© 2001 Elsevier Science. Reprinted with permission from The Journal of Systems and Software 58, No. 3, pages 181-198.

The Journal of Systems and Software 58 (2001) 181–198

The Journal of Systems and Software

www.elsevier.com/locate/jss

Case study of the evolution of routing algorithms in a network planning tool

Jyrki Akkanen, Jukka K. Nurminen [*]

Nokia Research Center, P.O. Box 407, FIN-00045 Nokia Group, Finland

Received 20 June 2000; received in revised form 14 October 2000; accepted 2 December 2000

Abstract

Traffic routing is a key component in a network planning system. This paper concentrates on the routing algorithms and follows their evolution over multiple releases of a planning tool during a period of six years. The algorithms have grown from the initial stage of finding shortest paths with Dijkstra's algorithm to cover more complex routing tasks such as finding protected and unprotected routes and capacity limited routing. We present the algorithms and in particular emphasize the practical aspects: how previous algorithms were reused and what were the practical experiences of using the algorithms. A conclusion of the study is that algorithms should be considered with an engineering attitude. It is not enough to focus on selecting the most sophisticated state-of-the-art algorithm for a given problem. Evolution capability, potential for reuse, and the development cost over the system lifetime are equally important aspects. © 2001 Elsevier Science Inc. All rights reserved.

1. Introduction

Developing computer solutions for real-life problems requires a model of the problem and a suitable algorithm to solve it. An important consideration is the right level of detail and complexity of the model and of the solution algorithm.

While a lot of research activity has been spent on developing better and better algorithms, for a practitioner the most sophisticated and efficient algorithm may not always be the best choice. It is possible to solve many important real-life problems with relatively simple techniques, at least after one is willing to accept suboptimal solutions. In many cases small implementation effort, reduced risk, reuse of existing work, clarity and simplicity as well as potential to adapt to new needs are more important aspects than running time or the optimality of the results.

Most modeling and algorithmic studies are static. They present models and algorithms that solve a given problem but they do not consider what happens when the problem changes. In practice most applications that do not evolve do not survive. Software, models, and algorithms have to be updated and further developed to meet the changing needs. Our opinion is that the evolution potential of algorithms has not received enough attention.

The goal of this study is to understand better how algorithms need to evolve within a software systems lifetime and to share our experience about what properties are important for an algorithm to cope with the evolution. The main motivation is to highlight that evolution potential is an important attribute for the practical use of an algorithm.

In this paper we will illustrate the potential of a relatively simple, old algorithm by considering Dijkstra's shortest path algorithm (Dijkstra, 1959) and its use within a network planning tool (Nokia Networks, 1999). Starting from the basic algorithm we present a set of practical problems that can be solved with minor changes to the algorithm. In some problems we use the basic algorithm and implement a very simple control structure on top of it. In other problems we modify the algorithm itself.

The structure of the paper is as follows. Section 2 discusses the evolution problem in more detail. Sections 3 and 4 briefly present some background information. In Section 3 we introduce the relevant network planning issues. In Section 4 we review the shortest path problem and represent Dijkstra's algorithm. Section 5 covers the applications of

[*] Corresponding author. Tel.: +358-7180-36442; fax: +358-7180-36229.
E-mail addresses: jyrki.akkanen@nokia.com (J. Akkanen), jukka.k.nurminen@nokia.com (J.K. Nurminen).

0164-1212/01/$ - see front matter © 2001 Elsevier Science Inc. All rights reserved.
PII: S 0 1 6 4 - 1 2 1 2 (0 1) 0 0 0 3 7 - 1

Dijkstra's algorithm to different network planning problems. Finally, in Section 6 we summarize our experiences and draw some conclusions.

2. Evolution problem

The need for a software system to evolve arises from different reasons. The trivial reason is that corrections have to be made to fix problems. A major need for modifications arises when the requirements to the software change. There are several, partly overlapping reasons for these changes

- The requirements are initially incomplete or misunderstood. With this view the need for evolution arises from a "bug" in an early phase of the software development process.
- The software is developed incrementally. This increasingly common viewpoint accepts that it is impossible to create a complete specification of the software in advance. The software is developed in incremental steps that build on top of previous releases and on user feedback.
- A new version of the software is needed. To preserve or increase their market share, the software vendors typically need to develop new versions of their products with enhanced functionality. Almost always the only reasonable way to go is to modify the old software.
- Software or a piece of it is reused in a different task than what is was initially intended for. The adaptation of a software to a new use typically requires changes.

Because of its importance the software evolution problem has received a lot of attention. At general level it has been one of the key driving forces behind the interest towards software processes and methodologies. A successful approach to solve the problem has been various improvements in software structure and modularity (e.g. software architecture, object-orientation, libraries, components, design patterns). Another major approach has been to create such software development processes that both aim to create clean, easy-to-evolve software and ensure that all relevant requirements are considered in the first stage. For example, Rajlich (1997), Burd and Munro (1998a) and Burd and Munro (1998b) focus explicitly on the methodologies and tools to support software evolution. Furthermore, there are various approaches that try to measure the evolution potential of the software, e.g., Ferneley (1999), Granja-Alvarez and Barranco-García (1998), Welker et al. (1997), Svahnberg and Bosch (1999) and Mattsson and Bosch (2000) present practical analysis software evolution in product families and frameworks, SER Consortium (1996) focuses on the evolution of banking software. Davidson et al. (2000) concentrates on modeling the costs and Mockus and Weiss (2000) on modeling the risks of software evolution at telecom applications.

Typically the efforts to deal with the evolution problem are focused on the general software system issues. The number of studies that focus on the evolution of algorithms and that try to investigate what the evolution means to the selection, implementation, and maintenance of algorithms is quite small.

Knuth (1989) is a detailed analysis of the development history of TeX typesetting system focusing on implementation mistakes. Ball and Magazine (1981) introduced various criteria for the comparison of heuristic algorithms, such as ease of implementation, flexibility, and simplicity, which to some extent characterize the evolution potential of the algorithms.

Some systems experiment with ideas of automating algorithm evolution by using genetic programming, e.g. Lukschandl et al. (1999) for routing algorithms and Teller (1998) for signal understanding algorithms. The systems are constrained to limited problem areas and well-defined solution spaces and are still quite far away from practice.

3. Network architecture planning

An essential task of transmission network planning is to combine traffic demands with a given network topology. This step is called routing. Fig. 1 illustrates the network planning process.

The traffic demands specify how much traffic needs to be sent between nodes x and y. An additional part of the traffic demand is the protection requirement. It specifies whether a single route or two (preferably disjoint) routes are needed. The protection affects the availability of the end-to-end connections in case of failure. Availability is an important planning criterion.

The topology specifies a candidate network structure. In addition to the nodes and edges it may have upper limits for edge capacities. The networks are typically very sparse since extra edges add to the network cost.

After the routing operation has combined the topology and traffic demands it is possible to study issues like how the capacities in the network changed, was there enough spare capacity for all the traffic, and what kind of routes the traffic is using.

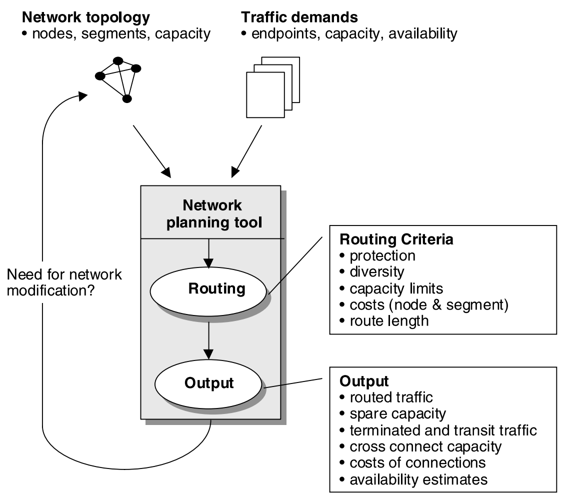

Fig. 1. Network architecture planning process.

Network planning is a complicated task with multiple goals. It requires a lot of user judgment to balance the different objectives and to anticipate future needs. Our experience is that a planning tool supporting the user in this task should be very flexible and visual. Tedious planning steps could be automated, but the user must have control to override all design decisions manually.

The tight user involvement sets some requirements for the planning algorithms. The algorithms should be fast enough so that the user is willing to wait while they are running. There should rather be a set of small algorithms than a single algorithm trying to solve the complete design problem. By running a sequence of smaller algorithms the user can make manual changes between the steps or iterate back to previous steps. Finally, it is easier for the users to understand what a small algorithm does. There is less need to enter parameters to the algorithm since the user can make decisions between the steps. This is especially true in balancing multiple objectives. The downside of the approach is that the user must be competent and cannot just press a button and wait for a ready answer.

Nokia NPS/10 is a transmission network planning system that has been built with the above ideas in mind. It consists of a set of simple tools, or commands, that the user can apply to any part of the network in any order he or she likes. The control of the planning process is thus tightly in the hands of the user. NPS/10 contains features for capacity and other calculations, a graphical user interface for visualization and editing the network topology and traffic, as well as numerous algorithms for routing and topology generation. The system works in the Microsoft Windows environment. The system has been in extensive internal use since 1993 and commercially available since 1996. According to the last count the system has over 100 active users within Nokia.

4. The shortest path problem and Dijkstra's algorithm

The shortest path problem is to find the path in a weighted graph connecting two given nodes a and b with the property that the sum of the weights of all the edges is minimized over all such paths (Sedgewick, 1995). The problem can be expressed as a linear optimization problem where the objective is to minimize

$$\sum_{(i,j)\in E} c_{ij}x_{ij}$$

subject to

$$\sum_{\{j|(a,j)\in E\}} x_{aj} - \sum_{\{i|(i,a)\in E\}} x_{ia} = 1,$$

$$\sum_{\{j|(b,j)\in E\}} x_{bj} - \sum_{\{i|(i,b)\in Et\}} x_{ib} = -1,$$

$$\sum_{\{j|(k,j)\in E\}} x_{kj} - \sum_{\{i|(i,k)\in E\}} x_{ik} = 0 \quad \text{for } k \in N - \{a,b\}, \text{ and}$$

$$x_{ij} \geqslant 0 \quad \text{for all } (i,j) \in E,$$

where N is the set of nodes, E is the set of directed edges, $a, b \in N$ are the endpoints for the path, $c_{ij} \geqslant 0$ is the weight of edge $(i,j) \in E$, and x_{ij} denotes how many times the path goes through edge $(i,j) \in E$.

Notice, that this formulation allows anomalous solutions that contain zero-weight cycles: in practical implementations these cycles can easily be avoided. The optimal solutions to the above problem are always integer, and if the zero-weight cycles are removed x_{ij} is always 0 or 1 in these solutions.

The network routing problem maps to the shortest path problem once the correspondence between "weights" and some network planning criteria has been defined. The weights can, for instance, represent the costs of using an edge for a route or they can simply be equal to one (to minimize the hop count).

Dijkstra's algorithm is a breadth-first search through the network with an ingenious heuristic to cut the search space. It requires that all weights in the network are non-negative.

```
Function Dijkstra (G:Graph, x:Node, y:Node)
returns Path // Returns the shortest path from x to y in G
var
  R: Set of Paths
  V: Set of Paths
do
  Add a trivial path from x to x with weight 0 in R.
  While R is not empty do
    Let p be the path in R which has the least weight.
    Move p from R to V.
    Let node v be the destination of path p.
    If v = y then // early termination rule
      Return p.
    end if.
    For every edge e incident from v in G do
      Let w be the destination of e.
      Let q be the path p extended by edge e.
      If V contains a path to w then // cutting rule
        Do nothing.
      else if R contains a path r to w then
        If weight of q is less than weight of r then
          Replace r by q in R.
        end if.
      else
        Add q in R.
      end if.
    end for.
  end while.
  Return empty path.
end function.
```

In the algorithm presentation we manipulate directed *paths* from source node x to other nodes in the graph G. We allow *trivial* paths as well, i.e. such paths that consist of a single node and no edges. The weight of a path is the sum of the weights of its edges.

The algorithm makes a breadth-first search for minimum weight paths from source node x to other nodes in the network until a path to the destination node is found. If the *early termination* rule (the first if- clause in the while-loop) is removed, the algorithm searches for shortest paths from source node to all the other nodes.

The algorithm maintains two sets of paths. The paths in set V are known to be optimal to their destination while paths in set R are the best paths the algorithm has found thus far. Due to the *cutting rule* (the nested if-clause within the inner loop) the sets together never contain more than one path to any node. (One can replace this rule with a simpler rule "add q in R if q is non-cyclic" and the algorithm still finds a shortest path, but with inferior performance. The set V will finally contain all non-cyclic paths from source node to other nodes.)

To implement the algorithm one can use very simple data structures. The elements of the sets are usually records having fields for the destination node, for the last edge, and for the total weight of the path. A Boolean attribute field is used to tell in which set the path is and a link field points to the element representing the initial segment of the path.

This representation is possible since elements that enter set V are never removed and no path can be inserted in R or V before its initial segments are in V. There is no need to store the whole path in each element since the path can be collected by traversing through the links.

Many methods to help finding the minimum weight element in the tree R have been suggested, e.g. Dial's implementation and various heap implementations (Ahuja et al., 1993; Cherkassky et al., 1994).

The key benefit of Dijkstra's cutting rule is the already mentioned fact: the sets R and V together never contain more than one element for each node in the network. This makes the algorithm very efficient. The maximum size of the sets is known in advance (once the size of the network is known) and the path replacement in the cutting rule is extremely easy: one just needs to modify the weight and parent link of a single element in the set. Thus the algorithm is very fast. For example, a binary heap implementation solves the shortest path problem in $O(e\log(n))$ time (where e is the number of edges and n is the number of nodes in the graph).

5. Applications to network planning

In this Section we will discuss various network planning problems and, in particular, how Dijkstra's algorithm has been used to solve them. We are in particular emphasizing what kind of modifications to the basic algorithm have been necessary and what have been the practical experiences of each step.

As any software development, the planning tool functionality is developed in stages. The main routing features in different NPS/10 releases are represented in Table 1. Once completed the releases have been immediately taken into use in real planning projects. The practical experiences have created new requirements and improvement needs, which have been satisfied in subsequent releases. Since the number of requirements is typically higher than the available time and resources permit, the development has to prioritize the features and constantly consider what is the most efficient way to satisfy the user needs.

5.1. Unprotected routing

Unprotected routing is the most straightforward application of the shortest path algorithm. The problem is to find a single route between given nodes with a minimum cost, assuming that each edge has a non-negative cost and an infinite capacity. The problem maps directly to the shortest path problem. It can be solved with the basic Dijkstra's algorithm.

Considering the practical network planning just having an efficient algorithm to route a single traffic requirement is not adequate. In real networks one always has multiple traffic requirements which need to share the same transmission links. This leads us to *multicommodity flow problems*: we have multiple commodities to be routed which are not allowed to be mixed and which share common resources. Much research has been done on this topic and several sophisticated techniques to solve such problems have been proposed (see e.g., Bertsekas, 1998; Balakrishnan et al., 1989; Assad, 1978; Schneur and Orlin, 1998). Unfortunately the problem is NP-hard and all these algorithms are computationally feasible only with small networks.

We can distinguish two different facets of this issue. Firstly, the capacity of real equipment is limited: this is discussed in greater detail in Section 5.4. Secondly, in practice edge costs are not linear to their capacity but they increase in steps. Edges have an initial fixed building cost (e.g., laying the cables), and after the initial investment the cost increases in steps when more powerful equipment is needed (see Fig. 2).

A simple way to handle the initial building cost is to assume that the user specifies the network topology. The building costs are thus implicitly taken into account already before traffic routing.

Table 1
Routing algorithms in different software releases

Release	Date	Features
1.0	12/92	Single path (Dijkstra)
1.1	2/93	Double path (Dijkstra)
		Semimanual routing
1.5	12/94	Heuristic double path
2.1	5/95	Cost model
		Capacity limited routing
3.1	9/96	Regions
3.3	5/97	Multilayer routing
3.5	2/98	Iterative heuristic double path
		Double path (Minimum cost flow)

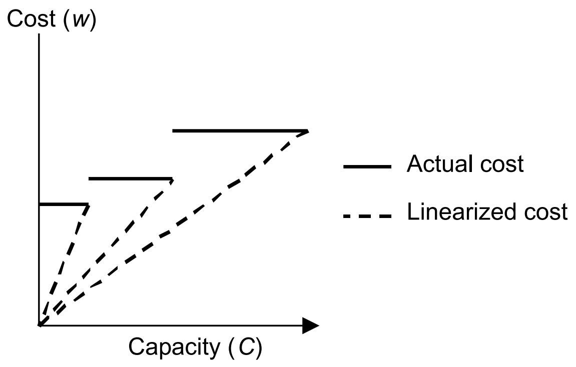

Fig. 2. Edge cost function.

There are several alternatives to handle the capacity dependent variable costs. The simplest one is to ignore the costs completely and rely on hop count or on some user specified cost. Hop count is often a useful starting point just to learn something from the network behavior. A more refined cost model can be based on user specified estimated capacities. We use a linear cost function

$$w_{ij}(c) = \frac{W(C_{ij})}{C_{ij}}c,$$

where C_{ij} is the user specified maximum capacity of the edge, W is the actual cost, c is the capacity of the traffic to be routed, and w_{ij} is the weight to be used in the routing. Fig. 2 shows in dotted line three alternative cost functions.

The linear cost function is a very rough estimate of the actual cost. Its nice feature is that by having a linear function the routing order is insignificant. Our scheme also favors edges that have been equipped with high capacity because the unit cost per capacity unit is lower for high capacity equipment.

For a more advanced solution where the initial costs are better taken in account, see e.g. (Holmberg and Hellstrand, 1998).

5.2. Semimanual routing

In some cases the user may wish to manually specify how traffic is routed. This often happens when there are some special issues that the routing algorithms do not take into account. The semimanual extension to basic unprotected routing makes this possible. The user specifies an ordered set of nodes on the routes and the semimanual routing algorithm finds the shortest path through this node set.

The algorithm uses Dijkstra's algorithm in an iterative fashion.

```
Function Single_Path_Through(G: Graph, S:Vector of Nodes)
returns Path // Returns the shortest path passing nodes S in given order
var
  P: Path
do
  Let v be the first node of S.
  Let P be the trivial path from v to v.
  while there are nodes in S after v do
    Let w be the next node in S after v.
    Append Dijkstra(G,v,w) to P.
    Set v to w.
  endwhile.
  return P.
endfunction.
```

Semimanual routine allows the user to have more control of the routing process but still makes it fairly easy to specify long routes. The major drawback of the approach is that the user must specify the nodes in the proper order. A failure to do so results into strange routes, especially when the selection between two equally long routes in arbitrary (see Fig. 3).

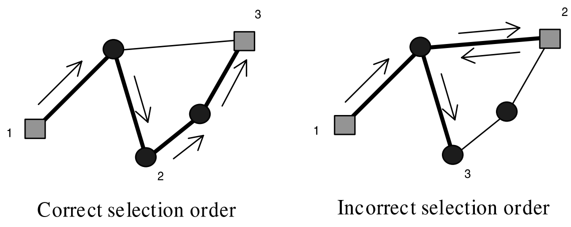

Fig. 3. Effect of selection order to the semimanual routing.

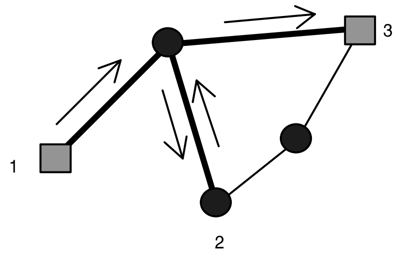

Fig. 4. Unexpected result of semimanual routing.

In addition the algorithm itself might produce an unexpected result even though selection order were correct since each path between two selected nodes is searched independently on the others (see Fig. 4).

There are techniques to improve the algorithm (e.g. branch-and-bound), but they were not tried in the software: users deemed other features of the software to be more important than fixing these defects.

5.3. Protected routing

5.3.1. Double path

The purpose of protection is to make sure that a network connection continues to work also in case of failures. In the $1 + 1$ protection scheme the same signal is sent both through the primary and through the secondary route. If one of the routes fails, the other one still continues to work. In terms of network planning the goal is thus to find two routes for the traffic that use disjoint paths. Since edge failures are more common than node failures (e.g., a cable is broken because of some construction work) the goal is to find two edge-disjoint paths for the traffic.

The following Double Path algorithm is a modification of Dijkstra's algorithm (differences to the original algorithm are underlined).

```
Function Double_Path_Dijkstra (G:Graph, x:Node, y:Node)
returns Pair of Paths // Returns the shortest path
var
  R: Set of Pairs of Paths
  V: Set of Pairs of Paths
do
  Add pair of trivial paths from x to x in R.
  While R is not empty do
   Let (p, p') be the pair of paths in R which has the least weight.
   Move (p, p') from R to V.
   Let nodes v, v' be the destinations of paths p, p', respectively.
   If v = y and v' = y then
     Return (p,p').
   endif.
   For every edge e incident from v in G do
     Let w be the destination of e.
     Let q be the path p extended by edge e.
     If V contains a pair of paths to (w, v') then
       Do nothing.
```

```
        else if R contains a pair of paths (r,r′) to (w,v′) then
          If weight of (q,p′) is less than weight of (r,r′) then
            Replace (r,r′) by (q,p′) in R.
          end if.
        else
          Add (q, p′) in R.
        end if.
      end for.
      For every edge e incident from v′ in G do
        Let w′ be the destination of e.
        Let q′ be the path p′ extended by edge e.
        If V contains a pair of paths to (v, w′) then
          Do nothing.
        else if R contains a pair of paths (r, r′) to (v, w′) then
          If weight of (p, q′) is less than weight of (r, r′) then
            Replace (r, r′) by (p, q′) in R.
          end if.
        else
          Add (p, q′) in R.
        end if.
      end for.
    end while.
  Return pair of empty paths.
  end function.
```

The only difference between the original and double path algorithm is that while the original deals with paths, the double path variant manipulates pairs of paths. The destination nodes of each pair can be the same or different. Of course some fundamental operations need to be refined.

- The inner for-loop in the original is duplicated here. The first loop goes through possible extensions to the first path p while the second loop tries to extend the second path p'. In other words, we extend only one of the paths in the pair, not both simultaneously.
- The weight of the pair (p, p') is the sum of weights of p and p' and of *common link penalties* for the common edges in p and p'

$$w(p,p') = \sum_{a \text{ in } p} w(a) + \sum_{b \text{ in } p'} w(b) + \sum_{c \text{ in } p \cap p'} p(c), \tag{1}$$

where $p(c)$ is the common link penalty for edge c. That can be chosen for each edge separately or we can use a global rule $p(c) = Pw(c)$, where P is a constant, *common link penalty factor*. By varying this factor the user can control the trade-off between route length and disjointness.

While it is relatively easy to show that the original Dijkstra algorithm really produces a shortest path, showing the same result for the double path variant is much more difficult, mainly because of the extra complexity due to the common link penalty in the weight function.

The double path algorithm is much more inefficient than the basic shortest path algorithm. The sets R and V can contain a pair of paths for every (ordered) node pair in the network, and a binary heap implementation runs in $O(ne\log(n))$ time. In practice this is not very severe limitation: as long as the networks are small and sparse the running times are reasonable.

A different kind of complication is the need to specify the trade-off between route length and disjointness. This is done via the common link penalty factor. As is often the case with this kind of parameter values, users find it very hard to adjust the value so that a reasonable compromise can be achieved. In practice the penalty is almost always set to a high value so that common edges are never preferred. Of course depending on the network topology common edges cannot always be avoided.

5.3.2. Heuristic double path

Heuristic double path is an approach to shorten the running times of double path algorithm. The algorithm uses a simple greedy approach: it finds a shortest path (primary route), increases the costs of the edges on the primary route, and reruns the Dijkstra's algorithm to find the secondary path.

```
Function Heuristic_Double_Path(G: Graph, x: Node, y: Node)
returns Pair of Paths
var
  P: Path
  Q: Path
do
  Let P be Dijkstra(G, x, y).
  For each edge e in P do
   Increase weight of e in G by common link penalty.
  endfor.
  Let Q be Dijkstra(G, x, y).
  Return (P, Q).
endfunction.
```

This algorithm is fast compared to the double path routing of the previous section. The drawback is that the results are not optimal in some cases. For instance in the network of Fig. 5 (assuming the weight of each edge is 1) the algorithm selects the primary path (A–B–F–E, drawn with thick solid lines) too greedily. After that it no longer can find an edge-disjoint secondary path between A and E (drawn with thick dashed lines where it differs from the primary one). The normal double path algorithm, on the other hand, finds the paths (A–H–G–F–E) and (A–B–C–D–E) correctly.

This kind of ring topology is common in real networks. Therefore the algorithm is not very useful in practical planning tasks. Its main use is to get some results quickly especially in the initial planning phase when the user just needs to become acquainted with the network behavior.

A similar heuristics to find several disjoint paths has been proposed by MacGregor and Grover (1994).

5.3.3. Iterative heuristic double path

Heuristic double path algorithm can be improved by iterating

```
Function Iterative_Heuristic_Double_Path(G: Graph, x: Node, y: Node)
returns Pair of Paths
do
  Let Paths P and Q be Dijkstra(G, x, y).
  Let w be the weight of (P, Q).
  While w contains common link penalties do
   Let graph H be equal to G where the weight of each
       edge in P is incremented by common link penalty.
   Let R be Dijkstra(H, x, y).
   If weight of (P, R) is less than w then
     Let w be the weight of (P, R) and let Q be R.
   else
     exit loop.
```

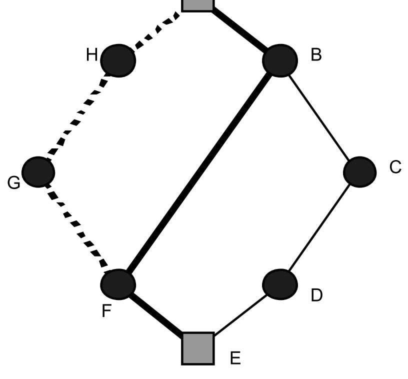

Fig. 5. Example of heuristic double path failure.

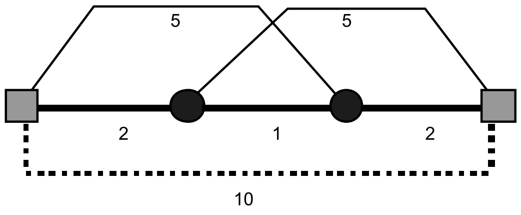

Fig. 6. Iterative heuristic double path does not find global optimum.

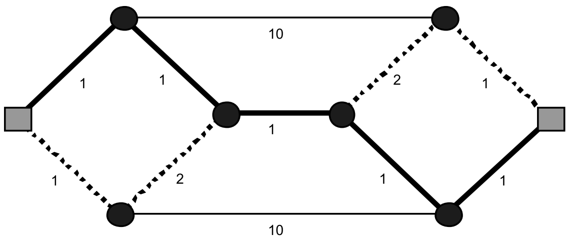

Fig. 7. Iterative heuristic double path does not find disjoint paths.

```
      Swap P and Q.
    endwhile.
    Return (P, Q).
  endfunction.
```

To start with we search the shortest path from source node to destination node and assign it to both paths. Then we try to improve the result step-by-step. We take one of the paths and reroute it trying to avoid the other one, and continue rerouting the paths as long as the situation improves. Finally we reach a local optimum: rerouting neither of the paths will improve the result.

Even though there is no upper bound for the number of iterations needed, usually a small amount (2–3) is enough, so the algorithm is reasonably fast. Sometimes it may fail to find a global optimum (Fig. 6) or two disjoint routes (Fig. 7). In the figures the numbers attached to edges denote edge weights and we assume common link penalties are high. The resulting primary route is shown with thick solid line and the secondary one, where it differs from the primary, with dashed line.

In spite of these weaknesses, the algorithm is useful in practice, giving acceptable routing in most practical cases. The main benefit is that the algorithm is very adaptable to different ways of calculating the weight of the double path.

5.3.4. Double path with minimum cost flow

When, in the course of time, the transmission network plans became more detailed and the networks grew in size and complexity, the double path Dijkstra algorithm presented above turned out to be too slow. As a replacement we selected a well-known solution in the literature, an algorithm which is generally known as *successive shortest path algorithm* to find a *minimum cost flow*, see (Ahuja et al., 1993; Suurballe and Tarjan, 1984).

```
Function Double_Path_Min_Flow(G: Graph, x: Node, y: Node)
returns Pair of Paths
var
  p: Mapping from Nodes to Reals // potential
  v: Mapping from Edges to Reals // reduced weights
do
  // find shortest path
  Let P be Dijkstra(G, x, y).
  // define node potentials
  For each node n in G do
    Let p(n) be the smaller from weight of P and weight of shortest path from x to n.
  endfor.
  // define residual graph
```

```
    For each edge e in G do
      If e is in P then
        Let v(e) be the sum of weight of e and common link penalty of e.
      else if reverse of e is in P then
        Let v(e) be the negated weight of e.
      else
        Let v(e) be the weight of e.
    endfor.
    // calculate reduced weights in the residual graph
    For each edge e from node a to node b in G do
      Add p(a) - p(b) in v(e).
    endfor.
    // find augmenting path
    Let H be graph G where weights are given by v.
    Let Q be Dijkstra(H,x,y).
    // combine shortest and augmenting path
    Return the superposition of P and Q.
  endfunction.
```

In the algorithm we assume the graphs are directed. One basic concept used in the algorithm is node *potential.* In general, potentials are just numbers attached to nodes. Once one has a potential $p(n)$ for each node n, one can calculate *reduced weights* for edges: the reduced weight of an edge e from node a to node b is the weight of edge $w(e)$ reduced by the difference of the potentials of its endpoints

$$w_{\text{red}}(e) = w(e) + p(a) - p(b).$$

The key fact is that, for searching shortest paths between two nodes, it does not matter whether one uses reduced weights or just plain weights. The reduced weight of a path, namely, equals to the actual weight of the path reduced by the difference of the potentials of its endpoints.

The basic idea of the algorithm is that to have a double path from source node x to destination node, we actually want a flow of two units from x to y. To get disjoint flows we restrict that each edge can carry at most one unit of flow without extra penalty. Conceptually this is achieved by duplicating all the edges and increasing the weight of the duplicate by the common link penalty weight. (However, in practice we don't need to add the duplicates, see later.)

We start by searching a single unit of flow, i.e. the shortest path from source node to destination node, using the known Dijkstra shortest path algorithm.

In the first for-loop we calculate a *potential* $p(n)$ for each node. In this particular case potentials are selected in such a way that the reduced weights on the shortest path P become zero while everywhere else they are non-negative. The calculation effort is in practice very small after one retains the result set V of the Dijkstra algorithm from the previous shortest path search. After a shortest path P from x to y has been caught by the early termination rule, we know that the paths in set V are shortest paths to those nodes which are closer to x than y, and for all the other nodes the length of shortest path is at least the length of P. Thus, to calculate the potential $p(n)$, we just need to check whether set V contains a path from x to n, and $p(n)$ is the weight of this path if it exists, and the weight of P otherwise.

Fig. 8 shows the network of Fig. 7 after this phase. The numbers attached to edges are edge weights (these are the same in both directions). The thick arrows denote the shortest path and the numbers attached to nodes are the potentials.

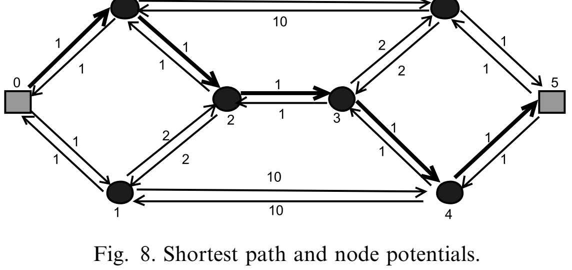

Fig. 8. Shortest path and node potentials.

In the second loop we calculate a *residual* graph H which one gets when one "removes" the already allocated flow *P* from graph G. In practice this means recalculating the weight function. We increase the weight of each edge in *P* by common link penalty: by doing so we "add" a duplicating edge with the increased penalty. To compensate, we turn the weight of each reversed edge negative: by using such edge when allocating further flow one "cancels" the allocation made earlier. All the other edges retain their weight. See Fig. 9. For clarity we have still retained the shortest path in the figure. The common link penalty has been chosen as 100.

The second for-loop calculates the reduced weights so that we get rid of negative edge weights. By this way a new graph is formed. Then we allocate a second unit of flow, *an augmenting path Q*, by searching the shortest path through H. See Fig. 10, where you can see edges with reduced weight, the shortest path *P* (thick solid arrows) and the new, augmenting path *Q* (thick dashed arrows).

The final step in the algorithm is to combine the unit flows *P* and *Q* into two paths. Of course *P* and *Q* are paths, but it may now be that *P* traverses one edge forwards while *Q* traverses the same edge backwards. In the superposition of the two paths these kinds of edges "cancel" each other and are removed. You can see the final result of our example case in Fig. 11.

Table 2 shows the speed comparison of double path (Dijkstra), heuristic double path (heuristic), and double path with minimum cost flow (min flow) algorithms. Density is the average degree of a node. The running times are seconds with a 90 MHz Pentium computer. The double path algorithm was not able to solve the largest networks within 10 min.

The minimum cost flow algorithm is much faster than the double path Dijkstra algorithm. It can be also easily extended to find any number of disjoint paths and not only two. On the other hand, it is more complex and requires deeper insight of the problem and more understanding of the graph algorithms. These factors together with the limited development resources and time pressure were the reasons why the double path Dijkstra algorithm was initially implemented in our application. Although clearly inferior in solution time it was used almost five years before the pressure to handle larger networks made it necessary to replace it.

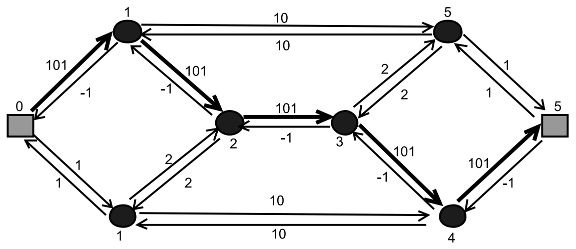

Fig. 9. Residual graph after one unit of flow.

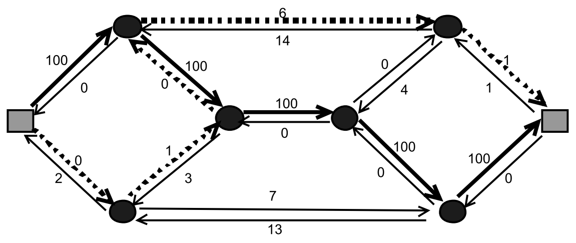

Fig. 10. Reduced weights in the residual graph, augmenting path.

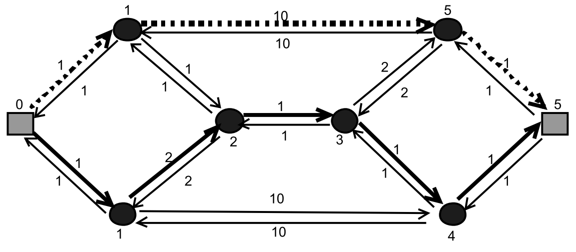

Fig. 11. Final result, original weights.

Table 2
Speed comparison of the double path algorithms

Network	Nodes	Edges	Density	Dijkstra	Heuristic	Min flow
S10.TRP	10	10	2	0.1	0	0.1
S50.TRP	50	62	2.5	3.1	0.1	0.2
S290.TRP	290	374	2.6	289.1	0.7	0.9
S1154.TRP	1154	1498	2.6	*	7.4	9.7
D10.TRP	10	22	4.4	0.2	0	0
D34.TRP	34	88	5.2	2.4	0.1	0.2
D130.TRP	130	358	5.5	60.4	0.5	0.8
D514.TRP	514	1474	5.7	*	4.6	3.3
D1026.TRP	1026	2974	5.8	*	14.7	6.7

In terms of speed heuristic double path and minimum cost flow algorithms are roughly equal (the difference comes out mainly from the different heap implementations in shortest path algorithms). The big advantage of the minimum cost flow algorithm is that it always finds an optimal solution whereas the heuristic algorithms cannot find good solutions with all topologies. The benefit of the heuristic solutions (other than being simpler) is that they are more adaptable. For instance when more complex calculation is needed for the route cost (e.g. to handle issues that arise in multilayer routing) the minimum cost flow algorithm cannot be used. One practical approach in such situation is to search for an initial solution with the minimum cost flow algorithm using an approximate cost function and then improve this solution with the heuristic algorithm with proper cost function.

5.4. Capacity limited routing

The above routing algorithms assume that the network has an unlimited capacity. In reality each edge has a finite capacity which depends on the devices that are located at the end nodes. The capacity can be incremented, but only with an extra cost. Capacity limitation complicates the routing problem because the traffic demands can no longer be treated independently from each other: the problem turns into multicommodity flow problem, which we already briefly discussed in Section 5.1.

In addition to the general multicommodity flow problem solving techniques, some special solutions to capacity limited routing problems have been suggested (e.g. Gavish and Neuman, 1989; Sanso et al., 1992). The problem is very hard to solve with larger networks.

Ignoring the capacity limits is often a good approach in the initial planning but at the later planning stages more accurate results can be achieved by taking in account the step behavior of capacity-to-cost function. One should then try to avoid capacities that are little over the threshold values since in these cases most of the capacity is not used.

One way to simplify the problem is to assume that the maximum capacities of the edges are predefined and the task is to route the traffic to the capacity limited network. The nice feature of this approach is that the users can control where in the network they want to have a high capacity area. The decision depends on a lot of issues, such as supply of space, easy access for maintenance, or future growth potential, which no model can completely cover. Another motivation for using predefined capacity in the network is that the resulting network is better structured and is thus easier to manage and maintain.

The resulting capacity limited routing is also a hard problem but it is easy to develop simple greedy heuristics for its solution. The following heuristic works quite well in practice.

```
Function Capacity_Limited_Dijkstra(G:Graph, x:Node, y:Node, c:Real)
returns Path // Returns the shortest path
var
  R: Set of Paths
  V: Set of Paths
do
  Add trivial path from x to x with weight 0 in R.
  While R is not empty do
   Let p be the path in R which has the least weight.
   Move p from R to V.
   Let node v be the destination of path p.
```

```
      If v = y then
        Return p.
      endif.
      For every edge e incident from v in G which has more spare capacity than c do
        Let w be the destination of e.
        Let q be the path p extended by e.
        If V contains a path to w then
          Do nothing.
        else if R contains a path r to w then
          If weight of q is less than weight of r then
            Replace r by q in R.
          endif.
        else
          Add q in R.
        endif.
      endfor.
    endwhile.
    Return empty path.
  endfunction.
```

The basic Dijkstra's algorithm is modified to check that the traffic capacity requirements do not exceed the maximum capacity of the edge. The only change to the basic algorithm is the underlined check that the capacity in the edge extending the already searched path *p* is large enough. The Capacity_Limited_Dijkstra algorithm selects the optimal path for a single traffic requirement (x, y, c). Similar modifications can also be made to the double path or heuristic double path algorithms. Notice that one can view this change as well as a change in the network topology: the edges having not enough capacity are removed.

Normally there is a set of traffic requirements that have to be routed. Capacity_Limited_Routing routine tries to route them in a decreasing order of capacities. The routing order is the heuristic step of the algorithm. The thinking behind our ordering is that finding routes for the largest demands is difficult and therefore it is done first. The approach is related to the bin packing problem and resembles the First Fit Decreasing heuristic (Kershenbaum, 1993).

```
Function Capacity_Limited_Routing(G: Graph, D: Vector of Connections)
returns Mapping of Connections to Paths
var
  P: Mapping from Connections to Paths
do
  Sort D in descending order of capacity
  For each connection d in D do
   Let a and b be the endpoints of d.
   Let c be the capacity of d.
   Let P map d to Capacity_Limited_Dijkstra(G, a, b, c).
  endfor.
  Return P.
endfunction.
```

The network planners have found the results useful although it is fairly easy to construct examples where it fails. For instance in the network of Fig. 12 the traffic A–D is routed first since it requires more capacity than A–F. The route (A–E–D) is the shortest one for it (assuming $w_{ij} = 1$) and there is enough capacity for it. Now an attempt to route the demand A–F does not succeed since the route A-E-D has already consumed the 10 units of capacity between A–E and E–D. In this case a reverse routing order would have produced a better result.

5.5. Other extensions

Transport networks are typically divided into separate regions for core, regional, and access networks with different design rules and targets (Salo, 1998). The most straightforward effect of this division to the routing algorithms was that

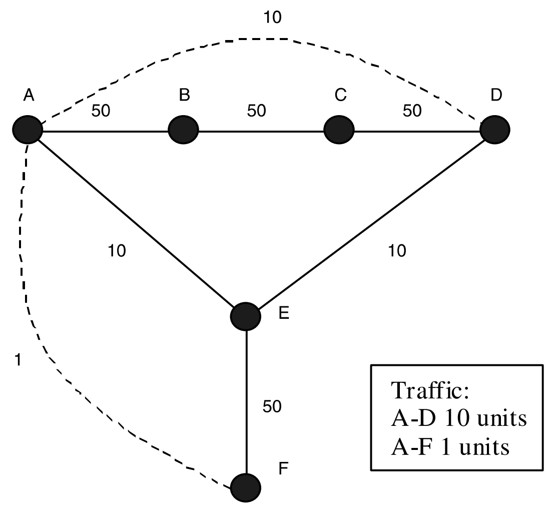

Fig. 12. Example of a capacity limited routing failure.

one should avoid such routes that unnecessarily visit a region. For example, traffic that is internal to an access region should not use other regions. This requirement was easy to include in the Dijkstra based algorithms: only a new way to select the weights and topology was needed.

In addition to the above horizontal division the network functionality is divided into different layers also in the vertical direction. For instance, the traffic is carried in low-capacity (2 M/1.5 M) links that are carried in high-capacity (SDH/SONET) links that are carried in optical fibers. The effect for double path routing is that routes must be disjoint both in the current layer as well as in the underlying layers. Instead of trying to generate the routes of all layers simultaneously we routed the traffic in only one layer at a time. This was done in such a way that the routes that had already been fixed in the lower layers were taken into consideration. This extra constraint could be added to the heuristic algorithms by only changing the weight calculation of path pairs. Common link penalty was added not only when there were same edges on both paths but also when the paths shared a common edge on some lower layer.

It is interesting to notice that the double path with minimum cost flow algorithm does not allow these kinds of changes to be made since it requires that the common link penalties are local. (Otherwise it is impossible to select weights in the reduced graph for the augmenting path.) This is a good example of a trade-off situation where a more efficient algorithm is more inflexible to new requirements.

These kinds of problems can be solved with branch-and-bound approach as well (Oellrich and von Puttkamer, 1998).

Having multiple layers makes it possible to protect at different layers: one may select to protect some traffic at the low-capacity layer while some other traffic is protected at the high-capacity layer. Moreover, not all equipment is able to protect the traffic. Therefore the protection function can happen only in certain nodes of the network. Routing becomes rather challenging when also these facets are taken in account.

6. Conclusions

In this paper we have considered the use of Dijkstra's algorithm to solve different routing problems encountered in transmission network planning. The basic algorithm together with our relatively minor modifications has turned out to be useful and practical in solving real-life problems.

The most important lesson of the study is the importance of evolution. The study shows that the requirements for the algorithms change during the evolution of the software system. In order to meet the changing needs the algorithms have to be modified and redeveloped. The ability of an algorithm to grow and meet the evolution requirements is an important feature.

Our findings underline the importance to consider the evolution potential when making the decision about the algorithms to be used. It is not enough to focus on selecting the most sophisticated state-of-the-art algorithm for a given problem. The evolution capability, the potential for reuse, and the development cost over the system lifetime are equally important aspects. Flexibility is often more important than strict optimality.

It is also important to understand that because of the nature of the problem domain, network architecture planning, the algorithms do not have to be very accurate. No model can completely handle all explicit and implicit goals and variables of a real-life planning case. A planning tool has to provide facilities for the user to use his or her judgment to control the planning process and to interpret and modify the results. An intuitive user interface and a set of fast, interactively used, easy-to-use and easy-to-understand algorithms makes it possible for the user to experiment with different alternatives and to compensate the inevitable deficiencies in the optimization algorithms of the software. As a matter of fact, since it is almost impossible to specify the optimization problem very exactly, a rigid state-of-the-art optimization algorithm may suggest theoretically optimal but practically infeasible solutions to the network architecture planning problem.

An interesting observation is that often the developers erroneously tend to set stricter requirements to the algorithms than what the users really need. After all, they understand better the deficiencies of the algorithms and easily worry too much. Users have certain priorities concerning different features of the software. Having the most efficient and sophisticated routing algorithms is not necessarily a top priority issue. The solution power of sophisticated algorithms is often needed for anomalous problem instances that seldom occur in practice. For example, in practical telecommunication network plans the network topologies cannot be very complex: after all, the plans must be maintained and understood by humans.

How is this interpreted to the requirements to the algorithms? What kind of algorithms would be most suitable to fulfil the above requirements?

First of all the algorithms should be simple. Our experience suggests that it is in many cases reasonable to sacrifice some accuracy or strict optimality if a simpler solution is available. Simple algorithms are easy to modify and apply to new needs, the initial implementation time is often short, and the number of implementation errors is likely to be small. As there is increasingly heavy pressure for fast time-to-market the speed of development to an actual product stage is a critical issue. Moreover the companies face a shortage of skilled developers. Ideally the developer should understand well the application area, programming, and the models and algorithms. The more complex the algorithms are the higher the demands are for the developers. Simpliness is also significant for the user since it is easier for them to understand what kind of calculations the software is performing for them. This increases their confidence towards the results.

Another important goal for algorithm is generality. Special purpose heuristics often increase the accuracy or solution speed of an algorithm but the heuristics used are often very vulnerable to changes. For instance, the double path with minimum cost flow algorithm (Section 5.3.4) could not be used for the multilayer routing task.

The speed is third key attribute. A fast algorithm has several benefits. It allows the user to work interactively and in that way to compensate for the shortcomings of the algorithm. A fast algorithm can also be used as a building block of a more complex algorithm, such as in the heuristic double path (Section 5.3.2) or in capacity limited routing of the whole network (Section 5.4). Finally, fast algorithm easily scales to larger problem instances. In our case, when we began our development work, the typical network sizes were tens of nodes, but, in the course of years, the plan sizes increased to thousands of nodes.

A problem with the above three key attributes is that they are often conflicting. For instance, special heuristics can improve the solution speed. Moreover, in addition to the above criteria which contribute to the evolution and reuse of the algorithms the normal issues, like accuracy of the results, should not be neglected completely. It is not easy to derive any simple rules or recommendations that would help in the selection process. In our case we typically selected the algorithm which took the least amount of development effort and was able to give acceptable results. The acceptability was tested through user feedback: we implemented a simple algorithm and made improvements only if requested by users.

This paper presents a single case of algorithm evolution and it suffers from the normal weakness of case studies that each development project is relatively unique (Zelkowitz and Wallace, 1998). More research would be needed to better understand the issues involved. The problem with studies of real-life cases of software evolution is that a long time span is needed. The goal of a real-life project is to create the necessary software and functionality – the collection of research data is a by-product. Performing controlled experiments on this area is difficult, slow, and expensive.

An interesting topic for further research would be the development of new measures and ways to compare algorithms in the evolution dimension. The “standard” scientific way compares algorithm running times and closeness to optimality. As the potential for evolution is not explicitly measured there is a risk that speed and optimality are overemphasized in algorithm research. Possibly the measures of software complexity, such as Halstead (1977) and McCabe (1976), could be used to somehow estimate the evolution potential of an algorithm. Even if the complexity measure would be a rough estimate of the evolution potential it would help in keeping this important practical perspective in focus.

References

Ahuja, R.K., Magnanti, T.L., Orlin, J.B., 1993. Network Flows: Theory, Algorithms, and Applications. Prentice-Hall, Englewood Cliffs, NJ.

Assad, A.A., 1978. Multicommodity network flows – a survey. Networks 8, 37–91.

Balakrishnan, A., Magnanti, T.L., Wong, R.T., 1989. A dual-ascent procedure for large-scale uncapacitated network design. Operations Research 37 (5).

Ball, M., Magazine, M., 1981. The design and analysis of heuristics. Networks (11), 215–219.

Bertsekas, D., 1998. Network Optimization: Continuous and Discrete Models. Athena Scientific, Belmont, MA.

Burd, E., Munro, M., 1998a. Investigating component-based maintenance and the effect of software evolution: a reengineering approach using data clustering. In: International Conference on Software Maintenance (ICSM'98).

Burd, E., Munro, M., 1998b. Reengineering support for software evolution: an evaluation through case-study. In: 22nd Annual Intenational Computer Software and Application Conference (COMPSAC 1998).

Cherkassky, B.V, Goldberg, A.V., Radzik, T., 1994. Shortest paths algorithms: Theory and experimental evaluation. In: Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms.

Davidson, J.W., Mancl, D.M., Opdyke, W.F., 2000. Understanding and addressing the essential costs of evolving systems. Bell Labs Technical Journal 5 (2), 44–54.

Dijkstra, E., 1959. A note on two problems in connexion with graphs. Numerische Mathematics 1, 269–271.

Ferneley, E.H., 1999. Design metrics as an aid to software maintenance: an empirical study. Journal of Network and Systems Management 11 (1), 55–72.

Gavish, B., Neuman, I., 1989. A system for routing and capacity assignment in computer communications networks. IEEE Transactions on Communications 37 (4), 360–366.

Granja-Alvarez, J.C., Barranco-García, M.J., 1998. A method for estimating maintenance cost in a software project: a case study. Journal of Software Maintenance: Research and Practice 9 (3), 161–175.

Halstead, M.H., 1977. Elements of Software Science. Elsevier, New York.

Holmberg, K., Hellstrand, J., 1998. Solving the uncapacitated network design problem by a Lagrangean heuristic and branch-and-bound. Operations Research 46 (2), 247–259.

Kershenbaum, A., 1993. Telecommunications Network Design Algorithms. McGraw-Hill, New York.

Knuth, D.E., 1989. The errors of TeX. Software Practice and Experience (19), 607–685.

Lukschandl, E., Borgvall, H., Nohle, L., Nordahl, M., Nordin, P., 1999. Evolving routing algorithms with the JBGP-system. In: Evolutionary Image Analysis, Signal Processing and Telecommunications. First European Workshops, EvoIASP'99 and EuroEcTel'99. Proceedings, Springer, Berlin.

MacGregor, M.H., Grover, W.D., 1994. Optimized *k*-shortest-paths algorithm for facility restoration. Software Practice and Experience 24 (9), 823–834.

Mattsson, M., Bosch, J., 2000. Stability assessment of evolving industrial object-oriented frameworks. Journal of Software Maintenance: Research and Practice 12 (2), 79–102.

McCabe, T.J., 1976. Complexity measure. IEEE Transactions on Software Engineering SE-5 (4), 308–320.

Mockus, A., Weiss, D.M., 2000. Predicting risk of software changes. Bell Labs Technical Journal 5 (2), 169–180.

Nokia Networks, 1999. NPS/10, Computer software.

Oellrich, M., von Puttkamer, J., 1998. Reliable links in transmission networks, in DRCN'98.

Rajlich, V., 1997. MSE: A methodology for software evolution. Journal of Software Maintenance: Research and Practice 9 (2), 103–124.

Salo, E., 1998. Resilience of open topology SDH networks. In: First international workshop on the design of reliable communication networks, IMEC, University Gent.

Sanso, B., Gendreau, M., Soumis, F., 1992. An algorithm for network dimensioning under reliability considerations. Annals of Operations Research 36, 263–274.

Schneur, R., Orlin, J.B., 1998. A scaling algorithm for multicommodity flow problems. Operations Research 46 (2), 231–246.

Sedgewick, R., 1995. Algorithms in C++. Addison-Wesley, Reading, MA.

SER Consortium, 1996. EUROBANQUET – Support for Effective Software Evolution in European Banks. http://dis.sema.es/projects/SER/eurobanq.html.

Suurballe, J.W., Tarjan, R.E., 1984. A quick method for finding shortest pairs of disjoint paths. Networks 14, 325–336.

Svahnberg, M., Bosch, J., 1999. Evolution in software product lines: two cases. Journal of Software Maintenance: Research and Practice 11 (6), 391–422.

Teller, A., 1998. Algorithms Evolution with Internal Reinforcement for Signal Understanding. Ph.D. Thesis, Carnegie Mellon University.

Welker, K.D., Oman, P.W., Atkinson, G.G., 1997. Development and application of an automated source code maintainability index. Journal of Software Maintenance: Research and Practice 9 (3), 127–159.

Zelkowitz, M.V., Wallace, D.R., 1998. Experimental models for validating technology. IEEE Computer 31 (5), 23–31.

Jyrki Akkanen received the M.Sc., Ph.Lic, and Ph.D. degrees in Mathematics from the University of Helsinki in 1989, 1992 and 1995, respectively, specializing in mathematical logic. Since 1995 he has worked as a senior software engineer at Nokia Research Center. His main interests have been routing and optimization algorithms for telecommunication networks and software architectures for network design tools.

Jukka K. Nurminen received the M.Sc and Tech.Lic. degrees from the Helsinki University of Technology in 1986 and 1989, respectively. In 1986 he joined the Nokia Research Center where he is working as a principal scientist on the area of network planning tools. His current interests are telecommunication network modeling as well as the model development and implementation processes.