is of minimum sparsity[[1]](#footnote-1), which minimizes congestion in the network. This decomposition process describes a natural tree, where a node's children are the sub-graphs partitioned from that node's graph. The tree that results, , is called the decomposition tree corresponding to , and each node corresponds to a set  in . An example decomposition is shown in Figure \ref{fig:routing:ex\_hierarchical\_decomposition}.

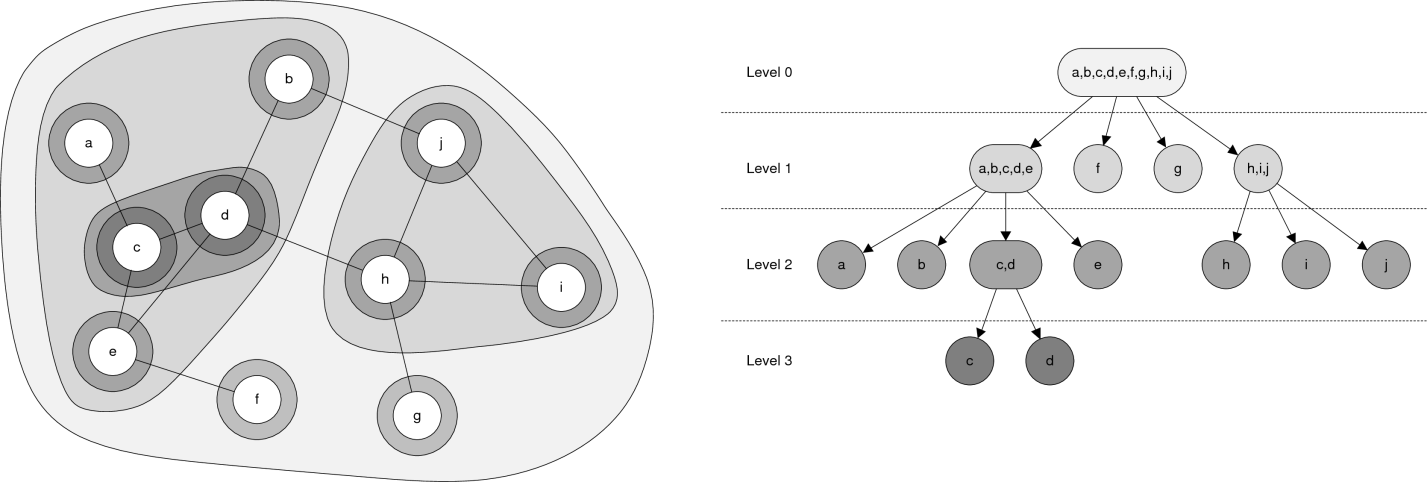


Figure x: Example Hierarchical Decomposition  
\label{ fig:routing:ex\_hierarchical\_decomposition}

The tree has several properties that are needed for later. The bandwidth of an edge in the tree is defined as . The level of a node in the tree is defined as the number of nodes between it and the root node. The weight of a level is defined by the function  for every level and is defined as:



Informally, the weight is the capacity of all edges leaving the set for nodes in the higher level nodes. A Concurrent Multi-Commodity Flow (CMCF) problem, from linear programming theory is defined for each cluster in the hierarchical decomposition. The CMCF for node  has a commodity  for each ordered pair . The source for the commodity is , the sink is , and the demand is defined as:



The flow for the commodity along an edge in  is decomposed[[2]](#footnote-2) into a set of convex paths between the endpoints. A path is selected from the set based on a randomized weighting according to  for a path  from the convex set.

# Objectives \label{sec:routing:objectives}

The routing algorithm must meet the following criteria:

* Self-configuring
* Applicable to general networks
* Utilize communication resources efficiently
* Utilize processing resources appropriately

As with the network, data link, and physical layers, the routing algorithm must be completely self-configuring to make it easy to use. In addition, the routing algorithm must be applicable to any type of network in order to avoid constraining the application developer. The routing algorithm must be appropriate to both the communication and processing resources available. In other words, it should avoid congestion and shouldn’t need to communicate much to support routing. Quick computation of routing tables and path selections must be possible. Of course these last few points are contradictory, so a balance needs to be struck.

# Methodology \label{sec:routing:methodology}

Räcke's method will be used for this project because it has the advantage of being applicable to general networks, which satisfies the second objective, and not needing the state of the network, which helps satisfy the third objective. The primary downside to Räcke's method is that solving CMCF’s involves using techniques from linear programming that are NP-complete (i.e. unsolvable by computers). Approximations do exist, such as the ellipsoidal method proposed by Räcke himself, but even these methods are still too complex to implement effectively on embedded hardware. To make Räcke's method suitable for embedded systems, the sparsest cut calculation and path decomposition must be replaced with elements that are simple to compute. This will, unfortunately, nullify some of the bounds on congestion that were determined for the method, but that is deemed as acceptable. The routing algorithm is divided into two sections: routing tree generation and path selection.

## Routing Tree Generation \label{sec:routing:methodology:routing\_tree\_generation}

The steps for generating the routing tree are:

1. Generate graph  from the node information
2. Decompose the graph into tree network 
3. Calculate the tree node weights 

Graph  is generated using the neighbor information obtained via the report neighbors protocol command. The list of nodes and edges are stored in a single data structure called an adjacency matrix . If two nodes  are connected, then , otherwise. By this definition,  is symmetrical along the diagonal.  is technically undefined, although is set to 0 in practice.

To decompose the graph, a graph partitioning algorithm must first be devised. A partitioning algorithm that performs a single cut (i.e. divides the graph in two) is simpler to implement than an algorithm that could potentially perform multiple cuts, so a binary partitioning algorithm is used. The minimal paths between all of the nodes in the (sub)graph are calculated first. The longest minimal path, , is found, the length, , is calculated, and the endpoints of the path, . The nodes are then sorted according to their distance from , and a cut-point, , is set for the middle of the sorted list. All nodes that are below the cut-point are grouped with , and the rest of the nodes are grouped with . In some cases this method can lead to nodes being grouped together that are far apart geographically if the network is not well distributed. In this case, a rule of thirds is applied: if the distance between  and , , is less than  or greater than , then  is shifted until . An example is shown in Figure \ref{fig:routing:example\_graph\_partition}. This process is repeated until the leaves of the tree only contain graphs with a single node. An example decomposition is shown in Figure \ref{fig:routing:example\_binary\_decomposition}.

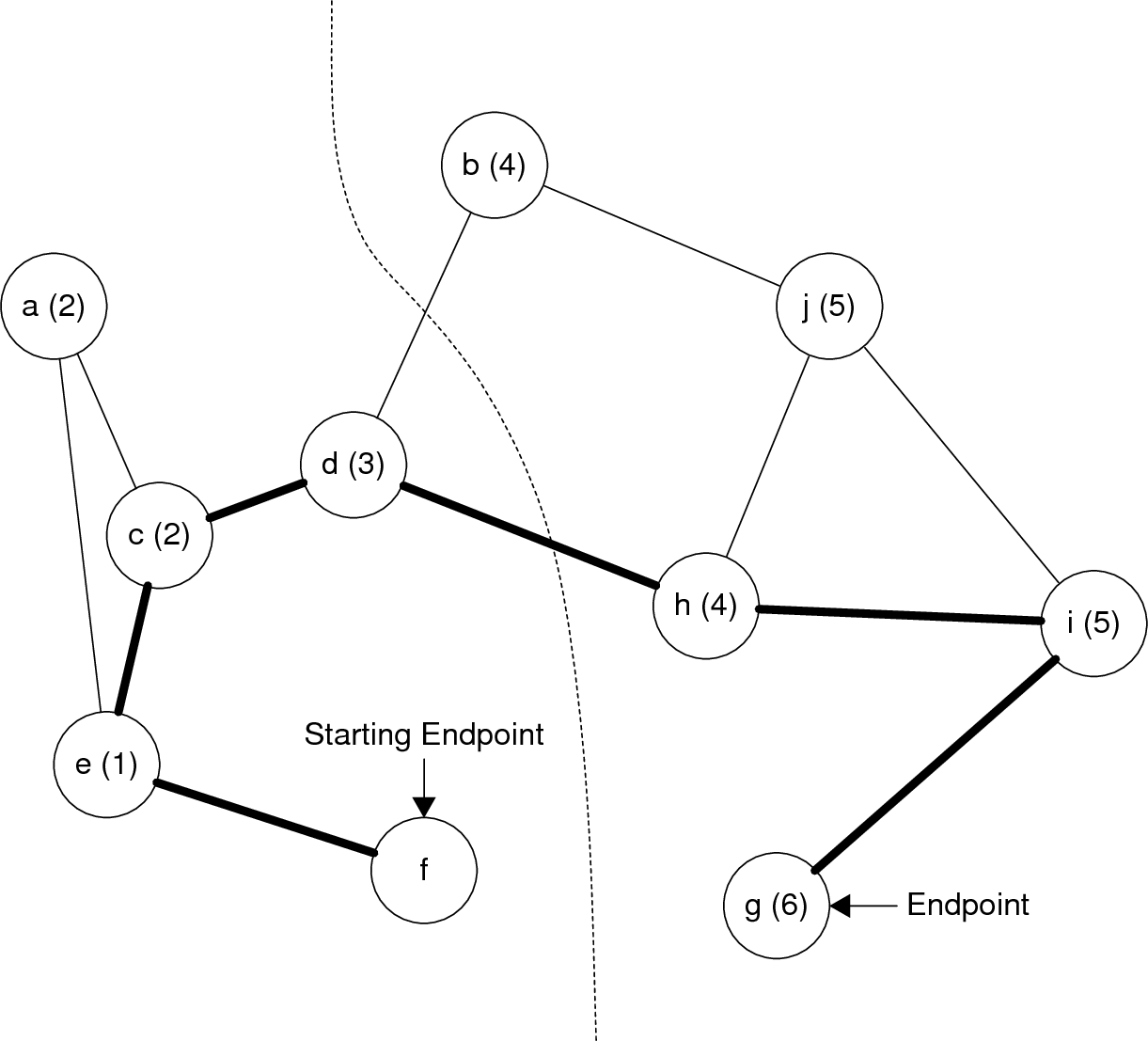


Figure x: Example graph partition  
\label{fig:routing:example\_graph\_partition}

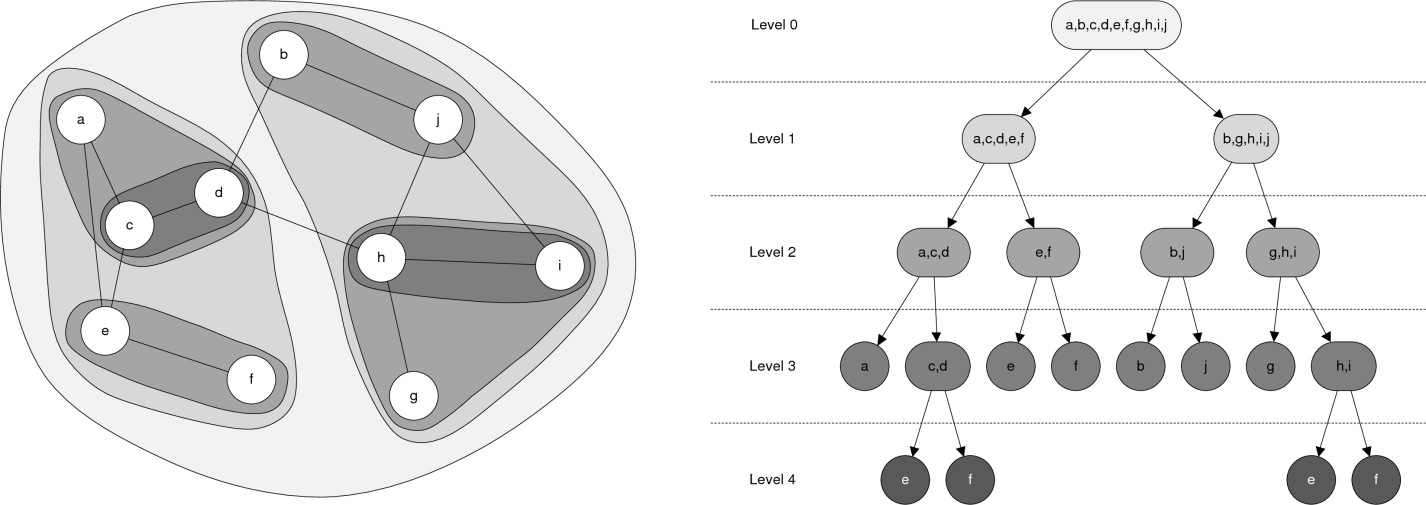


Figure x: Example Binary Decomposition  
\label{fig:routing:example\_binary\_decomposition}

The weights for each node are calculated using the formula from Section \ref{sec:routing:background:oblivious\_routing} for node capacity:



This formula states that the weight is defined as the normalized bandwidth between node  and all graph nodes in the tree node's parent. For the root node, the capacity is defined as the total capacity of the network.

After the tree is generated, it can be distributed to the rest of the system via the new routing table available and request routing table commands. The tree isn't a routing table in the strictest sense, but it serves the same purpose as a routing table.

## Path Selection \label{sec:routing:methodology:path\_selection}

The first step in determining the path that a packet will take is to find the path in the tree network, . This is done by find the path from the source leaf to the tree root, the path from the destination leaf to the tree root and concatenating the paths. An example is shown in Figure \ref{fig:routing:example\_tree\_path}.

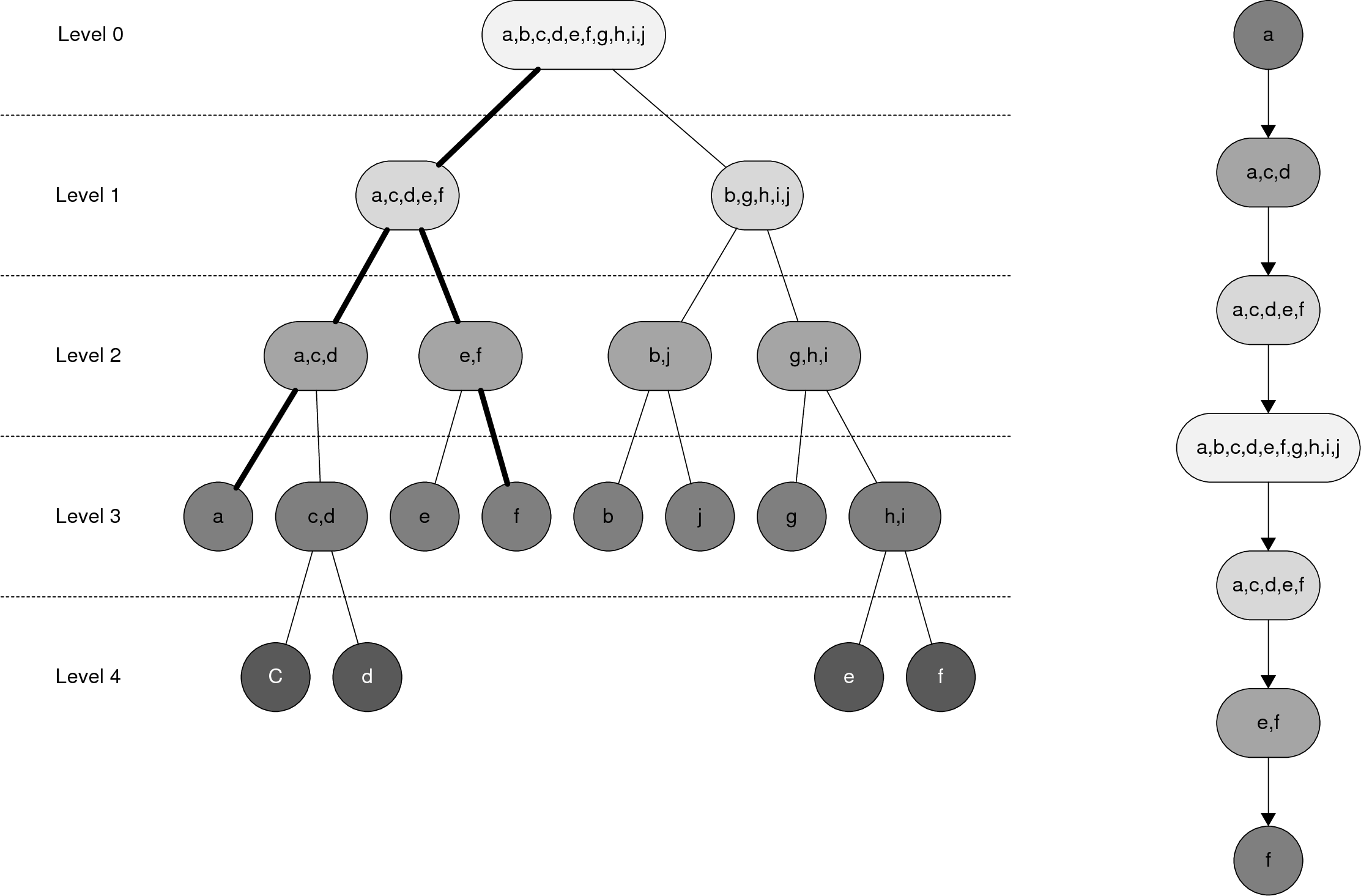


Figure x: Example Tree Path  
\label{fig:routing:example\_tree\_path}

The next step is to select a path  comprised of nodes in  from . This is done by generating randomized values for each node in a subgraph, and selecting based on this randomization and the weight. A side effect of not using linear programming to solve this problem is that sometimes a selected path gets ``stuck,'' where there are no nodes in the next subgraph connected to the currently selected subgraph. In this case, the algorithm backtracks to the previous tree node, marks the previously selected graph node as invalid, and tries again. An example path selection is shown in Figure \ref{fig:routing:example\_path\_selection}.

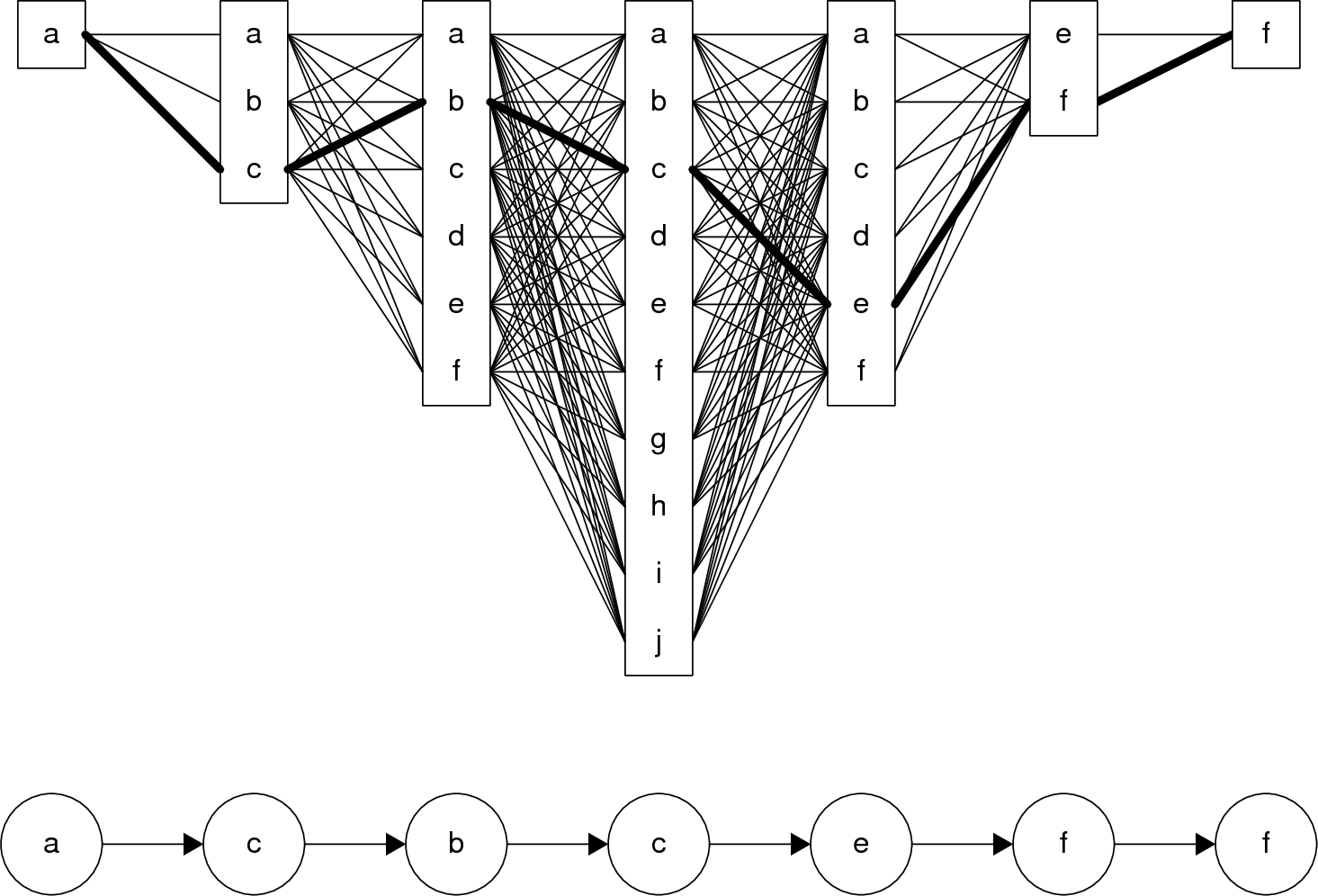


Figure x: Example path selection  
\label{fig:routing:example\_path\_selection}

The final step is to remove any redundant graph nodes. Nodes may be repeated in direct succession, or there may be other nodes in between, in which case this represents a path cycle. Once this is complete a valid path has been selected. An example path reduction is shown in Figure \ref{fig:routing:example\_path\_reduction}.

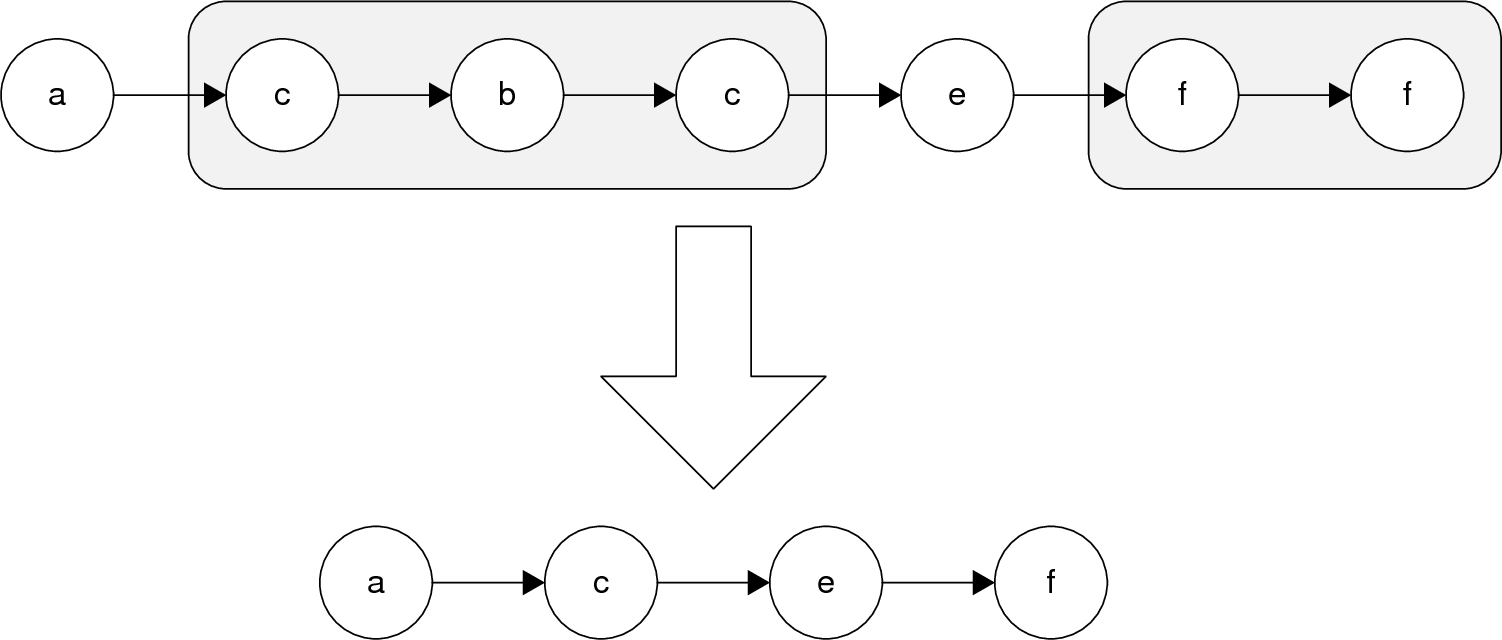


Figure X: Example reduced path  
\label{fig:routing:example\_path\_reduction}

# Implementation \label{sec:routing:implementation}

## Routing Tree Generation \label{sec:routing:implementation:routing\_tree\_generation}

The first step in the routing algorithm is, once again, generating the graph. In practice, an edge list is created before the adjacency matrix to aid in speeding up later steps. Edges are represented by a tuple consisting of the addresses of the two endpoint nodes. Each tuple is always ordered with the lower address first, and the list of tuples is ordered first by the lower address, then by the upper address. When an edge is added to the edge list, it is also added to the adjacency matrix. An adjacency matrix is used because determining whether or not two nodes are connected only requires a single lookup operation and no loops, which greatly increases performance. This is important because the routing algorithm checks this information many times, thus even slight increases in performance here result in large performance increases aggregately.

The second step in the routing algorithm is to generate the tree decomposition. The set of all minimal paths is created before the partitioning process begins to save computation time during the partitioning process. The algorithm for creating the minimal path list is borrowed from \ref{2007-lau-java\_library\_of\_graph\_algorithms}, and is essentially a modified version of Dijkstra's algorithm that re-uses information between destinations. The algorithm is shown in Table \ref{tab:routing:shortest\_minimal\_paths}.

|  |
| --- |
| Given:  nodes and  edges. Find the shortest paths, , given  from node 1 to all others |
| 1. Set |
| 1. For , do the following |
| * 1. Let  and compute |
| * 1. If  is less than current minimum, set |
| 1. Set |
| 1. If  then stop, otherwise go back to step 2. |

Table x: Shortest Minimal Path Algorithm \ref{2007-lau-java\_library\_of\_graph\_algorithms}  
\label{tab:routing:shortest\_minimal\_paths}

To partition a (sub)graph, the algorithm first checks if there are two nodes because that is a special case. If there are two nodes, then the two nodes are split and the branch is finished. If there are more than two nodes, then the longest path is found by comparing the minimal paths between all nodes in the (sub)graph. After the path is found, the distance from the start endpoint to all of the other nodes is found, and all of the nodes are sorted according to distance. The cut-point is calculated and the rule of thirds is checked. If the rule of thirds fails, then the cut-point is shifted until the rule is satisfied. After the groups have been determined, the new child nodes of a parent node are created. If there are two or more nodes in the new child node, then the partition algorithm is called again. The complete partition algorithm is shown in Table \ref{tab:routing:partition\_algorithm}.

|  |
| --- |
| Given: minimal path set  and (sub)graph |
| 1. If |
| * 1. Split  into two child nodes  and , each containing one graph node |
| 1. Otherwise |
| * 1. Find , where  is the minimal path set for |
| * 1. Sort the nodes in  according to distance from an endpoint |
| * 1. Set |
| * 1. If  or |
| * + 1. Shift  until |
| * 1. Group nodes below  with  and nodes above  with |
| * 1. Create child nodes of ,  and , containing the two groups |
| * 1. If  or  run the partition algorithm on the child node |

Table x: Partition Algorithm  
\label{tab:routing:partition\_algorithm}

Node weights are pretty simple to calculate. The capacity between a graph node in a child tree node  and all graph nodes in  is calculated and stored as the weight directly. The values are not normalized here because the process would require the path selection to use floating point math, which is slow on devices that don't have a hardware floating point unit, such as the F2808. The algorithm is shown in Table \ref{tab:routing:weight\_algorithm}.

|  |
| --- |
| Given: (sub)graph  and it's tree node parent |
|  |
| * 1. Calculate |

Table x: Node Weight Algorithm  
\label{tab:routing:weight\_algorithm}

## Path Selection \label{sec:routing:implementation:path\_selection}

The tree path between the source and destination is found by first creating a path from the source to the root, and the destination to the root. This is done by starting at the source/destination and storing the parent as the next step in the path. The parent of the parent is then stored, and so on to the root. Redundant nodes between the source and destination paths are then removed here, instead of later on, to gain efficiency. Starting with the source node, probabilities are generated by multiplying the weights and a random number for each graph node. The node with the highest probability is selected, and the process repeats itself at the next tree node. Before the probabilities are calculated, each node is checked for connectivity with the previously selected node, and if not the probability is set to 0. If all probabilities in a given tree node are set to 0, then the algorithm backtracks to the previous tree node, marks the previously selected node as a dead end by marking its probability as 0, and tries again. When the full path is selected, redundant nodes are removed to form a simple path. The full algorithm is shown in Table \ref{tab:routing:path\_selection}.

|  |
| --- |
| Given: source node  and destination node |
| 1. Generate the tree paths to the root node  and |
| 1. Remove redundant parts of  and  (i.e. path subsets ) |
| 1. Concatenate  and  to form |
| 1. Set the current tree node index, , to point to thesource node  (i.e. to index 1) |
| 1. Generate random numbers, , for all nodes in the next tree node |
| 1. Calculate  for all node weights |
| 1. Store the graph node, , with the highest probability  to the network path |
| 1. If there is no |
| * 1. Decrement |
| * 1. mark the previous node  with |
| 1. Otherwise increment |
| 1. Repeat steps 5 through 9 until  points to |
| 1. Remove any redundant nodes in |

Table x: Path Selection Algorithm  
\label{tab:routing:path\_selection}

# Results \label{sec:routing:results}

The routing algorithm is checked for correctness by viewing the results of the routing tree generation for two networks: one regular and well distributed, the other somewhat irregular. The regular network and its resulting decomposition tree are shown in Figure \ref{fig:routing:test\_1\_tree}. The algorithm gives a very nice decomposition tree that is a well-balanced, full tree. The first cut divided the network in half along its shorter axis, which encourages transmissions among nodes on one side of the network to stay in that network.

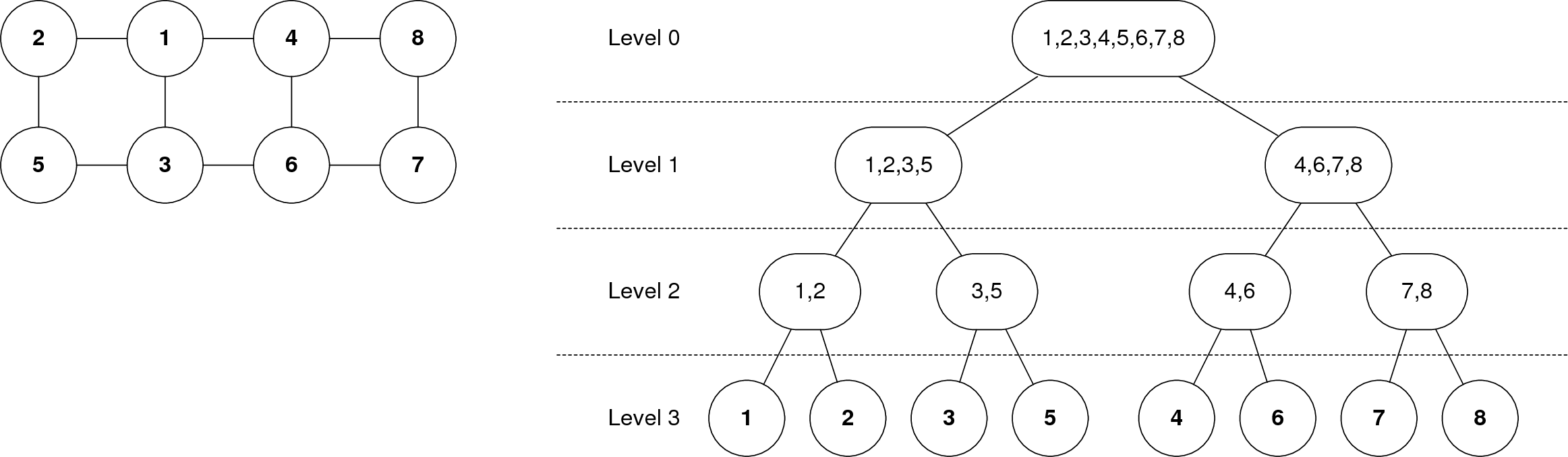


Figure x: Test network 1 and the resulting decomposition tree  
\label{fig:routing:test\_1\_tree}

The second network is much more irregular in design, and so produces a more irregular tree, with the results in Figure \ref{fig:routing:test\_2\_tree}. The tree still divided the network pretty well, although it couldn't quite figure out what to do with node 7. The network consists of two somewhat distinct parts, with node 7 straddling the middle, which is why node 7 is spun off so quickly, which is actually not a bad decision because it means each side of the network will have equal access, in terms of path lengths, to this node. The decision to lump nodes 2 and 4 together instead of nodes 2 and 8 may not be quite ideal, but it's not bad either.



Figure x: Test network 2 and the resulting decomposition tree  
\label{fig:routing:test\_2\_tree}

The next test is to determine the frequency that alternate paths are taken. The network from test 1 is used for the first test and paths from node 2 to node 7 are calculated. The results are in Table \ref{tab:routing:test\_1\_paths}. Four paths exist that are the shortest possible between nodes 2 and 7, and the routing algorithm selected all four of them. It also selected a more out of the way path on one occasion. This is the type of behavior that is desired because it favors the best path, but still distributes the traffic across the network.

|  |  |
| --- | --- |
| **Path** | **Number of times selected (out of 100)** |
|  | 62 |
|  | 10 |
|  | 20 |
|  | 7 |
|  | 1 |

Table x: Test 1 path selection results  
\label{tab:routing:test\_1\_paths}

The network from test 2 is used for the second test and paths from node 5 to node 2 are calculated. The results are in table \ref{tab:routing:test\_2\_paths}. This test showed a good distribution, once again. There are two shortest paths available, and the algorithm took both of them. It did also take a slightly longer path some of the time, but definitely favored one of the shortest paths most of the time.

|  |  |
| --- | --- |
| **Path** | **Number of times selected (out of 100)** |
|  | 77 |
|  | 11 |
|  | 12 |

Table x: Test 2 path selection results  
\label{tab:routing:test\_2\_paths}

# Future Work \label{sec:routing:future\_work}

Some rather obvious improvements exist that can be made. One of the reasons that Räcke's work was so important is that it proved oblivious routing is competitive with adaptive routing in regard to traffic. By modifying the algorithm, those bounds no longer apply. In the future, the bounds that do apply to this system could be found. A more intelligent partitioning algorithm could be conceived as well that would surely improve the bounds. Methods such as k-means clustering are worth investigating.

# Conclusions \label{sec:routing:conclusions}

In this chapter, a routing mechanism has been presented that is based on Harald Räcke's oblivious routing method. The mechanism works by mapping a general network onto a tree network. A path in the network is determined by finding the related path in the tree network and mapping it back to the general network. When the tree network is mapped back to the general network, multiple paths exist within the mapping that can be taken. Weights are assigned to the general graph nodes in the tree network based on their level of connectivity to the nodes in the tree nodes parent. Variable paths that tend toward the shorter paths are selected by mixing these weights with a random number generator. The mechanism produces routing trees that are sensible and well balanced. The mechanism works well and usually, but not always, selects the shortest available path(s) while also spreading the traffic around the network to reduce congestion.

1. The concept of a sparsest cut for a graph in this context is based on linear Programming and concurrent multicommodity flows, which are far too complex for this discussion. For information on this topic, see \ref{fig:routing:ex\_hierarchical\_decomposition} and \cite{ref:1993-ahuja-network\_flows}. [↑](#footnote-ref-1)
2. The idea of flow decomposition is a linear programming technique that is outside the scope of this paper. See \ref{AMO93 from Racke} for more information. [↑](#footnote-ref-2)