# Edge-Coloring Bipartite Multigraphs to Select Network Paths

Amy Ousterhout
aousterh@mit.edu

December 11, 2013

**Abstract**

We consider the idea of using a centralized controller to schedule network traffic within a datacenter and implement an algorithm that edge-colors bipartite multigraphs to select the paths that packets should take through the network. We implement three different data structures to represent the bipartite graphs: a matrix data structure, an adjacency list data structure, and an adjacency list data structure which tracks edges using bitmaps. We compare the performance of these three variations of the algorithm analytically and experimentally and conclude that the bitmap adjacency data structure performs best. This variation of the algorithm would allow us to select paths for a 10 Gbps datacenter with 800-1000 machines, or a 1 Gbps datacenter with 1500-2000 machines.

## 1 Introduction

In most packet networks, each component makes local decisions for individual packets. Routers determine the path each packet takes while endpoint congestion control algorithms (such as TCP) choose when to transmit packets using locally-observed congestion information (such as packet drops and round-trip-times). This distributed architecture is robust against failures of routers and links, but inhibits precise control of network behavior over short time scales. Bursts of packets from many different endpoints can overwhelm links, leading to large queues and dropped packets before the endpoints have time to respond. This can cause drastic variation in packet latencies and inefficient utilization of many links in the network. Furthermore, because TCP is sensitive to the ordering of packets, routers typically send all packets destined for the same endnode along the same path, even when multiple paths are available, preventing effective load balance. These problems are exacerbated when a network contains applications with different objectives; high-throughput services can fill up queues in the network, causing interactive applications to observe high latencies.

In this paper, we consider the idea of using a centralized controller to schedule network traffic within a datacenter. Instead of using distributed congestion control algorithms, this idea proposes to have each endnode request a send time and path from the controller for each packet that it wants to send. Packets are queued at endnodes until their scheduled send time, at which point they can zoom through the network with almost zero delay. This system is analogous to a hypothetical road traffic control system in which a central entity tells each car when to leave home and what path to take. Instead of sitting in traffic, you can zoom all the way to your destination (Figure 1).

For this idea to be feasible, a controller must be able to quickly determine when to send each packet and which path it should take to avoid any queueing within the network. To accomplish this, we divide time into many timeslots, so that each timeslot is just enough time for one endnode to

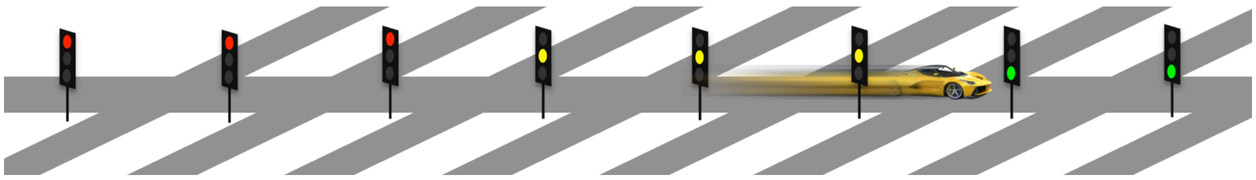

Figure 1: Traffic is queued at endpoints of the network until its allocated timeslot and then zooms through the network without delay.

transmit one packet, given the speed of the links in the network. In this paper, we assume that there exists a controller which chooses a set of packets that can feasibly be transmitted in each timeslot, and focus on the problem of assigning each packet to a path through the network. We model this problem as edge coloring a bipartite multigraph, where the nodes on one side are the senders, the nodes on the other side are the receivers, edges represent packets to be sent in a given timeslot, and colors indicate the path to take. In this paper, we implement the algorithm for edge-coloring bipartite multigraphs proposed by Kapoor and Rizzi [7] using three different data structures to represent the underlying graphs. We evaluate the performance of these three implementations for different network sizes, and conclude that the bitmap adjacency implementation is able to scale sufficiently well to make this idea feasible for small datacenter networks.

## 2 Background

### 2.1 Network Topology

Finding non-conflicting paths through a network with arbitrary topology is a difficult task, so we assume a specific network topology. We assume that our topology is a *multi-rooted tree* with three tiers. In this topology, each machine belongs to a single rack and has one link to its top of rack switch (ToRs). Each top of rack switch connects to several of the aggregate switches, and each aggregate switch then connects to several of the core switches, as shown in Figure 2. This topology is known as a multi-rooted tree because each core switch serves as a root.

We also assume that our network guarantees *full bisection bandwidth* [2]. This means that each switch has the same amount of bandwidth above it in the topology as below it. Consequently, the network can support any amount and combination of traffic that does not overwhelm the links into and out of the endnodes. In Figure 2, the links within the rack must have half the bandwidth of the links in the rest of the network to preserve full bisection bandwidth. For example, if the links from the machines to the top of rack switches were 1 Gbps links, the links to the aggregate switches must be 2 Gbps links, so that the bandwidth above and below each top of rack switch is 4 Gbps.

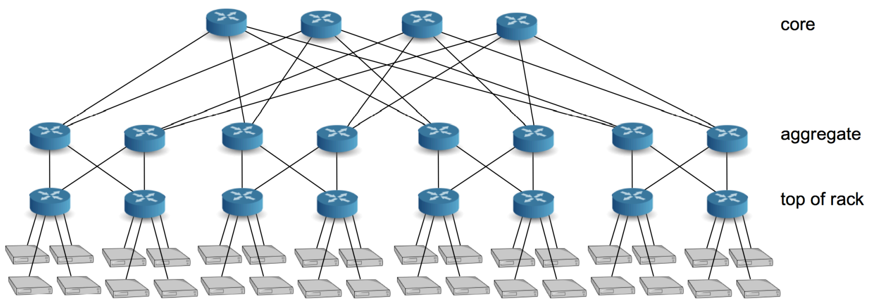

Figure 2: A simple datacenter network topology. Each of the 8 racks has 4 machines in it. The links above the top of rack switches are twice as fast as the links within a rack to maintain full bisection bandwidth.

Given this specific network topology, we can determine a set of paths for the packets transmitted in each timeslot by modeling this problem as edge coloring a bipartite multigraph. We assume that every packet goes through one of the core switches, even if it is destined for a machine in the same rack. This means that our traffic is pipelined so that all packets sent during the same timeslot will reach the top of rack switches, the aggregate switches, the core switches, and so on at the same time. Thus when assigning the path for a single packet, we need only consider the paths of other packets sent in the same timeslot. With the given topology, there is exactly one path from each machine to its corresponding top of rack switch, and exactly one (shortest) path from each top of rack switch to each core switch. Therefore, any path between two nodes can be uniquely specified by the core switch we send it through.

We seek to assign a core switch to each packet; this can be achieved with edge-coloring. We construct a bipartite multigraph where the nodes on the left represent the sending top of rack switches and the nodes on the right represent the receiving top of rack switches (each physical switch is represented as both a sender and a receiver). Each packet transmission from a sending ToRs to a receiving ToRs is represented by an edge in the bipartite multigraph. Note that each machine is connected to only one ToRs, so we can easily map from machines to their corresponding ToRs. We assume that the controller can choose a set of packets to transmit in a timeslot such that no machine sends more than one packet and no machine receives more than one packet. The only constraint we must now enforce with our assignment of paths is that all packets sent by a given ToRs go through a different core switch, and similarly for all packets received by a given ToRs. This assignment can be achieved by edge-coloring the bipartite multigraph with a number of colors equal to the number of core switches. We then send each packet along then path that goes through the core switch that corresponds to its assigned color.

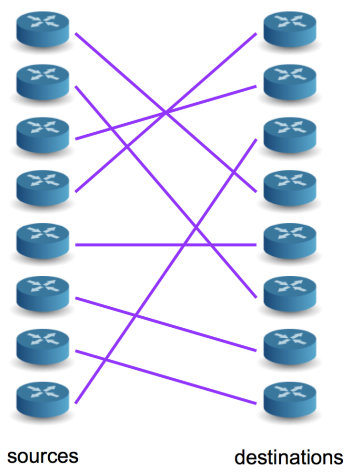

Figure 3: One perfect matching from source top of rack switches to destination top of rack switches. This traffic can be routed through one core switch in a single timeslot.

For example, a bipartite multigraph of packets to be sent on the topology of Figure 2 could be colored with four colors, because there are four core switches. Figure 3 shows one perfect matching that could be sent through one core switch in the topology of Figure 2. There are four machines attached to each top of rack switch, so each top of rack switch must send at most four packets per timeslot, one to each core switch. Thus any set of packets which can be admitted into this network during one timeslot can be decomposed into four matchings such as the one shown in Figure 3; four perfect matchings would fully saturate the network for one timeslot.

## 2.2 Speed Requirements

For this idea to be feasible, our path selection algorithm must have both low latency and high throughput. Each computation for an individual timeslot must be fast so that packets don't queue at endpoints for a long time while they wait for their allocations from the controller. The controller must provide high throughput so that it can select paths for every single timeslot. A small datacenter cluster might have 500-1000 machines in it and use 1 Gbps or 10 Gbps links; we assume these parameters.

To be competitive, the time required to perform the path selection must be less than typical queueing delays within a datacenter network. Typical queueing delays within a datacenter are on the order of tens or hundreds of microseconds [3]. Therefore, our path selection algorithm must take no more than a few hundred microseconds to determine the paths for traffic for a single timeslot.

Our algorithm must be able to deliver a path selection for a set of packets for every timeslot. Because the path selections for successive time slots are independent (the traffic scheduled for consecutive timeslots is likely to differ significantly), successive computations can be performed using different cores. The largest packet size allowed (maximum transmission unit or MTU) on Ethernet links is 1500 bytes. Thus it takes just over 1 $\mu$s to transmit a packet on 10 Gbps links, or 11 $\mu$s on 1 Gbps links. If each computation requires 100 $\mu$s, we will need 100 cores to provide enough throughput for 10 Gbps links, or 10 cores for 1 Gbps links. 10-50 cores can be provided by one machine, but 100 cores would be prohibitively expensive. For a network with 1 Gbps links, we can allow our algorithm 100 $\mu$s per computation, but for 10 Gbps links we can only feasibly allow 50 $\mu$s per computation to limit the number of cores necessary.

## 3 Problem Statement and Definitions

Our goal is to edge color bipartite multigraphs as quickly as possible using the minimum number of colors, or the minimum number of colors plus a small constant number of additional colors. An **edge-coloring** of a graph or multigraph is an assignment of a color to each edge in the graph such that no vertex has multiple incident edges with the same color. A **$k$-edge-coloring** is an edge coloring which uses $k$ colors. A **bipartite graph** is a graph who vertices can be divided into two disjoint sets such that all edges in the graph are incident upon one vertex in one of the sets and one vertex in the other set.

It is well known that a bipartite graph or multigraph with maximum degree $\Delta$ can be edge colored using $\Delta$ colors [8]. In this paper, we assume for simplicity that our multigraphs are regular multigraphs. A **regular graph** is a graph in which all vertices have the same degree (number of incident edges). Note that for any input graph that is not a regular graph, we can add edges until the graph is regular. Schrijver also proposes a method for converting a graph $G$ with $m$ edges and maximum degree $\Delta$ into a $\Delta$-regular graph $\bar{G}$ with $O(m)$ edges [9]. However, because our goal is to minimize the maximum time it will take to perform any edge-coloring and some timeslots will have traffic that consists of a regular graph with $m$ edges and degree $\Delta$ already, the more complex approach of Schrijver is unnecessary.

We now define a few additional terms which will be useful in Section §4. A **matching** in a graph is a subset of edges such that every vertex has at most one edge incident on it. A **perfect matching** is a subset of edges in a graph such that every vertex has exactly one edge incident on it. An **Euler tour** of a graph is a cycle that visits every edge exactly one.

## 4 Edge-Coloring Algorithms

We now describe the fastest known algorithms for edge-coloring bipartite graphs.

### 4.1 Graphs with Power of Two Degree

In 1976, Gabow proposed a divide-and-conquer algorithm for edge-coloring bipartite graphs with $m$ edges in $O(m \log \Delta)$ time when the degree $\Delta$ is a power of two [6]. This algorithm relies on repeatedly performing Euler-splits on the input graph. As described in Section §3, an Euler tour of a graph is a cycle that visits every edge exactly once. An Euler-split involves finding an Euler tour of a graph $G$, and then placing alternate edges in graph $G_1$ and alternate edges in graph $G_2$. This means that for each vertex $v$ in $G$, half of the edges incident on it will end up in $G_1$ and half will end up in $G_2$. Thus if $G$ is a regular graph with degree $\Delta$, then $G_1$ and $G_2$ are subgraphs of $G$

with degree $\frac{\Delta}{2}$ such that every edge in $G$ is in either $G_1$ or $G_2$. For an input graph with a degree $\Delta$ that is a power of two, we can recursively perform Euler-splits until we have a list of $\Delta$ graphs which each have degree one. Each of these graphs is a single perfect matching which can be colored using one color, for a total of $\Delta$ colors. Thus we have edge-colored our graph with $\Delta$ colors.

This algorithm requires $O(m \log \Delta)$ time. An Euler-tour of a regular bipartite graph $G$ with $m$ edges can be found in linear time by simply traversing the graph. In our first call to EDGE-COLOR, we perform one Euler-split on $m$ edges. For each of the two resulting subgraphs, we perform an Euler-split on $\frac{m}{2}$ edges, for a total runtime of $\frac{m}{2} \cdot 2 = m$, and so on. We must perform splits of $\log \Delta$ different sizes to produce perfect matchings from an input graph with degree $\Delta$. Each edge is present in exactly one split of each size, so our total runtime is $O(m \log \Delta)$.

## 4.2 Arbitrary Graphs

Repeatedly performing Euler-splits (§4.1) on a graph with degree $\Delta$ which is not a power of two will not yield a coloring with $\Delta$ colors. An Euler-split of graph $G$ with odd degree $\Delta$ will yield two subgraphs which might both have maximum degree $\lceil \frac{\Delta}{2} \rceil$. In the worst case, this could yield a coloring that requires $2\Delta - 2$ colors [5], which is unacceptable, because it would mean that we could only use about half of the network capacity in every timeslot. Thus we consider alternative algorithms, which are still based on Euler-splits.

### 4.2.1 Exact Algorithm with $\Delta$ Colors

In 1999, Kapoor and Rizzi proposed two algorithms for edge-coloring bipartite graphs of arbitrary degree [7]. They both leverage the idea that you can edge-color bipartite graphs of arbitrary degree in close to $O(m \log \Delta)$ time if you can find a single perfect matching quickly. The first algorithm they propose finds an edge-coloring for $G$ using $\Delta$ colors in time $T + O(m \log \Delta)$ time, where $T$ is the time required to find a perfect matching in a subgraph of $G$ with degree at most $\Delta$. They use the best known bound for finding a perfect matching in a bipartite graph at the time, which was $O(m + \frac{m}{\Delta} \log \frac{m}{\Delta} \log^2 \Delta)$ due to Hopcroft and Cole [4], yielding an algorithm that runs in time $O(m \log \Delta + \frac{m}{\Delta} \log \frac{m}{\Delta} \log^2 \Delta)$. Cole et. al. later found an algorithm to find a matching in a regular bipartite graph in $O(m)$ time, yielding an algorithm that can edge-color a bipartite graph of degree $\Delta$ using $\Delta$ colors in $O(m \log \Delta)$ time [5].

Unfortunately, though the asymptotic runtime of their algorithm is $O(m \log \Delta)$, Cole et. al. believe that the time to compute the perfect matchings dominates the runtime of the Euler-slits, yielding a complex algorithm which is likely to be slow in practice. Thus we considered an approximate algorithm which was likely to yield better performance in practice.

### 4.2.2 Approximate Algorithm with $\Delta + 2$ Colors

Kapoor and Rizzi proposed a second algorithm in 1999, which finds an edge-coloring for $G$ using at most $\Delta + 2$ colors in $O(m \log \Delta)$ time [7]; this is the algorithm we chose to implement. For our application it is acceptable to use a small number of additional colors, but this reduces the maximum utilization of our network from 100% to at most $\frac{d-2}{d}$ where $d$ is the number of machines per rack. This is acceptable for sufficiently large values of $d$. This algorithm is very similar to Kapoor and Rizzi's algorithm which involves computing a perfect matching, except that they utilize one or two arbitrary perfect matchings which are not originally in $G$ instead of computing perfect matchings from $G$, to handle subgraphs with odd degree. We outline their algorithm below and refer the reader to their paper for details [7].

In this algorithm, we work with a series of bins, where a *bin* is a list of regular graphs. The input to our algorithm is a bin which consists of the input graph $G$ with degree $\Delta$ and two arbitrary perfect matchings (we start with bin $B$ where $B[0]$ and $B[1]$ are arbitrary perfect matchings and $B[2] = G$). The output of our algorithm is a bin which includes $\Delta + 2$ perfect matchings. We proceed by performing a series of SPLIT-ODD and SPLIT-EVEN operations on the graphs in our bin. SLIT-EVEN is performed on a single graph with even degree $k$, and simply performs an Euler-split, resulting in two graphs each with degree $\frac{k}{2}$. SPLIT-ODD is performed on two graphs each with odd degrees $k_1$ and $k_2$, and involves adding the graphs together and then performing an Euler-split, resulting in two graphs each with degree $\frac{k_1+k_2}{2}$. SPLIT-ODD and SPLIT-EVEN are shown in Algorithm 1.

**Algorithm 1** Edge Coloring with Arbitrary Degree

```
**function** EDGE-COLOR(graph $G$) **returns** a list of perfect matchings
  $B \leftarrow$ ALMOST-SOLVE($G$)
  $M \leftarrow G_i \in B$ s.t. $G_i$ is a perfect matching
  $W \leftarrow G_i \in B$ s.t. $G_i$ is not a perfect matching
  **while** $W$ is not empty **do**
    $G_1 \leftarrow W$.pop() {treat $W$ as a stack with the graphs in order of increasing degree}
    **while** degree($G_1$) $\neq 1$ **do**
      **if** degree($G_1$) is even **then**
        $G_1, G_2 \leftarrow$ SPLIT-EVEN($G_1$)
      **else**
        $G_1, G_2 \leftarrow$ SPLIT-ODD($G_1$, $M$.pop())
      **end if**
      $W$.push($G_2$) {process $G_2$ later}
    **end while**
    $M$.push($G_1$) {$G_1$ is a perfect matching}
  **end while**

**function** ALMOST-SOLVE(bin $B$) **returns** an almost-solved bin $B'$
  see [7] for details

**function** SPLIT-EVEN(graph $G$) **returns** two subgraphs of $G$ with degree $\frac{d(G)}{2}$
**Ensure:** $G$ has even degree
  $G_1, G_2 \leftarrow$ EULER-SPLIT($G$)
  **return** $G_1, G_2$

**function** SPLIT-ODD(graph $G_1$, graph $G_2$) **returns** two graphs with degree $\frac{d(G_1)+d(G_2)}{2}$
**Ensure:** $G_1, G_2$ have odd degree
  add edges from $G_2$ to $G_1$
  $G_3, G_4 \leftarrow$ EULER-SPLIT($G_1$)
  **return** $G_3, G_4$
```

The first step of our algorithm is to convert the input bin into a bin which is almost solved. We use $d(G)$ to denote the degree of graph $G$. A bin $B$ is *almost solved* if the following two properties hold:

$$d(B[0]) = 1$$

$$d(B[j]) \leq \sum_{i=0}^{j-1} d(B[i]) \quad \forall\, j \text{ s.t. } 0 < j < \mathrm{length}(B)$$

An input bin can be converted to a bin which is almost solved using only SPLIT-ODD and SPLIT-EVEN operations. This requires $O(\Delta n \log \Delta)$ time, which is equivalent to $O(m \log \Delta)$ time for regular graphs; refer to their paper for details [7].

Once a bin is almost solved, it can be completely solved in time $O(m \log \Delta)$. To do this, we repeatedly solve a sub-bin of the current bin, which contains $p$ perfect matchings, followed by a graph $B[j]$ with degree at most $p$. To solve this bin, we recursively split $B[j]$. If $B[j]$ has even degree, we perform a SPLIT-EVEN. If $B[j]$ has odd degree, we perform a SPLIT-ODD with one of the perfect matchings. We recurse on the two graphs we obtain from splitting $B[j]$ until our original sub-bin of $p$ perfect matchings and graph $B[j]$ has been reduced to $p + d(B[j])$ perfect matchings. We iterate this process until the entire almost solved bin has been reduced to perfect matchings. As long as we proceed from lower degree bins to higher degree bins, we will always have enough perfect matchings to handle odd-degree graphs which might arise, by the almost solved properties described above. This algorithm is shown in Algorithm 1.

The time to solve a sub-bin which ends with graph $B[j]$ is $O(d(B[j])n \log d(B[j]))$, because adding the additional perfect matchings at most doubles the effective degree of $B[j]$, as compared to the case where $B[j]$ has a power of two degree, yielding the same asymptotic complexity. The total runtime to solve all sub-bins is then:

$$\sum_{i=0}^{\mathrm{length}(B)-1} O\left(d(B[i])n \log d(B[i])\right) = O(m \log \Delta)$$

Thus the runtime of the algorithm overall is $O(m \log \Delta)$.

## 5 Data Structures

Theoretically, we can assume that graph operations such as adding and removing edges and finding neighbors can be performed in constant time. However, in practice, the data structure that we choose to represent the graphs has a significant impact on the performance of our algorithm, and in some cases a structure with worse asymptotic complexity can yield faster runtimes in a real implementation. To implement the algorithm described above, we must be able to check if a node has any neighbors, find a neighbor of a node, and add and remove edges from graphs. In this section, we describe three different graph data structures that we implemented for this algorithm, and their asymptotic complexity for each of these operations.

### 5.1 Matrix

The first graph data structure that we implemented uses a matrix to represent edges. For a network with $n$ top of rack switches, we use an $n$ by $n$ matrix of 8-bit integers. The integer at location $[i][j]$ represents the number of edges from source $i$ to destination $j$.

This data structure is very simple to implement and to perform operations on, but has poor asymptotic complexity. Adding and removing edges to and from graphs can be performed in constant time. However, checking if a node has any neighbors can require $O(n)$ time, because the entire row or column must be scanned. Similarly, finding a neighbor of a node requires linear time. In addition, finding a neighbor of a destination node requires scanning a column (rather than a row) and has very poor locality, yielding poor cache performance.

## 5.2 Adjacency List

The second graph data structure that we implemented represents edges using adjacency lists. For a network with $n$ top of rack switches, we maintain a list of adjacent edges for $2n$ nodes: the $n$ senders and the $n$ receivers. For each node we store an array of edges, as well as the index of the first unused edge in the array (tail). Each edge includes whether this edge is currently in use or not, the number of the other vertex incident on this edge, and the index of this edge entry in the other vertex's list, so that given an edge in one adjacency list, we can find the other instance of this edge in the data structure in constant time.

This data structure is more complex than the matrix data structure (§5.1), but performs all of the required operations in constant or amortized constant time. In contrast to the matrix data structure, we can check if a node has a neighbor in constant time, using the tail index for this node. Edges can also be added to a graph in constant time, using the tail pointers of the two vertices. This approach works because our algorithm does not interleave add and remove operations; once we begin removing edges, we do not add any more edges to a graph, so we never need to fill in holes in the adjacency list caused by remove operations. We can simultaneously find a neighbor of a node and remove an edge to it in amortized constant time, using the tail pointer. We simply choose the last edge in this node's adjacency list. We can then find the entry for this edge in the other vertex's adjacency list in constant time. To remove this edge, we mark both of these edges as unused. Then, we update the tail pointer for each vertex by scanning backwards in the list until we find the previous valid edge, or reach the beginning of the edge list.

## 5.3 Bitmap Adjacency List

This final graph data structure also uses adjacency lists to represent edges, but does so in a way that allows us to perform all operations in constant time in practice. A single edge-color operation involves many graphs, but all edges utilize the same underlying graph structure; they only differ in the subset of edges that they involve. We leverage this fact to create a graph structure with two components, one which represents the graph structure, and one which indicates which edges of that structure are active in a particular graph. The structure component is very similar to the adjacency list representation (§5.2); for each of the $2n$ vertices, it stores a list of edges, where each edge includes the number of the other vertex and the index of this edge in the other vertex's list. The structure does not maintain a tail pointer or an indicator of whether each edge is in use or not. The edges component maintains a bitmap for each of the $2n$ edges, where a 1 in the bitmap indicates that the corresponding edge in the structure is included in this graph, and a 0 indicates that it is not included.

With this structure, all operations can be performed in constant time in practice. To check whether a node has any neighbors, we simply check if its bitmap is non-zero. We also must be able to find a neighbor of a given vertex, remove that edge from one graph $G_1$, and add the same edge to a second graph $G_2$. We can perform this operation all at once in constant time by leveraging a special hardware instruction. To find a neighbor of the given node, we must scan through the bitmap of this node to find an entry that is a '1'. Using the graph structure, we can find the corresponding entry for this edge in the other vertex's edge list. We remove this edge from $G_1$ by setting both bits for this edge to 0. We add this edge to $G_2$ by setting the corresponding bits to 1. The theoretical complexity of this operation is linear in the number of nodes, $O(n)$, because we must scan through the bitmap. However, most architectures offer *find first set* instructions which can find the first set (non-zero) bit in a bitmap using only one instruction.[1] Thus for graphs with degree of at most 64, this data structure allows us to perform all necessary operations in constant time. This structure has the additional advantage over the adjacency list structure that it can reuse one instance of the graph structure for all graphs in an edge-coloring, thereby requiring much less memory. This is particularly important for graphs that approach or exceed the size of the L1 cache.

## 6 Implementation Optimizations

All algorithms were implemented in C using static memory allocation and compiled will full optimization. We implemented two additional optimizations to improve the runtime of this implementation for all graph data structures.

### 6.1 Precomputation

For a given network topology, the sequence of SPLIT-ODD and SPLIT-EVEN operations which must be performed is fixed. This is because the sequence of operations depends only on the degree of the graph, $\Delta$, which depends only on the number of machines in each rack. Therefore, we can compute ahead of time the sequence of SPLIT-ODD and SPLIT-EVEN operations that must be performed during each EDGE-COLOR operation. For each such EDGE-COLOR operation, we can statically allocate an array of graphs to operate on. If we include in our precomputation which array indices should be the input and output graphs for each SPLIT-ODD and SPLIT-EVEN operation, our EDGE-COLOR operation simply needs to iterate through a series of steps, performing either a SPLIT-EVEN or a SPLIT-ODD operation at each step, on the specified indices of the array. Note that this precomputation is practical because one controller would typically only handle one specific network with a fixed topology; if the topology were to change this would be an infrequent occurrence, and this precomputation could be repeated for the new topology. This precomputation is done in Python.

### 6.2 Converting SPLIT-ODD to SPLIT-EVEN

We performed an additional speed-up by eliminating the need to explicitly compute SPLIT-ODDs. Consider a pair of graphs $G_1$ and $G_2$ on which we would need to call SPLIT-ODD. We can determine which of those graphs is generated last (via a preceding split), and have its edges added to the other graph during the split. For example, if $G_2$ was created after $G_1$ during the course of the edge-coloring, we can simply add its edges to $G_1$ during the split that created $G_2$. This avoids having to explicitly add edges from one graph to another, and appears to have improved performance by a small but noticeable constant factor.

## 7 Experimental Results

### 7.1 Experimental Setup

The approximate edge-coloring algorithm of Kapoor and Rizzi (§4.2.2) was implemented using the optimizations described above (§6) for each of the three graph data structures (§5). Experiments

[1]On the Intel 386 and later platforms, this instruction is bsf. Similar instructions exist for AMD (lzcnt), ARM (clx), MIPS (clz), and other platforms.

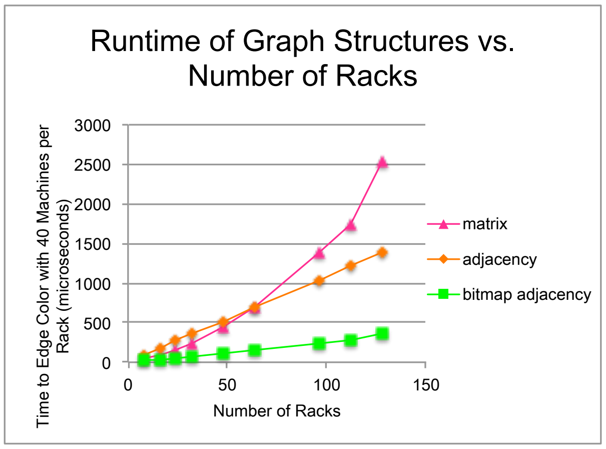

Figure 4: Runtime of the three graph structure implementations for different numbers of racks in the network.

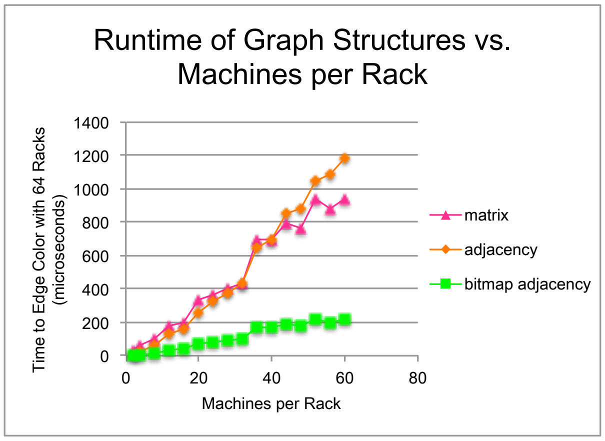

Figure 5: Runtime of the three graph structure implementations for different numbers of machines per rack in the network.

were run on a machine with a 2.7 GHz Intel Core i7 processor. This processor has 4 cores, each of which has 32 KB of L1 cache and 256 KB of L2 cache [1].

For each data point reported, we ran 100 experiments and reported the average runtime. The series of splits to be performed for each network topology was precomputed as described in Section §6.1 and only the total runtime of the splits was reported (not the precomputation time). For each experiment, a random regular bipartite graph with the appropriate degree and number of nodes was generated. The difficulty of edge-coloring depends primarily on the network parameters (machines per rack and number of racks) rather than on the traffic to be sent in a given timeslot. Thus we did not attempt to evaluate the performance of our algorithm on any specific input graph.

## 7.2 Results

In our first experiment, we compare the implementations of the three different graph data structures. Figure 4 shows the runtime of each implementation on a network with 40 machines per rack (degree = 40), as the number of racks (nodes in the bipartite graph) varies. Because each operation in the matrix implementation takes time linear in the number of nodes, the overall runtime of the algorithm is $O(nm \log \Delta) = O(n^2 \Delta \log \Delta)$, which scales quadratically with the number of top of rack switches. In contrast, the adjacency list implementations perform graph operations in constant time, yielding an overall runtime of $O(m \log \Delta) = O(n\Delta \log \Delta)$, which scales linearly with the number of racks. Though both adjacency list representations have the same asymptotic complexity, the bitmap adjacency implementation involves much smaller constants, because it uses simple bitmap operations (one instruction each) to find edges in the edge lists, whereas the first adjacency implementation must scan through the edge lists using while loops (several instructions). Therefore, the bitmap adjacency implementation's runtimes are much smaller, with the ratio between the two runtimes likely determined by the ratio of numbers of instructions used. It is also worth noting that for networks with fewer than 64 racks, the matrix implementation is actually faster than the first adjacency implementation; in some cases we might prefer to use an algorithm with worse theoretical complexity because it is faster in practice.

We also compare the performance of the three implementations as we vary the number of machines per rack in the network. Figure 5 shows the runtime of each implementation on a network with 64 racks (64 nodes in the bipartite graph) as we vary the number of machines per rack (degree

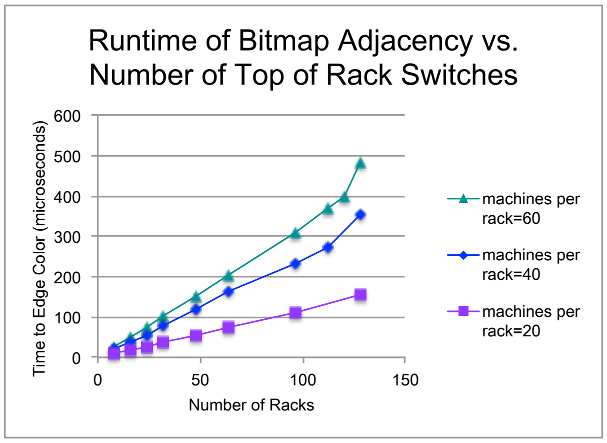

Figure 6: Runtime of the bitmap adjacency implementation for different numbers of racks and machines per rack in the network.

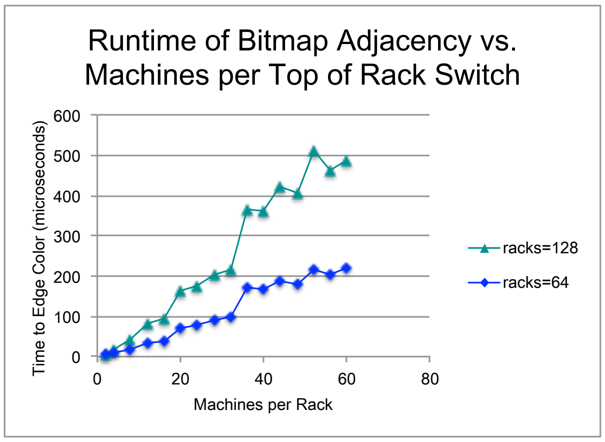

Figure 7: Runtime of the bitmap adjacency implementation for different numbers of racks and machines per rack in the network.

of vertices). The runtime of the graph operations for the three implementations are all independent of the degree of the graph. Therefore the complexity of the runtime as we vary the degree of the graph is simply given by the algorithms' complexity, which is $O(m \log \Delta) = O(n\Delta \log \Delta)$. Thus our runtimes overall should scale linearithmically with the number of machines per rack; this is the growth we see in Figure 5 for all implementations. In this case, the matrix and adjacency implementations demonstrate similar runtimes, while the bitmap adjacency implementation runs much faster.

We also observe that the runtime is not a smooth function; in some cases an increase in degree yields a large increase in runtime, while in other cases, an increase in degree yields a small decrease in runtime. This is likely due to the number of Euler splits necessary, based on the degree of the graph. If the degree is a power of two, then the runtime is exactly $m \log \Delta$, because each edge appears in exactly one Euler split for all power of two graph sizes from two to $\Delta$. However, for degrees that are not powers of two, some edges will be involved in more than $\log \Delta$ Euler splits, as they are re-used to help with odd-degree graphs. The precise number of splits necessary depends upon how many odd-degree graphs are encountered while splitting. This is why we see such a pronounced increase in runtime from 16 machines per ToRs to 20 and from 32 to 36, for all implementations.

Our first experiment demonstrated that the bitmap adjacency implementation runs the fastest regardless of the number of machines per rack or the number of racks in the network; in our second experiment we evaluate the performance of the bitmap adjacency implementation and consider what size networks this implementation could be applied to. Figure 6 shows the runtime of the bitmap adjacency implementation on networks with 20, 40, and 60 machines per rack, as the number of racks in the network varies. As in Figure 4, we see that the algorithm scales linearly with the number of racks. However, we note that for very large numbers of racks (ex: 128), the algorithm no longer seems to scale linearly. This is likely because the data structures begin to overflow the L1 cache at this point. For a network with 60 machines per rack and 128 racks, the data structure to store the graph structure requires about 32,800 bytes, and each data structure to store the active edges in a particular instance of the graph requires 2,048 bytes. This just begins to overflow the 32 KB L1 cache, incurring much longer latencies for some operations as they must access the L2 cache.

Figure 7 shows the runtime of the bitmap adjacency implementation on networks with 64 and

128 racks as we vary the number of machines per rack. As in Figure 5, we see that the runtime grows linearithmically with the number of machines per rack, and is not a smooth function.

Given our speed requirements, the performance of this implementation limits our centralized scheduling approach to certain sizes of networks. For a network with 1 Gbps links, we can allow up to 100 $\mu$s per computation (§2.2). For example, from Figure 7, we see that we could support a 1 Gbps network with 60 machines per rack and 32 racks (1920 machines), or a network with 40 machines per rack and 40 racks (1600 machines), or a network with 20 machines per rack and 88 racks (1760 machines). For a network with 10 Gbps links, we can allow up to 50 $\mu$s per computation (§2.2). Thus we could support a 10 Gbps network with 60 machines per rack and 16 racks (960 machines), 40 machines per rack and 21 racks (840 machines), or 20 machines per rack and 44 racks (880 machines). Choosing within these options depends upon the preferences of the network administrator. However, we see that in general our algorithm is fast enough to select paths for a 1 Gbps network with 1500-2000 machines or a 10 Gbps network with 800-1000 machines.

## 8 Conclusion

In this paper, we considered the idea of using a centralized scheduler to schedule traffic within a datacenter network, and implemented an algorithm to select the paths that packets should take through this network. We implemented the algorithm of Kapoor and Rizzi for edge-coloring bipartite graphs using three different data structures to represent the underlying graphs. We contribute a variant of the standard adjacency list graph representation that makes it more efficient, particularly when many graphs share the same underlying structure. Our results demonstrate that an adjacency list implementation generally performs better than a matrix representation, and that our enhanced bitmap adjacency list data structure yields an overall algorithm which is able to select paths in a datacenter network with 800-1000 machines with 10 Gbps links, or 1500-2000 machines with 1 Gbps links.

## Acknowledgments

This work was done as part of the FastPass research project, which is joint work with Jonathan Perry, Hari Balakrishnan, and Devavrat Shah. We thank them for their assistance, as well as Professor Karger for his well-taught Advanced Algorithms course.

## References

[1] Apple macbook pro “core i7” 2.7 15” early 2013 specs, 2013.

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[3] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *NSDI*, 2012.

[4] Richard Cole and John Hopcroft. On edge coloring bipartite graphs. *SIAM Journal on Computing*, 11(3):540–546, 1982.

[5] Richard Cole, Kirstin Ost, and Stefan Schirra. Edge-coloring bipartite multigraphs in o (e log d) time. *Combinatorica*, 21(1):5–12, 2001.

[6] Harold N Gabow. Using euler partitions to edge color bipartite multigraphs. *International Journal of Computer & Information Sciences*, 5(4):345–355, 1976.

[7] Ajai Kapoor and Romeo Rizzi. Edge-coloring bipartite graphs. *J. Algorithms*, 34(2):390–396, February 2000.

[8] Dénes König. Graphok és alkalmazásuk a determinánsok és a halmazok elméletére. *Mathematikai és Természettudományi Ertesito*, 34:104–119, 1916.

[9] Alexander Schrijver. Bipartite edge coloring in o($\delta$m) time. *SIAM Journal on Computing*, 28(3):841–846, 1998.