Hierarchical Demands

KEVIN MITCHELL
Agilent Laboratories

Initial deployments of Multiprotocol Label Switching (MPLS) were typically confined to the network core. Offline tools were retargeted from ATM networks to MPLS to optimize the routing of label switched paths (LSPs) across the network. As the technology matures MPLS deployment is moving into the access networks. This creates a scaling problem for the optimization tools, with many more demands, nodes and paths that have to be considered. This paper develops a range of techniques for decomposing network topologies into a set of components, allowing us to solve a group of small optimization problems rather than a single large one. Access trees are the simplest access structures we can exploit, and in some cases may be sufficient to yield a tractable problem. For more complex cases we develop a strategy based on the identification of biconnected components and the construction of a component tree rooted at the network core. Finally we describe how additional virtual nodes can be introduced to increase the flexibility of the approach, allowing us to split components even further.

Key Words and Phrases: MPLS, hierarchical topologies, traffic engineering, offline optimization, multicommodity flows

1. INTRODUCTION

When Multiprotocol Label Switching (MPLS) was first introduced in the late 1990s, one of the primary drivers was the desire to make routers faster. ATM switches had better packet forwarding rates than routers because fixed length label lookup was faster, and easier to implement in hardware, than the longest match lookup used by IP routing. MPLS allowed a device to do the same job as a router but with the performance of an ATM switch. Initially MPLS deployment was confined to the network core where the improved routing speed could be fully exploited. The traffic between any two points within the core will be heavily aggregated, consisting of a large number of microflows. For each traffic class we can construct traffic matrices showing how much traffic of this type flows across the core during each measurement period. In some cases, the demand will be sufficiently predictable over time that we can assign the traffic to MPLS paths, and then determine the best routes for these paths that minimize congestion. As long as the variability in bandwidth requirements is not excessive, offline path placement, coupled with the use of autobandwidth mechanisms to adjust the reservations at runtime, e.g. [Cisco 2003], can yield a useful network optimization strategy. ATM networks had previously been used within network cores, and offline tools had been developed to optimize the routing of paths through ATM clouds. These tools were quickly adapted to support the offline optimization of bandwidth guaranteed label switched paths (LSPs) through MPLS clouds. Given the small size of typical core networks, this optimization problem was reasonably tractable. The main constraint was that demands must not be split. They represent a collection of aggregated flows and it would be difficult to split these across multiple paths without introducing unnecessary packet reordering within the individual flows.

With the advent of better hardware support for IP longest prefix matching the speed differential between MPLS and IP routing became less of an issue. However, the need for service differential between flows became increasingly important as operators struggled to find profitable revenue streams. Techniques such as DiffServ allow packets to be treated differently within each router. When a DiffServ marked packet reaches the MPLS core the DiffServ code point can be mapped, albeit imperfectly, to the EXP bits in the MPLS header, allowing the service differential to be continued across the core. DiffServ provides few guarantees about the quality of service that can be expected for a flow, which is understandable as no

Author's address: K. Mitchell, Agilent Labs Scotland, South Queensferry, Scotland EH30 9TG

resources are reserved for it. Furthermore, all packets with the same source and destination typically follow the same path, irrespective of their DiffServ codepoint. There is clearly a limit to how much service differential you can achieve with such an approach. The obvious solution is to use MPLS to provide multiple paths across the network, and to then assign flows to paths based on their traffic class. Whilst this can be done in a DiffServ environment, by using the DiffServ markings to guide the choice of LSP at the ingress to the MPLS cloud, it introduces additional complexity into a part of the network that is already heavily stressed. More recently, the MPLS boundary has been gradually moving outside the core into the access networks. This allows packets to be classified and assigned to LSPs prior to reaching the core, using tunneling to choose the desired path across the core and minimizing the signaling and state that must be supported by the core routers. In scenarios that are more complex it also allows the operator to choose different paths across the access networks themselves.

This trend has a number of consequences for the offline optimization problem. The size of the MPLS cloud is no longer restricted to the size of the core network. This creates a serious scaling problem for the optimizer, requiring the development of techniques to decompose the problem into something more manageable. The traffic originating within the access layers will typically be less aggregated than that in the core, and so exhibit greater fluctuations. This makes it difficult to identify LSPs that are persistent and stable enough to be worth routing offline. Whilst in most cases the source of a flow will be some distance from the ingress of the LSP it will be carried by, these points are starting to converge in some important cases. For example, some voice gateways are already MPLS-enabled and individual calls could be assigned to LSPs during the call setup process. TDM over MPLS (TDMoMPLS) access devices, e.g. [RAD 2004], could also potentially support call mapping to LSPs. This has an impact on the optimization process, as in these cases it may be acceptable to split demands across multiple LSPs without introducing additional packet reordering within each flow/call.

Systems such as CPlane [CPlane 2003] and WANDL [Wandl 2002] allow an operator to optimize MPLS demands across the core of a network. They assume a predefined partition of the network into access and core routers, and the demands to be optimized are then restricted to the core (edge). In the case of an MPLS-enabled VoIP gateway, our demands may originate in the access layers. We will call these *access LSPs*. There are a number of reasons why we must treat the access LSP optimization problem differently to the equivalent core problem.

—Scalability. Globally optimizing a large number of demands, spanning many routers, is computationally very expensive. This cost increases rapidly as we increase the number of demands and/or routers. Decomposing the problem into a collection of simpler problems may be essential if we are to achieve realistic optimization times.

—Administration. Many organizations use different groups of people to manage the core and access networks. Even if we could route the demands across the whole cloud, we might not be able to deploy such a solution because of these administrative divisions. A better approach might be to use the access demands to construct a set of requirements for core demands necessary to support this traffic. These requirements could then be passed to the Core group who can optimize the placement of these demands using traditional or new optimization techniques. The Access group would use the solutions to these requirements to build LSPs to support the original demands. We argue that the requirements projected onto the core from a set of access demands may be rather different in character to a traditional set of core demands, potentially requiring different core optimization tools.

—Tunneling. Provisioning an LSP hop-by-hop across the whole route between ingress and egress may be inefficient. Each router along the way will need to process the signaling traffic necessary to keep the LSP alive, and to reserve state within the router. It will be more efficient to define a set of LSPs across the core, and then use these as tunnels for the permanent LSPs that originate in the access layers. Only the access routers would store state specific to the access LSPs.

Our goal in this paper is to explore the problem of optimizing access level demands. Our strategy will be to infer from these demands a set of requirements for LSPs crossing the core. Having optimized the core LSPs, we will then show how these can be used to route the access LSPs. The organizational structure of many operators makes it natural to split networks into core and access components. However,

as networks become more complex it may be beneficial to decompose the network further. An access LSP may pass through a number of different layers before reaching the "core", and each of these could be exploited in the same way as the core/access distinction. The decomposition into core and access routers may be specified a-priori, for example via data extracted from an OSS system. Nevertheless, it is unlikely that any finer level of hierarchy will be explicitly stated. Even in the core/access case, we may have to infer the boundary, perhaps with a little help from the operator, and heuristics developed for such a task may complement topology discovery mechanisms. For these reasons we will start by considering the decomposition problem in a setting where no hierarchical structure is known in advance. Later we will consider the effects of a predefined core/access split, particularly in the case where different groups are responsible for these components. We may need to merge components to cover the entire core. We may also need to split some components to achieve the required separation. The optimization tools themselves may need to be split so they can be used by separate groups. The interface required between these tools may be richer than that provided by existing offline optimization tools if we wish to support access LSPs.

Replacing demands that span the access and core networks with aggregated demands restricted to the core is trickier than it seems. We want to ensure that any solution we find across the core can be extended to a solution that satisfies the original demands, particularly where the demands have QoS constraints associated with them. We argue that we either need to have some flexibility in where we draw the core/access boundary, or map the access demands into a more complex form of core demand, for example where multiple ingresses and egresses are supported, or where multiple paths are acceptable.

The approach followed in this report is to develop heuristics to cluster routers into components, with the components organized in a hierarchical fashion, and with the network "core" at the root of this hierarchy. Demands that originate or terminate at components outside the core, but that traverse the core, are temporarily replaced by demands that originate and terminate within the core component. Having optimized the resulting set of demands, we show how to use the solution to satisfy the original demands. Clearly, the success of this approach, judged by the optimality of the final solution, depends crucially on our choice of components, and the clustering of routers into these components. We start with a very simple clustering mechanism that gives us good guarantees about the final solution. Multi-access networks cause some complications, and we discuss these and how to handle them in this setting. However, the definition of "access component" is rather restricted, leading to large "core" components. We then develop further heuristics to widen the set of access components, but our solutions may be less optimal as a result. The report concludes with a discussion of more general access graph structures, and the problems these generate. In particular, we consider the case where our components must respect an existing partitioning of the routers, e.g. into core and access routers.

2. THE OPTIMIZATION PROBLEM

The starting point for our optimization problem is a set of demands across a network. Each demand represents a requirement for a certain amount of bandwidth, with an associated traffic class or QoS requirement. The traffic class constrains the acceptable routes that can be used to service the demand, such as the maximum delay or cost. We will allow the optimization process to generate solutions that split demands over multiple paths. Although the need for demand splitting is rare, it can be useful with high-bandwidth demands. We assume that there will be a mechanism to split demands at the ingress if necessary, for example in a VoIP gateway. We wish to find the best route(s) for each demand to minimize some global criteria, such as average link utilization. Clearly there may be many optimization criteria we could choose, but the exact choice is unimportant for the following discussion.

Optimizing the placement of demands across a network is computationally expensive. There are two main approaches to tackling this multi-commodity flow problem. In an edge-based strategy, a linear program (LP) attempts to compute the amount of each demand carried by each link in the network. In the worst case, with a full mesh of demands, and a highly connected graph, there will be $O(n^2)$ demands and $O(n^2)$ edges. The paper by Köhler [Köhler and Binzenhöfer 2003] contains five example topologies of increasing size. Timings from a prototype solver [Mitchell 2004], presented in Figure 1, illustrate how quickly the computation times increase as the network size grows. An edge-based strategy has another disadvantage when the demands have additional QoS constraints attached to them. The paths found by

the optimizer may not satisfy the constraints, e.g. on delay or hop length, resulting in an invalid solution. Attempts to enforce these constraints during the optimization process quickly lead to intractable models, even for small networks. The edge-based approach is therefore most suited for demands with liberal QoS constraints, such as best-effort traffic.

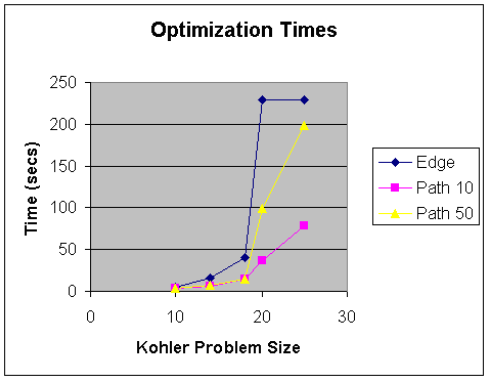


Fig. 1. Optimization times for various network sizes

The other main approach is to first identify a set of potential paths for every demand, each of which satisfies the QoS constraints. We can then use a linear program to calculate how much of the demand should be carried by each path. If we only use a small number of paths for a demand then we can limit the size of the optimization problem to something tractable. The downside is that our solution is only as good as our choice of paths. If we increase the number of paths, to increase the chance an optimal solution is found, then the optimization times grow very quickly, particularly for highly connected graphs. This behavior is illustrated in Figure 1, where `Path 10` uses a maximum of ten paths for each demand, and `Path50` uses a maximum of fifty. For demands with strict QoS properties, our best strategy is to use a path-based approach and hope that the number of valid paths for each demand is fairly small. However, as the network size increases, we quickly reach the stage where optimization times become intractable. The case where there are multiple paths through the access network also creates difficulties for the path-based approach. If we just enumerate the first "n" paths, using something like A*Prune [Liu and Ramakrishnan 2001], we may find that most of these paths share a common subpath across the core. In many cases we will want more variability in the paths in order to find a good solution to the optimization problem.

For these examples to scale to networks with hundreds or thousands of endpoints we need to discover hierarchical structures within the network, together with heuristics to exploit such structures without departing too far from the optimal solution.

3. ACCESS TREES

Figure 2 illustrates a typical backbone network. There are approximately 120 nodes in this network. Even with a single traffic class, with demands between each pair of devices, we generate a problem that would be very expensive to optimize directly. It seems clear we need to find heuristics for simplifying the demand placement problem if we want to tackle problems of this scale, or larger.

One way of simplifying the optimization problem is to identify, and then exploit, particular properties of the network under investigation. For example, if some form of hierarchical structure can be identified then we may be able to split the problem into two or more simpler problems. Of course, the challenge in such cases is to ensure that the combined solution to the simpler problems is close to the optimum solution of the original problem. The problem is complicated by the fact that networks come in lots of different shapes and sizes. An optimizer might need to exploit a range of different heuristics for the demand placement problem depending on the structure of the network being optimized.

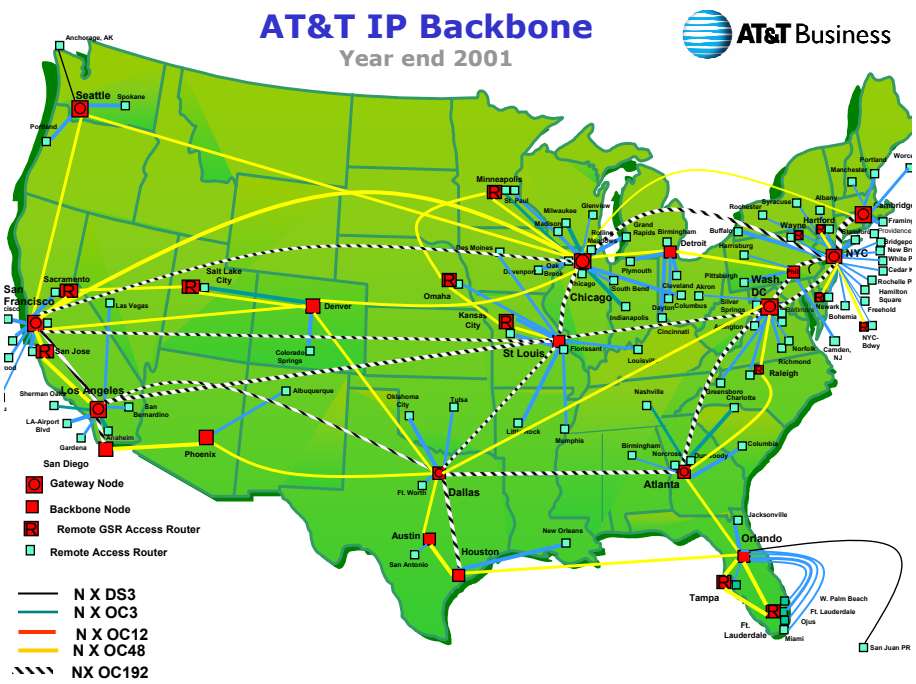


Fig. 2. AT&T IP backbone, 2001

We start by developing a heuristic that might be appropriate for the AT&T example shown in Figure 2. One obvious feature of the example is the large number of leaves, corresponding to the "access" part of the network. Therefore, whilst there are about 120 nodes in total, approximately three quarters of these are "leaf" nodes. If we could somehow remove these from the problem, optimize the residual graph, and then extend the solution to incorporate the demands associated with the leaf nodes, we might have a tractable solution to this particular network.

How do we identify structures such as leaf nodes, or the edges forming the access part of the network? We could exploit information provided by the network topology. For example, a node may have a type and be associated with an autonomous system (AS). We could standardize the type information, allowing us to identify access routers. We could also use the AS information to suggest a hierarchical decomposition of the graph. Nevertheless, whilst such information may be useful as "hints", a more robust solution would examine the structure of the graph itself.

3.1 The Core/Access Tree Partitioning Algorithm

We start by imposing a forest structure, i.e. a set of trees, on the nodes in the network graph. Let Nodes be the set of nodes, or vertices, in the graph. We define a map, parent, representing a total function on Nodes, i.e.

$$\text{parent} : \text{Nodes} \rightarrow \text{Nodes}$$

By construction, we will ensure this map implicitly defines a set of trees, i.e. there will be no sequence of vertices, $v_1, v_2, \ldots v_n$, such that $\forall 1 \leq i < n \cdot \text{parent}(v_i) = v_{i+1}$ and where $v_1 = v_n$. However, we indicate tree roots by vertices v such that $\text{parent}(v) = v$, and so we relax the acyclic rule accordingly. By this we mean that the only cycles permitted are those involving root nodes. We use the parent map to partition the set of vertices into *root* nodes and *tree* nodes.

We initialize the map such that $\text{parent}(v) = v$ for all nodes, i.e. they all start as root nodes. We then apply the following procedure repeatedly:

Find a root node v with n outgoing edges in the graph, for some $n > 0$. Let the destinations

of these edges be $v_1, v_2, \ldots, v_n$. If there exists a root v_p, $p \in \{1..n\}$ such that

$$\forall i \in 1..n, i \neq p \cdot \text{parent}(v_i) = v$$

then set $\text{parent}(v) := v_p$.

Each time this transformation is applied a vertex changes from being a root node to a tree node. Furthermore, the procedure never changes a tree node back to a root node, and so the process is guaranteed to terminate. Although not proved here, the map we end up with does not depend on the order in which we transform the nodes.

At the end of this process the root nodes will form the core of our network, and the tree nodes the access component, as illustrated in Figure 3. For the AT&T example each tree will have a depth of one,

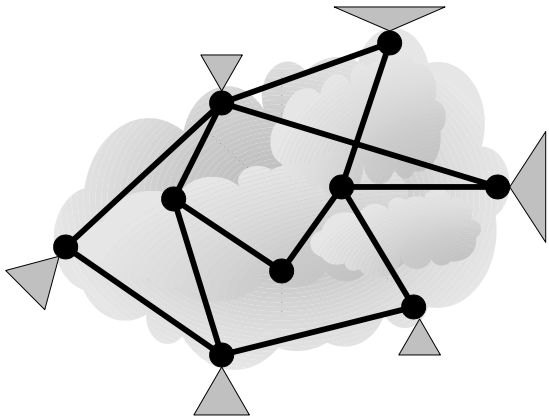


Fig. 3. The core and access components

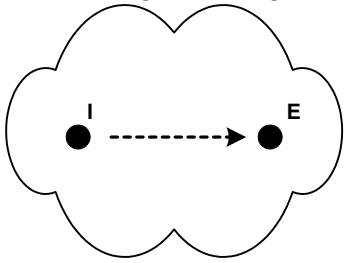
but in other examples it may be larger than this.

3.2 Demand Simplification

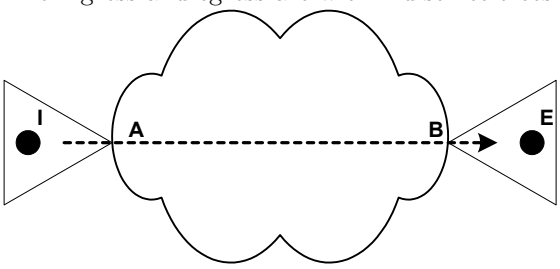
For each root node r there is a set of nodes, possible empty, that form the tree attached to this node, i.e. the set of nodes that can reach r using the parent map. We will call this set the access tree attached to r. For each such tree, T, there may be some demands with both the ingress and egress local to T. By definition, there is a unique shortest path between the ingress and egress in such a case. Whilst longer paths exist, these must traverse the same links as the shortest path, plus possibly others, and so there is no point in considering longer paths for such demands in an optimum solution. Placing such demands is simple. Our only choice is whether to place them first, and then optimize the rest of the demands, or to place them afterwards. If we assume that all demands can be satisfied then it will make no difference in which order we place these demands.

Having taken care of the "local" demands then we are left with demands of the following forms:

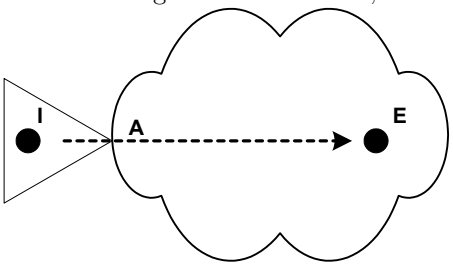
(1) Both the ingress and egress are within the core.

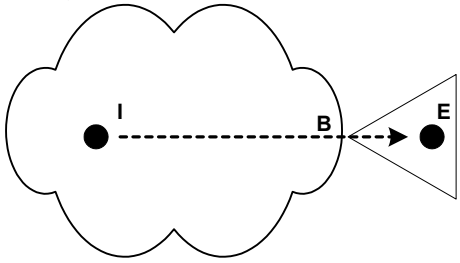


(2) The ingress and egress are within distinct trees.



(3) Either the ingress is in the core, and the egress in a tree, or vice versa.





Our strategy is to transform demands of form 2 or 3 into demands purely internal to the core. We can then solve an optimization problem on the core and use this to place the type-2 and type-3 demands. Consider a type-2 demand. Any path from I to E must pass through nodes A and B. Furthermore, any path from I to A must pass along the edges forming the unique shortest path from I to A. Similarly for B to E. So if we find an optimum placement of the demand from A to B we can easily extend it to a solution from I to E. Furthermore, our intention is to map paths into MPLS label switched paths (LSPs). MPLS supports tunnels, and so a demand from I to E could be implemented by an LSP from I to E that used another LSP to tunnel from A to B. This suggests the following approach. Given a demand d_{IE} from I to E we find the demand d_{AB} from A to B with the same traffic class as d_{IE}, creating a new one if necessary. We then add the bandwidth requirement of d_{IE} to d_{AB} and (temporarily) remove the demand d_{IE}. We do this repeatedly until there are no more type-2 demands left.

The type-3 demands are treated similarly, either adding the bandwidth to a demand from A to E or from I to B. At the end of this process we will be left with just a set of type-1 demands. We now need to place the demands across the core network, i.e. the subnetwork only involving core nodes. Depending on the size of the core network, we may be able to solve this problem directly, or need to use further heuristics to break the problem down to a more manageable size. For the AT&T example the initial problem has 122 nodes, and 14762 demands, a complete mesh with a single traffic class. After the transformations, we are left with a core network of 27 nodes, and 702 demands. Clearly this is a much more manageable problem which we should be able to solve directly in this case.

Most demands will be satisfied by the solver using a single path. This will be mapped to a single LSP by the provisioning system. Now consider our example of a type-2 demand from I to E. The demand from A to B will be mapped to LSP_{AB}. The demand from I to E will be mapped to an LSP that follows the unique path from I to A, followed by the tunnel LSP_{AB}, and then the unique path from B to E. In a few cases the solver may split the demand from A to B across multiple paths. These will map into multiple LSPs, LSP_{AB_1} and LSP_{AB_2} say. We have two choices here. We could split the demand from I to E into two paths using the same proportions as the core demand was split. One path would be tunneled over LSP_{AB_1} and the other over LSP_{AB_2}. But if there are multiple demands being tunneled over this core demand then it would make sense to assign each access demand to a single tunnel where possible, only splitting them where necessary.

A more verbose example might help here. Suppose we had the situation illustrated in Figure 4. Furthermore, we assume the following three demands need to be satisfied:

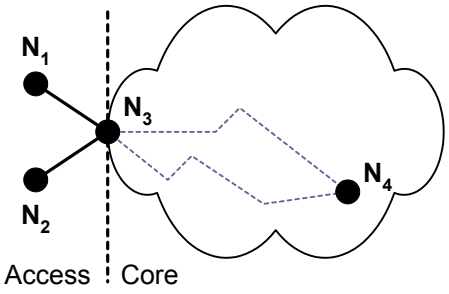


Fig. 4. Splitting demands across the core network

D_1	$N_1 \Rightarrow N_4$, 10 units
D_2	$N_2 \Rightarrow N_4$, 10 units
D_3	$N_3 \Rightarrow N_4$, 10 units

As N_1 and N_2 are part of the access network we would (temporarily) remove D_1 and D_2, and increase the reservation for D_3 to 30 units. Suppose, after calling the solver, we have two paths between N_3 and N_4, with one path, P_1, having a reservation of 10 units, and the other, P_2 having a reservation of 20 units. We would generate two LSPs, LSP_1 and LSP_2, corresponding to these two paths. We now have to map the original demands to LSPs, and there are multiple ways of doing this. We could map D_3 to LSP_1, and then map D_1 to LSP_3 which starts at N_1 and tunnels to N_4 using LSP_2. Similarly we would map D_2 to LSP_4, that starts at N_2 and tunnels to N_4 using LSP_2.

Now suppose the solver had allocated 15 units to both P_1 and P_2, i.e. LSP_1 and LSP_2 can carry 15 units each. We would then map D_3 to LSP_1 and D_1 to LSP_3, tunneling over LSP_2. Finally, we would map D_2 to two LSPs, LSP_4, tunneling over LSP_1, with 5 units capacity, and LSP_5, tunneling over LSP_2, with 5 units capacity. Given that the solver rarely splits demands, in most cases there will be no choice about how to map the solution to the original demands. In the cases where there are choices, each solution will have the same network utilization, but some may be better than others in terms of the total number of LSPs. This could be formulated as another optimization problem, but the situation will occur sufficiently infrequently that any mapping will probably lead to an acceptable solution.

The situation with type-3 demands is treated similarly. In the case where the demand starts in the tree, we could use label merging, rather than tunneling, if the LSP signaling mechanism supports this. In the case where the egress is in the tree then merging isn't an option, and so we just push both labels on the tree at the ingress.

4. QOS CONSTRAINTS

The approach of mapping demands across the access trees into demands across the core becomes more complex when there are quality of service (QoS) constraints attached to the demands. To see why, consider a simple scenario where we attach an additive constraint, such as a hop limit, to each demand. Suppose all demands are associated with a traffic class with a hop limit of five. By this we mean that the only paths that can satisfy such a demand must have a length no greater than five. Now consider a type-2 demand between I and E. Our algorithm translates this into a demand between A and B. But which demand? If there is already a demand between A and B it is tempting to just increase the bandwidth reservation on this demand. After all, the optimization times are influenced not just by the size of the network but also by the number of distinct demands being placed across the network. In the absence of QoS constraints, such a translation would be valid. However, in our example it will lead to erroneous results. To see why, consider the case where I and A are connected by a point-to-point link, and similarly for B to E. Given that it takes one hop to enter the core network, and one hop to reach E from the core network, we are only allowed three hops to traverse the core from A to B if we wish to satisfy the QoS constraint. However, if the existing demand from A to B has a QoS limit of 5 then the placement solution

for this demand may not be suitable as a solution for the IE demand. This suggests that in such cases we should construct a new demand from A to B with a more restrictive QoS constraint, in this case a hop limit of three. A similar argument applies with other additive QoS constraints, such as maximum delay or cost.

In the worst case, we may end up with hundreds of demands between A and B, differing only in their QoS constraints. Our reduced problem would have a smaller number of nodes, but the same number of demands, potentially still giving us a problem no more tractable than the original. Nevertheless, there are reasons for hope here. In practice there are likely to be many nodes in the access tree containing I that wish to reach E and other nodes within E's access tree. If they all have the same number of hops to travel to reach the core, which will be a common case, then they can all share a demand from A to B with the more restrictive constraint. Delay and cost constraints are obviously trickier, as it will be less likely there will be an exact match. However, we can trade off optimality for speed here, by mapping an access demand to an existing core demand with a more restrictive constraint, rather than creating a new one. We anticipate using such demand and traffic class merging even within the core network when there are many different traffic classes. This scenario is just another example of where it would be useful. The path-based approach to optimization introduces further potential for demand merging. The purpose of the demand classes is to restrict the set of paths that are acceptable for a given demand. In many cases, we may generate the same set of paths for two demands, even though they have different traffic classes. In a path-based approach we can merge any demands for which we generate the same set of candidate paths, and then use the aggregated demands in the linear program.

In the case of the edge-based optimization strategy we could adopt a different approach to this problem. The edge-based optimizer ignores QoS constraints during the optimization process. The edge allocations are translated to paths during a post-processing phase, and then these paths are checked to see if they respect the desired QoS constraints. Whilst this has the disadvantage that some demands may violate their QoS constraints in the "optimal" solution, it does simplify our current problem. We would map the access demands to the core demands in the same way as if there were no QoS constraints. When constructing paths for these access demands we would choose the tunnels based on the strictness of the QoS constraints. For example, consider the situation illustrated in Figure 4 again, where the solution identified two paths between N_3 and N_4, with P_1, having a reservation of 10 units, and P_2 having a reservation of 20 units. If P_2 were shorter than P_1 then we would use it as our preferred tunnel for D_1 and D_2, using P_1 for D_3. Of course, there may be no way of assigning the tunnels to the access demands that satisfies the QoS constraints. However, this is just another symptom of the mismatch between edge-based strategies and tight QoS constraints.

5. MULTI-ACCESS NETWORKS

All the links in our previous examples have been point-to-point links. However, these are not the only kinds of link we have to consider. Multi-access networks, built using ethernet or similar technologies, will often be found in the access component of the network. We need to be able to recognize such networks, separating them out from the core where possible. They must also be treated specially from the perspective of the optimizer.

Figure 5 a) illustrates a common situation. By introducing a new "network" node, and replacing each multi-access link by a point-to-point link to the network node, we reach the position shown in Figure 5 b). Most of the routers attached to the network will be leaves, and the tree/core partitioning algorithm will place them in the access network, as desired. However, in the case of some multi-access networks, the logical links connected to the network node may not be truly independent of each other. Assigning some bandwidth to one path traversing the network node may constrain how much bandwidth can be assigned to other paths traversing this node. So simply assigning bandwidth constraints to individual links may be insufficient in such cases. One approach would be to attach a bandwidth constraint to the network nodes themselves. The sum of the bandwidth required by all demands traversing this node would then be constrained to be less than this limit. However, since most networks do not exhibit such dependencies, at least not to the extent that they need to influence the placement process, we ignore such complexity in this document.

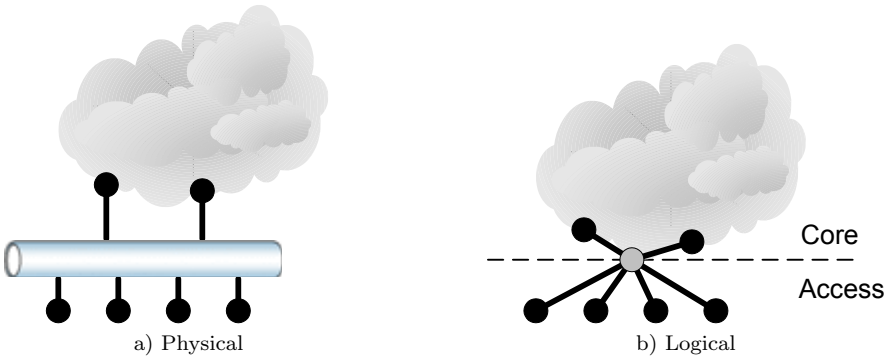


Fig. 5. A multi-access subnetwork

As with the pure point-to-point case, there may be some demands where the ingress and egresses are confined to a single access tree, in this case comprising nodes attached to the same multi-access network. We can either choose to place these demands before or after we optimize the core network. We can often make the simplifying assumption that link delay and cost between each pair of nodes connected to the multi-access network will be the same. If there were substantial differences in these metrics, for example because the access network is provided by an ATM cloud, then we would have to model the multi-access component by a full mesh of point-to-point links. This allows us to specify the link metrics for each pair of end-points, but our access/core distinction would break down in this case, forcing us to treat all these nodes as part of the core. We would adopt the same approach if something like an ATM cloud were deployed within the core of the network.

Whilst the introduction of virtual network nodes may allow us to decouple a multi-access network from the network core it complicates the post-optimization processing. To see why consider the scenario where a demand originating in the multi-access network is replaced by a demand originating at the network node. If the multi-access network has multiple entry points into the core then the network node will end up being treated as part of the core during the optimization process. The optimizer will compute one or more paths to carry the demand originating at the network node. However, this node doesn't really exist, so we can't simply map these paths to LSPs. The first hop in each of these paths will be to a real router within the core, and we can use this router as the ingress for the LSP associated with the path. The original demands would tunnel through these LSPs, just as in the point-to-point case.

6. ACCESS GRAPHS

It is not hard to find examples of access networks that are more complex than simple trees. Figure 6 illustrates two such examples. In this network router A has two paths into the core, whilst router E

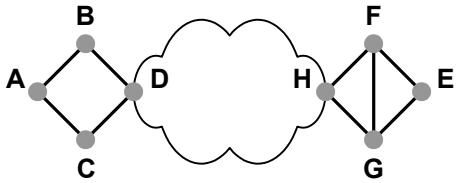


Fig. 6. Examples of Access Graphs

has four paths. The access tree analysis is insufficient for such networks, resulting in everything being treated as part of the core. Not only may this lead to a graph that is too large to optimize, but our path selection process may also produce a poor set of candidate paths. Ideally we want to ensure that most

of the variability in these paths isn't confined to links close to the ingress. If there are multiple paths through the access component it can be difficult to achieve this ideal.

When an access graph has many links into the core it will be difficult to tease it out from an optimization standpoint. The access/core distinction is more administrative than topological in these cases. However, in our example the situation is simpler. We know where demands originating at A will reach the core, and if they are destined for router E we also know where the paths for these demands will leave the core. Routers D and H act as bottlenecks that can be exploited to split off the access components.

The problem with trying to replace a demand originating at node A and terminating at node E in our example is that there are multiple paths both to and from the core. So what QoS constraint should we associate with the replacement demand from D to H? Each path from A to D may have a different cost, and so the additive constraints, such as hop limit, delay or cost, for the replacement demand originating at D would depend on the path taken to reach D. We could imagine building a system where the optimizer could work with demand choices. I.e. given a choice set $d_1, d_2, \ldots, d_n$ the optimizer would only have to satisfy one of these demands. We could enumerate all the different paths to reach the core network, calculate the different costs, and then generate one demand for each cost. But this just replaces the original problem with an even worse one. For example, if we assume that all the paths through the access networks have different costs, we would end up replacing the original demand from A to E by eight different demands from D to H! Of course, not all of these will necessarily result in different candidate path sets, and so the linear program might not have to deal with eight demands. Nevertheless, this isn't the road to a scaleable solution.

For access graphs with a unique entry point into the core, we can make the simplifying assumption that all traffic will reach this point using the least cost route through the access network. Having made this assumption, we can calculate the QoS constraint for the replacement path across the core. Having optimized the core demands we can then build the access demands using the core LSPs as tunnels. Loosely routing these LSPs, i.e. just constraining them to use particular core LSPs, leaves the runtime with some flexibility for traversing the access networks. Of course, there is no guarantee that we will be able to satisfy all these demands in the access component with the required QoS bounds.

For this approach to work we need to identify access components with single entry-points into the rest of the network. Our approach starts by identifying biconnected components within the graph. First, we need some definitions.

DEFINITION 6.1. *A vertex v in a connected graph G is an* **articulation point** *if the deletion of v from G, along with the deletion of all edges incident to v, disconnects the graph into two or more nonempty components.*

DEFINITION 6.2. *A graph G is* **biconnected** *if and only if it contains no articulation points.*

DEFINITION 6.3. *$G' = (V', E')$ is a* **maximal biconnected subgraph** *of G if and only if G has no biconnected subgraph $G'' = (V'', E'')$ such that $V' \subseteq V''$ and $E' \subseteq E''$. A maximal biconnected subgraph is a* **biconnected component**.

LEMMA 6.4. *Two biconnected components can have at most one vertex in common and this vertex is an articulation point.*

Partitioning the graph edges into a set of biconnected components is a fast operation [Sedgewick 2001]. Figure 7 illustrates the result of such an analysis, where the edges have been tagged with their component identifiers. Vertices with edges from more that one component are the articulation points. The cloud represents the "core" that we are trying to identify.

A biconnected component consists of all the edges with the same component number, together with the nodes they connect. What can we do with such an analysis? How do we (reliably) distinguish core from access? Given an ingress and an egress they are either both in the same component, or there is a unique path of components to get from the ingress to the egress (assuming the two points are connected). The component path is guaranteed to be unique as the components form a tree, as illustrated in Figure 8. Note that this is an unrooted tree, and so at this stage there is no concept of moving from the edge of a

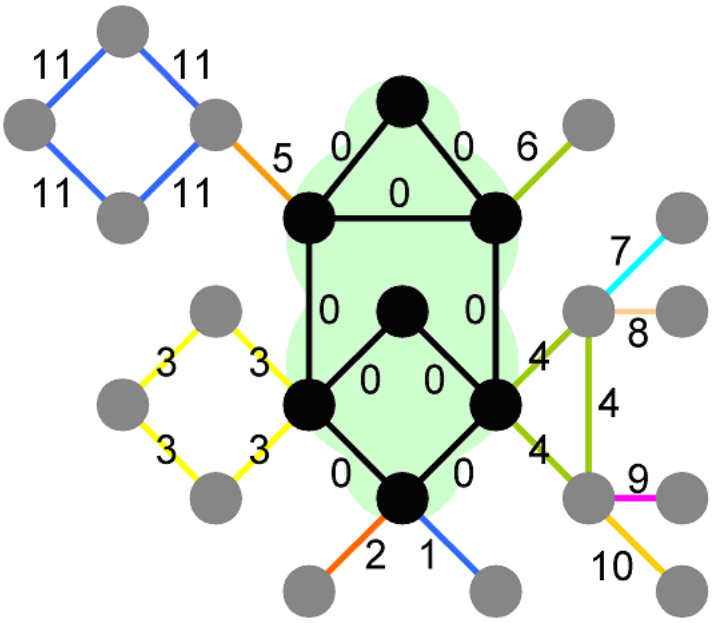


Fig. 7. Complex access graphs

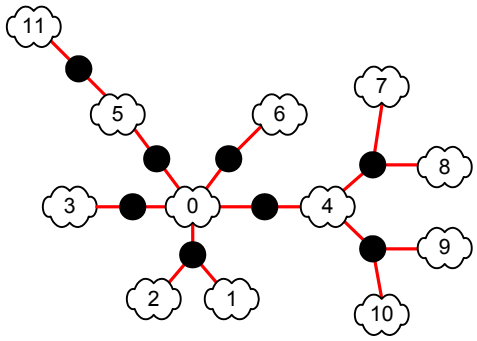


Fig. 8. Biconnected components

network to the core. The black nodes are the articulation points. Therefore, a router will either be an articulation point in this tree or part of a component in the tree, i.e. in one of the clouds.

6.1 Merging access trees

Before trying to order the components, to give us the core/access distinction we require, we need to simplify the graph. Access trees occur frequently in networks, and produce many small components when the network is split into biconnected components. To illustrate this behavior consider the network in Figure 9. The components generated by this example are shown in Figure 10, where the components, excluding the articulation points, contain the following nodes:

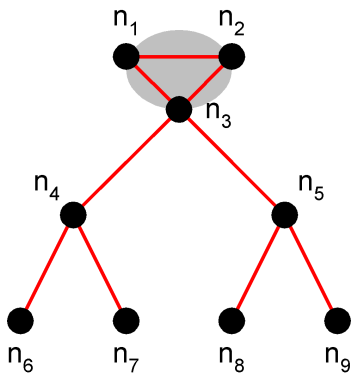


Fig. 9. A simple access tree

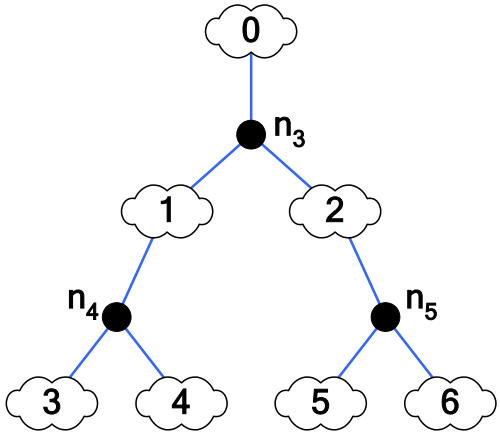


Fig. 10. The components generated for the simple access tree

0	$\{n_1, n_2\}$
1	$\{\}$
2	$\{\}$
3	$\{n_6\}$
4	$\{n_7\}$
5	$\{n_8\}$
6	$\{n_9\}$

Given our intended usage of these components, to identify bottleneck nodes between the access networks and the core, we can merge the components containing the access trees to simplify the graph.

DEFINITION 6.5. *A component C is a* **tree component** *if it is connected to a single articulation point, x say, and the nodes within the component, including x, are connected to each other by edges that form a tree in the original graph, with x as the root of this tree.*

In our previous example we can immediately identify component 3 as a tree component. It only has a single node, and this node, together with the articulation point n_4, form a trivial tree. Similarly for components 4, 5 and 6. We now define rules for merging tree components. The first rule is illustrated by Figure 11. In this figure we are using T as an annotation on the component clouds to indicate that they represent tree components. Why is this transformation legitimate, i.e. why is the merged component also a tree component? This follows trivially from the fact that there can be no edges between C_1 and

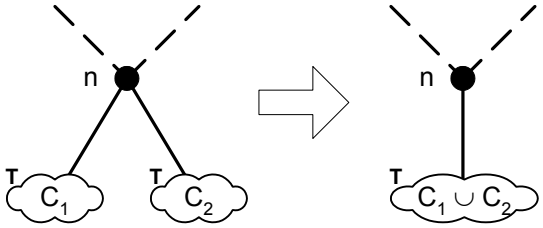


Fig. 11. First tree merging rule

C_2, because if there were they would not have been separate components in the first place. To see why, suppose there were an edge e between a node in C_1 and a node in C_2. Then there would be two paths between any node in C_1 and a node in C_2, one via e and the other via n. So C_1 and C_2 would be biconnected, and would not have been split in the first place.

Our transformation step can be used repeatedly to merge siblings, but we also need a rule to merge the children with their parent when a tree has a depth greater than one. A transformation to perform this task is illustrated in Figure 12. Why is this transformation valid? The node r and n must be

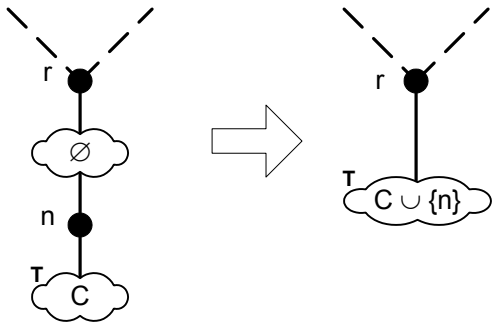


Fig. 12. Second tree merging rule

connected by a single edge, as the intermediate component is empty. Furthermore, n is the root of the tree component C. Therefore, r must be the root of the tree component $C \cup \{n\}$. By repeated application of these transformations, we can collapse the access trees into tree components. For our original example this process results in the graph presented in Figure 13. In this figure we have used $x \cup y$ to denote the component containing the union of the nodes in component x and y.

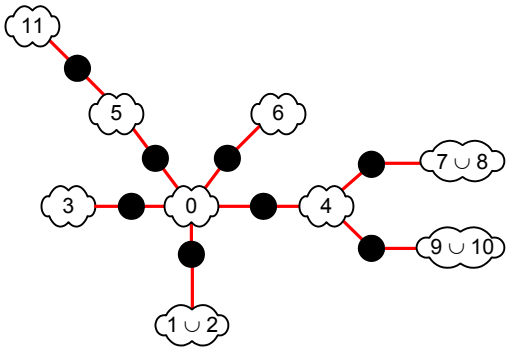


Fig. 13. Merged components

We will still use the terms “articulation point” and “biconnected component” when discussing the transformed tree, even though strictly-speaking this is incorrect. Trees that connect more general access graphs to a core component will not be detected by these transformations. It remains to be seen whether such situations occur frequently enough to be worth detecting and exploiting. It is hard to formalize transformations such as these when the component tree has no root, and therefore no “direction” at this stage. This suggests a second stage of simplification should take place after the network core has been identified.

6.2 Determining the network core

Our component graph imposes some structure on the original graph, but there is still no concept of moving from an access component towards the core. We need to make the component tree rooted, where the ”core” is the root. This will then define an ordering between components, and supply the required concept of moving from the access to the core. We could clearly choose any of the clouds as being the root, but which one is most likely to represent the core of the network? There are a number of heuristics we could employ to pick the core component. However, they will all fall foul of pathological examples, potentially requiring user intervention to guide this process. For example, picking the largest component, including its articulation points, seems plausible, except that we might have a very large access tree with more nodes than the true core. Similarly choosing the component with the smallest maximum path length to all the other components seems reasonable, as it would tend to find the component in the ”center” of the tree. However, an access graph with many hops will tend to derail such an approach, with one of the components at the top of this access graph chain potentially being identified as the core.

A more robust heuristic might be to choose the component whose average path length to all the other components is minimized. The path lengths for our example are presented in Figure 14. In this figure we

	0	1∪2	3	4	5	6	7∪8	9∪10	11	Average
0	0	1	1	1	1	1	2	2	2	1.2
1∪2	1	0	2	2	2	2	3	3	3	2.0
3	1	2	0	2	2	2	3	3	3	2.0
4	1	2	2	0	2	2	1	1	3	1.6
5	1	2	2	2	0	2	3	3	1	1.8
6	1	2	2	2	2	0	3	3	3	2.0
7∪8	2	3	3	1	3	3	0	2	4	2.3
9∪10	2	3	3	1	3	3	2	0	4	2.3
11	2	3	3	3	1	3	4	4	0	2.6

Fig. 14. Average path lengths from each component

have counted the hops from component to component, i.e. ignoring the articulation point nodes. Using the average length as our measure would result in component 0 being chosen as the root, which matches with our intuition. Given this choice of root we can order the tree links to introduce the concept of moving towards and away from the core, as illustrated in Figure 15.

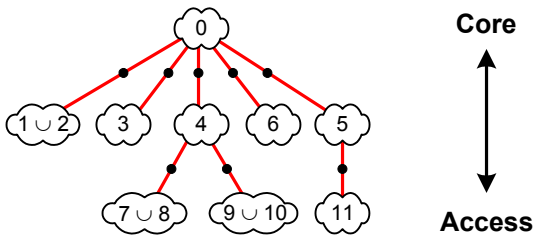


Fig. 15. Rooted tree of components

6.3 Fine-tuning

No heuristic is always going to be perfect. There are two things that could go wrong with this one. We might pick the wrong root entirely. This is easy to remedy. The optimizer application would highlight what it thought constituted the core component. If this were incorrect then the application would just need a mechanism for the user to select an alternative router. The component containing this router would then be treated as the core component.

The other problem stems from our intended usage of these distinctions. The next section describes in more detail the demand replacement process. However, crudely speaking, the strategy is to move demands up the component tree, optimize the demands once they reach the top, and then use the resulting LSPs as tunnels to support the original demands. This approach creates an asymmetry between components. To see why, consider an example where the core itself is partitioned by an articulation point, illustrated in Figure 16. One of these components will be chosen as the "core" and the other as just part of the

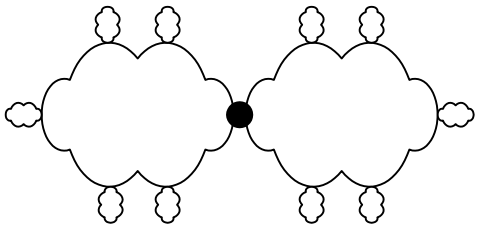


Fig. 16. A core with two components

access graph. The problem here is that we will end up optimizing paths though only part of the core, with the other part using effectively shortest-path first routing. At this point we presumably need human intervention, to make our true intentions clear. However, the extent of this intervention could be relatively painless. Just as we can alter the core component by selecting a single node in another component, rather than all the nodes that make up the new core, here we could add the facility to merge components to make a bigger core. For example, suppose the system had chosen the left-hand large component as the core, and the right-hand component contained the node n. All we would have to do, having noticed that n wasn't in the core, would be to select it and ask that the component containing n be added to the core. This would result in the component graph shown in Figure 17. The modified component graph would

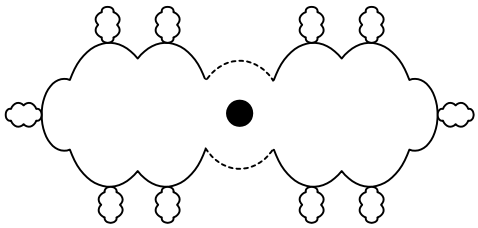


Fig. 17. Merging the core components

allow all paths through the core to be optimized. The user could also extend the core into some of the more complex access structures if desired, using the same mechanism. Such a mechanism could give us a flexible way of bounding the running time and complexity of the optimization step without requiring a lot of manual intervention from the user. Of course, we may have situations where the core component, rather than being too small, is too large. We address this situation in Section 8.

7. DEMAND REPLACEMENT AND OPTIMIZATION

Section 3.2 replaced demands that originated or terminated in the access trees with new demands that just spanned the core. The core demands were then optimized and the solution used to build paths for the original demands. This approach is no longer adequate when we have more general access topologies. We need an alternative mechanism that can deal with components made from graphs, not just trees. Our strategy will be similar in spirit to the tree case; we will replace demands that originate in the access network with others that just span the core. However, there will be two crucial differences. We may need to traverse multiple components before we reach the core. We may also need to perform an optimization step for each of these components as there are now multiple paths across (some of) these access components. Before describing the demand replacement and optimization process in detail, we need some definitions.

OBSERVATION 7.1. *All rooted component trees will have a component at the root, components at the leaves, and with adjacent components being separated from each other by articulation points.*

A component is a set of nodes with a unique identity. Whilst no node can appear in more than one component, components may contain an empty set of nodes. The sets themselves are therefore insufficient to distinguish between components. Where it causes no confusion, we will use a component as if it were the node set. Given a component C we will use $AP(C)$ to denote the articulation points directly attached to C, i.e. the nodes connected to C by a single edge in the component tree. We will use C^+ to stand for the set $C \cup AP(C)$.

DEFINITION 7.2. *Every node n in the original network is either an articulation point in the component graph, or a member of a component in this graph. The function $\mathcal{C}$ records this relationship:*

$$\mathcal{C}(n) = \begin{cases} n & \text{if } n \text{ is an articulation point;} \\ C & \text{if } n \in C. \end{cases}$$

THEOREM 7.3. *Given any two nodes n_1 and n_2 in the original graph, with a path between them, there will be a unique acyclic path between $\mathcal{C}(n_1)$ and $\mathcal{C}(n_2)$ in the component tree. This path will consist of a chain of alternating components and articulation points.*

Proof: A node cannot be both an articulation point and a member of a component. Furthermore, all articulation points are distinct, and the component sets are all disjoint. So $\mathcal{C}(n_1)$ and $\mathcal{C}(n_2)$ are unique nodes in the component tree. By definition, there is a unique path between these points in the tree. By Observation 7.1 this path must consist of an alternating chain of components and articulation points. ■

DEFINITION 7.4. *The* **component path** *between two nodes n_1 and n_2 in the network is the sequence of components obtained by removing the articulation points from the unique acyclic path through the component tree between $\mathcal{C}(n_1)$ and $\mathcal{C}(n_2)$.*

LEMMA 7.5. *There is a unique component path for each demand; it is the component path from the ingress to the egress.*

DEFINITION 7.6. *A* **singleton** *demand has identical ingress and egress nodes.*

Section 7.2 illustrates how singleton demands may be created as part of the process of lifting the demands up the component tree.

DEFINITION 7.7. *A demand is* **local** *if the component path for the demand has length ≤ 1, and is* **non-local** *otherwise.*[1]

DEFINITION 7.8.
The **ingress component** *of a demand d is the first element in the component path for d.*

[1]We allow the length to be zero for the degenerate case of a singleton demand at an articulation point.

The **egress component** *is the last element in this path.*
A demand d **traverses** *component C if C is in the component path for d and is neither the ingress or egress component.*

Associated with every component C is a set of demands $\mathcal{D}(C)$. These are the demands whose component path includes C. Every component also has a set of child components, $\mathcal{CS}(C)$, possibly empty. These are the descendents of C in the component tree connected to C via a single articulation point.

DEFINITION 7.9. $\mathcal{D}(C) = \{d \mid C \text{ is an element of the component path for demand } d\}$

DEFINITION 7.10. $\mathcal{CS}(C) = \{C' \mid \exists AP \cdot C' \rightarrow AP \rightarrow C \text{ is a path in the component tree }\}$

If all the demands in $\mathcal{D}(C)$ are local to C then we can optimize the routing of these demands across the component without considering any other components. Where we encounter demands that are not local, our strategy will be to replace them by demands that start or finish higher in the component tree. By repeatedly applying this process, all the demands will eventually become local demands of some component. More precisely, given a non-local demand from n_i to n_e we will lift it up to the lowest common ancestor of $\mathcal{C}(n_i)$ and $\mathcal{C}(n_e)$ in the component tree. This might be a component or an articulation point.

The complicating factor in this process is the QoS constraints attached to each demand. These define the acceptable paths for the demand. When we replace a demand by another one that starts or finishes higher in the tree, we must calculate new QoS constraints that take into account the cost of reaching the new endpoint from the original one. When all components were access trees we could do this in a single step. The paths though the access trees were unique, and so it was trivial to compute the cost of traversing these paths. However, in the more general case there may be multiple paths though each component, raising the question of which cost we should use. Consider the case where the lowest common ancestor of a demand d, with ingress n_i and egress n_e is the component C_a. Furthermore, for simplicity assume that neither n_i or n_e are elements of C_a^+. This situation is illustrated in Figure 18. There will be a unique sequence of components, $\langle \mathcal{C}(n_i) = C_1, C_2, \ldots, C_k = C_a \rangle$, for some $k >= 2$, that must be traversed in order to reach C_a from the ingress. There will be another sequence of components that must be traversed to reach the egress from C_a. Furthermore, the path from $\mathcal{C}(n_i)$ to C_a will enter via some articulation point, AP_i say, and leave C_a via another articulation point, AP_e. These two nodes will be different, as otherwise, this articulation point would be the lowest common ancestor, not C_a.

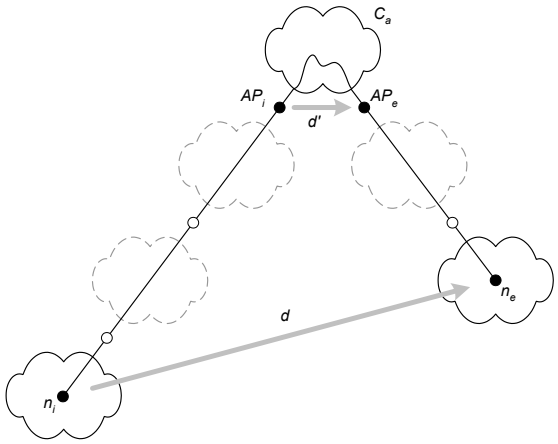


Fig. 18. Lifting a demand up the component tree

Let d' be the replacement demand with ingress AP_i and egress AP_e. What QoS constraint should be attached to this demand? We need to determine the cost of traversing all the components between n_i and AP_i. We also need the cost from AP_e to n_e, but this can be treated in a similar way, and so will be ignored in the following discussion. We could use the shortest route costs across the C_i components.

This would allow us to lift the demand up to AP_i in a single step. However, we may find that the solution we generate may be invalid. There may be insufficient capacity in the access components to route all the non-local demands using the shortest possible routes. The other extreme, where we use the worst-case cost estimate for each component, may result in a demand between AP_i and AP_e with a QoS constraint that is so restrictive that no path can satisfy it. Our strategy will be to lift the demands up the tree one component at a time, optimizing as we go along. This guarantees that each demand can be routed through the child component(s), whilst ensuring that the lifted demand is not excessively restricted by its QoS constraint.

If d is a non-local demand originating at I then we will split it into two parts, d_l and d_r. This process is illustrated in Figure 19. In the case of an ingress component demand d_l will form the local leg, and d_r the remainder of the route to E. The original demand d has a QoS constraint associated with it. This might constrain the total delay permissible along any path used to carry traffic for d for example. Clearly such a limit has to be split between d_r and d_l. The more freedom we give to the routing across d_l the less we have for routing d_r, and vice versa. If there is a unique path from I to the articulation point AP then there is no choice. The cost for d_l is fixed by this path, and so we can just subtract this from the original cost to determine the QoS constraint to use for d_r. However, in the more general case there will be many ways of splitting the QoS constraints. Our strategy will be to first solve an optimization problem for the component containing I. We will give preferential treatment to demands such as d_l to increase the likelihood they will be allocated the shortest possible routes through the component. Once we have assigned a path to d_l we can use this to compute the remaining QoS quota for the d_r demand. A similar strategy is used when the arrows are reversed and we are processing the egress component for d. Having determined the QoS constraint required for d_r we can then delegate its placement to the parent component. Once the whole tree has been optimized we use the paths chosen for d_r and d_l to determine the path to use for d.

From this discussion we see that a demand d will either be assigned a set of paths, in the case of a local demand, or a pair of demands (d_l, d_r) otherwise. We refer to these alternatives as the *carrier* of demand d, and define a partial function $Carrier\big(\big)$ to record this association.

$$Carrier = \text{Demand} \nrightarrow (\text{Demand} \times \text{Demand}) \oplus \mathbb{F}(\text{Path})$$

Initially the carrier will be undefined for all demands. The purpose of the algorithm is to set the carriers in a way that satisfies the QoS constraints of the demands. We describe more precisely what we mean by "satisfies" after presenting the algorithm. Strictly speaking, it is not sufficient to just assign a set of paths to a demand; we also need to know how much of the bandwidth should be allocated to each of the paths. For ease of exposition we ignore this detail in what follows.

7.1 The algorithm

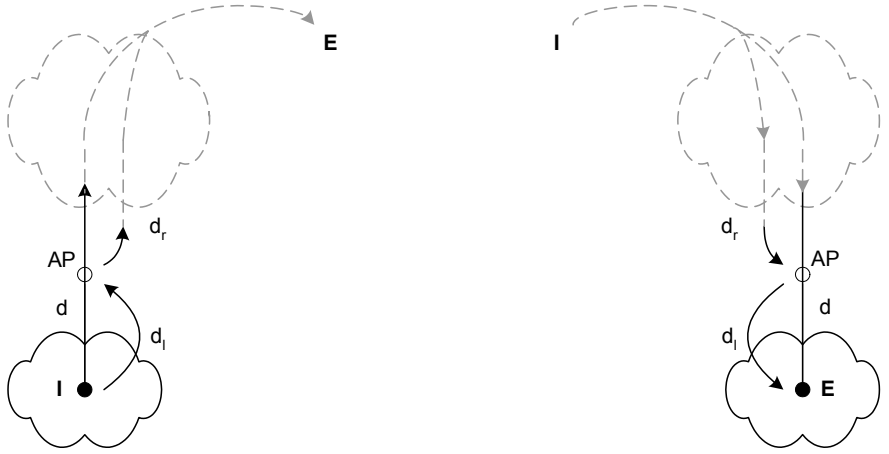
We start by constructing a queue Q of components by performing a postorder traversal of the component tree, skipping the articulation points. The purpose of the algorithm is to define paths for all the demands in the system. A local demand will be allocated one or more paths during the optimization of a component. In the case of a non-local demand the association with paths is implicitly defined by the carrier (d_l, d_r). We use the set $\mathcal{U}$ to record the collection of unprocessed demands. Initially $\mathcal{U}$ will contain all the demands. The algorithm processes each component in Q until the queue is empty, maintaining the following invariant:

INVARIANT 7.11. *Let C be the component at the head of Q.*

$$\forall d \in \mathcal{U} \cap \mathcal{D}(C) \cdot \forall C' \in \mathcal{CS}(C) \cdot d \notin \mathcal{D}(C')$$

In other words if a demand is not local then it must leave or enter C via the unique parent articulation point for C.

LEMMA 7.12. *Let C be the component currently at the head of Q. If Invariant 7.11 holds then for all demands in $\mathcal{U} \cap \mathcal{D}(C)$ either the ingress or the egress (or both) are members of C^+; the demand cannot just be traversing this component.*



a) Ingress component b) Egress component

Fig. 19. Demand carriers

Proof: Consider a non-local demand d, with ingress I and egress E. As no demands traverse child components of C this implies C must have a parent component C_p and a unique articulation point AP connecting C to C_p. If I is not a member of C^+ then the demand must pass through C_p and AP before reaching C. If E is not a member of C^+ then the demand must also pass through AP and C_p when leaving C. This is impossible as component paths are acyclic. ■

ALGORITHM 7.13.

Construct queue Q by performing a postorder traversal of the component tree, skipping the articulation points.
Set $\mathcal{U}$ to be the set of all demands.
while Q is not empty **do**

1. Let C be the head of the queue.
   Construct an empty map $\mathcal{L} : \text{Demand} \nrightarrow \text{Demand}$.
   If all the demands in $\mathcal{D}(C)$ are local then skip to 5.
2. Otherwise, by Invariant 7.11, there must be a parent articulation point AP.
3. **for each** non-local demand d, with ingress n_i and egress n_e
   Either C is an ingress component or an egress component for d due to Lemma 7.12.
   If it is an ingress component then create a new local demand d_l from n_i to AP;
   otherwise create d_l from AP to n_e.
   $\mathcal{L} \leftarrow \mathcal{L} \oplus (d \rightarrow d_l)$.
4. $\mathcal{U} \leftarrow (\mathcal{U} - \operatorname{dom} \mathcal{L}) \cup \operatorname{ran} \mathcal{L}$
5. At this point all the demands in $\mathcal{U} \cap \mathcal{C}(C)$ are local, and so we can compute a set of paths for each of these. If C is a component tree then the solution is purely deterministic. In the more general case we have to solve an optimization problem. We want to minimize the routing cost of the local carrier demands, i.e. the demands in $\operatorname{ran} \mathcal{L}$, to give the corresponding continuation demands the maximum routing freedom. We assume a path-based optimization strategy is being used and start by assigning the shortest weight path (or paths) to the local carrier demands, and a more complete set of paths to the remaining demands. We then try to solve this network problem; the exact strategy is not too important from a global perspective. If no solution can be found, i.e. we can't satisfy all these demands, then we may need to widen the set of paths for the carrier demands and try again. Set $Carrier(d)$ to the set of paths assigned to d by the optimization process for all local demands d. If the optimization strategy allows multiple paths to be assigned to a demand we limit this flexibility to the non-carrier demands. I.e. we assume that if

$d \in \operatorname{ran}\mathcal{L}$ then $|\mathit{Carrier}(d)| \leq 1$. If a demand cannot be satisfied, for example because the QoS metric is too restrictive, then the carrier will be the empty set.

6. **for each** pair $(d, d_l) \in \mathcal{L}$
   C may be the ingress or egress component for d. The cases are symmetrical so we just consider the ingress case. Create a new demand d_r, starting at AP, with the same properties as d, except that the QoS constraint is reduced by the weight of the path allocated to d_l.
   $\mathit{Carrier}(d) \leftarrow (d_l, d_r)$.
   if d_r is a singleton demand
   $\mathit{Carrier}(d_r) \leftarrow \{\epsilon\}$
   else
   $\mathcal{U} \leftarrow \mathcal{U} \cup \{d_r\}$
7. $\mathcal{U} \leftarrow \mathcal{U} - \mathcal{C}(C)$
   $Q \leftarrow \mathrm{t}ail\ Q$

LEMMA 7.14. *Algorithm 7.13 preserves Invariant 7.11.*

Proof: In the initial state the component at the head of Q is by definition a leaf in the component tree, and so has no child components. The invariant therefore trivially holds in the initial state.

The postorder traversal of the component tree guarantees a component reaches the front of the queue only after all its children have been processed. To ensure the invariant holds each time the queue is popped we merely have to check that all the demands in $\mathcal{C}(C)$ are removed from $\mathcal{D}$ before C is removed from the queue. This is guaranteed by Step 7. ■

LEMMA 7.15. *Algorithm 7.13 preserves the following relationship between $\mathcal{U}$ and Q after each loop iteration:*

$$\forall d \in \mathcal{U} \cdot \forall C \in \text{ component path of } d \cdot C \in Q$$

Proof: The relationship holds initially as Q contains all the components. The algorithm adds demands to $\mathcal{U}$ at step 4. But these are all local to C, and all local demands get removed from $\mathcal{U}$ at the end of each iteration in step 7. However, demands also get added in step 6. But each demand d_r added in this step will have the same component path as an existing demand, d, except that it won't include C. At the start of the loop we know that all the components in the path for d are in Q. So at the end of the loop we can remove C from Q whilst satisfying the invariant for d_r. ■

DEFINITION 7.16. *A demand d is placed if either*

a) $\mathit{Carrier}(d) \in \mathbb{F}(\mathrm{Path})$, *or*

b) $\mathit{Carrier}(d) = (d_l, d_r)$ *for some placed demands d_l and d_r.*

LEMMA 7.17. *On completion of the algorithm if $\mathit{Carrier}(d) = (d_l, d_r)$, for some d, d_l and d_r, then $\mathit{Carrier}(d_l) \in \mathbb{F}(\mathrm{Path})$ and $|\mathit{Carrier}(d_l)| \leq 1$.*

THEOREM 7.18. *After running Algorithm 7.13 every demand d is placed.*

Proof: Step 5 guarantees that each local demand will be assigned a set of paths. Step 6 guarantees that each non-local demand will have the carrier set to a pair of demands (d_l, d_r). The demand d_l will be satisfied by Step 5, as it is a local demand. Lemma 7.15 guarantees that the demand d_r only involves components still in Q, and so will be placed when these components are processed. ■

7.2 Singleton demands

The algorithm may create singleton demands as part of the lifting process. This is illustrated in Figure 20. In the left-hand tree we push the demand from AP_1 to AP_2 up to a demand from AP_3 to AP_4. This demand will be local to C, and so will be routed when C is optimized. If we try the same thing in the second example then we create a replacement demand from AP_3 to AP_3. Step 6 in the algorithm assigns an empty path to such demands. When the paths are concatenated to determine the routes for the original demands such paths will disappear.

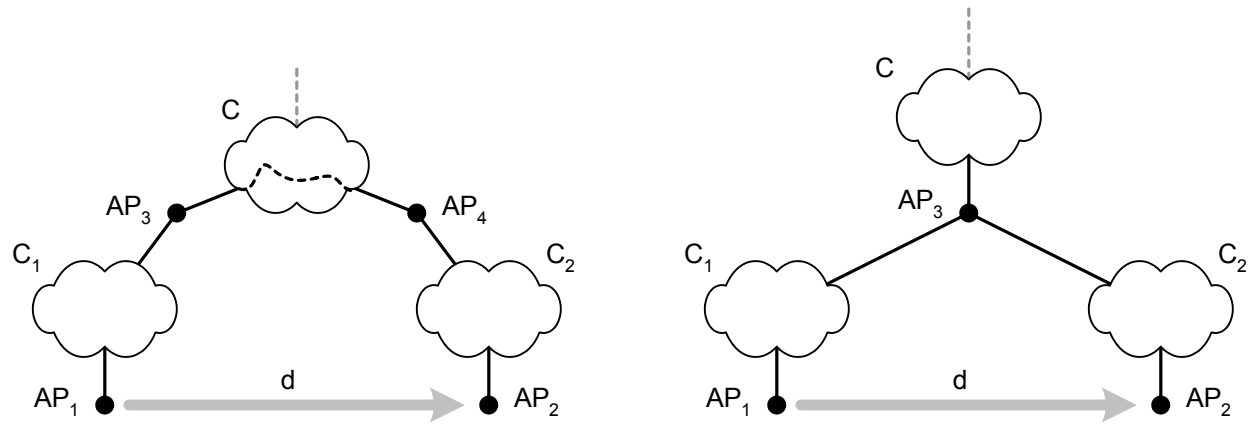


Fig. 20. Singleton demands

7.3 Constructing the paths

The following lemma shows the carriers form a chain of finite length.

LEMMA 7.19. *If* $Carrier\big(d\big) = (d_l, d_r)$ *then* $|$*component path for* $d$$| > |$*component path for* $d_r$$|$

The chain terminates when we reach a demand d_r where $Carrier\big(d_r\big) \in \mathbb{F}(\text{Path})$. Let $\oplus$ be a binary operator that concatenates paths, reordering them if necessary. For example, if paths were represented by a sequence of nodes then we would define $\oplus$ as follows.

DEFINITION 7.20. *Let* p_1 *be the path* $\langle n_{11}, n_{12}, \ldots, n_{1i}\rangle$, $i >= 0$, *and* p_2 *be the path* $\langle n_{21}, n_{22}, \ldots, n_{2j}\rangle$, $j >= 0$. *Then*

$$p_1 \oplus p_2 = \begin{cases} p_1 & \text{if } j = 0 \\ p_2 & \text{if } i = 0 \\ \langle n_{11}, n_{12}, \ldots, n_{1i}, n_{22}, \ldots, n_{2j}\rangle & \text{if } n_{1i} = n_{21} \wedge n_{2j} \neq n_{11} \\ \langle n_{21}, n_{22}, \ldots, n_{2j}, n_{12}, \ldots, n_{1i}\rangle & \text{if } n_{2j} = n_{11} \wedge n_{1i} \neq n_{21} \\ \text{undefined} & \textit{otherwise}. \end{cases}$$

A similar definition could be written for the case where paths were represented by a sequence of edges. We can now expand the carriers into explicit paths using the following definition, which is well-defined as a consequence of the previous lemma.

DEFINITION 7.21. *Let*

$$\mathcal{P}(d) = \begin{cases} Carrier\big(d\big) & \text{if } Carrier\big(d\big) \in \mathbb{F}(\text{Path}); \\ \{p \mid \exists p' \in \mathcal{P}(d_r), p_l \in Carrier\big(d_l\big) \cdot p = p_l \oplus p'\} & \text{if } Carrier\big(d\big) = (d_l, d_r) \end{cases}$$

The set $\mathcal{P}(d)$ may be empty if the algorithm has been unable to place the demand. This would typically be caused by the QoS metric for the demand being too restrictive, or the network overutilized. We would like to prove that if the algorithm fails to place one or more demands then all other algorithms would also fail. However, this is not true. To see why, consider the case of a heavily utilized access component C, with two non-local demands d_1 and d_2. These will be split into the carriers (d_{1l}, d_{1r}) and (d_{2l}, d_{2r}). It may not be possible to assign the shortest routes to both d_{1l} and d_{2l}, so one of them will end up getting preferential treatment. This, in turn, will effect the QoS constraints assigned to d_{1r} and d_{2r}. At the time these decisions are made we don't know how difficult it will be to route d_{1r} and d_{2r} across the remainder of the network. So one choice may result in both demands being routed, whereas the other choice may result in one failing to be placed. This is the nature of such heuristics. By splitting the problem into a number of smaller optimization steps, we lose optimality. However, by giving preference to demands such

as d_{l1} and d_{l2} during the local optimization steps we hope to minimize the problem, particularly where networks are not heavily overutilized.

7.4 Parallelism

Our description of the optimization process is more sequential than it needs to be. Instead of a queue, we could just allow a component to be processed once all its descendents in the component tree have been processed. Relaxing the ordering introduces the possibility that the carrier may already be set when the algorithm reaches Step 6. Consider the situation illustrated by Figure 21. Moreover, suppose we are processing component C_1. We replace demand d by a local demand d_l from I to AP_1 and then optimize C_1. At the end of the optimization process we construct a demand d_r, which originates at AP_1 and terminates at E. We must then set $Carrier(\mathsf{d}) = (\mathsf{d}_l, \mathsf{d}_r)$. But if a separate thread or process has been processing C_2 in parallel, we may find that $Carrier(\mathsf{d})$ has already been set to $(\mathsf{d}'_l, \mathsf{d}'_r)$, for some demand d'_r from I to AP_2, and d'_l from AP_2 to E. In this case, our new demand, d_r should be from AP_1 to AP_2, and we would set $Carrier(\mathsf{d}'_r)$ to be $(\mathsf{d}_l, \mathsf{d}_r)$. Of course, we may find that $Carrier(\mathsf{d}'_r)$ has also been set. Therefore, the algorithm must follow the chain of carriers until it reaches a demand d_u with an undefined carrier. This demand is locked to prevent any other thread updating it. The appropriate demand, d_r would be constructed and the carrier for d_u updated. The demand would then be unlocked, allowing other threads to extend the chain if blocked.

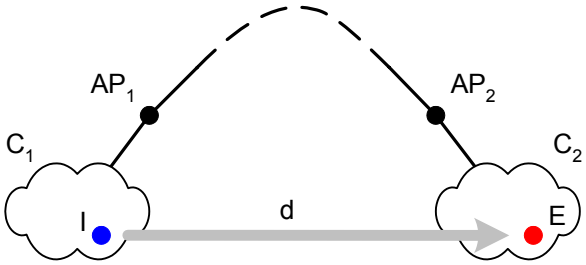


Fig. 21. Updating carriers

7.5 Tunneling

The function $\mathcal{P}(d)$ converts the carriers into an explicit set of paths for each demand. We can convert each of these into an LSP to support d. Consider the common situation where the lowest common ancestor component of the ingress and egress of d is the core component. Let the core ingress articulation point for d be AP_i and the egress point be AP_e. There may be many access demands that cross the core from AP_i to AP_e with the same traffic class. It may make more sense to aggregate the bandwidth for these demands and construct a single LSP from AP_i to AP_e. We can then use this as a tunnel for demands such as d. This reduces the LSP state that must be maintained by the routers in the core.

We could go further, using the carriers to suggest further scope for tunneling. However, the structure they expose is really an artifact of the lifting process. For example, in the case of parallel execution, the exact sequence of carriers will depend on the relative speed of the threads or processes. They are therefore unlikely to generate an efficient set of tunnels.

The component hierarchy could be used to suggest further potential tunnels. For example, suppose we processed all components in reverse breadth-first order. All components at the same depth in the tree would be processed before any components higher in the tree. The carrier chains would then be more constrained, allowing us to build tunnels that spanned components of equal depth. If the LSPs are constructed after the algorithm has run, rather than incrementally, then the demands in the carrier chains could be rearranged to achieve a similar effect without altering the traversal order. However, it's not clear that the hierarchy we identify via biconnectivity has sufficient semantic importance that even this would yield a good set of tunnels. A more satisfactory solution may require exploiting attributes of

the network itself, such as organizational boundaries, to suggest ways of grouping the carrier demands to best exploit the potential for tunnels.

Ideally, we would like to aggregate demands with common properties as we proceed up the component tree. The order in which we process components may also affect the potential for such aggregation. We conjecture that traversal orders that attempt to optimize tunnel production may also increase the likelihood of demand aggregation.

8. COMPLEX ACCESS GRAPHS

Consider the scenario illustrated in Figure 22. An access component and the core component are merged

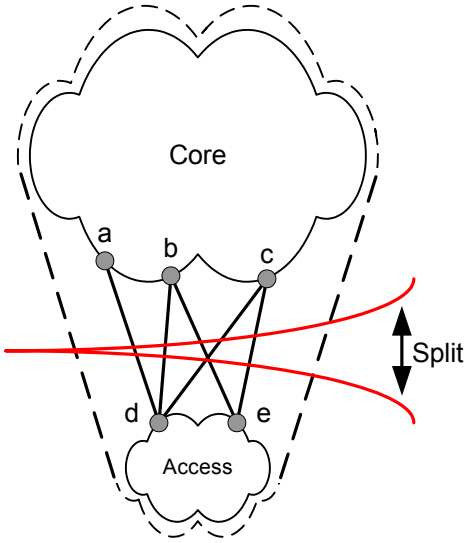


Fig. 22. Splitting a component

into one large component due to the rich connectivity between them. We would like to split these components, either to reduce the size of the optimization task, or to enforce administrative boundaries. In our approach components are separated by articulation points. To separate these two components we would need to chop the real links between them and introduce a new virtual node to play the role of the articulation point. Figure 23 illustrates this process. Whilst case a) appears simpler, it is not clear what link metrics should be assigned to the new edges. In case b) we use the same link metrics as in the original graph. For example, the edge bd will have the same cost/weight as the edge from b to d in the original graph. At first glance, it appears we will count the weights twice as we traverse to/from the core via AP. However, our optimization process will take care of this duplication.

Having split the component in two, we can use the technique of Section 6.2 to root the component tree, and hence identify one of these components as a grandparent of the other. We optimize the access component as before, treating the virtual articulation point just like any other articulation point connected to this component. However, when we reach the stage where the external demands must be lifted to the component above we need to treat these virtual nodes specially. To see why, consider the situation where an external demand d has been partially replaced by a local demand d_l terminating at AP. Suppose this demand has been satisfied by a path that reaches AP via the link db. When constucting the continuation demand for d_l we should not start it at AP, as our solution has already committed us to entering the core at node b. The continuation demand should therefore be created with b as the ingress. Demands terminating in this access component would be treated in a similar fashion. Using this approach we see that the links from AP to the core are redundant. When we decompose two components we will not necessarily know which component will be end up as parent, and which as child, and therefore in which

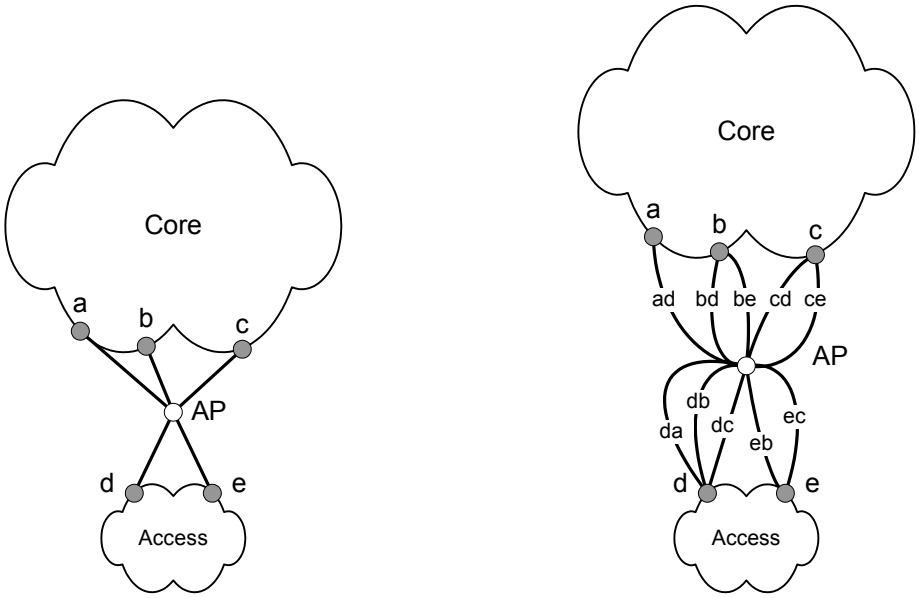


Fig. 23. A virtual articulation point

order the components will be processed by the optimizer. By including both sets of links initially, the approach will work irrespective of which order is eventually chosen.

This strategy has the advantage of simplifying the interface between the core and the access components. The access optimization phase is responsible for choosing how access demands reach the core, i.e. which core router they should use. It simplifies the interface between the components, which may be an important issue where they are managed by different organizations. However, it is worth noting that this is not the only approach we could adopt. For example, suppose the links from the access component to AP had zero cost, and there was a single link from d to AP, and from e to AP. Optimizing the access component would now only commit us to the exit points for each external demand, e.g. node d or e. The continuation demand across the core would then have either d or e as the ingress point. Whilst this gives the core optimizer more freedom, allowing it to choose where a demand should enter the core in some cases, it complicates the process. For example, we would have to (temporarily) view d and e as core routers, or articulation points, for the duration of the core optimization. This may be impossible when organizational boundaries and procedures are involved.

Even with this additional freedom, the decisions are still asymmetric. For example, if the access component has routed a demand via node e then this prevents such a demand reaching the core via node a. A global optimizer might find a better solution by routing the demand to d, and thence to a. We may also find there is insufficient link capacity to cross from the access to the core with the exit points we are committed to. This is the price we pay for decomposing the problem. In some cases, we should optimize the access graph and let the solution constrain the core problem. In other cases it may be better, albeit more complex, to optimize the core and hope we can extend the solution to one across the core.

Figure 22 illustrated the situation where we started with a single component. Of course, in practice, our starting point is likely to be more complex than this, with other articulation points already connected to the component. Figure 24 contains an existing articulation point that has links to routers in both the new core and access components. Introducing the new virtual node does not split the components in this case as they are still biconnected. The transformation only works when the existing articulation points impinging on the original component have links that do not act as bridges between the new components. Whilst this is frequently the case, it will not always be so. Further heuristics need to be developed for these cases, leading us even further from any pretence at optimality. For example, we may need to resort

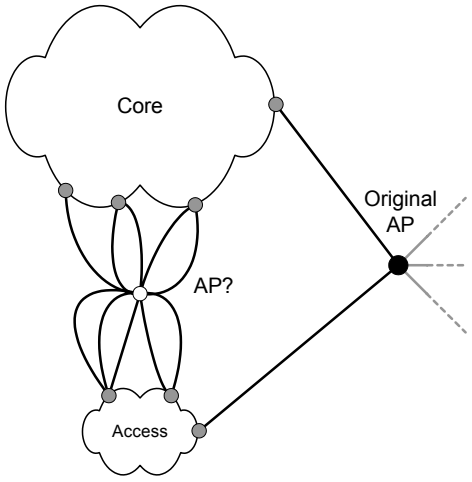


Fig. 24. An invalid split

to marking links as being "backup" connections, ignoring them for placement purposes, and simplifying the connectivity.

9. ADMINISTRATIVE PARTITIONS

Many network operators split the management of the network across multiple organizational boundaries. It is important to align our components with this organization so we don't attempt to optimize a collection of routers under the control of multiple groups. Note that this does not imply we want to construct only as many components as there are organizational entities. Nevertheless, we must ensure that no components are split across such entities. Section 6.1 discussed component merging, and Section 8 discussed splitting. Given a predefined grouping of routers there will be a need to automate the merging and splitting of components identified by the biconnected component analysis so our components respect this grouping. Our description of the algorithm in Section 7.1 attempted to place all the demands. However, when the network is partitioned along administrative boundaries we have to refine this approach. For example, suppose you were managing the access network. You would run the algorithm until the demands were lifted to the core component(s). The resulting demands would be presented to the core team as a set of requirements. These would eventually be satisfied by a set of LSPs, which would then be fed back into the access optimizer. This component could then complete the provisioning of the access LSPs. In some scenarios, such as the VoIP gateway case, it may be acceptable for these core demand requirements to be satisfied by a collection of LSPs, to spread the load. A mechanism for specifying such flexibility would need to be developed.

10. CONCLUSIONS

In this document we have discussed various algorithms to decompose network topologies in a way that simplifies the optimization of demand placement. Access trees are simple to identify, and in some cases may be sufficient to yield a tractable problem. An approach based on the identification of biconnected components was developed for those examples where the access elements of the network are more complex in structure. The optimization process is more involved in this case, but allows a far richer collection of networks to be tackled. To align the components with administrative boundaries, and to split individual components that are still too large to optimize as a whole, we introduced virtual articulation points. Of course there will still be some networks where none of these techniques will be sufficient, requiring the introduction of further heuristics to assist in their decomposition. Access to real network topologies is often difficult, with operators frequently treating such information as being sensitive. At this stage it is therefore difficult to assess the extent to which these techniques will be sufficient, and what, if any,

additional techniques will be required. Only by implementing these techniques, and then deploying them widely, will we be truly able to assess their utility.

One important aspect of MPLS network optimization is the need for path protection. Constraining a demand to an explicit path may make efficient use of the network resources. But if one of the links or routers along this path fails then we need to be adaptable, preferably without just falling back to best-effort routing. Many approaches have been proposed to tackle this problem, e.g. [Bejerano et al. 2003]. Our optimization strategy is based upon exploiting bottleneck nodes, either naturally occurring in the network, or artificially created to help the decomposition process. There is an obvious conflict here, as bottlenecks are undesirable from a path-protection standpoint. We may need to group multiple nodes and links into virtual nodes, allowing redundancy at the physical level whilst looking like a single object to the optimizer. It would also be interesting to investigate whether the hierarchical structure produced by our technique could be used to simplify the path restoration problem as well.

REFERENCES

Bejerano, Y., Breitbart, Y., Orda, A., Rastogi, R., and Sprintson, A. 2003. Algorithms for computing QoS paths with restoration. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies*. Vol. 2. IEEE, 1435 – 1445.

Cisco. 2003. Autobandwidth allocator for MPLS traffic engineering:. White paper.

CPlane. 2003. Traffic optimizer product overview. White paper.

Köhler, S. and Binzenhöfer, A. 2003. MPLS traffic engineering in OSPF networks - a combined approach. Tech. Rep. 304, Institute of Computer Science, University of Würzburg. February.

Liu, G. and Ramakrishnan, K. G. 2001. A*prune: An algorithm for finding k shortest paths subject to multiple constraints. In *INFOCOM*. 743–749.

Mitchell, K. 2004. The TOAD optimizer. Tech. rep., Agilent Labs. In preparation.

RAD. 2004. RAD introduces TDMoIP PMC for OEM developers. http://www.rad.com/Article/0,6583,18087,00.html.

Sedgewick, R. 2001. *Algorithms in C++ Part 5: Graph Algorithms*. Addison-Wesley.

Wandl. 2002. IP/MPLSView: Integrated network planning, configuration management & performance management. White paper.