Bandwidth Allocation Planning in Communication Networks

Christian Frei, Boi Faltings
Artificial Intelligence Laboratory, Swiss Federal Institute of Technology (EPFL), CH-1015 Lausanne, Switzerland
E-mail: Christian.Frei@epfl.ch

The fundamental issue of quality of service routing has triggered a lot of research during the last years. However, the proposed algorithms attempt to route communication demands only on a call by call basis, without taking into account future traffic. There are nonetheless cases where the traffic profile is known. In this paper, we address this related problem to quality-of-service routing, the off-line planning of bandwidth allocation to demands known in advance. Shortest path routing is the traditional technique applied to this problem. However, this can lead to poor network utilization and even congestion. We show how an abstraction technique combined with systematic search algorithms and heuristics derived from Artificial Intelligence make it possible to solve this problem more efficiently and in much tighter networks, in terms of bandwidth usage. In some cases, the abstraction technique also allows to prove during search that some allocation problems are indeed infeasible, without having to explore the whole search space. The regions between which bandwidth must be added are then identified.
Keywords: QoS-routing, constraint-based routing, resource allocation planning, abstraction, constraint satisfaction.

1. Introduction

A central problem in the field of communications is the automatic routing of traffic through a given network. Currently, shortest path routing according to some metric is most often used to route traffic across a network on a call by call basis. However, the upcoming high-speed networks are expected to offer a wide range of services to an increasingly large number of users, with a diverse range of Quality of Service (QoS) requirements. This raised the issue of quality of service routing of communication demands: the goal is now not only to achieve global efficiency in network resource utilization, but also to satisfy the QoS requirements of each admitted connection. Wang and Crowcroft [31] proved that the allocation of every single demand is NP-hard by itself, when demands are subject to multiple additive or multiplicative quality of service (QoS) criteria (such as delay and loss probability). This triggered much research during the last years [3], and the proposed QoS-routing algorithms are mainly variations of the shortest path method, differing in the metric (or metric combination) for link length and the exploration of the network for path computation.

The above mentioned routing algorithms do not make any assumption about future traffic. However, there are cases in which the traffic profile is known. The use of *global information*, including not only the available link capacities but also the expected traffic profile for the period in question, can lead to routing strategies designed to minimize congestion and make better use of available network resources. In this paper, we consider the problem of allocating in an *off-line* manner a set of demands known in advance within the resource capacities of a communication network. This situation may arise for instance when setting up virtual private networks in a connection-oriented network (e.g., ATM, TDM) of a provider; planning the routing of virtual path connections (VPC) in an ATM network; planning the routing of virtual channel connections (VCC) in the VPC network of an ATM backbone; or optimizing the routing tables of an IP network (demands are then derived from objective traffic measurements, which almost every network operator carries out).

From the routing point of view, the key resource to manage in networks is bandwidth. Therefore, in order to make better use of available network resources, there is a need for planning bandwidth allocation to communication demands, in order to set up routing tables (or any other route selection criterion) more purposefully. We define the problem of *resource allocation in networks* (RAIN) as follows:

Given a network composed of nodes and bidirectional links, where each link has a given bandwidth capacity, and a set of communication demands to allocate,
Find one and only one route for each demand so that the bandwidth requirements of the demands are simultaneously satisfied within the resource capacities of the links.

It is important to note that because of technological limitations (for ATM typically) and/or performance reasons, it is impossible to divide demands among multiple routes. (Nonetheless, there may be several demands between same endpoints, and each of these demands can be allocated over a different route.)

Vedantham and Iyengar [28] showed that the bandwidth allocation problem in the ATM network model is NP-complete. This result can be easily extended to the RAIN problem. The RAIN problem is combinatorial due mainly to the exponential number of routes in a network. Suppose the network is simple but complete (this is not even the worst case, since a communication network is a multi-graph: it allows multiple links between same endpoints) with n nodes. A route is a simple path, its length in number of links is therefore bounded by $n-1$. Since a route of length j has $j-1$ intermediate (and distinct) nodes, the number of routes of length j is $(n-2)!/(n-j-1)!$. The total number of routes between two nodes is therefore equal to $\sum_{i=1}^{n-1}(n-2)!/(n-i-1)!$. For instance, in a complete graph with 10 nodes, there are 69'281 routes between any pair of nodes.

Currently, most network or service providers use shortest path-based algorithms, without any backtracking on routing decisions, due to the complexity of the problem: given an order of the demands, each demand is assigned the shortest possible route supporting it, or just skipped if there is no such route. Although the use of shortest path routing for each single demand ensures the best possible route for that particular request, Wang and Crowcroft [30] have shown that it can lead to suboptimal routing or even highly congested network routing solutions when one considers the network utilization as a whole. If the shortest path method is very fast, it clearly works well, in terms of accepted demands, only in over-dimensioned networks.

This creates a new situation for the networking community, as traditional routing algorithms such as shortest paths do not perform very well on this problem. In order to do better, we must allow (1) other routes than the shortest paths, (2) *backtracking* to previous allocation in order to squeeze in new demands. Basically, two types of algorithms can be used to solve the RAIN problem: incomplete and complete methods. *Incomplete* algorithms, such as the shortest path method, explore only partially the search space, i.e., they only consider a subset of possible routes (or a subset of route combinations). These algorithms are generally fast, however, they are not guaranteed to find a solution if there is one. On the other hand, *complete* algorithms perform an exhaustive search, and thereby do always find a solution if one exists. However, due to the huge search space, their worst case behavior is exponential. In order to cope with this complexity, there is a need for heuristics to guide the search, and thereby to allow solving most problem instances in a reasonable amount of time.

In practice, the RAIN problem poses itself in the following way: a network or service provider receives a request from the customer to allocate a number of demands, and must decide within a certain time decision threshold whether and how the demands can be accepted. The trade-off we must consider is then completeness (that is satisfying all users if possible in order to ensure customer satisfaction) versus time efficiency (customers require an answer in a reasonable amount of time), and hereby complete versus incomplete algorithms.

In this paper, we restrict QoS to bandwidth requirements. We propose the use of complete search algorithms to address the RAIN problem. We show how an abstraction of the network called Blocking Islands, create a compact representation of the domains which allows the application of *Constraint Satisfaction Problem* techniques such as forward checking, variable and value ordering to the RAIN problem with manageable complexity. We present empirical results that show a phase transition behavior: even if it is NP-complete, this problem can be solved efficiently by our complete algorithms in many problem instances.

The outline of the paper is the following. In Section 2, we introduce our problem model, we give a quick overview of Constraint Satisfaction and the difficulties encountered while the technique is applied to the RAIN problem, and we review some of the related work. In Section 3, we briefly recall the Blocking Island paradigm and outline its major properties. Section 4 illustrates how blocking islands help to route a single demand while attempting to preserve bandwidth connectivity inside the network. In Section 5, we present a generic algorithm and some heuristics to solve the RAIN problem. Section 6 examines how blocking island abstractions allow to prove in some cases that the problem is infeasible by identifying global bottlenecks in the network, or to identify culprit assignments of routes to demands that prevent the allocation of another demand. Empirical results are summarized in Section 7. Finally, we conclude with some directions for future work.

2. Problem definition

2.1. Problem modeling

A communication network is composed of nodes interconnected with communication links. We model it as a connected *network graph* $\mathcal{G} = (\mathcal{N}, \mathcal{L})$ (see Figures 1.d and 8), an undirected multi-graph without loops, i.e., edges whose endpoints are the same vertex. The set of nodes $\mathcal{N}$ are processing units, switches, routers, etc., and the links $\mathcal{L}$ correspond to bidirectional communication media, such as optical fibers. Each link l is characterized by its *bandwidth capacity*, the (currently) *available bandwidth*, and other quality of service parameters, such as delay, loss probability, etc. In the network graph, the weights on the links denote their available bandwidth.

Our network must fulfill communication needs between pairs of nodes, or *demands*. Each demand $d_u = (x_u, y_u, \beta_u)$ is characterized by a source (x_u) and destination (y_u) node, and a bandwidth requirement β_u, the sole QoS parameter taken into account here. A network $\mathcal{G}$ satisfies a set of demands $\mathcal{D}$ by allocating a *route* for each demand. A route is a simple path in the network graph that satisfies the bandwidth requirement. Assigning a route to a demand and reserving the resources needed is called establishing a *connection*.

2.2. RAIN as a constraint satisfaction problem

Constraint satisfaction [26] is a technique which has been shown to work well for solving certain NP-hard problems, and has been applied to a variety of domains [29]. A *Constraint Satisfaction Problem* (CSP) is defined by a triple (X, D, C), where $X = \{x_1, ..., x_n\}$ is a set of *variables*, $D = \{D_1, ..., D_n\}$ a set of finite *domains* associated with the variables and $C = \{C_1, ..., C_n\}$ a set of *constraints*. The domain of a variable is the set of all values that can be assigned to that variable. A constraint between variables restricts the combinations of values that can be assigned to those variables.

Solving a can be done using a *backtracking algorithm* with *forward checking* (FC), a systematic search process. Such an algorithm maintains a partial solution (initially empty) which satisfies all the constraints and attempts to extend it to a full solution. Its basic operation is to pick one variable (demand) at a time, assign it a value (route) of its domain that is compatible with the values of all instantiated variables so far, and propagate the effect of this assignment (using the constraints) to the future variables by removing any inconsistent values from their domain. If the domain of a future variable becomes empty, i.e., there is no value for that future variable that is consistent with the instantiated variable, the current assignment is undone, the previous state of the domains is restored, and an alternative assignment, when available, is tried. A *dead-end* is reached during search when all values of the current variable are rejected. In such a case, the previously instantiated variable is uninstantiated, i.e., it is removed from the current partial solution. This process is called *backtracking*. FC proceeds in this fashion until a complete solution is found or all possible assignments have been tried unsuccessfully, in which case there is no solution to the problem.

Indeed, the RAIN problem is easily formulated as a CSP in the following way: variables are demands, the domain of each variable is the set of all routes between the endpoints of the demand, and constraints on each link must ensure that the resource capacity is not exceeded by the demands routed through it. A solution is a set of routes, one for each demand, respecting the capacities of the links.

However, this formulation presents severe complexity problems, because of the exponential number of routes (Section 1). It is thereby too expensive to compute, represent, and store the domain of a variable, i.e., all the routes that join the endpoints of a demand. The Blocking Island abstractions (Section 3) allows to overcome this problem since they are a compact representation of possible routes satisfying a demand.

2.3. Related work

Surprisingly, there has been little published research on the RAIN problem. Currently, most network providers use (incomplete) shortest path methods, as described in Section 1, due to the complexity of the problem. There are some proprietary tools for this, about which nothing much is known.

Operation Research (OR) techniques are also applied to the RAIN problem. Most often, a fixed number of shortest paths for each demand are precomputed, and the problem is solved using linear programming with very large constraint systems of equations [1,5,16,33]. However, because only a given number of routes are considered, these techniques are not guaranteed to find a solution if one exists. Moreover, OR techniques are not as flexible as CSP-based methods (see Section 8).

Mann and Smith [22] search for routing strategies that attempt to ensure that no link is over-utilized (hard constraint) and, if possible, that all links are evenly loaded (below a fixed target utilization), for the predicted traffic profile. Finally, the routing assignment attempts to minimize the communication costs. Genetic algorithms and simulated annealing approaches were used to develop such strategies. However, they limit the search space being explored to the $k = 8$ shortest paths for each demand. Therefore, as they do not perform an exhaustive search, they may not find a solution even if one exists (using the $j > k$ shortest path for at least one demand).

To our knowledge, the closest published work to ours is the CANPC framework [23]. It is based on the successive allocations of shortest routes to the demands, without any backtracking when an assignment fails. They propose several heuristics to order the demands (such as bandwidth ordering) to provide better solutions, i.e., to route more demands. They are currently developing an optimization tool that takes the partial solution as input to try to allocate all demands. Results show that the methods we propose clearly outperform theirs.

It has long been observed that the complexity of solving a problem can depend heavily on how it is formulated. Giunchiglia and Walsh define abstraction as follows in [13]: *"Abstraction is the mapping of a problem representation into a simpler one that satisfies some desirable properties in order to reduce the complexity of reasoning. The problem is solved in the abstract space and the solution is then mapped back to the more complex ground space."* The ground space refers to the original problem representation. Abstractions are naturally used my humans to solve problems. Reducing problem complexity is a major reason for using abstraction techniques. *Pertinent support to users* is another motivation. It is often desirable to keep the user in the decision loop, since they consider more decision criteria that can be modeled. However, the amount of information generally overwhelms the user. Abstractions should then be used to hide unimportant information and only summarize in a pertinent manner the data he must see. Abstraction were for example used in theorem proving [13], planning [20,19], first order logic [17], learning [27], digital circuits troubleshooting [14], and knowledge bases [21].

Abstraction and reformulation techniques have already been applied to permit more efficient solution of a CSP. Choueiry and Faltings [4] relate interchangeability to abstraction in the context of a decomposition heuristic for resource allocation. Weigel and Faltings [32] cluster variables to build abstraction hierarchies for configuration problems viewed as CSPs, and then use interchangeability to merge values on each level of the hierarchy. Freuder and Sabin [8] present abstraction and reformulation techniques based on interchangeability to improve solving CSPs. A recent collection of papers addressing abstraction, reformulation, and approximation techniques in a variety of Artificial Intelligence domains can be found in [34].

The phenomenon of phase transitions occurring in many types of problems as a control parameter is varied has been recognized and studied extensively in recent years. Cheeseman *et al.* [2] first reported a phase transition between a region where almost all problems have many solutions and are relatively easy to solve, and a region where almost all problems have no solution and their insolubility is relatively easy to prove. In this intervening region, the probability of problem solubility falls from close to 1 to 0, and the cost of searching these problems is highest. Cheeseman suggests the following conjecture: *"All NP-complete problems have at least one order parameter and the hard to solve problems are around a critical value of this order parameter"*. The critical value (or a range) of the order parameter is where phase transition occurs. The value of the order parameter for a problem instance is often called the *tightness* of the problem instance. Noteworthy, Gent *et al.* [11] observed that the relative behavior of algorithms on large and small problems is the same when plotted against this parameter. Comparison of different algorithms can therefore be performed on small problems, and results can be expected to scale to larger problems. Phase transition behavior has been reported in an increasing number of NP-complete problems, including satisfiability problems [18,24], Hamiltonian paths [2], and the traveling salesman problem [10].

3. The Blocking Island paradigm

Frei and Faltings [6] introduce a clustering scheme based on Blocking Islands (BI), which can be used to represent bandwidth availability at different levels of abstraction, as a basis for distributed problem solving. A β-blocking island (β-BI) for a node x is the set of all nodes of the network that can be reached from x using links with at least β available resources, including x. Figure 1.d shows all 64-BIs for a network. Note that some links inside a β-BI, i.e., the links that have both endpoints in the β-BI, may have less than β available resources. In such a case, it simply means that there is another route with β available resources between the link's endpoints. As a matter of fact, link (a, b) has both endpoints in 64-BI N_1 but has less than 64 available resources. However, there are at least 64 available resources along route $\{(a, c), (c, b)\}$.

β-BIs have some fundamental properties. Given any resource requirement, blocking islands partition the network into equivalence classes of nodes. The BIs are *unique*, and *identify global bottlenecks*, that is, inter-blocking island links. If inter-blocking island links are links with low remaining resources, as some links inside blocking islands may be, inter-blocking island links are links for which there is no alternative route with the desired resource requirement. The *inclusion property* states that for any $\beta_i < \beta_j$, the β_j-BI for a node is a subset of the β_i-BI for the same node. Moreover, BIs highlight the *existence* and *location* of routes:

Proposition 1 (route existence property). There is at least one route satisfying the bandwidth requirement of an unallocated demand $d_u = (x, y, \beta_u)$ if and only if its endpoints x and y are in the same β_u-BI. Furthermore, all links that could form part of such a route lie inside this blocking island.

Blocking islands are used to build the β-blocking island graph (β-BIG), a simple graph representing an *abstract* view of the available resources: each β-BI is clustered into a single node and there is an abstract link between two of these nodes if there is a link in the network joining them. Figure 1.c is the 64-BIG of the network of Figure 1.d. An abstract link between two BIs clusters all links that join the two BIs, and the abstract link's available resources is equal to the maximum of the available resources of the links it clusters (since a demand can only be allocated over one route). These abstract links denote the critical links, since their available resources do not suffice to support a demand requiring β resources.

In order to identify bottlenecks for different βs, e.g., for typical possible bandwidth requirements, we build a recursive decomposition of BIGs in decreasing order of the requirements: $\beta_1 > \beta_2 > ... > \beta_b$. This layered structure of BIGs is a *Blocking Island Hierarchy* (BIH). The lowest level of the blocking island hierarchy is the β_1-BIG of the network graph. The second layer is then the β_2-BIG of the first level, i.e., β_1-BIG, the

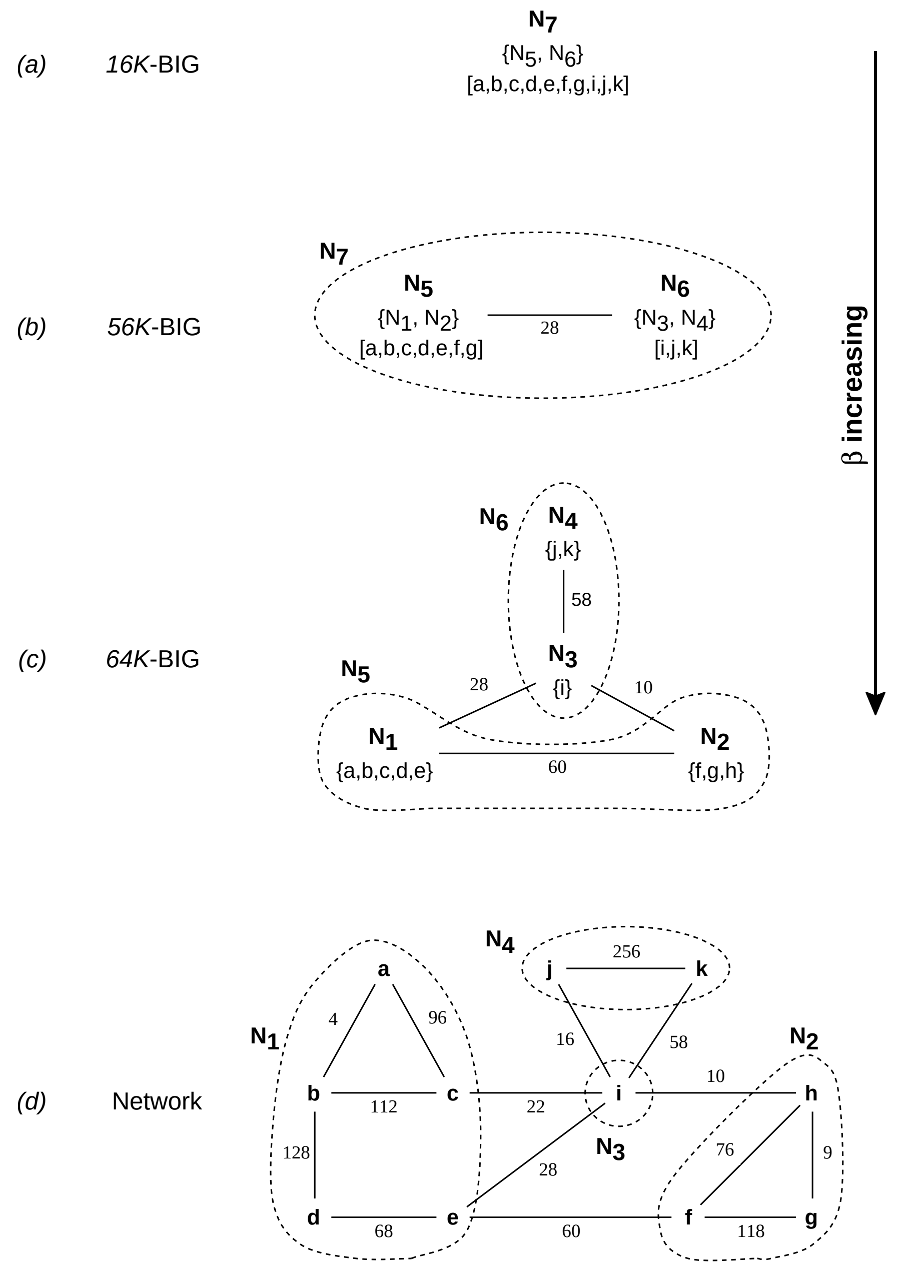


Figure 1. *The blocking island hierarchy for resource requirements* $\{64, 56, 16\}$. *The weights on the links are their available bandwidth. Abstract nodes' description include only their node children and network node children in brackets. Link children (of BIs and abstract links) are omitted for more clarity, and the* 0*-BI is not displayed since equal to* N_7. *(a) the* 16*-BIG. (b) the* 56*-BIG. (c) the* 64*-BIG. (d) the network.*

third layer the β_3-BIG of the second, and so on. On top of the hierarchy there is a 0-BIG abstracting the smallest resource requirement β_b. The abstract graph of this top layer is reduced to a single abstract node (the 0-BI), since the network graph is supposed connected. Figure 1 shows such a BIH for resource requirements $\{64, 56, 16\}$. The graphical representation shows that each BIG is an abstraction of the BIG at the level just below (the next biggest resource requirement), and therefore for all lower layers (all larger resource requirements).

A BIH can not only be viewed as a layered structure of β-BIGs, but also as an *abstraction tree* when considering the father-child relations. In the abstraction tree, the leaves are network elements (nodes and links), the intermediate vertices either abstract nodes or abstract links and the root vertex the 0-BI of the top level in the corresponding BIH. Figure 2 is the abstraction tree of the BIH in Figure 1.

The β-BI S for a given node x of a network graph can be obtained by a simple greedy algorithm, with a linear complexity of $O(m)$, where m is the number of links. The construction of a β-BIG is straightforward from its definition and is also linear in $O(m)$. A BIH

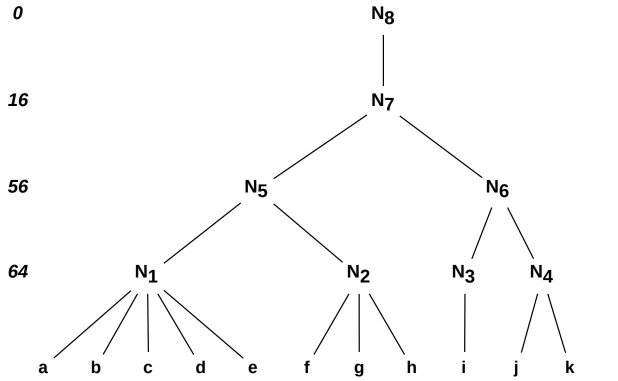


Figure 2. *The abstraction tree of the BIH of Figure 1 (links are omitted for clarity).*

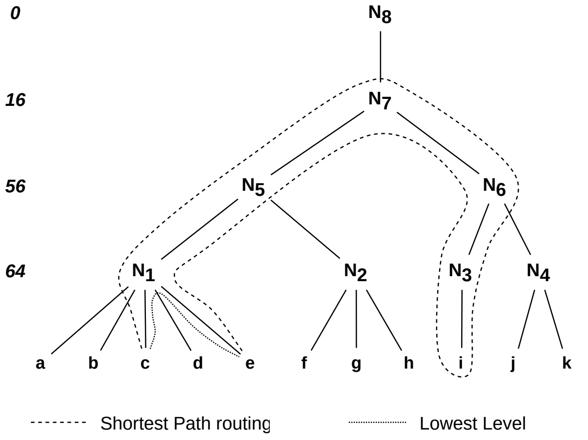


Figure 3. *Mapping the shortest path (SP) and the lowest level (LL) routes onto the abstraction tree.*

for a set of constant resource requirements ordered decreasingly is easily obtained by recursive calls to the BIG computation algorithm. Its complexity is bound by $O(bm)$, where b is the number of different resource requirements. The adaptation of a BIH when demands are allocated or deallocated can be carried out incrementally with complexity $O(bm)$. Therefore, since the number of possible bandwidth requirements (b) is constant, all BI algorithms are linear in the number of links of the network.

A BIH contains at most $bn + 1$ BIs: one BI for each node at each bandwidth requirement level plus the 0-BI. In that worst case, there are $\min\{m, n(n-1)/2\}$ links at each bandwidth level, since multiple links between same BIs are clustered into a single abstract link. The memory storage requirement of a BIH is thus bound by $O(bn^2)$.

4. Routing from the BIH perspective

Consider the problem of routing a single demand $d_u = (c, e, 16)$ in the network of Figure 1.d. Since c and e are clustered in the same 16-BI (N_7), we know that at least one route satisfying d_u exists. Classical wisdom would select the shortest route, that is the route r_S: $c \to i \to e$. However, allocating this route to d_u is here not a good idea, since it uses resources on two critical links in terms of available bandwidth, that is (c, i) and (i, e): these two links join 64-BIs N_1 and N_2 in the 64-BIG of Figure 1.c. After that allocation, no other demand requiring 16 (or more) between any of the nodes clustered by 56-BI N_5 and one of the nodes inside 56-BI N_6 can be allocated anymore. For instance, a demand $(c, i, 16)$ is then impossible to allocate. A better way to route d_u is r_L: $c \to b \to d \to e$, since r_L uses only links that are clustered at the lowest level in the BIH, that is in 64-BI N_1, and no critical links (that is inter-BI links). The only effect on latter assignments is that no demand $(d, e, 64)$ can be allocated after that anymore, which is a less drastic restriction than before.

r_L is a route that satisfies the *lowest level* (LL) heuristic. Its principle is to route a demand along links clustered in the lowest BI clustering the endpoints of the demand, i.e., the BI for the highest bandwidth requirement containing the endpoints. This heuristic is based on the following observation: the lower a BI is in the BIH, the less critical are the links clustered in the BI. By assigning a route in a lower BI, a better *bandwidth connectivity* preservation effect is achieved, therefore reducing the risk of future allocation failures. Bandwidth connectivity can therefore be viewed as a kind of *overall load-balancing*.

Another way to see the criticalness of a route is to consider the *mapping* of the route onto the abstraction tree of Figure 3: r_S is by far then the longest route, since its mapping traverses BIs N_1, N_5, N_7, N_6, N_3, and then back; r_L traverses only BI N_1. This observation (also) justifies the LL heuristic.

Even better, a BIH gives also the means to compare *a priori* equivalent routes in order to decide for the "best" one, besides the length criterion. The *minimal splitting* (MS) heuristic selects the route that causes the fewest splittings of blocking islands in the BIH: obviously, the more splittings, the more links become critical, leading to more allocation failures of demands. MS has therefore an even greater bandwidth connectivity preservation effect than LL. Unfortunately, only an approximation of the MS heuristic can effectively be used in practice, since an exact implementation of MS requires to compute all routes beforehand in order to compare them. A possible implementation is to compute a given number of routes using LL, and then to order them according to the MS heuristic to select a route. The evaluation of the MS heuristic is left for a later paper.

5. Automatically solving a RAIN problem

Solving a RAIN problem amounts to solving the CSP introduced in Section 2.2. Blocking islands allow to overcome the formulation complexity of the CSP formulation, since they are abstractions of the domains of each demand: any route satisfying a demand lies within the β-BI of its endpoints, where β is the resource requirement of the demand (Proposition 1). Therefore, if the endpoints of a demand are clustered in the same β-BI, there is at least one route satisfying the demand. We do not know what the domain of the variable is *explicitly*, i.e., we do not know the set of routes that can satisfy the demand; however we know it is non-empty. In fact, there is a mapping between each route that can be assigned to a demand and the BIH: a route can be seen as a path in the abstraction tree of the BIH. Thus, there is a route satisfying a demand if and only if there is a path in the abstraction tree that does not traverse BIs of a higher level than its resource requirement. For instance, from the abstraction tree of Figure 2, it is easy to see that there is no route between a and f with 64 available resources, since any path in the tree must at least cross BIs at level 56.

This mapping of routes onto the BIH is used to formulate a forward checking criterion, as well as dynamic value and variable ordering heuristics. (We note that a patent for the methods given below is pending.)

5.1. Forward Checking

Thanks to the route existence property, we know at any point in the search if it is still possible to allocate a demand, without having to compute a route: if the endpoints of the demand are clustered in the same β-BI, where β is the resource requirement of the demand, there is at least one, i.e., the domain of the variable (demand) is not empty, even if not explicitly known. Therefore, after allocating a demand, forward checking is performed first by updating the BIH, and then by checking that the route existence property holds for all uninstantiated demands. If the latter property does not hold at least once, another route must be tried for the current demand. Domain pruning is therefore implicit while maintaining the BIH.

5.2. Value Ordering

A backtracking algorithm involves two types of choices: the next variable to assign (see Section 5.3), and the value to assign to it. As illustrated above, the domains of the demands are too big to be computed beforehand. Instead, we compute the routes as they are required. In order to reduce the search effort, routes should be generated in "most interesting" order, so to increase the efficiency of the search, that is: try to allocate the route that will less likely prevent the allocation of the remaining demands. A natural heuristic is to generate the routes in *shortest path* order (SP), since the shorter the route, the fewer resources will be used to satisfy a demand.

However, Section 4 shows how to do better using a kind of min-conflict heuristic, the lowest level heuristic. Applied to the RAIN problem, it amounts to considering first, in shortest order, the routes in the lowest blocking island (in the BIH). Apart from attempting to preserve bandwidth connectivity, the LL heuristic allows to achieve a computational gain: the lower a BI is, the smaller it is in terms of nodes and links, thereby reducing the search space to explore. Moreover, the LL heuristic is especially effective during the early stages of the search, since it allows to take better decisions and therefore has a greater pruning effect on the search tree, as shown by the results in Section 7. Generating one route with the LL heuristic can be done in linear time in the number of links (as long as QoS is limited to bandwidth constraints).

5.3. Variable Ordering

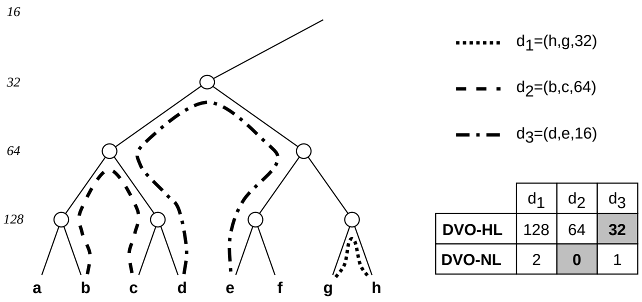
The selection of the next variable to assign may have a strong effect on search efficiency, as shown by Haralick and Elliot [15]. A widely used variable ordering technique is based on the "fail-first" principle: "To succeed, try first where you are most likely to fail". The idea is to minimize the size of the search tree and to ensure that any branch that does not lead to a solution is pruned as early as possible when choosing a variable.

There are some natural static variable ordering (SVO) techniques for the RAIN problem, such as first choose the demand that requires the most resources. Nonetheless, BIs allow dynamic (that is during search) approximation of the difficulty of allocating a demand in more subtle ways by using the abstraction tree of the BIH:

DVO-HL (Highest Level): first choose the demand whose lowest common father of its endpoints is the highest in the BIH (remember that high in the BIH means low in resources requirements). The intuition behind DVO-HL is that the higher the lowest common father of the demand's endpoints is, the more constrained (in terms of number of routes) the demand is. Moreover, the higher the lowest common father, the more allocating the demand may restrict the routing of the remaining demands (fail-first principle), since it will use resources on more critical links.

DVO-NL (Number of Levels): first choose the demand for which the difference in number of levels (in the BIH) between the lowest common father of its endpoints and its resources requirements is lowest. The justification of DVO-NL is similar to DVO-HL.

The behavior of DVO-HL and DVO-NL are illustrated in Figure 4. In the implementation, both latter heuristics use the required bandwidth as a secondary criterion to break ties: in case two or more demands have the same value for the criterion, the one with



	d_1	d_2	d_3
DVO-HL	128	64	**32**
DVO-NL	2	**0**	1

Figure 4. *Selecting the next demand to allocate using DVO-HL and DVO-NL. The DVO-HL line shows the value for the lowest common father level for each demand. The DVO-NL line shows the number of levels between the lowest common father and the bandwidth requirement level. The selected demand by each heuristic is shaded in gray.*

highest requirement is preferred. Besides obeying the fail-first principle, this secondary criterion attempts to minimize one side effect of the lowest level heuristic for route selection: by avoiding the routing through critical links, LL may cause the split of BIs at very low levels in the hierarchy, i.e., for high bandwidth requirements, thereby preventing the allocation of demands with high requirements.

There are numerous other *Dynamic Variable Ordering* (DVO) heuristics that can be derived from analyzing the BIH. For instance, let β_d-BI be the blocking island clustering the endpoints of a demand d at its bandwidth requirement level β_d. Then, select first the demand d for which the β_d-BI contains the fewest nodes. This heuristic is called *DVO-N*. It follows the fail-first principle, since one can assume that the fewer nodes, the fewer routes connect the endpoints of the demand. Two similar heuristics, with same justification, are to select first the demand d for which the β_d-BI contains the fewest links (*DVO-L*) or the lowest link density (*DVO-D*). The evaluation of these heuristics (and others) is left for a later paper.

5.4. A forward-checking algorithm for the RAIN problem.

Now that the different parts have been examined, we can put everything together into a systematic search algorithm. Figure 5 shows a the pseudo-code for `FCRAIN`, a recursive forward-checking backtracking algorithm. `FCRAIN` can be read as follows: if there are still unallocated demands, it selects the next demand to allocate with `NextDemand` using a dynamic demand ordering heuristic (Section 5.3). `NextBestRoute` computes the best route according to the dynamic route ordering heuristic (Section 5.2). FCRAIN then performs forward-checking: it verifies that all remaining unallocated demands can still be allocated, using generic function `ExistsRouteForAll`. `ExistsRouteForAll` checks that all unallocated demands have both endpoints in the same BI at their bandwidth requirement level (as explained in Section 5.1). If it is the case, it recursively allocates the next demand. Otherwise, the current allocation is undone, and the best next route is tried. If all routes have been tried unsuccessfully, backtracking to the previously allocated demand occurs. This amounts to revoking the current and the previously allocated demand, and then selecting the next best route for the latter.

6. Conflict identification and resolution

Blocking islands, and especially blocking island graphs, identify *global bottlenecks* in the network, as shown for instance in Figure 1.c for 64 demands. The abstraction of the network into blocking islands allow to take measures when a demand cannot be allocated (because its endpoints are not clustered in the same BI at its bandwidth requirement level).

Suppose a set of demands $\mathcal{D}$ were allocated in the network and that the network's available resources is given by Figure 1.d. Now we are to allocate a new demand $d_n = (c, h, 64)$. Since c and h are not clustered in the same 64-BI, it is impossible to satisfy d_n without rerouting some previously allocated demands. Now, there are two cases:

1. It may be that the problem of allocating $\mathcal{D} \cup \{d_n\}$ is in fact unsolvable.
2. Rerouting one or more already allocated demands may resolve the problem.

6.1. Unresolvable problem identification

Given the situation explained above where d_n cannot be allocated, it is possible, in some cases to prove that the RAIN problem is indeed unsolvable by approximating it with a network multi-flow problem. If there is any cut in the network graph whose capacity is smaller than the sum of the bandwidth requirement of the demands that have one endpoint on each side of the cut. However, there is an exponential number of possible cuts in a network and examining them all is intractable. Nonetheless, BIs identify critical parts of the network (for the possible bandwidth requirements of the demands), and therefore the search for infeasibility can be restricted to the cocycle[1] of some BIs only.

We call *primary blocking islands* (PBI) of a demand that cannot be allocated the two BIs of its endpoints at its bandwidth requirement level. For d_n, the PBIs are N_1 and N_2. If for any of the two PBIs the sum of the bandwidth requirements of the demands that have one and only one endpoint inside the PBI is higher than the capacity of the links of the PBI's cocycle, then it is

[1] The cocycle of a subset of nodes A is the set of all links that have one and only one endpoint in A.

```
function FCRAIN(𝒟, Γ, ℋ)
    /* 𝒟 is the set of demand still to be allocated */
    /* Γ is the set of already established connections */
    /* ℋ is the current BIH */
    if 𝒟 = ∅ then
        return Γ
    else
        d ← NextDemand(𝒟, ℋ)
        r ← NextBestRoute(d, ℋ)
        repeat
            γ ← connection (d, r)
            PropagateConnectionAdditionInBIH(ℋ, Γ, γ)
            /* Forward checking: verify that (d, r) does not prevent the allocation of an open demand */
            if ExistsRouteForAll(𝒟 − {d}, ℋ) then
                res ← FCRAIN(𝒟 − {d}, Γ ∪ {γ}, ℋ)
                if res ≠ ∅ then return res          /* We have a solution */
                PropagateConnectionRemovalInBIH(ℋ, Γ, γ)
                r ← NextBestRoute(d, ℋ)
        until r = ∅
        return ∅          /* Backtrack */
```

Figure 5. *The forward checking algorithm* FCRAIN *for the RAIN problem. Its input is threefold:* $\mathcal{D}$ *is the set of still unallocated demands,* Γ *the set of established connections, and* $\mathcal{H}$ *the current blocking island hierarchy. It returns either a set of connections* Γ *(a solution) or* $\emptyset$ *(in which case there is no solution).*

obvious that the problem is infeasible. We call this *unresolvable conflict detection* (UCD). Search can then be aborted, thereby saving us much effort. Note that if a problem is infeasible, it does not mean that unsolvability can be always be proven that way, because the RAIN problem is not a network flow problem. By examining only the PBIs of demands that cannot be allocated, we overcome the intractability issue raised above, while still being able to find such a cut with high probability.

Moreover, when the problem can be proven infeasible that way, the BIH helps the human operator in deciding where to add bandwidth resources, since global bottlenecks are identified. In the above case, in order to be able to satisfy d_n, we have to add bandwidth resources between the two primary blocking islands, that is between one node of N_1 and one node of N_2, since routing inside the BIs of the bandwidth requirement level does not pose any problem. For instance, suppose the network is the one of a service provider. Depending on link prices, the service provider may buy some more resources from a network operator either between e and f, or directly between c and h (link addition), or even along a longer path, for instance between a and j, and i and f. The BI then provide meaningful abstractions of the network by hiding "easy" parts of the network and only confronting a human user with the real problems.

6.2. Culprit assignment identification

In a classical backtracking algorithm, when a dead-end is reached, the most recent assignment is undone and an alternate value for the current variable is tried. In case all values have been unsuccessfully assigned, backtracking occurs to the preceding allocation. However, if we have the means to identify a culprit assignment of a past variable, we are able to directly *backjump* to it (the intermediate assignments are obviously undone), thereby possibly drastically speeding up the search process.

In case infeasibility cannot be proven when a dead-end is reached while solving a RAIN problem, analyzing the situation on the links of the PBIs cocycle indeed helps to identify a *culprit assignment.* In this case, we either find that one or more demands were at fault by being routed through at least one link on the blocking island's cocycle, or that there is in fact no solution. The latter cannot be established, however a culprit assignment can be identified by analyzing the demands routed over the cocycle of the PBI. There are two possibilities:

1. The sum of the bandwidth requirements of all unallocated demands that have one and only one end-point in the PBI is less than the sum of the available bandwidth on the links of the PBI's cocycle. This means that there is at least one already allocated demand that is routed over more than one

link of the PBI's cocycle, thereby using up many critical resources. We therefore have to backjump to the point in the search where the total available bandwidth on the cocycle was enough to support all unallocated demands that have one and only one endpoint in the PBI.

2. Otherwise, reallocating some of the demands that are routed over the PBI's cocycle over different routes may suffice to solve the problem. Therefore, the most recent culprit assignment is the latest demand that is routed over the PBI's cocycle.

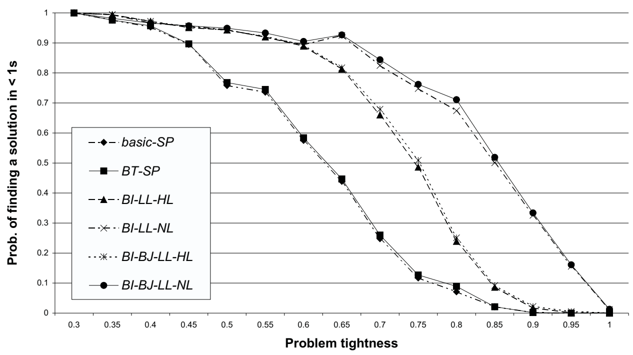
7. Empirical results

We report here some empirical results on solving feasible and impossible RAIN problems. All results were obtained on a Sun Ultra 60 with a LISP program.

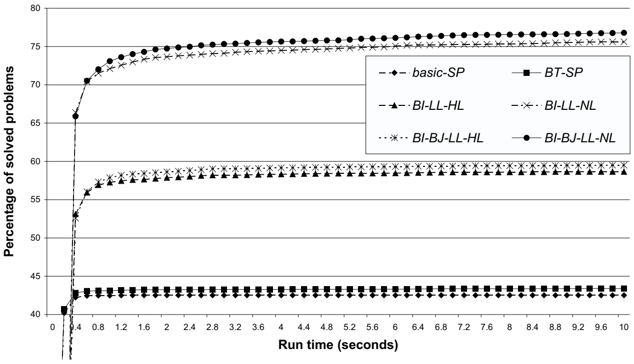
7.1. Solving feasible RAIN problems

A meaningful analysis of the performance of the heuristics we propose would analyze the probability of finding a solution within the given time limit, and compare this with the performance that can be obtained using common methods of the networking world, in particular shortest-path algorithms. For comparing the efficiency of different constraint solving heuristics, it is useful to plot their performance for problems of different tightness. In the RAIN problem, tightness is the ratio of resources required for the best possible allocation (in terms of used bandwidth) divided by the total amount of resources available in the network. Since it is very hard to compute the best possible allocation, we use an approximation, the best allocation found among the methods being compared.

We generated 22'000 instances of RAIN problems, each with at least one solution. Each problem has a randomly generated network topology of 20 nodes and 38 links, and a random set of 80 demands, each demand characterized by two endpoints and a bandwidth constraint. A solution must allocate all demands within the bandwidth capacities of the links. No other restriction was imposed on the routes. We especially supposed no hop-by-hop routing table constraints for instance. A solution is thus applicable to a connection-oriented network such as ATM. The problems were solved with six different strategies: *basic-SP* performs a search using the shortest path heuristic common in the networking world today, without any backtracking on decisions; *BT-SP* incorporates backtracking to the previous in order to be able to undo "bad" allocations. The next search methods make use of the information derived from the BIH: *BI-LL-HL* uses the LL heuristic for route generation and DVO-HL for dynamic demand selection, whereas *BI-LL-NL* differs from the latter in using DVO-NL for choosing the next demand to allocate. *BI-BJ-LL-HL* and *BI-BJ-LL-NL* differ from the previous in the use of backjumping to culprit decisions, as described in Section 6.2.



(a)



(b)

Figure 6. *Statistics on solving 22'000 randomly generated solvable problems with 20 nodes, 38 links, 80 demands each. (a) the probability of finding a solution within 1 second, given the tightness of the problems. (b) the probability of solving the problems according to run time.*

Figure 6.a gives the probability of finding a solution to a problem in less than 1 second, given the tightness of the problems (as defined above). The curves clearly display a phase transition behavior. BI search methods prove to perform much better than brute-force, even on these small problems, where heuristic computation (and BIH maintenance) may proportionally use up a lot of time. Backjumping methods show slightly better performance over their purely backtracking counterparts. The benefits of backjumping seem to be somewhat canceled by the computing overhead associated with calculating the culprit assignment, at least on these small problems. On problems of much larger size, we noticed that BJ algorithms do perform much better in average than their non-backjumping counterparts. Noteworthy, NL outperforms HL: NL is better at deciding which demand is the most difficult to assign, and therefore achieves a greater pruning effect. The shape of the curves are similar for larger time limits.

Figure 6.b provides the probability of solving a problem according to run time. Two conclusions can be de-

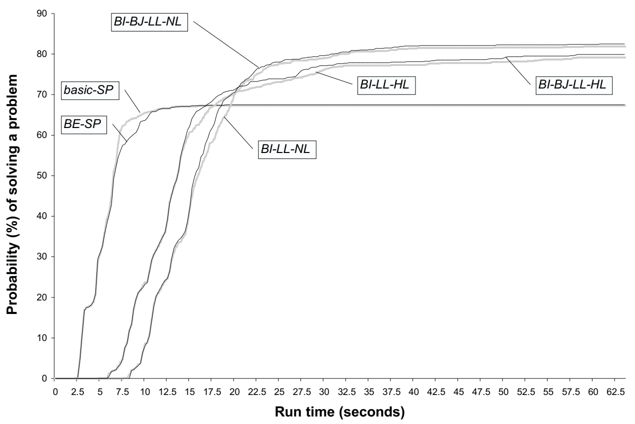


Figure 7. The probability of solving a problem according to run time (800 problems, with 38 nodes, 107 links, 1200 demands).

rived from the latter figure: first, BI methods curves continue to grow with time, albeit slowly, which is not the case for basic-SP (and obviously BT-SP); second, maintaining the BIH on these small problems does not affect the BI algorithms very much. The quality of the solutions, in terms of network resource utilization, were about the same for all methods. However, when the solutions were different, bandwidth connectivity was generally better on those provided by BI methods.

The experimental results allow quantifying the gain obtained by using our methods. If an operator wants to ensure high customer satisfaction, demands have to be accepted with high probability. This means that the network can be loaded up to the point where the allocation mechanism finds a solution with probability close to 1. From the curves in Figure 6, we can see that for the shortest-path methods, this is the case up to a load of about 40% with a probability of 0.9, whereas the NL heuristic allows a load of up to about 55%. Using this technique, an operator can thus reduce the capacity of the network by an expected 27% without a decrease in the quality of service provided to the customer!

According to phase transition theory, relative performance can be expected to scale in the same way to large networks. This is corroborated by first results on larger problems. We generated 800 different RAIN problems with 38 nodes, 107 links, and 1200 demands. The probability of solving such a problem according to run time is given in Figure 7. Here, we see that the maintenance of the BIH has a larger effect on solving "easy" problems. However, after 20 seconds of run time the BI methods clearly are more efficient than the non-BI techniques. The facts noticed for the smaller problems (Figure 6) remain valid: NL outperforms HL, even if it requires slightly more run-time, and backjumping brings only a small advantage over their non-backjumping counterparts. Further, as another result, BI-BJ-LL-NL solved a much larger RAIN problem (50 nodes, 171 links, and 3'000 demands) in 4.5 minutes. BT-SP was not able to solve it within 12 hours.

Table 1
The comparison in terms of run time (seconds), generated routes, and backtracks, of a brute-force backtracking method to BI-based search methods when finding all solutions (6'504) on a small RAIN problem (a leased-line network), with 8 nodes, 9 links, and 18 demands.

Search method	Time (sec)	Routes	Backtracks
BT- SP	191.140	795'425	788'921
BI-LL-HL	10.523	16'668	10'164
BI-LL-NL	8.466	10'694	4'190

The advantages of the BI methods over naive shortest path allocation are best illustrated when searching all solutions of a RAIN problem, as shown in the comparisons of Table 1 for a small problem. Thanks to their much better pruning power and more purposeful search guidance heuristics, they are much faster (about 20 times), generate fewer routes (between 48 and 74 times) and backtrack even less (between 78 and 188 times).

7.2. Performance evaluation on infeasible RAIN problems

In order to test the capability of identifying in some cases that a RAIN problem is infeasible, we generated problems using an existing network topology, and randomly generated demands until we could prove that it was infeasible (within 20 minutes of computation time), either by identifying an unresolvable conflict (as described in Section 6.1), or by having to explore the whole search space. The network topology used is the FUDI sub-network covering the main cities in Europe. It comprises 12 nodes and 14 links, each link with a bandwidth capacity between 6 and 34 [Mb/s], as depicted in Figure 8.

We generated two sets of problems based on the FUDI sub-network[2]:

- Fore the *FUDI 1 set* of 300 problems, we randomly generated demands with relatively high bandwidth requirements with respect to the link capacities. The bandwidth requirement of a demand was uniformly distributed in $\{5, 6, 7, 8, 9, 10\}$ [Mb/s]. For each problem, there are between 20 and 24 demands. The maximal bandwidth a demand could ask for is therefore higher than the lowest link capacity. The goal was to make it easier for the BT-SP algorithm to identify the problems as impossible. These problems can be considered as "easy", since their search space is small.

[2] There are no "trivial" unsolvable problems in these sets, such as a node for which the total bandwidth capacity of the links adjacent to the node exceeds the total requirement of the demands of which the node is an endpoint.

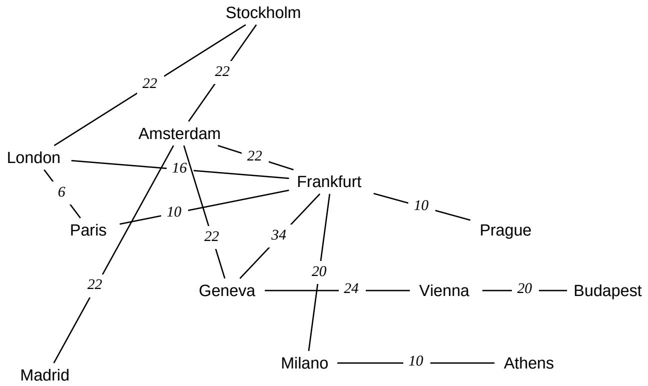


Figure 8. *The FUDI sub-network, comprising 12 nodes and 14 links. The weights on the links denote their bandwidth capacity (in Mb/s). Source:* `http://www.caida.org/Tools/Mapnet/Backbones`.

- The *FUDI 2 set* of 300 problems is intended to be more difficult to solve. The bandwidth requirements of the demands are much smaller compared to the link capacities. The bandwidth requirement of a demand is uniformly distributed in $\{0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1\}$ [Mb/s]. There are between 78 and 164 demands for each problem. The search space of these problems is therefore much bigger than for those of the first set. Noteworthy, the BIH for these problems contains more levels than for the first set of problems.

We compare the same algorithms as in Section 7.1, but instead of BI-LL-HL and BI-LL-NL, we use *BI-U-LL-HL* and *BI-U-LL-NL*, which are the same as BI-LL-HL and BI-LL-NL respectively, however incorporating unresolvable conflict detection. Note that BJ methods incorporate UCD. basic-SP is not taken into account here, since it does not perform a complete search, and therefore cannot prove that a problem is unsolvable. Recall that BT-SP must perform an exhaustive search (i.e., it must examine all route combinations) in order to conclude that the problem has no solution.

The percentage of problems identified infeasible according to run time is given in Figure 9. Part a displays the statistics for FUDI 1 set, and part b are results for FUDI 2 set. For the set of easy problems (FUDI 1 set - see Figure 9.a), we also display the percentage of problems identified as impossible thanks to the UCD mechanism. Since these statistics are very close for non-backjumping and backjumping algorithms, only those for BI-U-LL-HL and BI-U-LL-NL are depicted. The UCD mechanism performs only slightly better with the backjumping algorithms (BI-BJ-LL-HL and BI-BJ-LL-NL). The difference between the UCD curve and the curve of the associated solving method identify the percentage of problems proven infeasible by exhaustive search. The UCD curves for the BI-based algorithms are not displayed in Figure 9.b since all problems that were proven unsolvable were so thanks to the UCD

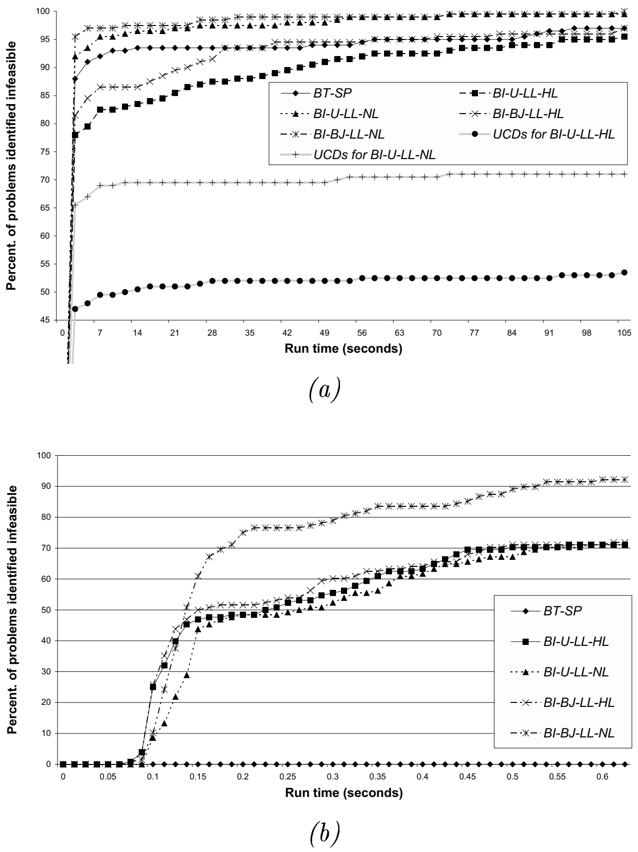


Figure 9. *The percentage of problem identified as infeasible according to run time by different search methods. (a) the 300 problems of the FUDI 1 set; the percentage of problems identified as unsolvable thanks to the UCD mechanism is also depicted for BI-U-LL-HL and BI-U-LL-NL. (b) the 300 problems of the FUDI 2 set; for these problems every problem that was declared infeasible was so thanks to the UCD mechanism.*

mechanism. Several observations can be made based on these statistics:

- BI-based algorithms with UCD logically outperform BT-SP, which has to perform an exhaustive search before declaring a problem unsolvable.
- Performance of BT-SP is only acceptable on the easy problems (FUDI 1 set), where the search space is reasonable in size. On the second set of problems (FUDI 2 set), the search space gets too big to be manageable by BT-SP despite the small network size. Even after 6 minutes, not a single problem was declared unsolvable by this method. The size of the search space for the problems of FUDI 2 set also prevents the exhaustive search by BI-based algorithms within 6 minutes.
- BI-based algorithms without UCD (which are not displayed in Figure 9) are only slightly better than BT-SP (they also require to explore the whole search space before being able to prove that a problem in infeasible) on the FUDI 1 set, albeit slower in their in-

crease. This is due to the computing overhead when maintaining the BIH. We could therefore not establish whether the exploration of the search space for infeasible problems is better for non-UCD BI-based algorithms than for BT-SP in terms of run time. On the FUDI 2 set, no BI-based algorithm without UCD was able to determine a problem as unsolvable.

- The NL demand ordering again outperforms HL, and the main reason for it is that it is able to locate critical parts of the network (i.e., identify an UCD) much better and faster: in Figure 9.a, around 70% of the problems are declared unsolvable by the UCD mechanism for NL, and only slightly more than 50% for HL. HL is then forced to perform many more exhaustive searches to identify infeasible problems.
- The UCD mechanism is very fast. On the FUDI 1 set, most problems declared impossible by means of UCD are so within 8 seconds. From then on, UCD then grows only very slowly. On the FUDI 2 set, UCD happens within the first second. It then increases only slightly (0.8%) for HL methods until 9 seconds elapsed. After that, no increase is established until the run time limit of 360 seconds.
- On the easy problems, BI-BJ-LL-NL proved that all problems are unsolvable after 104 seconds, BI-BJ-LL-HL after 205 seconds. After 360 seconds, BT-SP is still not able to characterize 2% of the problems. This means that the UCD mechanism does perform well also on problem instances that require a lot of search to be proven infeasible, and not only on "easy" cases, and that search space exploration is most efficient for BI-BJ-LL-NL.
- The benefit of backjumping algorithms is greater for identifying impossible problems than for solving feasible ones. This is especially the case for BI-BJ-LL-NL applied to the FUDI 2 set of problems (Figure 9.b).
- If we compare the search statistics on the problems identified as impossible by all methods, we can clearly establish that the number of generated routes and backtracks are significantly better for BI based algorithms, especially for those incorporating the UCD mechanism. For instance, after a run-time of 30 seconds on the FUDI 1 set problems solved by all methods, BI-BJ-LL-NL backtracked about 40 times fewer (this includes backjumps) and generated over 25 times fewer routes than BT-SP. These values are even higher for lower time thresholds, and only very slightly decrease with time (37 and 24 respectively after 240 seconds).

8. Conclusions and future work

The current technique for routing communication demands in a network is to select the shortest route for each particular demand. However, this strategy can lead to suboptimal routing or even highly congested network utilization as a whole. Information about the expected traffic allows to make better use of network resources. However, on-line routing processes cannot make use of this knowledge since they must be very fast to ensure customer satisfaction. Instead, bandwidth allocation can be planned in an off-line manner with this information, thanks to a systematic (complete) search algorithm that is capable of backtracking to faulty routing decisions in order to satisfy all demands.

We have shown that using blocking island abstractions coupled with CSP search mechanisms and heuristics, it is possible to solve the bandwidth allocation planning problem in reasonable time in many problem instances with a complete search algorithm, and to get better solutions than shortest path routing algorithms. Network operators (or service providers) can now plan the allocation of bandwidth in much tighter networks and more often than before. Noteworthy, it is possible to simulate link and node failures and, using the same technique, compute alternate routing plans for these situations. The blocking island paradigm also proves to be very efficient in identifying infeasible problems very quickly, and thereby also constitutes a powerful aid to the network operator, since the UCD mechanism highlights the regions in the network between which bandwidth must be added in order to be able to satisfy all demands. An on-line demo illustrating these features is available on the WWW at `http://www.iconomic.com`.

The presented techniques are not only applicable to connection-oriented networks (such as ATM), but also to connection-less networks (such as IP). The only difference is then in the route generation process, since IP uses hop-by-hop routing tables, and the generated routes must respect that constraint. Even if the route existence property does not hold in this context, its natural counterpart, the route *inexistence* property does. Thus, it is likely that solving a RAIN problem for an IP network will be as efficient as for the connection-oriented case.

In this paper, we restricted demands to point-to-point traffic. However, this need not be. The same techniques can be applied for multipoint demands: routes are then trees instead of simple paths. Generalizing the presented heuristics, such as the lowest level (LL) for route generation or the number of levels for demand selection (DVO-NL) are straightforwardly generalized to multipoint demands. Moreover, quality of service was expressed in terms of bandwidth only. It is however easy to incorporate other QoS parameters (such as maximal delay and maximal loss ratio) into the technique by checking that the generated routes do verify those additional constraints. Taking into account other QoS parameter may even improve search drastically since search space is thereby restricted.

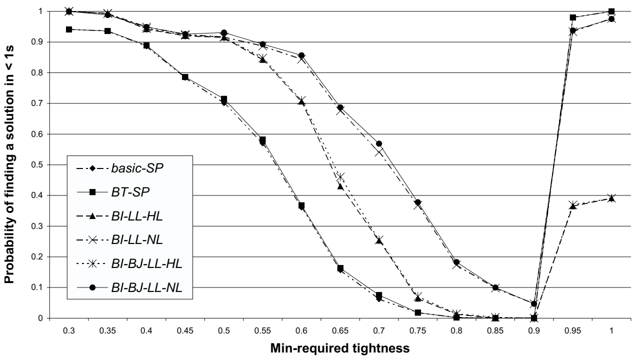


Figure 10. *The probability of finding a solution within 1 second, given the* min-required *tightness of the problems (22'000 random problems with 20 nodes, 38 links, 80 demands).*

Current and future work is conducted along several axis:

Order parameter identification Phase transition theory supposes an "order parameter", and hard instances of problems occur at a critical value of this parameter. We have shown that the RAIN problem also has a phase transition behavior. However, the order parameter we used, the tightness as defined in Section 7.1, is a parameter that can only be computed when the problem is solved. We intend to identify other types of tightness measures that can be computed on an unsolved problem in order to estimate the difficulty of solving it. A natural *a priori* order parameter is the *min-required* tightness. It is based on the *minimal demand load* (MDL). The MDL for a demand minimal amount of resources required to satisfy it, that is its bandwidth requirement times its minimal hop count. If we sum the MDL for all demands and divide the result by the total bandwidth capacity, we get the min-required tightness of the problem. Phase transition plots for this order parameter are given in Figure 10. Again, the curves display a clear phase transition behavior[3]. The min-required tightness is however only a very raw characterization, and we are looking into other types of measures that better capture the idea of tightness.

Extending the backjumping concept We presented how the explanation of allocation failures during search (when a dead-end is encountered) allows to determine a culprit assignment, in order to directly backjump to the cause of the failure. However, this backjump is only performed once. *Conflict-Directed Backjumping* (CBJ), introduced by Prosser in [25], has even more sophisticated backjumping behavior than BJ: it has the ability to perform multiple backjumps, that is, after the initial backjump from a dead-end, it can continue backjumping across conflicts, which may potentially result in significant savings. This comes at the price of maintaining additional data structures however, the conflict sets for each variable (demand). We intend to extend this technique to the RAIN problem. However, we need to find an efficient way to identify those conflict sets in our particular problem.

New search algorithms and heuristics We are concerned with improving the search process by looking at other types of search algorithms, such as Ginsberg's *Dynamic Backtracking* [12]. We also want to incorporate the explanation of allocation failure into iterative search techniques, such as hill-climbing, GSAT, Genetic Algorithms, Simulated Annealing, and Tabu Search. We are currently evaluating new heuristics for route generation and demand selection to increase search efficiency.

New abstraction techniques We are developing new types of hierarchies that decompose even better the network according to resource availability. One major goal is here to get rid of the fixed levels in the hierarchy that depend on the possible bandwidth requirements of the demands. We have developed such a hierarchy, called Ω-BIH, whose levels depend only on available resources on the links (and not at all on the demand requirements), and preliminary results are very promising despite a larger maintenance overhead.

Incorporation of other QoS parameters We need to experiment the behavior of our algorithms and heuristics when QoS not only depends on bandwidth, but also on other parameters (such as delay and loss probability). We are also seeking to take into account node limitations, such as buffer capacity. CSP modeling has the facility to easily take such additional restrictions into account, by just adding these additional constraints "as is". CSPs have in this case a major advantage over Operations Research techniques, that do not allow the integration of new constraints in such a straightforward manner.

One first result was the development of a generalization of the BI paradigm to several concave metrics [7]. A resource metric μ is said to be *concave* if for any path p, $\mu(p) = \min_{l \in p} \mu(l)$, where $\mu(l)$ is the metric for link l. Bandwidth is typically a concave resource, however there are others: for instance, the number of connections routed over a link may be limited, due to various factors, such as the number of allowed identifiers of connections over a link (e.g., virtual path/connection identifiers in ATM).

On-line routing by intelligent agents We also intend to examine how the blocking island paradigm can help routing demands *on-line*. The idea is to assign one intelligent agent to each BI, each agent

[3] Noteworthy, we see that after a min-required tightness of 0.9, the probability drastically increases. This is normal, since in these regions, a solution can almost only contain shortest routes, by definition of the order parameter.

then being responsible for the allocation of demands inside its domain, thereby possibly cooperating with agents below it in the hierarchy. In this case, the demands are not known in advance and allocated one after the other. Demand ordering heuristics as well as backtracking algorithms are then not applicable. We presented some first considerations in [6] and some results in [7].

Acknowledgments

This work is partly a result of the IMMuNe (Integrated Management for Multimedia Networks) Project, supported by the Swiss National Science Foundation (FNRS) under grant 5003-045311. The authors are grateful to Dean Allemang, Beat Liver, Steven Willmott, and Monique Calisti for their invaluable comments, suggestions, and encouragements during the last years. The authors also wish to thank George Melissargos and Pearl Pu for their work on the graphical user interface of the developed tool.

References

[1] G. R. Ash, *Dynamic Routing in Telecommunications Networks*, McGraw Hill, 1998.

[2] P. Cheeseman, B. Kanefsky and W. M. Taylor, Where the Really Hard Problems Are, in: *Proc. of the 12 th IJCAI*, pp. 331–337, 1991.

[3] S. Chen and K. Nahrstedt, An Overview of Quality of Service Routing for Next-Generation High-Speed Networks: Problems and Solutions, *IEEE Network*, pp. 64–79, November/December 1998.

[4] B. Y. Choueiry and B. Faltings, A Decomposition Heuristic for Resource Allocation, in: *Proc. of the 11 th ECAI*, pp. 585–589, 1994.

[5] S. Cosares and I. Saniee, An optimization problem related to balancing loads on SONET rings, *Telecommunication Systems*, 3:165–181, 1994.

[6] C. Frei and B. Faltings, A Dynamic Hierarchy of Intelligent Agents for Network Management. in: *2nd Int. W. on Intelligent Agents for Telecommunications Applications (IATA'98)*, pp. 1–16, Springer-Verlag, 1998.

[7] C. Frei and B. Faltings, Abstraction and Constraint Satisfaction Techniques for Planning Bandwidth Allocation, *IEEE INFOCOM'2000*, March 2000. In print.

[8] E. C. Freuder and D. Sabin, Interchangeability Supports Abstraction and Reformulation for Multi-Dimensional Constraint Satisfaction, in: *Proc. of the 15 th AAAI*, pp. 191–196, 1997.

[9] J. Gaschnig, Experimental case studies of Backtrack vs. Waltz-type new algorithms, in: *Proc. of 2 nd Biennial Conf. Canadian Society for Computational Study of Intelligence*, 1978.

[10] I. P. Gent and T. Walsh, Computational Phase Transitions from Real Problems, in: *Proc. ISAI-95*, pp. 356–364, Mexico, 1995.

[11] I.P. Gent, E. MacIntyre, P. Prosser and T. Walsh, Scaling Effects in the CSP Phase Transition, in: *First International Conference on Principles and Practice of Constraint Programming (CP'95)*, pp. 70–87, 1995.

[12] M. L. Ginsberg, Dynamic Backtracking, *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[13] F. Giunchiglia and T. Walsh, A Theory of Abstraction, *Artificial Intelligence*, 57:323–389, 1992.

[14] W. C. Hamscher, Modeling digital circuits for troubleshooting, *Artificial Intelligence*, 51:223–271, 1991.

[15] R. M. Haralick and G. L. Elliott, Increasing Tree Search Efficiency for Constraint Satisfaction Problems, *Artificial Intelligence*, 14:263–313, 1980.

[16] M. Herzberg, D. J. Wells and A.Herschtal, Optimal Resource Allocation for Path Restoration in Mesh-Type Self-Healing Networks, *International Teletraffic Congress (ITC)*, 15:351–360, 1997.

[17] T. Imielinski, Domain Abstraction and Limited Reasoning, in: *Proc. of the 10 th IJCAI*, pp. 997–1003, 1987.

[18] S. Kirkpatrick and B. Selman, Critical Behaviour in the Satisfiability of Random Boolean Expressions, *Science*, 264:1297–1301, 1994.

[19] C. A. Knoblock, J. D. Tenenberg and Q. Yang, Characterizing Abstraction Hierarchies for Planning, in: *Proc. of the 9 th AAAI*, pp. 692–697, 1991.

[20] R. E. Korf, Planning as search: A quantitative approach, *Artificial Intelligence*, 33:68–88, 1987.

[21] A. Y. Levy, Creating Abstractions Using Relevance Reasoning, in: *Proc. of the Workshop on Theory Reformulation and Abstraction*, pp. 2-139/2-153, 1994.

[22] J. W. Mann and G. D. Smith, A Comparison of Heuristics for Telecommunications Traffic Routing, in: *Modern Heuristic Search Methods*, eds. V. J. Rayward-Smith, I. H. Osman, C. R. Reeves and G. D. Smith, John Wiley & Sons Ltd, 235–254, 1996.

[23] B. T. Messmer, A framework for the development of telecommunications network planning, design and optimization applications, Technical Report FE520.02078.00 F, Swisscom, Bern, Switzerland, 1997.

[24] D. Mitchell, B. Selman and H. Levesque, Hard and Easy Distributions of SAT Problems, in: *Proc. of the 10 th AAAI*, pp. 459–465, 1992.

[25] P. Prosser, Hybrid Algorithms for the Constraint Satisfaction Problem, *Computational Intelligence*, 9(3):268–299, 1993.

[26] Edward Tsang, *Foundations of Constraint Satisfaction*, Academic Press, 1993.

[27] A. Unruh and P. S. Rosenbloom, Abstraction in Problem Solving and Learning, in: *Proc. of the 11 th IJCAI*, 1989.

[28] S. Vedantham and S.S. Iyengar, The Bandwidth Allocation Problem in the ATM network model is NP-Complete, in: *Information Processing Letters*, 65:179–182, 1998.

[29] R. J. Wallace, Practical Applications of Constraint Programming, *Constraints. An International Journal*, pp. 139–168, 1996.

[30] Z. Wang and J. Crowcroft, Analysis of shortest-path routing algorithms in a dynamic network environment, *ACM SIGCOMM CCR*, 22(2):63–71, 1992.

[31] Z. Wang and J. Crowcroft, Quality-of-Service Routing for Supporting Multimedia Applications, *IEEE Journal on Selected Areas in Communications*, 14(7):1228–1234, September 1996.

[32] R. Weigel and B. Faltings, Structuring Techniques for Constraint Satisfaction Problems, in: *Proc. of the 15 th IJCAI*, pp. 418–423, August 1997.

[33] T.-H. Wu, *Fiber Network Service Survivability*, Artech House, 1992.

[34] Symposium on Abstraction, Reformulation and Approximation (SARA-98), 1998.