University of Southern Denmark

Speciale

Using Online Algorithms for Call Admission Control

Author:
Sebastian Eklund Larsen
<selar16>

January 1, 2022

Resume

I dette speciale er der blevet studeret online algoritmer i forhold til "call admission control" problemet. Specialet starter med at forklare den viden konkurrencedygtighed der vil blive brugt igennem specialet. Bagefter bliver der introduceret for "call admission control", og den viden om grafer der daner grandlag for problemet. Der er blevet gennemgået algoritmer, som har beviser for konkurrencedygtighed når de bliver brugt på specifikke typer af grafer. De typer af grafer som indegåri specialet er linjer, træer og gitre. Disse algoritmer er forsøgt at blive implementeret for at kunne sammenligne deres brugbarhed.

Contents

1 Competitiveness

An instance of a problem P consists of a set of inputs I and a cost function C. The feasible solutions to input I is defined as $F(I)$, and for feasible solution $O \in F(I)$ has a positive cost defined as the function $C(I, O)$. For an algorithm ALG which solves problem P, $ALG[I] \in F(I)$ is defined as the solution which ALG returns given input I, and $ALG(I)$ denotes the cost of the solution O of the algorithm on I.
In the space of possible algorithms we have the concept of an optimal algorithm OPT, which always finds the optimal feasible solution.

$$OPT(I) = \min_{O \in F(I)} C(I, O)$$

Algorithms are judged in their relation to OPT, such that for all inputs I an algorithm ALG is within $ALG(I)$ is within some bound of $OPT(I)$. The term for the relation between an algorithm ALG and OPT is called the competitive ratio. For an minimization algorithm we say that ALG is c-competitive if there exists constants c and α for which

$$ALG(I) \leq c * OPT(I) + \alpha$$

holds for all legal inputs I and $0 \leq \alpha$. When an algorithm is c-competitive we can also say it attains a competitive ratio of c.

1.1 Online and Offline problems

Problems can be framed in either an online or an offline way. For offline problems the solving algorithm ALG has access to the entirety of input I, and as such can explore the entirety of the feasible solutions $F(I)$ in order to find the cheapest solution. Even though ALG can explore $F(I)$ it may not be able to find the optimal solution as it may be infeasible to check every solution due to the size of $F(I)$.
The difference between an online and an offline algorithm is the access to input I, in online algorithms we do not have access to the entirety of I from

the start, and instead get parts of I in a sequence. These parts must be processed by the algorithm without any knowledge of the parts later in the sequence, or the length of the sequence.
For both online and offline problems we consider competitiveness against OPT, where OPT is not limited by online restraints.

1.2 Adversaries

One way to analyse online algorithms is to create an adversary. Adversaries will generate the input for the algorithm to increase the cost of the online algorithm, while simultaneously decreasing the cost of the offline optimal algorithm, in order to maximize the competitive ratio. For deterministic algorithms the adversary knows exactly how the algorithm will choose to do with each item in the sequence, but for randomized algorithms it does not know which choice will be chosen. The adversary will instead be trying to increase the expected cost of the algorithm.

1.3 Potential Function

Potential functions can be used to prove what competitive ratio an algorithm has. Given an algorithm and a request sequence, we can define an operation sequence. We can combine the operation sequences of the algorithm and the optimal algorithm, in any way such that any operation servicing request $j+1$ does not precede an operation which services j.

We can categorize the segments of the combined sequence into events, where event e_i is the segment of operations servicing request i.
Let S_{ALG} be all the possible states of the algorithm and S_{OPT} be all the possible states of the optimal algorithm. We now define potential function Φ which map states of ALG and OPT to a real number: $\Phi : S_{ALG} \times S_{OPT} \to \mathbb{R}$.
Given an event sequence $e_1, e_2, ..., e_n$, let Φ_i be the value after event i for each $i \in [1; n]$. We also have Φ_0 which is the potential before any events.
Two styles which use potential functions potential functions to prove competitiveness are through amortized costs and interleaving modes.

1.3.1 Amortized Costs

For amortized costs we count the change in potential as part of the cost incurred by the algorithm after a request. ALG_i is the direct cost of the i'th request. Let a_i be the amortized cost for event e_i in the event sequence $e_1, e_2, ..., e_n$, for algorithm ALG. Then a_i is defined as:

$$a_i = ALG_i + \Phi_i - \Phi_{i-1}$$

In order to prove that an algorithm has a competitive ratio of c, one must prove both of the following.

1: For every event e_i, $a_i \leq c * OPT_i$
2: there exists a constant b, where for all requests i, $\Phi_i \geq b$

1.3.2 Interleaving Moves

In interleaving moves the competitiveness can be proved using a potential function which holds for the following 3 constraints.
1: if only OPT acts during an event e_i and pays x, then $\Delta\Phi = \Phi_i - \Phi_{i-1} \leq c * x$. In other words the potential increases by at most $c * x$
2: if only ALG acts during an event e_i and pays x, then $\Delta\Phi = \Phi_i - \Phi_{i-1} \leq -x$. In other words the potential decreases by at least x.
3: there exists a constant b, where for all requests i, $\Phi_i \geq b$

1.4 Randomized Algorithms

Randomized algorithms are which can have varying results on the same input by using randomization to make choices. There are 3 types of adversaries when it comes to randomized algorithms. Oblivious adversaries which know the code of the algorithm but not the results of its decisions, and creates the input with no knowledge of those decisions. The entire input gets generated at the start and the adversaries pays optimally.
Adaptive online adversaries which can adapt to the choices of the algorithm, when a choice is made the next request will be generated with knowledge of

the algorithms decisions. This type of adversary must itself also service the request in an online manner, before serving it to the player.
Offline adaptive adversaries chooses the next input based on the algorithms decisions, but pays the optimal offline cost on the generated sequence.

The general requirement for a randomized algorithm ALG being c-competitive against an adversary ADV is that there exists an constant α for which the following holds:

$$E[ALG(\sigma) - c * ADV(\sigma)] \leq \alpha$$

Meaning that the expected cost of ALG is at most the expected value of ADV multiplied by c plus the constant α.

For oblivious adversaries we view the competitiveness in relation to an algorithms expected cost, on the distribution of possible input sequences. Since the input is generated before any decisions, the adversary's cost is just the optimal offline cost, and is not a random variable. Because of this, the competiveness of against oblivious adversaries can be simplified as:

$$E[ALG(\sigma)] \leq c * ADV(\sigma) + \alpha$$

For both of the other types of adversaries the adversary's cost is a random value due to the input sequence being a result of the randomness in the player algorithm.

2 Call Admission control

Call admission control[1] is a routing problem that appears in capacitated graphs, when we want to connect pairs of nodes along a path in the graph. These pairs are specified in calls, which along with two nodes specify the profit of the call together with bandwidth and duration requirements. Call admission control then pertains to the problem of accepting calls to achieve maximum profit.

There are special instances of the call admission control problem which apply different variations on the definition of the problem. These variations can be having permanent calls, being able to preempt previously accepted calls, the profit gained from accepted calls, and having single unit capacity edges and single unit bandwidth calls.

2.1 Definition

We have a graph $G = (N, E)$ with nodes N and and edges E. Each edge has a capacity for the bandwidth which can run through it. We identify the edge connecting nodes a and b as $e_{a,b}$ or equivalently $e_{b,a}$
Calls are defined as $r_i = (s_i, t_i, b_i, p_i)$, where r_i denotes the i'th call. From this definition $s_i, t_i \in N$ are the starting node and ending node respectively. b_i describes the required bandwidth of the call. And p_i is the profit of accepting the call.

2.2 Graph classes

The structure of graphs can be used to group them into classes. One can use the known aspects of these classes, in order to pick an algorithm which has a good competitive ratio when run on those classes. In this speciale we look at the classes line, tree and mesh. Some aspects which can be used to describe these classes are connectivity, edge expansion and routing number.

2.2.1 Connectivity

Connectivity is used to describe connected graphs, where there exists a path between any pair of nodes. There exists two ways to measure the degree to which a graph is connected, these being edge connectivity and vertex connectivity. Edge connectivity is defined as the minimum number of edges one would need to remove from the graph, to make it so there exists pairs of nodes for which no path between exists. When the number of edges to delete is k we say that the graph is k-edge connected.
For vertex connectivity we measure connectivity as the minimum number of vertices to delete from the graph, so there exists pairs of nodes in the graph with no path between them. When the minimum number is equal to k, we say that the graph is k-connected.

2.2.2 Edge Expansion

For a graph $G = (V, E)$ with a set $S \subset V$ and $|S| \leq |V|/2$, we can classify the graph based on the set of edges going between S to V/S. We then denote a given graph as a β-expander, where $|out(S)| \geq \beta|S|$.
In order to identify the β of a class of graphs we attempt to find the set S for which the ratio $\frac{|out(S)|}{|S|}$ is as small as possible.

2.2.3 Routing Number

The routing number describes the minimum congestion on of a permutation routing problem on the graph.
We define C as the maximum congestion on an edge given a collection of paths, the congestion of an edge is equal to the amount of paths using that edge. D is the dilation of the path collection, which is the maximum length among any of those paths. Let S_n be the set of all permutations of $\{1, ..., n\}$, For any permutation $\pi \in S_n$, and any D at least the diameter of a graph G. $C(G, D, \pi)$ is the minimum possible congestion of a routing on paths $p_1, p_2, ..., p_n$. Here each p_i is a path from node i to $\pi(i)$, which length does not exceed D.

The routing number is the upper bound for the minimum congestion of all the permutations. The routing number can be found when we have a graph G and a dilation D. The definition goes as follows:

$$R(G, D) = \max_{\pi} \max\{C(G, D, \pi), D\}$$

When we do not have a bound for the length we instead describe the routing number as $R(G)$, which is equivalent to $\min_D R(G, D)$.

2.2.4 Line

The line class is the class of graphs where you have a set of vertices $v_1, v_2, ..., v_n$, and the set of edges $E = \{e_{i,i+1} \mid i \in \{1, 2, ..., n-1\}\}$. This class has the aspect where for every pair of vertices, there only exists a single path between them. Its edge expansion is $\Theta(1/n)$, since we can take an end node as S, for which there is only one edge going out. Any set of nodes containing an end node, which can be used to form a path on the line will have the one outgoing edge. The routing number is $\Theta(n)$ because there is a permutation routing where every node is routed to a node on the other half of the line. This means that every path intersects at the middle edge, and we have a path for every node. The line class of graphs is 1-connected for both nodes and edges.

2.2.5 Tree

A tree graph has a set of N vertices $v_1, v_2, ..., v_n$, and set of edges E. The edges have to be such that the entire graph is connected, but no edge may form a cycle on the graph. This means that for every pair of nodes there only exists one valid path. This also means that this class is 1-connected.
Since there only is one valid path for all pairs, there is a permutation routing where all path routings will go through some middle node, but it is possible that not every path has to use a shared edge. This is because the middle node could have multiple edges, of which any path would only use two of. The specific routing number of a tree will therefore be graph dependent, but

the upper bound would be that of the line $\Theta(N)$, and the more subtrees the middle node is connected to, the smaller this number becomes, down to a minimum of 2.
The expansion has the same caveat, since the structure of the tree can vary. In the minimum case it approaches the same expansion of the line. In the maximum case where all nodes are connected to a single node the expansion approaches $\Theta(N)$.

2.2.6 Mesh

A mesh is a graph which has the nodes set up in a grid structure, and adjacent nodes have edges between them. It can have any kind of rectangular structure, but the one considered in this speciale is the square of N nodes, with dimensions $\sqrt{N} \times \sqrt{N}$. This class of graph is 2-connected due to its corners having 2 edges.
The routing number for a mesh is $\Theta(\sqrt{N})$. This can be seen from the routing where we split the square up into 4 squares, and have each node routed to a node in the opposite corner. These squares contain $(\sqrt{N}/2) * (\sqrt{N}/2)$, so counting the incoming and outgoing paths from the opposite square, there would be $\frac{N}{2}$ paths with an end in this square. The half of the paths from the other pair of squares would also be routed through this square, which results in another $\frac{N}{4}$, but these go both in and out of the square so they are counted twice. From this we can see that a corner square has a total of N entries and exits of paths. The corner square has $\sqrt{N}$ edges on its border, which the paths need to be routed on, and if the paths are perfectly distributed we get a maximum congestion of $\frac{N}{\sqrt{(N)}} = \sqrt{N}$.
The mesh has an expansion of $1/\sqrt{N}$, this due to the fact that we can choose a wall as S, so we would have $\frac{\sqrt{N}}{\sqrt{N}}$ expansion from this set. If we then include the line next to the set we increase the size, but the outgoing edges stays the same, and we get the value $\frac{\sqrt{N}}{2\sqrt{N}}$. We continue until half the nodes are in the set $\frac{\sqrt{N}}{\sqrt{N}^2/2}$, and this shows that the expansion of this class is $\Theta(\frac{1}{\sqrt{N}})$.

2.3 Problem instance

In this speciale we consider call admission control as an online problem where we start with complete information of the graph, and receive calls in a specific sequence. When a call is received the call must either be accepted or rejected without any knowledge on the rest of the sequence. This also means that we do not know the length of the sequence. We do not consider duration as part of this problem, and accepted calls are all considered permanent, so once a call is accepted it stays in the solutions until the end. We also have that all edges have capacity 1, all calls have a bandwidth of 1 and a profit of 1. This makes the problem equivalent to finding the maximum size set of edge disjoint paths on the graph, from the paths which can be generated from the call sequence.

2.4 Solution algorithms

Given an instance of the call admission control problem, we want to find the largest set of calls of calls among the requests, which can be routed on the graph, without violating edge capacities. We have the limitation that the problem is online so we must make decisions in the sequence of the requests. We want to find a set of calls, that is as close to the size of the set given by the optimal offline algorithm as we get. In order to do this we must use online algorithms whose performance is based on their competitive ratio against the optimal offline algorithm.

These algorithms have two phases to them. The first is the reprocessing phase where they process the information of the problems graph, and initialize the structures specific to that algorithm. The second is the routing phase, which receives a single call, and the algorithm either finds a path for which it will accept the call, otherwise it rejects it.

From the section about graph classes we saw that there are different properties on different graph classes. An algorithm can use this information to help on how to decide which calls to accept. Because of this many algorithms are designed to be used on a specific class of graph, and cannot be used on graphs not of that class. And the proof for their competitiveness is based

on the structure of that class of graphs. Some algorithms might be designed for average case, where the request sequence is generated by an oblivious adversary, while others may be designed with another type of adversary in mind. So the choice of algorithm is also impacted by the expectation of the request sequence.

2.5 Graph Implementation

Now that we have defined the definition of call admission control and explained how algorithms uses the graph, we can implement the base structure for which algorithms will interface with. The graph will have both a reference to all the nodes in the graph, and a reference to all edges in the graph. Nodes are identified by an id, and edges are identified by the pair of node ids which are connected with the node. The structure will have functions for finding a node, or edge given node ids.
Every node contains a list of the adjacent nodes. Edges contain knowledge on their capacities, and a list of paths which run through it. Edges also have a method for finding the total bandwidth of the paths which run through the edge. Paths contain the route which forms the path, and the bandwidth required by the path.
This covers the basic structure of a capacitated graph. When we have chosen an algorithm, and it receives a call, it tries to find an acceptable path for that call. The path it finds will then be added to the graph, by adding it to the path list of every edge along the path.

3 AAP

AAP is an algorithm defined by Awerbuch, Azar and Plotkin[1][p.228], which works on arbitrary capacitated networks. It focuses on maximizing throughput, and therefore uses it's own measurement for the profit of calls. It defines an exponential edge cost function, which determines the cost of an edge on a path. It also defines a bound on the cost of a legal path. When a request is received it will try to find a legal path, and if it can it accepts by applying the bandwidth of the call to the edges of the legal paths. Since this algorithm does not use randomization it is a deterministic algorithm.

3.1 Terms

$u(e)$ is the capacity of edge e. $L_j(e)$ is the total normalized load on edge e after request j, defined as $L_j(e) = \frac{1}{u(e)} \sum_{k \in A_j; e \in P_k} b_k$. A_j is the set of the first j calls in the request sequence. We define $c_j(e)$ to be the exponential edge cost of edge e such that $c_j(e) = u(e) * [\mu^{L_j(e)} - 1]$. μ is a constant defined as $2^{1+1/\epsilon}D$ where D is the maximum length of any path, which can be up to the number of nodes in the network. ϵ is chosen so $0 \leq \epsilon \leq 1$ holds. For a call $r_i = (s_i, t_i, b_i, p_i)$, the profit p_i is redefined as $D * b_i$.
For each call we also want

$$b_i \leq \min_e \frac{u(e)}{\epsilon \log \mathrm{D} \ + 1 + \epsilon} = b^*(\epsilon)$$

such that for a fixed ϵ the following holds:

$$b_i = O(\frac{u(e)}{\log \mathrm{D}})$$

Theorem 1 (Theorem from textbook [1][p.229])Under the assumption that we can find an assignment of ϵ, for which all $b_i \leq b^*$. And the assumption that all calls r_i costs $p_i * D$. The algorithm attains a competitive ratio of

$$2^{1+1/\epsilon} \log \mu + 1 = O(2^{1/\epsilon} + 2^{1/\epsilon} \log D)$$

Proof Let A be the set of calls accepted by aap, and $\overline{A}$ be the set of calls rejected by aap, but routed on some path in *opt*. We consider the sum of exponential costs $C = \sum_e c_n(e)$, and if the following two inequalities hold, we can prove the theorem.

$$C \leq (2^{1+1/\epsilon}\, log\, \mu) \sum_{j \in A} p_j$$

$$\sum_{j \in \overline{A}} p_j \leq C$$

This is due to the following two equations.

$$aap(\sigma) = \sum_{j \in A} p_j$$

$$opt(\sigma) \leq aap(\sigma) + \sum_{j \in \overline{A}} p_j$$

The first of the inequalities is proved by use of induction. We say that r_k is the last accepted call, with P_k as the routing. We want to show that the acceptance of a given call results in both of the following inequalities holding.

$$\sum_e c_{k-1}(e) = 2^{1+1/\epsilon}\, log\, \mu \sum_{j \in A-k}$$

$$\sum_e c_k(e) = 2^{1+1/\epsilon}\, log\, \mu \sum_{j \in A}$$

In order to do that we need to show that the addition of the call, results in a in a value that is bounded as follows:.

$$\sum_e c_k(e) - \sum_e c_{k-1}(e) = \sum_{e \in P_k} (c_k(e) - c_{k-1}(e)) \leq 2^{1+1/\epsilon} * p_k * log\, \mu$$

For every edge $e \in P_k$, we have the following bound for the change in the value of the exponential cost function.

$$c_k(e) - c_{k-1} = u(e)[\mu^{L_{k-1}(e)+b_k/u(e)} - \mu^{L_{k-1}(e)}]$$

$$= u(e) * \mu^{L_{k-1}(e)}[\mu^{b_k/u(e)} - 1]$$

$$\leq u(e) * \mu^{L_{k-1}(e)} \, [log\, \mu * \frac{b_k}{u(e)}] \, 2^{1/\epsilon}$$

$$= 2^{1/\epsilon} * \log\, \mu * [\frac{c_{k-1}(e)}{u(e)} + 1] * b_k$$

$$= 2^{1/\epsilon} * \log\, \mu * [\frac{b_k}{u(e)} * c_{k-1}(e) + b_k]$$

When we sum all edges on the path we get the following inequality:

$$\sum_{e \in P_k} [c_k(e) - c_{k-1}(e)] \leq log\, \mu \sum_{e \in P_k} [\frac{b_k}{u(e)} c_{k-1}(e) + b_k] * 2^{1/\epsilon}$$

$$= log\, \mu \, (\sum_{e \in P_k} \frac{b_k}{u(e)} c_{k-1}(e) + \sum_{e \in P_k} b_k) * 2^{1/\epsilon}$$

We can find another bound by substituting the first summation with p_k, since r_k is accepted and that is the definition of what the cost may not exceed.

$$\leq log\, \mu \, (p_k \sum_{e \in P_k} b_k) 2^{1/\epsilon}$$

$$\leq log\, \mu \, (p_k + D * b_k) * 2^{1/\epsilon}$$

$$= log\, \mu \, * 2p_k * 2^{1/\epsilon}$$

This proves the upper bound on C.

We now go on to prove the lower bound on C
Given a call r_j which is accepted by *opt*, we define P_j^* to be the path used to route it. For any call r_j not accepted by *aap*, but accepted by *opt* we know that there is no path for which the acceptance criteria is true, so P_j^* must be a path which exceeds that.

$$p_j < \min_{s_j - t_j \text{ path } P} \sum_{e \in P} \frac{b_j}{u(e)} c_{j-1}(e) \leq \sum_{e \in P_j^*} \frac{b_j}{u(e)} c_{j-1}(e)$$

We use this fact to prove the bound on C, by bounding the sum of profits for calls in $\overline{A}$. We can further bound this for $c_n(e)$, where n is the last request, since c is a non decreasing function. Then we have the sum of all edge capacity ratios multiplied with $c_n(e)$. The ratio is a number in $[0, 1]$, which means that it is upper bounded by the sum of $c_n(e)$ for all edges, which is also the same as C.

$$\begin{aligned}
\sum_{j \in \overline{A}} p_j &< \sum_{j \in \overline{A}} \sum_{e \in P_j^*} \frac{b_j}{u(e)} c_{j-1}(e) \\
&\leq \sum_{j \in \overline{A}} \sum_{e \in P_j^*} \frac{b_j}{u(e)} c_n(e) \\
&= \sum_{e} \sum_{j \in \overline{A}\,;\, e \in P_j^*} \frac{b_j}{u(e)} c_n(e) \\
&= \sum_{e} c_n(e) \sum_{j \in \overline{A}\,;\, e \in P_j^*} \frac{b_j}{u(e)} \\
&\leq \sum_{e} c_n(e) \\
&= C
\end{aligned}$$

□

3.2 Algorithm

Have D be a bound on the length of routing paths, like the number of nodes in the network. And assign ϵ such that $0 \leq \epsilon \leq 1$, and for each call r_i, $b_i \leq b^*(\epsilon)$ holds. Then we can assign $\mu = 2^{1+1/\epsilon} D$

When receiving j'th call r_j the algorithm tries to find a path between s_j and t_j for which the following holds:

$$\sum_{e \in P} \frac{b_j}{u(e)} * c_{j-1}(e) \leq p_j$$

If it does find such a path then it accepts the request by including the requests b_i on edges along the path. It also updates $c_j(e)$ for the affected edges.

3.3 Implementation

The routing of a call r_i is implemented via breadth first search, starting from node s_i. For each node n we have a variable K_n for the shortest cost found to reach it, and a variable F_n for the node from which it was reached. These are initialized to ∞ and $None$ respectively. We have a queue which starts only containing s_i.
We then loop until the queue is empty. Each loop removes the first item n from the queue. For each neighbor m of n we check if adding the cost of the edge between n and m to K_n is still within the legal bound for a paths cost.

$$cost = K_n + \frac{b_j}{u(e_{n,m})} c_{j-1}(e_{n,m}) \leq p_j$$

If this holds and $cost \leq K_m$ we assign $K_m = cost$ and $F_m = n$, and add m to the queue.
Once the loop is finished we can check if F_{t_j} is $None$, if it is then we did not find a legal path for the call. If it is not $None$ then we can construct a sequence of nodes which creates a legal path. The sequence starts with node F_{t_j} and then recursively finds nodes in F from the previous node, until $None$ is reached.

Algorithm 1 AAP route

given call r_j
for $n \in N$ **do**
    $K_n = \infty$
    $F_n = None$
end for
$start \leftarrow s_j$
$K_{start} \leftarrow 0$
$list \leftarrow [start]$
while list is not empty **do**
    $current \leftarrow list.pop()$
    for neighbor in current.neighbors **do**
        $edge \leftarrow e_{(current,neighbor)}$
        $cost \leftarrow K_{current} + \frac{b_j}{u(edge)} * c_{j-1}(e)$
        if $cost \leq p_j$ and $cost \leq K_{neighbor}$ **then**
            $K_{neighbor} \leftarrow cost$
            $F_{neighbor} \leftarrow current$
            $list.append(neighbor)$
        end if
    end for
end while
$end \leftarrow t_j$
if $K_{end} == \infty$ **then**
    return None
else
    $parent \leftarrow F_{end}$
    $path \leftarrow [\quad]$
    $current \leftarrow end$
    while parent is not None **do**
        $edge \leftarrow e_{(current,parent)}$
        path.add(edge)
        $current \leftarrow parent$
        $parent \leftarrow F_{parent}$
    end while
    return path
end if

4 Line CRS

Now we introduce an algorithm which is designed with respect to a class of graphs, as defined by Allan Borodin and Ran El-Yaniv[1][p.238]. This algorithm is called *CRS* and is used with graphs of the line class, here CRS stands for classify and randomly select. The algorithm assigns all the possible calls into classes, and randomly selects a single class of calls. If the algorithm receives a call request which is a member of that class it accepts, otherwise it rejects it.

4.1 Terms

We need to define how each call is given a classification, this is obtained from the edges needed to satisfy the call, and since the graph is a line, there is only one possible path. Every edge in the graph is assigned a number representing its level. Given a line graph with N nodes and D edges. we create a list of the edges such that they are ordered the same way as in the line. Such that neighbors in the list share an endpoint. The level of the edge is then defined by its placement in that list.
The middle edge of the list is assigned the level of 1. From the list we create two other lists, one contains the edges preceding the middle edge, and the other contains the edge following it. We then assign the middle edges of these list the previous the level of the earlier list incremented by 1. We recursively do this until the lists are of length 1. From each list at most 2 new lists are formed and each list represents the removal of an edge, so there is at most $log_2(D)$ levels. In figure 1 a representation of the process is illustrated. we see that it forms a binary tree.
Using the levels of the edges, we can get the level of a call. This is done by taking the minimum level among all edges, on the path associated with the call.

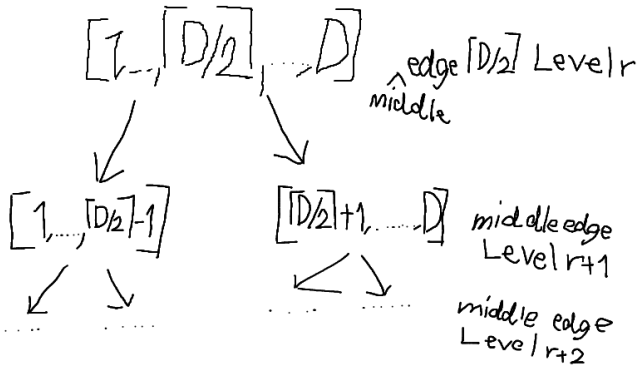

Figure 1: Recursive edge level assignment: Middle edge is assigned a level, afterwards remove the middle and split the list in two. then increment the level to assign to the new middle edges

4.2 Algorithm

CRS starts by having the size of the line N, and chooses a number from $\{1, 2, ..., log(N)\}$. It will then accept calls of the class associated with that number and reject all calls not of that class.

Theorem 2 *CRS* is $\lceil log(N) \rceil = \lceil log(D+1) \rceil\}$-competitive, for the on-line problem of finding disjoint paths on a the line class. Theorem is from reference[1][p.239]

Proof When we only accept calls of one level, we can accept at most an amount equal to the number of edges assigned that level. From the binary tree it can be seen that there are at most $2^{(i-1)}$ edges at level i, where the only level with less is the highest level.

When viewing the calls as having a level we have that when (crs) accepts a level i call then the optimal algorithm can also only accept one level i call. If level i is chosen, let c_i denote the calls crs would accept. And let o_i represent the number of calls opt accepts from level i. since opt cannot accept more

at the chosen level we have that $c_i \geq o_i$. and from this we can show that crs is strictly $log(N)$ competitive.

$$\mathbb{E}[crs(\sigma)] = \sum_{i=1}^{log_2(D)} Pr[\text{CRS chooses level i}] * c_i$$

$$\geq \sum_{i=1}^{log_2(D)} \frac{1}{log_2(N)} * o_i$$

$$\frac{1}{log_2(N)} * \sum_{i=1}^{log_2(D)} o_i$$

$$\frac{1}{log_2(N)} opt(\sigma)$$

□

Theorem 3 The lower bound of *CRS* competitive ratio against an oblivious adversary is $\frac{log(N)}{2}$. Theorem is from reference[1][p.240]

Proof The number of possible calls is C(N,2)= $\frac{N*(N-1)}{2}$.

At level i the amount of possible calls is

$$(\frac{N}{2^i})^2 * 2^{i-1} = \frac{N^2 * 2^{i-1}}{2^{2i}} = \frac{N^2}{2^{i+1}}$$

This means that $i+1$ has half the possible calls of i and as such half the probability of a call appearing of that level. The probability of a call being level i is :

$$\frac{1}{\frac{N*(N-1)}{2}} * \frac{N^2}{2^{i+1}}$$

$$\frac{2}{N*(N-1)} * \frac{N^2}{2^{i+1}}$$

$$\frac{1}{(1-1/N)*2^i}$$

$$\frac{2^{-i}}{1-1/N}$$

We then generate an input sequence from this probability, we pick a level i in $\{1, ..., log(N)\}$ with probability $\frac{2^{-i}}{1-1/N}$.
For l in 1 to i we generate 2^{l-1} edge disjoint level l calls.
The optimal algorithm will accept the edges at the lowest level. So it's expected value is the number of calls at a level multiplied by the probability that level was the last level for which calls were generated.

$$\mathbb{E}[opt(\sigma)] = \sum_{i=1}^{logN} 2^{i-1} * \frac{2^{-i}}{1-1/N}$$

$$> \frac{LogN}{2}$$

We have that if the calls of level i have been generated, then the probability that the calls of level $i+1$ is at most $1/2$. And the calls at level $i+1$ are twice that of level i. Level 1 is always generated and has 1 call. Level 2 is generated with half the probability of level 1 and has 2 calls. If we continue this then we see that for every level we can pick the expected value is at most 1. Therefore the expected value of CRS is

$$\mathbb{E}crs(\sigma) \leq 1$$

From this we can get the lower bound on the competitive ratio, which proves the theorem

$$\frac{\mathbb{E}opt}{\mathbb{E}crs} \geq \frac{log(N)}{2}$$

□

4.3 Implementation

When the graph is received we identify which edges are which ranks. We do this by by recursively assigning the middle edge of the line graph increasing levels. When an edge has been assigned a level the two lines on either side of the edge are processed the same way by going to the middle of them and

assigning the level to the middle edge. At run time we also pick a level for which the algorithm will accept requests. When a request is received we identify the level of the call by the lowest level edge on the path needed to accept it.

5 Bounded greedy algorithm

An algorithm by Petr Kolman and Christion Scheideler was presented in a paper[2]. The bounded greedy algorithm is an algorithm which accepts calls when there exists a legal path within a defined cost bound to satisfy it. We denote this bound by L. For each call we check if such a path exists, and reject the call if it does not.

5.1 algorithm

It defines the bound for the length of paths, and when a call is received it tries to satisfy it with a path within that length. If it cannot find a legal path within that length, the call is rejected.

Theorem 4 For any network G, with maximum degree $Delta$ and routing number R, the competitive ration of BGA with $L = 2 * R$ is at most $(\Delta + 4) * R + 1 = O(\Delta^2 a^{-1} logn)$

Proof B is the set of paths for the requests accepted by BGA, and O is the set of paths accepted by an optimal solution. In the case that O only contains paths of at most length L, the competitive ratio can at most be $L + 1$. This is due to the fact that any edge in a path from B can at most interfere with one path from O, since the paths in a solution are edge disjoint, and paths from B are at most of length L. This means that if we only assume that O consists of paths at most length L the following inequality holds $|O| \leq (L + 1) * B$

We describe $q \in B$ as a witness for a path p if both p and q share an edge on the network G. We describe short paths as those with maximum length L and long paths those that are longer. Any short path accepted by O but rejected by B must have a witness in B. Let $O' \subset O$ be the set of paths in the optimal solution which are longer than $2 * R$, which do not have a witness in B. Every other path in O have either also been accepted by B, or have a witness in B. As a result $|O - O'| \leq (1 + L) * |B|$, or more

specifically $|O - O'| \leq (1 + 2R) * |B|$
In order to bound the size of O' we convert the long paths into a set of short paths S. For any path which is a member of S it will hold that it has at least a witness in B and the paths in B are witnesses to at most $(\Delta + 1) * R * |B|$ paths in S. For each path in O' there will be a path in S of at most length L which connects the same vertices.
Given a long path $p \in O'$ which connects vertices s and t we create a short paths. We have that $a_{p,1} = s, a_{p,2}, ..., a_{p,R}$ denotes the first R nodes on the path, and $b_{p,1}, ..., b_{p,R-1}, b_{p,R} = t$ denotes the last R vertices on the path. We define a multiset of pairs between these nodes $\mathcal{L} = \bigcup_{p \in O'} \bigcup_{i=1}^{R} (a_{p,i}, b_{p,i})$. Because the paths of O' are edge disjoint each node will at most appear Δ times in $\mathcal{L}$. We can create a graph $H = (V, \mathcal{L})$, which can be colored with $\Delta+1$ colors such that no adjacent vertices has the same color. By using combinations of two color classes we can create $\lceil (\Delta + 1)/2 \rceil \leq \Delta + 2)/2$ permutation routing problems. The routing number R is defined as the maximum among all minimum congestions among permutation routing problems. This implies that the routing of pairs in $\mathcal{L}$ can be done with at most $(\Delta+2)/2) * R$ congestion, and dilation R. We denote by $\mathcal{P}$ a set of such paths connecting the pairs of $\mathcal{L}$, such that a path $l \in \mathcal{P}$ exists, which connects $a_{p,i}$ to $b_{p,i}$ for each i.
We then create a short path by picking an $i \in \{1, 2, ..., R\}$, which corresponds to the path from $a_{p,1}$ to $a_{p,i}$ from the original path, followed by $l \in \mathcal{P}$ which connects $a_{p,i}$ to $b_{p,i}$, and then $b_{p,i}$ through $b_{p,R}$ from the original path. This path has a length of at most $2 * R$, R vertices from the original path, and at most R vertices from the shortcut l.

We want to pick a subset $P' \subset \mathcal{P}$ such that each path in O' has a shortcut in P' and the paths in B at most witnesses $(\Delta + 2)/2) * R * |B|$ paths in P'. Each path has R shortcuts in $\mathcal{P}$ which can be chosen uniformly at random.

$$X = \{(l, q) l \in P', q \in B, \text{ q is a witness for l } \}$$

Every path P' must have a witness in B, therefore $|P'| \leq |X|$ for any P'.

Consider the following 2 definitions of variables:

For every $l \in P'$ let $v_l = |\{q | q \in B, q \text{ is a witness for } l\}|$
For every $q \in B$ let $w_q = |\{l | l \in P, q \text{ is a witness for } l\}|$

For these variables $\sum_{l \in P} v_l = \sum_{q \in B} w_q$ holds.
For every $l \in P$, X_l is the random variable equal to 1 if l is included in P'.
From

$$w_q \leq |q| * (\frac{\Delta + 2}{2} * R) \leq 2R * (\frac{\Delta + 2}{2} * R)$$

we can get

$$E[|X|] \leq E[\sum_{l \in P} v_l * X_l] = \sum_{l \in P} \frac{1}{R} * v_l = \frac{1}{R} \sum_{q \in B} w_q \leq (\Delta + 2) * R * |B|$$

From this we can see that there exists a P', where $|X| \leq (\Delta + 2) * R * |B|$. From the definition of P' it holds that $|P'| = |O'|$, and $|P'| \leq |X|$. This implies that $|O'| \leq (\Delta + 2) * R * |B|$.

Recall that $|O - O'| \leq (1 + 2R) * |B|$. When we combine this with our bound for $|O'|$ we get

$$|O| \leq (1 + 2R) * |B| + (\Delta + 2) * R * |B| = ((\Delta + 4) * R + 1) * |B|$$

Proving that BGA is $(\Delta + 4) * R + 1$ competitive.

□

5.2 Implementation

We get the routing number passed together with the graph. We bound the paths to be 2 times the routing number due the proof on competitiveness. When we receive a call a breadth first search is performed which tries to find a path which does not exceed the bound.

6 Tree CRS

This is an algorithm on tree graphs which uses the same technique as line CRS[1][p.241], where it chooses a class of calls which it will accept. This however assigns the levels to the nodes on the graph. The class of a call is based on the path which would be used to serve the call, and for trees there only exists one possible path for each pair of nodes.

6.1 Terms

In order to get the level of a call in the tree *CRS*, we identify a node which removal would result in subtrees which contain no more than $\frac{N}{2}$ nodes. This node is assigned as level 1, and for each subtree the process is repeated but with level incremented by 1. There are at most $log(N)$ levels. From the path of a call, the class is defined as the minimum level node among all the nodes on the path.

6.2 algorithm

At the start it picks a number among the levels of the nodes, this will be the class of calls it accepts. When it receives a call request it, it checks whether it is of the class selected at the start, and whether the path is free. If both of those conditions are true it accepts the call, otherwise it rejects it.

Theorem 5 *TreeCRS* is strictly $(2log(N))$-competitive on tree graphs.[1][p.241]

Proof We say that edges have the level of the minimum level node it is adjacent to. Any accepted level i call will only intersects at most a single level i node and at most 2 level i edges. So two calls of level i are either edge disjoint, or they share a level i node on their paths, and share one or two level i edges. This means that for every call accepted by the algorithm, the optimal algorithm may at most accept two calls that edge intersect, one for each of the at most two level i edges.
We can then proceed with proof similar to that of *crs*. c_i is the amount of level i calls *TreeCRS* accepts, and o_i is the level i calls accepted by *opt*.

The expected value of $TreeCRS$ is the summation of the multiplication of each levels probability and number of calls accepted at that level c_i.

$$\mathbb{E}[tcrs(\sigma)] = \sum_{i=1}^{log_2(N)} Pr[\text{TCRS chooses level i}] * c_i$$

For a given level *opt* at most accepts the same number of calls that $TreeCRS$ will accepts. there is a $\frac{1}{log(N)}$ chance that we pick a level. And for each call accepted by $TreeCRS$, there are at most 2 calls accepted by *opt*. Which gives us a lower bound on the expected value of the algorithm.

$$\mathbb{E}[tcrs(\sigma)] \geq \sum_{i=1}^{log_2(N)} \frac{1}{2log_2(N)} * o_i$$

$$= \frac{1}{2log_2(N)} * \sum_{i=1}^{log_2(N)} o_i$$

$$= \frac{1}{2log_2(N)} opt(\sigma)$$

□

6.3 Implementation

In order to do classify and randomly select on trees we start by ranking the nodes. The way we do that is by starting with every leaf moving inward, simultaneously moving inward while nodes have at most degree 2, keeping track of how many nodes is seen by each entity. When a node with more than that is encountered, we know it is connected to multiple subtrees. Given such a node with degree d, if d-1 entities are waiting at that node, we can combine them and their number into one. If we make sure that it is always the smallest entity that moves at a time, we make sure that the node for which they all end up at at the end is the one which is deepest in the tree. This node is then assigned the level, and removed from the set of nodes, we now recursively do the same to the subtrees by the removal.
We then pick a random number among the amount of levels created, and

later only accept calls from that level. The class of graph is tree, so the paths between nodes are still singular, which makes routing trivial. When a call is received we consider it to be the level of the lowest level node on its path, which we find by iterating through it, where after we check whether to reject it.

7 Tree aap

We now introduce an algorithm for tree graphs, which makes use of the aap algorithm[1][p.243]. The algorithm makes a copy of the original tree with increased edge capacities. When a call is received we first try to route it on the copy, and if the copy accepts it then with some probability the original graph accepts too.

7.1 Terms

We say that the original graph is called T, and its copy is called T'. T' is created by copying the original graph, and changing all edge capacities to $2\,log(D)$, where D is the diameter of the graph. aap is run with $\epsilon = 1$.
Given a tree T and paths P, the intersection graph $(\mathbb{P}, E_{\mathbb{P}})$ is the graph for which each path is a node, and overlapping paths have an edge connecting them. A graph $G = (V, E)$is said to be d-inductive if its vertices $v_1, v_2, ..., v_n$ can be ordered in a way where for all i $|\{(v_i, v_j)|i < j \text{ and } (v_i, v_j) \in E\}| \leq d$.

7.2 algorithm

When a call is received the algorithm uses aap to route it on T'. If the the routing on T' is accepted, the call is considered a candidate. The algorithm accepts the same path for T with probability $\frac{1}{4(1+log(D))}$, if an overlapping path has not already been accepted.

Theorem 6 Tree aap is $O(log(D))$ competitive, for trees with diameter D

Proof For a request sequence we have that the number of candidate calls, is a value that scales more than the the number of calls accepted by *opt*

$$|\{r_i \mid \text{ where } r_i \text{ is a candidate}\}| = \Omega(opt(\sigma))$$

Since *opt* only has a fraction of the bandwidth this changes the inequality from the proof of aap to:

$$\sum_{j \in \overline{A}} p_j \leq \frac{C}{2 + \log D}$$

In order to prove the theorem it will be shown that:

$$|\{r_i | r_i \text{ is accepted}\}| \geq \frac{1}{8(1 + \log D)} |\{r_i \,|\, r_i \text{ is a candidate}\}|$$

We say that request r_i is a lucky candidate, if it wins the $\frac{1}{4(1+log(D))}$ coin toss. And we define the following two random variables $X = r_i | r_i$ is accepted and $Y = (r_i, r_j) | r_i$ and r_j are lucky candidate calls. Let $\mathbb{P}$ be the set of candidate calls.

$$|\{r_i | r_i \text{ is accepted}\}| \geq |X| - |Y|$$

The expected size of X is the amount of candidate calls multiplied with the probability that a call is lucky.

$$\mathbb{E}[|X|] = |\mathbb{P}| * \frac{1}{4(1 + \log D)}$$

The expected size of Y is the number of edges in the intersection graph for the set of candidate calls. The upper bound on the number of edges on an intersection graph is $2(g-1)$ where g is the maximum congestion on T'. This results in a maximum of $2(2 + \log\, D - 1) = 2 + 2 \log D$ edges.

$$\mathbb{E}[|Y|] = |E_{\mathbb{P}}| * \frac{1}{16(1 + \log D)^2} \leq |\mathbb{P}|(2 + 2 \log D) \frac{1}{16(1 + \log D)^2}$$

$$\mathbb{E}[|X| - |Y|] \geq \frac{\mathbb{P}}{8(1 + \log D)}$$

□

7.3 Implementation

We need to find the diameter of the tree. We do this by a similar tactic as the other tree algorithm, we start from all leafs and move inwards, keeping track of how many nodes we have traveled inwards. When we meet a node of degree d more than 2, we wait until d-1 entities are waiting. When enough entities are their we merge them by taking the maximum number among them and moving inwards on the node. Again we make sure that the smallest entities move first. Doing that they should all converge on the deepest node. From the deepest node we need only combine the two neighbors with the longest distance traveled to get the diameter of the graph.
Now that we have found the diameter, we can create a copy of the original graph with increased capacities. When we receive a call we run aap on said graph, and if it accepts we use randomization to choose whether to accept it in the original graph. The path is also only accepted if we have not already accepted a conflicting path.

8 Mesh Algorithm

We introduce an algorithm defined by Jon Kleinberg and Eva Tardos [?] This algorithm is designed to be used on graphs which have a the shape of an $n \times n$ square mesh. The idea is to separate the mesh into blocks, and then creating a graph where each nodes represents a block and there are edges between adjacent blocks. These edges will have a capacity based on the size chosen for the blocks. Then the AAP algorithm is used to route on this new graph. The result of this algorithm gives a sequence of blocks, which is converted into a path on the original graph. The conversion has as an objective to route a path from the starting node to a vertex on the blocks boundary. We then route from that vertex to a vertex on the boundary of the next block in the sequence. We repeat until we enter the final block in the sequence, where we finish by routing to the terminating vertex inside the block.

8.1 terms

Given a mesh $G = (V, E)$, $G[x, y]$ denotes the vertex at row x and column y. When we want to describe a sub square in the graph refer to $G[x : x', y : y']$ as the sub square containing the following set of vertices:

$$\{G[p, q] : x \leq p \leq x', y \leq q \leq y'\}$$

The distance between two vertices on the mesh can be represented by the manhattan distance between their coordinates, which defines the distance function as $dist(G[x, y], G[x', y']) = |x - x'| + |y - y'||$.

8.1.1 simulated network

We build a simulated network from graph G, by considering it divided into $\gamma \log n \times \gamma \log n$ blocks, where $\gamma > 1$. These blocks are referred to as γ-blocks. A sub square of a graph is a γ-block if it adheres to the following

definition for some natural numbers i and j.

$$G[(i-1)\gamma \log n : (i)\gamma \log n, (j-1)\gamma \log n : (j)\gamma log\, n]$$

The vertex at the center of such a γ-block is the vertex found in the middle of that block.

$$v = G[(i - \frac{1}{2})\gamma \log n, (j - \frac{1}{2})\gamma \log n]$$

When a block is centered at v, the block is denoted as C_v. In regards to a block, some of its vertices are described as boundary vertices, which together form walls. These are the vertices which are at the maximum or minimum row or column. A wall is defined as a set of boundary vertices which all share column or row number.We describe the rest of the vertices in a block as internal vertices. Blocks which share boundary vertices are considered neighbors. Given a block X, let X^* be the sub square representing the union of X and its at most 8 neighbors.

V' is the set of vertices which are centers for all γ-blocks in the graph. We now build a graph G' with these vertices. In this graph we create edges between $v, u \in V'$ if C_v^* and C_u^* intersect at internal vertices. A randomized version Luby's maximal independent set algorithm is run on G' to find a maximal independent set of vertices. We find this set by having each vertex pick a random number between 1 and j. Vertices which have a higher number than all of their neighbors are added to the independent set, while those neighbors are removed from consideration. We repeat this until every node is either in the set or removed from consideration. It can take multiple iterations because there is a chance of ties, but by picking a big enough j, those ties become less likely.
We now have a maximal independent set $M \subset V'$. For any of the blocks in the graph C_v^* would at most intersect internally with 24 other C_u^* for $u \in V'$. This means that the probability of v being both $v \in V'$ and $v \in M$, is $\frac{1}{25} - o(1)$

For $v, u \in V'$ where $dist(v, u) \geq 11\gamma \log n$, the probability that they both

appear in the maximal independent set is constant. This is due to them not sharing any neighbors in G'

We can now create enclosures of vertices from $v \in M$. We consider C_v^* as the base for the enclosure of v, which we refer to as D_v. We do this for every vertex in the maximal set, but this still leaves some blocks which does not get represented by any C_v^* for $v \in M$. We want to have it so that all vertices of G belong to some D_v. Due to the maximality of M we have that the blocks not inside any enclosure must share a boundary with some C_v^* for some D_v. When this happens one adds the block to an arbitrary enclosure for which it shares a boundary vertex. Now D_v is a union of blocks and may no longer be a square. And the union of all enclosures D_v contains every vertex of G.

Let $\mathcal{N}$ be a network containing the vertices of M. Where vertices $v, u \in M$ share an edge if D_v and D_u share a wall in some γ-block. The blocks added to enclosure D_v can at most be blocks contained in the 5×5 block centered at C_v. This means that the enclosure can at most share a wall with 20 γ-blocks outside of D_v. And therefore the degree of vertices in $\mathcal{N}$ is at most 20.

If we set the capacity of the edges connecting $u, v \in \mathcal{N}$ to be equal to the number of vertices on walls shared by γ-blocks of D_u and D_v, we would have edges with capacities of $\Theta(log\, n)$ multiplicity. When we receive a call to connect some s_i and t_i, we want to identify the enclosure which they belong to, and use a routing algorithm on $\mathcal{N}$ which returns a series of enclosures from which we want to generate a path between s_i and t_i in G.

In order to generate paths we define the term v-rings. Given a cluster C_v^* we have $\frac{1}{2}\gamma\, log\, n$ v-rings. These v-rings are the cycles of the inner half of C_v^*/C_v, if the cluster is on the wall of the graph then it will be the paths in said inner half. Given two v-rings from the same cluster R' and R, we say that R' is inside R if the distance from $R'$to v is less than the distance

from R to v. We define the distance of a v-ring to the center of its cluster through L_∞ of any vertex on the ring.

If we have an $X \subset V$, we say that $\delta(X)$ are the edges leaving $\pi(X)$ are the vertices incident to those edges. $\delta(X, Y)$ are the edge going between X and Y. For neighboring enclosures we want to pick $p\,log\,n$ edges. We also want to pick $p\,log\,n$ edges going out of the enclosures center block. we want it such that these edges for a given enclosure sum to less than $\gamma\frac{1}{2}\,log\,n$, so we can associate a different v-ring with each of them. We then associate a path between the v-ring and the edge with them. For a pair of v-rings, ones path must intersect a node of the other, so we can combine if we want to make a path between two of our chosen edges, it could be routed in the union of the two v-ring path unions.

8.2 Algorithm

The algorithm has two modes, one for routing long paths and one for short. A path is said to be short if the distance is shorter than $16\gamma logn$. The algorithm randomly decides whether to service long or short paths at the beginning.

8.3 Long routing

When a call is received during long routing, we first check whether it begins and ends in center blocks for some enclosure, and then route it with aap on the simulated graph. If it accepts we get a sequence of enclosures for which we choose edges between, whereafter we combine the v-ring paths as stated in the previous section, then for both ends we use an edge going out of their center cluster, and add the v-ring path to the route. Then for both sides we route from the chosen edge to the destination nodes.

8.4 Short routing

We choose to set $r = 32\gamma logn$ and use *Luby*'s maximum independent set algorithm where to find a maximum set where $L_\infty(u, v) > 4r$. For a u from this set we say that X_u is the set of nodes within an L_∞ distance of r, Y_u

the the nodes within a distance of $2r$.
When we receive a call with both ends in an X_u, we try to route them in Y_u. We keep track of the number of free edges in Y_u m. We always route the shortest path within the subgraph, we only accept paths whose distance is $\leq \sqrt{m}$

8.5 Implementation

We first start by loading the graph into a square array. We do this by identifying the corners of the mesh, and for two corners which share a wall we find distances to all nodes. With these two sets of distances we can find the position of all the other nodes. This is because the nodes of a mesh exists in a geometric space. We have that both of these corners share one axis coordinate, but are offset on the other. The distance to a node can be defined as the sum of x and y distance. For the two corners, one of these distances will be equal, and the other will be below the distance between the corners. If we say that y is the axis they are equal, then we also have that the sum of x distances to the node is equal to the x distance between the two corners. So by having 2 distances to a node, we need only find the point between them where the remainder of the distance is equal, and then we have both the x and the y coordinate.

I did not manage to properly implement this algorithm

9 Testing

The information which we want to inspect in this speciale is how the size of the graph, and the length of the request sequence affects the performance of the algorithms. Since the material has been focusing on the competitive performance of the algorithms this testing will be focusing on those aspects too. And therefore will not comment much on run time.

9.1 Line

9.1.1 Graph size

For the line we test how graph size impacts the bga, aap and crs algorithms, by testing different sizes of graph from the line class, with a request sequence of length proportional to the size. The instances tested on will be on lines of 200 node increments up to 2000, each with running with 30 different call sequences of length equal to number of nodes. That size of call sequence chosen because it is just above the maximum amount of calls you could possibly accept.
the results can be seen in figure 4, we see that both aap and bga have a higher growth rate than crs. We see that crs performs poorer, but this is because it was designed to work against an adversary, who would create a bothersome sequence of calls. If reordered the call sequence such that the distant pairs were all requested irst it would impair the two greede algorithms but crs would accept the same number of calls as otherwise. The number of calls do affect probability of some calls existing, and because of this it may in the average case become better for crs to pick a higher level for its acceptance in order to get more profit.

9.1.2 Requests length

The request size was tested by fixing the graph to 1000 nodes then, then at each phase the algorithms where tested in 100 call increments up to 1000. The results can be seen on figure 8.
We see that they all trend upwards. Crs increases more steadily and this is

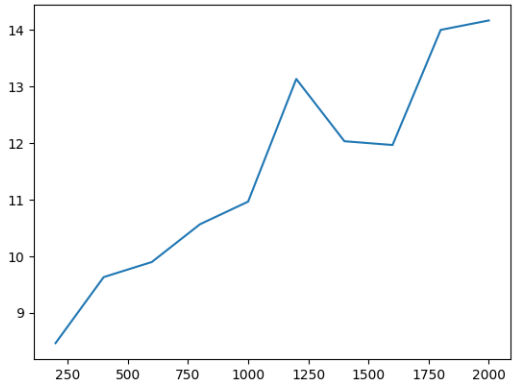

Figure 2: result from bga size test

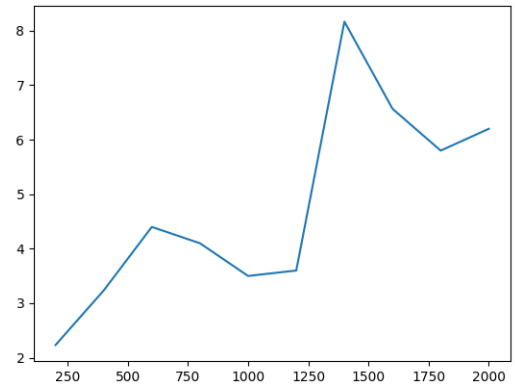

Figure 3: result from lcr size test

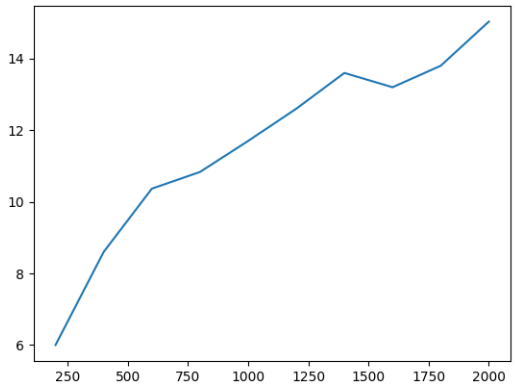

Figure 4: result from aap size test

Figure 5: x-axis is number of nodes , and y-axis is number of accepted calls. These results were reached with number of generated calls equal to number of nodes

likely because the amount of levels in the graph stays the same, and as the number of calls increases it gets more likely that a call of the chosen level gets generated. We've seen for both this test and the previous that bga tends to more varying results compared to aap, start and end at similar values, but bga spikes in the middle

9.2 Tree

9.2.1 size

For trees we have $N - 1$ edges, and as such we decide on N as the number of calls here too. This time the graphs are generated by picking one node as the baseline for a set, and adding a node to that set by creating an edge to a random node from the set. For each size we try 30 different instances of the problem, so both the graph and the request sequence varies each run. The results can be seen in 11. We see that the tree aap algorithm does not accept that many calls, this could either be because of how the probability in the algorithm rejects a lot, or because something does not work as it should. The copy graph seems to accept plenty of calls. I have found that the acceptance does not properly link the path on the original graph afterwards, so now i know where the problem is.
For tree lcr we have that it increases with the size, due to more nodes filling tree and therefore more nodes at the different levels.

9.2.2 Sequence length

The results from the tests on sequence length can be found in figure 14, we see not much of a change in tree aap. tree lcr has the same effect as the previous test. It increases because more calls mean more chances for call to be the level it picked.

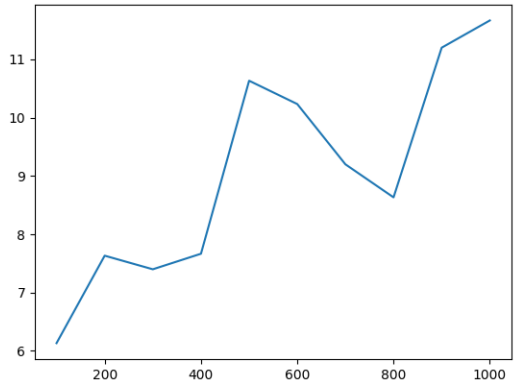

Figure 6: result from bga sequence size test

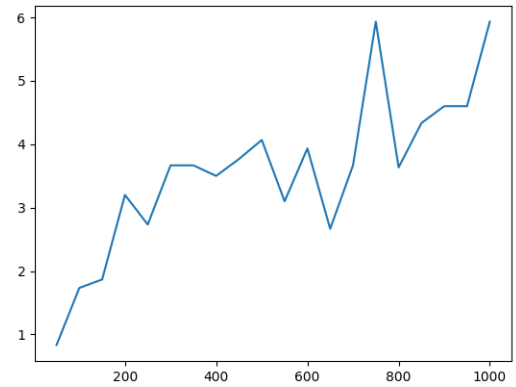

Figure 7: result from lcr sequence size test

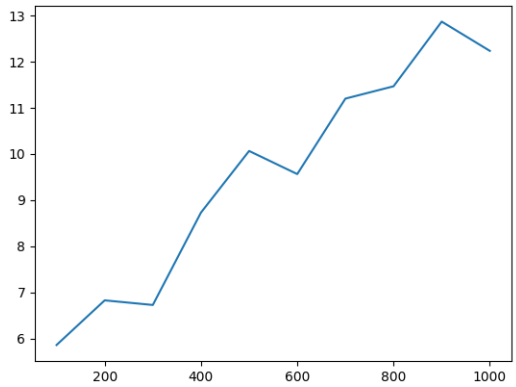

Figure 8: result from aap sequence size test

Figure 9: x-axis is number of calls , and y-axis is number of accepted calls. These results were reached on a line graph of size 1000

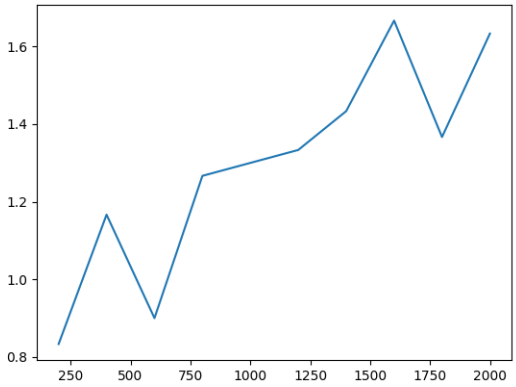

Figure 10: result from aap tree size test

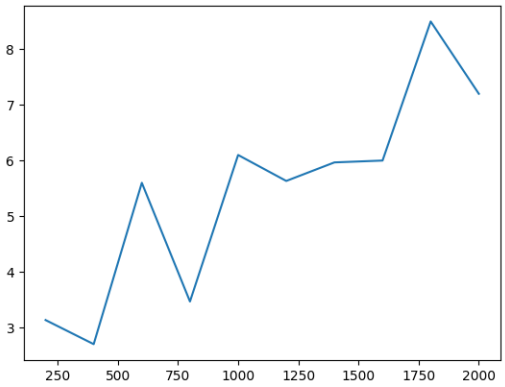

Figure 11: result from tree lcr size test

Figure 12: x-axis is number of nodes , and y-axis is number of accepted calls. These results were reached with number of generated calls equal to number of nodes

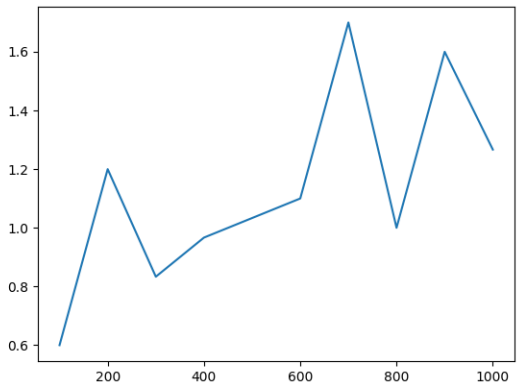

Figure 13: result from aap tree seq size test

Figure 14: result from tree lcr seq size test

Figure 15: x-axis is number of calls , and y-axis is number of accepted calls. These results were reached with number nodes = 1000

10 Conclusion

This thesis showed algorithms for different instances of the call admission control problem, and proofs together with them. This also means that it covered many of the fundamental features about capacitated graphs which could be used to prove bounds on other algorithms. The thesis also contained implementations of these algorithms, but there are some things that could have gone better , there were a couple of algorithms which didn't end up working quite right. The tree aap algorithm likely only needs a couple of lines edited. I feel like i understand how the mesh works, but i was having difficulties tying it together, if i had time to redo the code for it at the end i would have.

11 Appendix

References

[1] Allen Borodin, Ran El-Yaniv (1998) Online Computations and Competitive Analysis.

[2] Petr Kolman, Christian Scheideler (2004) Simple On-Line Algorithms for the Maximum Disjoint Paths Problem

[3] Jon Kleinberg, Eva Tardos (1995) Disjoint Paths in Densely Embedded Graphs