Technische Universität München
Institut für Informatik
Forschungs- und Lehreinheit Informatik VIII

Untersuchungen von Metaheuristiken bei der Planung von Rechnernetzen

Diplomarbeit

Christian Kreibich

Technische Universität München
Institut für Informatik
Forschungs- und Lehreinheit Informatik VIII

Untersuchungen von Metaheuristiken bei der Planung von Rechnernetzen

Diplomarbeit

Christian Kreibich

Themensteller: Prof. Dr. Eike Jessen
Betreuer: Dr. Volker Gerd Fischer
Abgabedatum: 15. Dezember 2000

Ich versichere, daß ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Berkeley, im Dezember 2000

Contents

Vorwort

Diese Diplomarbeit entstand an der TU München in Zusammenarbeit mit dem Lehrstuhl von Prof. Dr. Eike Jessen unter der Betreuung von Dr. Volker Fischer. Dr. Fischer besitzt damit Rechte über die Verwertung der Ergebnisse dieser Diplomarbeit.

Preface

Designing large and complex networks is a difficult business, independent of the actual nature of the network. In the scope of this document, the focus will be on wide-area computer networks. As will be seen, the number of potential solutions to the problem is overwhelming, thus rendering full enumeration in order to find the overall optimal solution (in terms of e.g. implementation cost) infeasible.

Instead, heuristic approaches that generally produce good results in comparatively little time are used, e.g. Genetic Algorithms, Tabu Search or Simulated Annealing. In order to yield good results in reasonable time, the "ruggedness" of the search space must be tolerable and the objective function which determines the quality of a solution must be evaluated quickly.

This thesis gives a general overview of the issues involved in network topology design and then focuses on the behaviour of meta-heuristic approaches to the problem. To increase the runtime performance of the various algorithms, an incremental all-pairs shortest path algorithm suggested by [Roh85] was implemented and improved in order to be able to handle paths of equal lengths, a typical problem with administrator-assigned link lengths. The convergence behaviour of Tabu Search, Genetic Algorithms, Simulated Annealing, Randomized Neighbourhood Search and pure Randomized Search is then investigated using convergence plots and Search Space Matrices, a search space visualization technique introduced in [Col98], proving Tabu Search to be a slightly more powerful meta-heuristic than Genetic Algorithms in this particular problem setting. Heuristic Link Reduction is shown to be a successful approach to overcome the limitations encountered when basing the search on a complete graph.

Chapter 1 introduces the topic of network topology design, Chapter 2 gives an overview over meta-heuristic search and various algorithms in that domain, Chapter 3 defines the network model used in this thesis and introduces the approach used to analyze the behaviour of the techniques implemented. Chapter 4 introduces means of improving the solvers' runtimes and performances, and also describes the implementation of the incremental all-pairs shortest path algorithm, before the results are presented in Chapter 5. Finally, Chapter 6 summarizes the results.

1

Introduction to Topological Network Design

Networks are an omnipresent part of our everyday life, whether it is in form of computer networks like the Internet, communication networks like the telephone network or transport networks like road or airplane infrastructures. In spite of the great variety of network types, the difficulties in the design process overlap to a large degree. In this thesis the focus will be on computer networks. Even though the various types of networks are commonly used today, there is still no agreed-upon methodology for producing high-quality and cost-effective networks, although research in this area has been going on for decades.

Lack of standard methodology

The reason for this lies in the fact that topological network design comprises a surprisingly large number of aspects, literally all of which have to be taken into account to solve the problem satisfactorily.

The issues to be addressed cover such questions as

- What is the design goal — a fast or low cost network?
- How to define a "good" network design? A metric is needed that can compare various designs to each other.
- What kinds of input data are available to the design task? What models are to be used for those not available?
- What network structure is suited best? Can the design be analyzed on different levels, such as access and backbone networks?
- How to layout the network topology on each level, i.e. which links to actually implement?
- What equipment to implement for the links and nodes in the network — what equipment is available? Are special cost-savings possible by upgrading existing installations? Can they cope with future changes in traffic patterns?
- What traffic routing mechanism is to be implemented? Routing schemes significantly influence the resulting topology and choice of equipment.
- How to calculate the effective cost of a design — what kinds of tariffs are taken into account? Over what period of time is the network designed to be feasible?

This list is by no means exhaustive, but already indicates the complexity of the design process. A definition of the network design problem will be given in section 1.1 after the necessary terminology has been introduced. The following sections of this chapter will elaborate on how the various aspects mentioned above are addressed.

1.1 Structuring the Network Design Task

As was illustrated in the previous section, the network design problem is complex. In order to achieve consistency in the design task, a formal approach to the problem is necessary.

Graph theory terminology

The terminology stemming from graph theory is usually adopted to network design: the hosts in the network are being referred to as *nodes* or *vertices*, the connections between the hosts are called *links* or *edges*. Assigning a numerical value w_e to each edge yields a *weighted* graph. The set of nodes in the graph is denoted V, the set of edges is E, the full network graph is $g(V, E)$. A graph with few edges is called *sparse*, one with many edges *dense*. A graph in which all node pairs are connected is called *complete*.

A sequence of links that is being considered e.g. when investigating routing issues is called a *path*, denoted as $p(n_1, n_2)$ where n_1 is the source and n_2 is the destination node. Paths can be concatenated, denoted as $p_1 \circ p_2$. The inner path p of $p_{prefix} \circ p \circ p_{suffix}$ where either p_{prefix} or p_{suffix} can be empty is called a *subpath*. Paths are assumed to be loop-free. The *path weight* w_p is the sum of all edge weights in path p, $w_p = \sum_{e \in p} w_e$. The set of all paths between two nodes will be denoted $P_{(n_1,n_2)}$. When it is clear which nodes are meant the indices will be left out. In context of counting the number of links in a path, the links are sometimes also called *hops*. The number of hops in a path p is denoted as $|p|$. An edge/link is a path with $|p| = 1$. A graph in which there exists a path from each node to each other is called *connected*.

In a *directed* graph or *digraph*, the edges are defined by ordered tuples $V \times V$. An edge from node a to node b in a directed graph is denoted as $(a \triangleright b)$. A digraph in which there is a directed path between all node pairs is called *strongly connected* (see Fig. 1.1).

Figure 1.1: Undirected (left) and directed (right) graphs. Both are connected, but the right graph is not strongly connected: there is no directed path from node 3 to node 2.

Unless explicitly stated otherwise, all traffic is assumed to be directed, i.e. the traffic flowing from node a to node b is exclusively using link $(a \triangleright b)$, the traffic flowing from node b to a occupies link $(b \triangleright a)$. The directed model can more adequately address the common situation of asymmetric traffic flows between two nodes. Our networks are supposed to be strongly connected.

The following definition of network design is adapted from [Fis00]:

Definition 1.1: Network Design

Network design (also called network planning) deals with the initial setup of a network as well as the ongoing reconfiguration and redesign necessary to accommodate changes in traffic, services and technology in order to satisfy demands imposed by the users of the network, while optimizing the network towards a certain design goal like cost-effectiveness or performance.

Definition 1.2: Network

In the model used in this thesis, a network is defined as a directed graph (V, E) consisting of a set of nodes V representing routers, workstations or subnets, and a set E of directed edges $V \times V$ between the nodes. Each edge is attributed with a certain *length* l that need not necessarily be an Euclidean or geographical distance, and a certain *capacity* c which can commonly be chosen from a finite set of available capacities (e.g. 155 Mbit, 622 Mbit etc).

To reduce the complexity inherent to the problem, the design task is commonly modularized into a *hierarchical* approach as shown in Fig. 1.2.

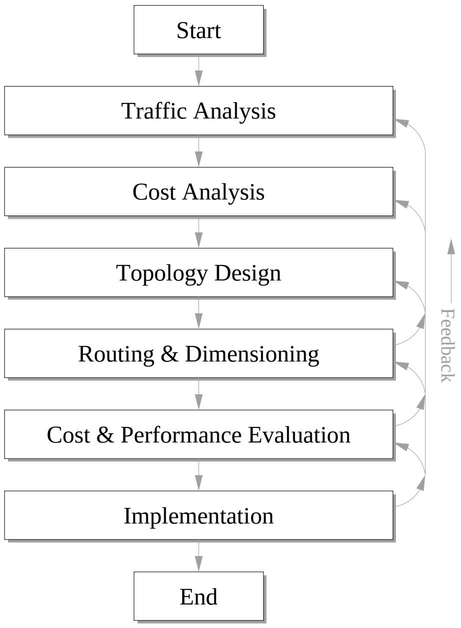

Figure 1.2: Network Design Process

Iterativeness

It is important to note that the process is iterative: most design steps provide feedback that can be used to improve on the results of preceding steps. Thus it is common that an initial design gets improved in successive, refined reinvestigations using better knowledge of the specific problem at hand. The implementation itself also provides feedback: after the network has been in place for some time, new information about traffic growths and weaknesses in the design become apparent that serve as input into a following, incremental design.

1.2 Traffic Analysis

One of the first steps in a network design process is traffic planning & analysis. It is worth noting that it is not necessarily the topology design task that consumes the largest amount of time: depending on the input data available, the effort spent on traffic and cost analysis (see next section) as well as the generation of input files can be substantial, since it may not be possible to automate them. [Cah98] puts it as follows:

> If genius is "1% inspiration and 99% perspiration", then we are lucky, since network design is 90% file preparation and 10% optimization.

The information on traffic requirements in the network is crucial, since most of the calculations use or indirectly depend on this input. Moreover, it is usually difficult to gather comprehensive information. For all node pairs (i, j) on the network, the directed amount of traffic flowing between i and j needs to be determined and recorded in the *requirements matrix Req*. Where traffic flow data cannot be obtained through traffic measurements or similar means, a model to generate traffic requirements is needed.

Traffic requirements matrix

Various models for traffic generation are conceivable. [Cah98] mentions the following, listed in increasing order of realism:

- Uniform traffic volume:
  For all undetermined links (i, j), set $Req(i, j)$ to a fixed amount of traffic.
- Randomized traffic volume:
  For all undetermined links (i, j), set $Req(i, j)$ to a random number that was generated uniformly distributed between the minimum and maximum allowed traffic volumes.
- Population-Distance-based traffic volume:
  A more realistic approach is basing the calculation on the concept of population sizes on both sites i and j (e.g. the human population of cities), and the distance of the sites.

Usually an additional step of traffic normalization in order to move the results in the desired range will be necessary. After the requirements are set up, it is a good idea to calculate the *node weights*.

Definition 1.3: Weight of a Node

The weight W_n of a node n is the sum of all traffic flowing into and out of a node:

$$W_n = \sum_{i \in N} Req(n, i) + \sum_{i \in N} Req(i, n)$$

The node weights serve as a good indicator of the relative importance of nodes. Nodes with a large weight need to handle a lot of traffic and are thus of greater importance in the design process. Fig. 1.3 shows nodes with larger weights using larger circles.

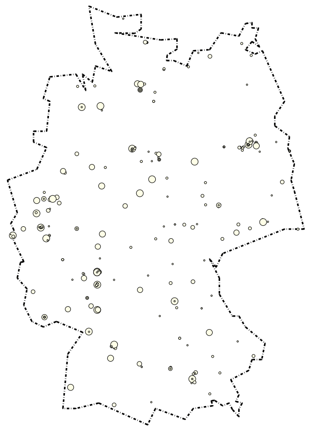

Figure 1.3: Node weights

1.3 Cost Analysis

Cost matrix

The other important input to the design process besides traffic volumes is the cost matrix. It defines the cost of implementing a link between two nodes i and j at a given capacity c, and will be denoted as $Cost_c(i, j)$. If the designer can choose from multiple link capacities, the full cost structure becomes three-dimensional with the capacity as an additional index.

In the real world, the cost of implementing a link depends on the tariffs available. While there is an immense number of different tariffs in existence, two basic billing models can be distinguished:

- Usage-based models
  In usage-based models, the customer's bill is mostly determined by the amount of time the line is used or the amount of data that is transmitted. The telephone system is the classical example. Usually, fixed periodical charges also apply. Depending on the nature of the connection (circuit-switched or packet-switched), this is also called a dial-up channel or pipe.

- Lease models
  In the other model, the customer rents a dedicated line, which usually remains installed or configured for months or years[1]. The setup expenditure is significant, but the usage itself is free of charge. The bandwidth is commonly larger than that of dial-up lines. Again depending on the type of the connection, this type is called fixed channel or pipe.

Tariff diversity

Due to the complexity and fluctuation of existing tariffs, there is currently no comprehensive database available that allows simple querying for the "best" tariff. Therefore, finding a suitable cost model can be time-consuming. If a link cost cannot be determined from available tariffs, cost generators need to be used.

[1]This does not necessarily mean that the line is dedicated to a single customer, but rather that the customer is guaranteed to be able to transfer a certain amount of traffic over that line at any time.

One way of implementing cost generators is using distance-based cost models, based on piecewise-linear or step functions [Cah98]. A cost function could look like this:

$$\mathsf{cost}(d, t, c) = \mathsf{fix}(c) + \mathsf{dist\_cap}(c, d) \cdot t$$

with d being the link length in kilometers, t the duration of the lease (e.g. in months) and c the desired link capacity in Mbit/s. The cost is the combination of a fixed installation or socket cost fix for the desired capacity and the distance function dist_cap which returns the cost of leasing a link of the given capacity and distance, times the desired lease period. Fig. 1.4 shows examples of piecewise-linear and stepped functions, using Table 1.1 as input.

Charge	Distance (km)	Charge (€/km/month)
fix	0	1.998,00
Distance	0 - 100	9.98
Distance	100 - 200	6.95
Distance	200 - 300	5.90
Distance	300 - ∞	4.99

Table 1.1: Example of a cost table for a certain link capacity c

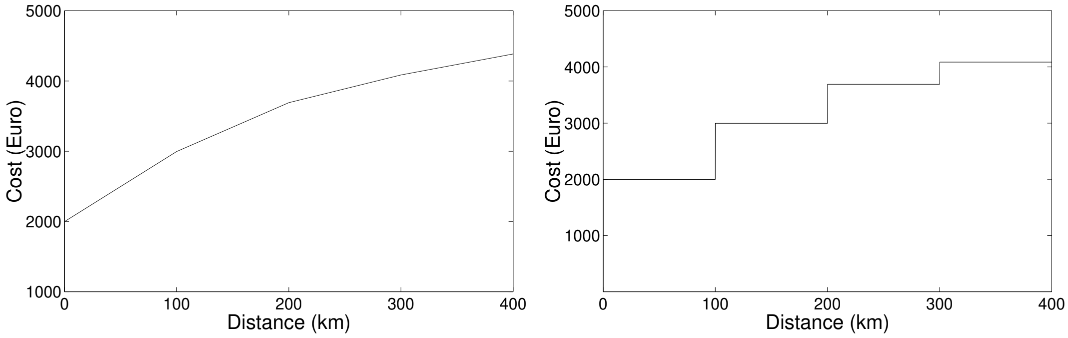

Figure 1.4: Piecewise-linear and stepped cost functions

Depending on the amount of realism desired, further "overlay" functions can be added, reflecting e.g. inter-country tariff characteristics or the user's adaptions. In either case, the cost function needs to be evaluated for all members of $Cost_c$ for which actual values cannot be determined.

Economy of scale

In 1953, IBM employee H.A. Grosch stated that computing power increases at the square of its cost [Gro53], a statement that became an accepted truth in computer installation planning for over 20 years. While it is arguable that this law still holds (see e.g. [ED85]), it was adapted to network bandwidth: the cost of bandwidth increases roughly as the square root of the capacity (see Fig. 1.5). Therefore, there is an *economy of scale* effect to be considered when buying bandwidth: the cost per bit actually decreases with growing capacities.

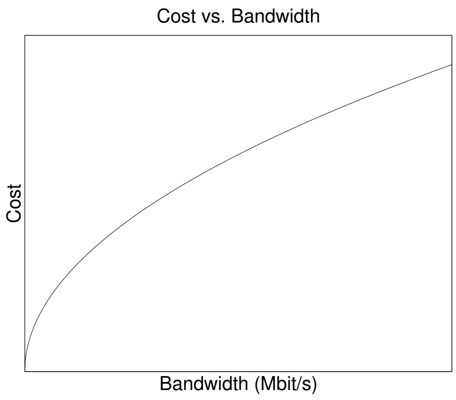

Figure 1.5: Grosch's law for network bandwidth

1.4 Topological Design

After the traffic requirements are established, the designer needs to decide on the topology of the network.

Definition 1.4: Topology

In context of network design, the topology of a network is equivalent to the set of nodes and the link layout of the network, i.e. a partition of the links into a subset E_{imp} of links to include in the network and a set E_{noimp} of edges which are to be left out of the design.

Fig. 1.6 shows the most common network topologies.

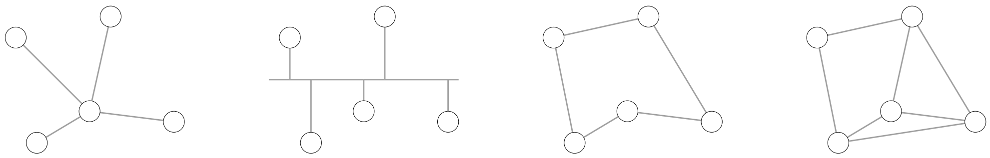

Figure 1.6: Typical network topologies: star, bus, ring and mesh.

Determining the topology will usually be achievable directly only for comparatively small networks: given a network consisting of n nodes, there are $2^{n(n-1)}$ different topologies, since a network with n nodes has $n(n-1)$ (theoretical) links that can be implemented or left out of the design. Many of those designs are not be feasible, since they represent an unconnected network. Cayley's theorem [Cay89] gives the number of possible spanning trees:

$2^{n(n-1)}$ different topologies

Theorem 1.1: Cayley's Theorem

A graph with n nodes has n^{n-2} different spanning trees.

To reduce the complexity, a *tiered* or *hierarchical* approach is commonly used. This means that only on the lowest level of the hierarchy a node actually represents a host — on higher levels, a node is the abstraction of a sub-network (see Fig. 1.7). The lowest level is commonly called the *access network* or *local access design*, the tier that builds onto this access network is called the *backbone network*. It is possible to incorporate further levels of abstraction, depending on the size of the problem. The public switched telephone network (PSTN) is an example of a structure that usually comprises more than two levels [Tan97].

Access design
Backbone design

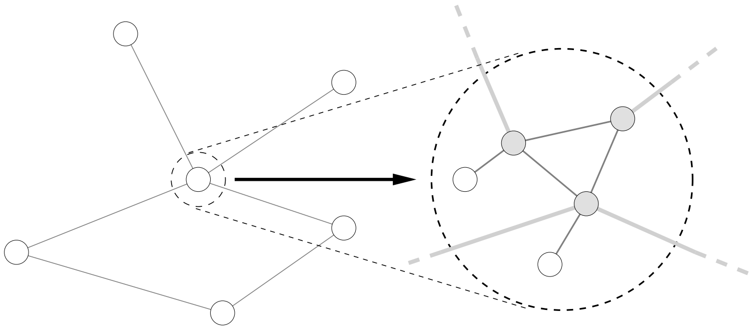

Figure 1.7: Hierarchical network structure

Fig. 1.8 shows an example of a two-tiered design. There is a great variety of algorithms available to the designer. Before describing access- and backbone design in more detail, the following paragraphs will quickly review the building blocks of many of the more sophisticated algorithms. Note that for all of these algorithms the position of the nodes is irrelevant — the outcome is solely based on the attributation of the graph's edges.

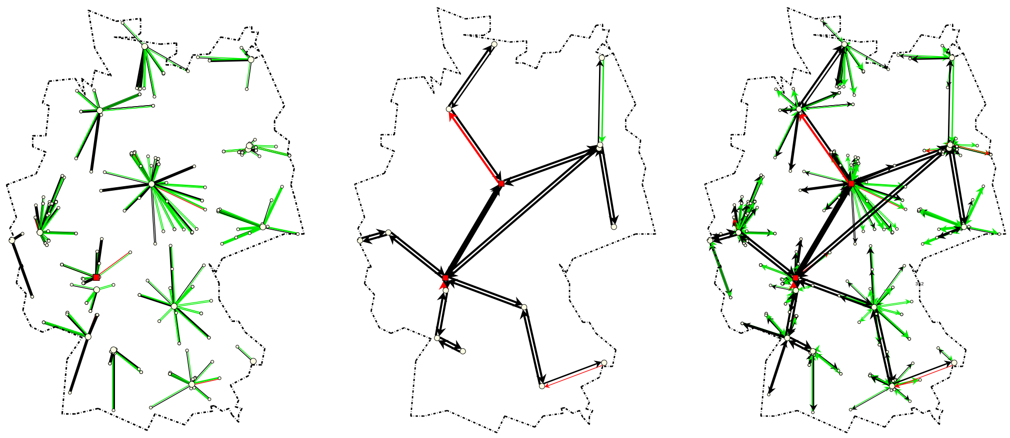

Figure 1.8: Access Design, Backbone Design, Full Design

- Shortest Paths
  Given a weighted graph and a start node, shortest path algorithms find those paths P to all other nodes for which $\sum_{e \in P} w_e$ is minimal, plus the length of these paths. If a path doesn't exist, a symbolic value for infinity is often used. In network design, the values of the cost matrix $Cost_c$ serve as the edge attributes.

Dijkstra's algorithm [Dij59] solves the shortest paths problem in $O(|E|+|V|\log|V|)$, building the shortest path tree by iteratively finding shortest paths of increasing length and using a labeling technique discussed below. The Bellman-Ford algorithm [Bel58] iteratively increases the hop count of the shortest paths found, calculating the shortest path in $O(\min(|V|, d)\cdot|E|)$ where d is the maximum number of hops in the shortest paths found.

Assuming the triangle inequality holds, shortest path algorithms produce star topologies on complete graphs, with the start node as the star's center. This has both advantages and disadvantages: while star topologies generally have low utilization and short path lengths, they are expensive, since they implement a large number of links. Furthermore, reliability is a problem, since the star's center is crucially important for the connectedness of the network.

- All-Pairs Shortest Paths
  Sometimes not only the shortest paths from a single node to all others is needed, but rather the shortest paths between all node pairs. This is called the all-pairs shortest path problem. The Floyd-Warshall algorithm [Flo62] solves the problem in $O(|V|^3)$, being faster than running Dijkstra's algorithm on each node for increasing density of the graph.

- Minimum Weight Spanning Trees
  A minimum weight spanning tree (also minimum spanning tree, MST) T is a subgraph of the network that covers all nodes, keeps it (strongly) connected and for which $\sum_{e\in T} w_e$ is minimal. There are two well-known algorithms that produce minimum spanning trees: Kruskal's algorithm [Kru56] solves the problem in $O(|E|\log|E|)$, Prim's algorithm [Pri57] in $O((|E|+|V|)\log|V|)$.

  While minimum spanning trees implement fewer links than star designs, they have a disadvantage: they are not scalable. For larger networks, two problems become dominant: the average length of the paths increases, since there is only one path between two nodes in a tree. This leads to a dramatic increase in the amount of traffic carried though the network, and consequentially multiple links between node pairs or larger capacity links. The other fundamental problem of the tree design is, again, missing redundancy. A single link failure renders the network unconnected. The above said applies to ring topologies as well.

- Prim-Dijkstra Trees
  With both shortest path threes and minimum spanning trees having advantages and disadvantages, it seems to be a good idea to find a compromise between the two: these are called Prim-Dijkstra trees [KKG91].

  During calculation, both algorithms distinguish between visited and unvisited nodes and update a distance matrix $Dist$. The neighbourhood of the currently visited node is inspected and depending on the algorithm, different criteria are used for selecting a node and labelling it. With s being the starting node of Dijkstra's algorithm, c being the current node in the iteration and n a neighbouring node to c, Prim's algorithm uses

  $$Crit_{Prim} = Dist(c, n)$$

  whereas Dijkstra's algorithm uses

  $$Crit_{Dijkstra} = Dist(s, c) + Dist(c, n)$$

Thus, by introducing a balance factor α within range $0 \leq \alpha \leq 1$ it is possible to balance between shortest paths and minimum spanning trees:

$$Crit_{Prim-Dijkstra} = \alpha \cdot Dist(s, c) + Dist(c, n)$$

$\alpha = 1$ produces a shortest path tree, $\alpha = 0$ yields a minimum spanning tree.

There is a plethora of design algorithms that are based on those basic algorithms, however only a few of them can be explained in detail in the scope of this thesis (cf. Chapter 2). For a more thorough introduction, the reader is referred to [Ker93], [KKG91] or [Cah98]. Appendix A of [Fis00] gives an overview of the various classes of design algorithms.

[MS81] shows that already subproblems of the general network design problem are strongly NP-complete. Therefore one needs to be aware of the following:

> **Assuming $\mathcal{P} \neq \mathcal{NP}$, it is not possible for most network design problems to find an optimal solution efficiently (i.e. in polynomial time) or even efficiently approximate it using pseudo-polynomial algorithms.**

The following is an overview of access and backbone design principles.

- Access Network Design
  The access network design (also referred to as local-access design) needs to solve three problems:

  1. Finding a clustering
     Since it is desirable to route traffic from small nodes via more important ones but not vice versa, it is a good idea to group small nodes together and detour their traffic via one or more nodes per cluster to the upper tier. Therefore, the first step of access network design addresses the problem of grouping the nodes into suitable clusters, each of which will represent a virtual node in the upper layer of the design.

     One way of doing this is using *threshold clustering*. Those nodes whose weight exceeds a certain threshold become backbone nodes, while the others are grouped with their closest backbone neighbour.

     This method does not work well if the relative differences between the node weights is small, since this leads to either too many or too few of the nodes passing the test. Using other clustering algorithms, this problem can be avoided. The *k-means* algorithm builds k clusters by calculating natural regions of large weight among the nodes such that the clusters are evenly distributed.
  2. Finding the cluster center
     The clusters need a designated *proxy*, *concentrator* or *center* node that will accept/deliver the traffic to/from the upper layer. There need not necessarily be a single proxy node, multiple proxies can also be used (as shown in Fig. 1.7). Depending on the number of proxies the design is also called *single-centered* or *multi-centered*. In Fig. 1.7 the proxy nodes are shaded.

     Oftentimes, the center will implicitly be defined by the clustering procedure. Again, a simple technique for determining the cluster's center is looking at the node weights (recall Definition 1.3). The node with the largest weight is a good candidate for the cluster center, since it needs to handle the most traffic.

3. Finding a cluster topology
   The clusters still need a topology. It depends on the nature of the network whether more time should be invested in access or backbone topologies. When the focus is on the backbone network, a simple topology (e.g. a star design) may be sufficient. More sophisticated results can be produced using the heuristic Esau-Williams algorithm ([EW66], [Cah98]), which builds trees starting from the cluster center while being aware of the problem of accumulating traffic towards the center of the tree. The algorithm can be extended to deal with multiple available capacities, producing larger links towards the center.

- Backbone Network Design
  The backbone design needs to build a connected network from the clusters. In contrast to many access networks, backbone designs typically are mesh networks rather than tree-shaped[2].

  In order to be able to use the cluster centers produced in the access network design, again useful traffic and cost matrices are needed. While traffic originating and terminating in a cluster remains unaffected, inter-cluster traffic needs to be detoured via the cluster center. Therefore, the following calculation needs to be performed for all cluster centers:

  1. Adjusting the outgoing traffic
     For all nodes connected to the cluster center, add the produced traffic to that of the cluster center.
  2. Adjusting the incoming traffic
     For all links between cluster centers (i.e. the backbone links), set the traffic requirement of the link to that of the cluster nodes as previously allocated.

  One of the most commonly used algorithms for backbone topology design is the *MENTOR* (*ME*sh *N*etwork *T*opology and *R*outing) algorithm [KKG91] and its derivatives, which basically build a Prim-Dijkstra tree on the backbone and then iterate "outside-in" over node pairs in the tree, deciding whether aggregated traffic volumes satisfy adding direct links between them. Therefore, MENTOR designs are mostly based on trees, with additional direct links. The MENTOR algorithm is fast, all of its basic loops run in $O(|V|^2)$, making it fast enough to be used in inner loops of more sophisticated algorithms. MENTOR is also quite rich in parameterization, which makes a certain degree of experimentation necessary. *MENTOR*

  One problem of the algorithm is that an implementation of a MENTOR-designed network has no way to understand MENTOR's reasoning in adding direct links. With an incompatible routing scheme (see next section), the actual performance of the network can get quite bad.

It is important to note that access design and backbone design mutually influence each other, so that an iterative design process is desirable. Furthermore, it depends on the characteristics of the network which of the two steps is more important in the case at hand.

[2]There are exceptions — for instance the Spanish National Research Network has a star backbone topology [CSI].

1.5 Routing and Dimensioning

The network now has a defined topology, however, the network has not been dimensioned yet, i.e. the actual link capacities are still undetermined. The paths along which traffic moves through the network obviously have direct impact on the utilization and capacities needed for the links in the network. Moreover, the importance of choosing paths wisely is critical in the meshed backbone, whereas in a tree-shaped access network with a single path between each pair of nodes the problem is trivial.

The following will briefly overview the various routing schemes before the influence of the routing scheme on network topology design and link dimensioning are discussed.

Definition 1.5: Routing

Routing is the process of determining the paths along which traffic flows in a network. The two predominant problems routing addresses are finding a suitable path for the traffic and providing a means to guide traffic through the network along this path.

Router, end node

The routing problem has fostered significant amounts of research over the last decades, which have led to a variety of different routing protocols. The hosts of a network that are concerned with forwarding traffic from source to destination hosts are called *routers*[3] and are connected to at least two separate networks, in contrast to the *end nodes* in a network which only produce or consume traffic. Through a routing protocol, a router maintains information on the current reachability status of other hosts in the network and the correct outgoing link on which to put packets on their way to the destination.

Routing protocols can be classified by a number of characteristics [RK87], including but not limited to:

- centralized/distributed
  In centralized routing schemes, the decision on how to route traffic is made in one central location, whereas in distributed routing all forwarding nodes decide autonomously.
- static/dynamic
  Static routing uses fixed routing tables that define how to forward traffic. Dynamic routing can adapt to congestion or equipment failure by automatically redirecting traffic.
- local/global
  Local routing algorithms can decide on routes directly, whereas global algorithms need to be forwarded information from other nodes to perform their work.

Many different routing strategies were conceived and implemented, such as [Nar89]:

[3]The routing terminology is confusing. In the earlier days of internetworking, routers were called *gateways*. To the knowledge of the author, the term *gateway* is nowadays used rather when theoretical properties of networks are discussed, whereas *router* is more often used in the marketplace, referring to the actual machine. In context of protocol stacks, it is also common to further differentiate by the protocol layer on which the router operates.

- Network flooding
  This is one of the simplest and oldest approaches to routing. Instead of worrying about how to calculate the best paths, the sending node simply transmits the data to all neighbouring nodes, these in turn send it to each of their neighbours etc. Some care had to be taken to keep the large amount of traffic under control, such as *backward learning* by using sequence numbers and packet aging information [BG92].

- Static source routing
  Another way that is easy to administrate for small networks is static source routing. The routing paths are *statically* set up by the network administrator, and each data packet upon being sent contains the full route in its header. Each router on the way to the destination extracts the necessary data, reducing the size of the routing information in the header. This is called *source routing.*

  Default route

  A common technique is the concept of a *default route.* Using default routes (sometimes also called *wildcard routes*), configured paths are used for routing when available — when not, a default router is used as the next hop, with the idea that this router has better knowledge of how to handle the packets.

- Distance-vector routing
  Bellman-Ford

  A dynamic approach is distributed *distance-vector* routing: when a router boots up, its horizon is limited to the networks it is directly attached to. The router periodically communicates its knowledge to the neighbouring routers and also keeps forwarding information it receives. The basic algorithm for doing this is similar to Bellman-Ford's: traffic is routed via the link for which the link metric plus the adjacent node's metric to the destination is minimal. Thus, with more incoming updates, the router increases its knowledge hop-by-hop and sends entire routing tables through the network.

  Counting to infinity

  One of the main weaknesses of distance-vector routing is the fact that since a router communicates topology updates only to adjacent routers, the network takes relatively long to converge to the up-to-date status. During the diffusion of update information through the network it is prone to the formation of routing loops because of the *counting to infinity* problem. This problem occurs when a topological change has taken place (e.g. a link failure) and the update information has not yet fully traversed the network while routers keep increasing the distance metric until the limit imposed by the implementation is reached (see [Hed88] for an example).

  Triggered updates

  Holddown rule

  Split horizon rule

  Route poisoning

  There are a number of countermeasures to prevent the counting to infinity phenomenon: *triggered updates* are routing tables that are sent as soon as a topological change occurs and that are forwarded by other routers immediately. The *holddown rule* says that when a topological change occurred, the new route that is established afterwards has to be kept up for a certain amount of time, thus giving the update information more time to propagate through the network. The *split horizon* rule says that there is no need to tell a router distance metrics about his adjacent networks, effectively preventing routing loops between adjacent routers. *Route poisoning* interprets large jumps in a path's distance metric as an indicator for the existence of a routing loop and prohibits the path's use. See again [Hed88] for more detailed explanations.

- Link-state routing
  Link-state protocols distribute the status of the topology throughout the network by

periodically flooding the network with information (e.g. the average packet delay) about adjacent links. Other routers in the network pick up the information and update their image of the network topology. Furthermore, the routers also keep recalculating the shortest paths to the nodes in the network and send packets along these paths.

A problem with link-state routing in large networks is the significant cost of broadcasting link-state updates throughout the network. Therefore, link-state schemes do not scale well. However, due to the fact that routers receive topology updates directly from the source of the update, they can adjust quickly to topological changes.

- Hierarchical routing
  One problem of the techniques described above is the amount of traffic that is transported throughout the network. An approach called *hierarchical routing* tries to reduce this amount by limiting the area that an information needs to cover. The terminology depends on the protocol; examples are *autonomous systems*, *areas*, *domains* or *zones*. The ARPANET is an example of a network that used hierarchical routing: within each zone, an *interior gateway protocol (IGP)* was used, effectively limiting the scope of broadcasts to the hosts within the zone. Inter-zone traffic was handled by an *exterior gateway protocol (EGP)*, that was able to translate the distance metrics available from within one zone into the metrics of other zones.

IGP

EGP

There is a wide variety of existing protocols in use in enterprise networks today, using various combinations of the approaches described above. Some of these are the following:

- HELLO
  HELLO ([Mil83]) was the routing protocol of NSFnet. It was a dynamic distance-vector protocol, using packet delay as distance metric. To avoid counting to infinity (where infinity means a delay of 30 seconds), HELLO implemented a holddown timer of two minutes.

- Routing Information Protocol (RIP)
  The RIP protocol ([Hed88]) appeared as part of the 4.x BSD UNIX distribution's routed daemon and quickly became a *de facto* standard. It was designed to be used in LAN environments, running the IP protocol. It is a distance-vector protocol in which each router periodically sends updates.

  While it served and still serves as a widely used IGP, it has a number of drawbacks: firstly, RIP uses hop-counts as distance metrics, ignoring other metrics such as traffic congestion. RIP limits the hop count to 15, making it unusable in large networks. Also, RIP doesn't know the concept of splitting host addresses for subnetting, and typically takes a long time (several minutes) to converge after a topological change occurs.

- Interior Gateway Routing Protocol (IGRP)
  IGRP ([Cis91]) is a distance-vector IGP patented by Cisco Inc. and conceptually similar to RIP, however, it is intended for larger networks. The main differences to RIP are

  - More sophisticated distance metric
    Instead of having a single distance metric, IGRP takes into account the delay time on an unloaded network, the minimum bandwidth on a path, traffic congestion and path reliability.

- Support for different traffic classes
  IGRP supports multiple traffic classes. For example, the distance metric calculation can be based on traffic classes with different needs, such as small delay or large bandwidth.
- Support for traffic splitting
  IGRP can handle traffic splits, i.e. traffic is not dedicated completely to a single path, rather, traffic is split up proportionally on those paths for which the distance metric falls into a certain range
- Counting to infinity countermeasures
  IGRP implements holddowns, triggered updates, a split horizon rule and route poisoning to prevent counting to infinity.

- OSI Routing
  The Open Systems Interconnection (OSI[4]) standard issued by the International Organization for Standardization (ISO) contains the description of a full suite of routing protocols, both IGPs and EGPs [Cisb]. OSI routing uses specific terminology: hosts are called *end systems* (ESs), routers are *intermediate systems* (ISs).

  Routing is based on a three-level hierarchy; an administrator groups systems into *areas*, areas are grouped into *domains*. Within areas, the *End System-to-Intermediate System* (ES-IS) protocol is used, inter-area traffic is contained in a domain and handled by the *Intermediate System-to-Intermediate System* (IS-IS) protocol. The protocol that routes between domains is called *Interdomain Routing Protocol* (IDRP). *ES-IS, IS-IS, IDRP*

  In ES-IS, routers inform hosts about their presence sending *hello packets*. A host can send data to another host by first contacting a router in its area. The router can decide whether the destination is within the same area by listening to hello packets sent from the ESs. Depending on the location, the data is forwarded to an IS en route to the destination area, or directly to the destination host in the same area, with a possible redirect message that informs the hosts that they can communicate directly. IS-IS is a link-state protocol, flooding the network with information about the ISs and and ESs it can reach, with four possible distance metrics: administrator-assigned, delay, link-use cost and link error rates. Finally, IDRP routing is a distance-vector EGP, featuring loop suppression by checking the domains contained in a route, compression of path information, cryptographic signatures and more.

- Open Shortest Path First (OSPF) protocol
  OSPF [Moy89][Moy98] is a distributed, dynamic, hierarchical link-state IGP used on IP networks and intended as replacement for RIP. It has several advantages over its ancestor: OSPF converges much faster after topological changes and it can assign a cost metric individually to each IP service, based on delay, throughput, round-trip time and other measures.

  OSPF uses dimensionless distance metrics assigned by the network administrator. Based on the value of the distance metric on each link, the routers periodically update their routing tables by calculating a single-source shortest-path algorithm to the other nodes in the network. When multiple routes of equal best cost exist, traffic is distributed evenly among them. Each router only uses the first hop in the

[4]The author intuitively tried `http://www.osi.org` to obtain information about OSI. While this proved to be misleading, a visit is highly recommended for the reader inclined to humour.

forwarding process. Routing updates are authenticated, thus only trusted routers can participate in the routing process.

Update messages are sent periodically only every 30 minutes, however, link failures or new links are reported immediately. Still, large networks cause the amount of data caused by updates to be a concern. To reduce traffic volume, OSPF uses a two-level hierarchy. Administrators can group systems into *areas*, which do not report topological changes occurring within them to the outside world. The backbone connecting the areas is handled as just another area.

For a practical introduction to OSPF see [Cisa].

The routing protocol greatly influences the network topology and link dimensioning. Before decisions can be made about the capacity of the links in the implementation, the designer needs to decide on a routing model. Using this model, first the traffic routes need to be established, then the links in the network need to be loaded with the required volumes of traffic. The technical details of the routing protocol (e.g. whether it is a link-state or distance-vector protocol) are of less concern here, interesting point are e.g. whether traffic streams can be split, as when using IGRP.

Traffic load matrix

In order to record the traffic loads on the network links, an additional matrix *Load* is introduced that contains for each link $(i \triangleright j)$ the currently assigned utilization. Algorithmically, the following needs to be done:

Algorithm 1.1: Loading the Network

```
initialize Load matrix;

for all node pairs (n1, n2)
  {
     p = path from n1 to n2 according to routing scheme;

     for all links in path (p)
       Load(n1, n2) += Req(n1, n2);
  }
```

The above algorithm is incomplete in that it cannot account for traffic splitting. If the routing model allows traffic splitting, the requirement taken from *Req* needs to be partitioned and put on the different paths accordingly, using an procedure similar to algorithm 1.1. *Load* now contains each link's projected load, based on the requirements matrix *Req* and the routing scheme.

Network dimensioning

The next step is *network dimensioning*, i.e. determining the capacity that each link will be given. This decision is based on the utilization assumption made (what is the maximum and/or minimum average utilization allowed for the links) and the available equipment options. If the load on a link exceeds the maximum available capacity, the designer can either reject the design or implement parallel links. The latter case a ratio must be found for the traffic amounts to put on the parallel links. One can e.g. load the first link to the maximum allowed utilization and install an additional, “thin” link with the overflowing traffic, or split the traffic evenly among two links of smaller capacity etc.

Algorithm 1.2: Dimensioning the Network

```
for all node pairs (n1, n2)
  {
     p = path from n1 to n2 according to routing scheme;

     for all links l in path (p)
       set capacity on link l to best available capacity for Load(l)
  }
```

1.6 Cost and Performance Evaluation

Finally the network can be evaluated by summing up the cost of the equipment installed. Depending on the network model, nodes also need evaluation. Relevant criteria in this case are for example the number of ports a switch offers (the number of lines connecting to a node can be large, e.g. in access networks with star topologies, see Fig. 1.8) or its queueing delay. Note that when many designs are examined, the computational effort needed to evaluate the solutions can become significant.

When considering the cost of equipment, it is important to be aware of the timespan the planning process covers (the *observation period*) — when planning for a given amount of time, setup costs, reoccuring costs, upgrade costs and others need to be considered. See appendix C in [Fis00] for details. *Observation period*

Depending on whether the quality of the solution has reached a satisfying range or the number of iterations exceeded a certain limit, the designer (resp. the algorithm) may choose to examine other designs or stop.

1.7 General Design Rules and Principles

During this chapter a number of fundamental design rules were mentioned. This section will restate them and introduce a number of new ones.

- Well-utilized links
  Links with low utilization waste money. Therefore it is desirable to have well-utilized links, taking into account a certain amount of slack capacity in order to be able to cope with peak traffic hours. This especially applies to packet-switches networks, where the upper limit for average utilization is commonly set within 50-70%.

- Economies of scale
  The cost per bit for lines grows roughly by the square root of the capacity (Grosch's law).

- Access size versus backbone size
  It is important to be aware of the relative size of the access and backbone networks, because that relation determines how much effort to spend on the design of both tiers.

- Shortest path designs are expensive
  Shortest path designs implement a lot of links and can thus yield very expensive networks.

- Minimum spanning trees do not scale, nor do rings
  They both produce a large average length of paths, thus increasing the amount of traffic carried through the network and necessitating parallel links or larger link capacities. They also do not provide redundancy.

- Traffic from small nodes is to be routed via large ones, not vice versa
  Routing large amount of traffic via small nodes will usually produce longer paths than traffic flowing rather directly between large nodes and more detoured between small ones.

- No green field
  Network design usually is an incremental process, i.e. that an existing infrastructure needs to be taken into account. In this context, *evolutionary* network planning [Fis00] becomes important.

- Iterativeness of the process
  Network design is subject to an overwhelming number of influences. It will usually not be possible to take all of them into account in a single design cycle, since most design steps provide feedback and improved input to previous steps. Therefore, an iterative approach is necessary.

2

Meta-Heuristic Approaches to Network Design

In the following sections the concept of heuristic search is introduced and some of the most commonly used optimization techniques are explained. First, some terminology needs to be introduced.

2.1 Concepts and Definitions

Definition 2.1: Heuristic

A heuristic search or optimization technique refers to a methodology that generally produces good-quality solutions, however, the quality of the solutions cannot be guaranteed or proved mathematically.

Heuristic algorithms are usually the only efficient means of producing feasible solutions to NP-complete problems [Hoc95].

Definition 2.2: Meta-heuristic

A meta-heuristic optimization technique is a strategy that supervises and controls other heuristic techniques, oftentimes local improvement heuristics. Meta-heuristics are therefore not problem-specific in nature and applicable to a wide variety of problems.

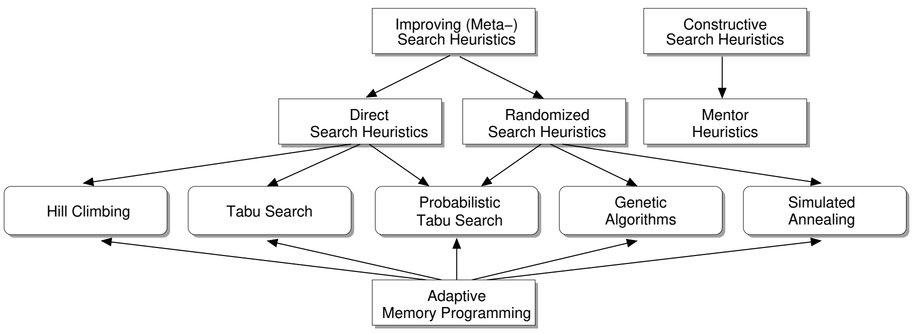

Figure 2.1: A coarse classification of some heuristic optimization techniques.

Improving & constructive heuristics

Fig. 2.1 (adapted from [Fis00]) shows a taxonomy of heuristic search algorithms[1]. There are two general classes: *improving* heuristics which build a solution (or a set thereof) using a certain strategy and iteratively try to improve on the result using the previously generated solutions; each of the temporary solutions is a valid solution itself. Some examples that belong to this category will be described below. In contrast, *constructive* heuristics produce the solution according to a building plan that usually comprises multiple steps, each of which contributes to the solution. However, only the final result is a valid solution. The MENTOR algorithm introduced in section 1.4 is an example of this concept.

Although each heuristic works in its own individual way, many improving heuristics share the common principle of memorizing characteristics of encountered solutions, trying to gain information on where to further direct the search. This class is called *adaptive memory programming* heuristics.

The network design problem is an instance of *discrete optimization problems*.

Definition 2.3: Discrete Optimization Problem, Objective Function

In an instance of a discrete optimization problem, each element of a discrete set S of items is assigned a numerical value, obtained using an objective function. The goal of the optimization process is to find the element(s) of S for which the objective function reaches a certain value, which needs to be defined as *optimal* in a way that is suitable for the given problem setting.

More formally, a discrete optimization problem can be defined as a triple $(S, \mathcal{O}, c)$, where S is the set of items, $\mathcal{O}$ is the objective function $\mathcal{O} : S \mapsto \mathbb{R}$ that maps each element of S to a real number, and a constraint function $c : \mathcal{O}(S) \mapsto \mathbb{B}$ that defines for all resulting values of $\mathcal{O}$ whether a certain optimality condition is met.

For a network design problem, S is the set of potentially implemented networks, evaluated with the optimality criterion being the goal of minimizing the objective function. S is also called the *search space*, those networks for which the optimality criterion is satisfied are also called *feasible solutions*:

Definition 2.4: Search Space, Solution

In a discrete optimization problem $(S, \mathcal{O}, c)$ the set S is also referred to as the search space, the members of S are solutions; those solutions whose objective function value satisfies c are called feasible solutions. The set of feasible solutions may be empty if the optimality constraint cannot be met.

Given this model, the goal is to discover an algorithmic means of traversing the search space that will ideally yield the optimal solution in the shortest amount of time possible. When traversing the search space, it usually makes sense to look at close neighbours of the solution at hand, i.e. to examine the *neighbourhood* of the solution (adapted from [Fis00]):

[1] The taxonomy is not complete, rather, only the techniques relevant to this thesis are categorized.

Definition 2.5: Neighbourhood, Transition

The neighbourhood $\mathcal{N}$ of an item i is defined by a mapping $\mathcal{N} : S \mapsto 2^S$ that assigns to each element in S a subset of S, assuming that

- $\forall s \in S : s \notin \mathcal{N}(s)$ (a solution cannot be part of its neighbourhood)
- $\forall s \in S : |\mathcal{N}(s)| \geq 1$ (each solution has a neighbourhood of at least one solution)

After investigating a solution s, an optimization process usually takes into account $\mathcal{N}(s)$, trying to substitute s with a "better" solution $s' \in \mathcal{N}(s)$. This substitution is called a transition to another solution.

For the success of the search process it is important to use a neighbourhood structure that is meaningful, i.e. the neighbourhood of a solution s should be closely related in some sense to s. Also, the performance of a solver can crucially depend on the computational effort necessary to pick a neighbouring solution s' from $\mathcal{N}(s)$, since $\mathcal{N}(s)$ can be quite large. Closely related to the concept of a neighbourhood of a solution is the *search space graph* $g_\mathcal{S}$:

Definition 2.6: Search Space Graph

The search space graph of a discrete optimization problem is the directed graph $g_\mathcal{S}(V_\mathcal{S}, E_\mathcal{S})$ whose nodes result from a bidirectional mapping $\mathcal{V} : S \leftrightarrow V_\mathcal{S}$ that associates each solution in the search space s with a node n_s in the search space graph, and that contains edges that are constructed as follows:

$$\forall s \in S : s' \in \mathcal{N}(s) \rightarrow (n_s \triangleright n_{s'}) \in E_\mathcal{S}$$

That is, the edges in the search space graph correspond to the neighbourhood structure.

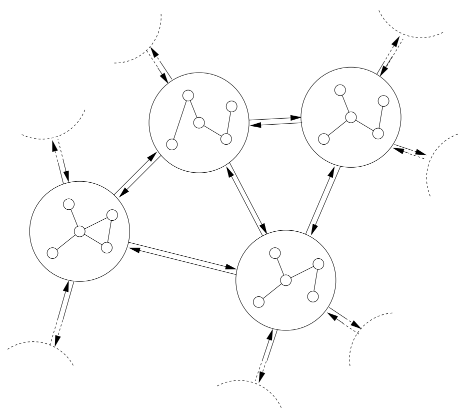

Figure 2.2: Part of a search space graph for a network design problem.

A transition as given in Def. 2.5 can be thought of as a token that marks a node in the search space graph and moves through the graph. In the network design problem, each node in V_S represents a network topology, see Fig. 2.2. Meaningfulness of the neighbourhood here usually means that the neighbouring solutions only contain small topological differences.

Attributing each node in the search space graph with the value of the objective function of the associated item in the search space yields a search space *landscape*, as shown in Fig. 2.3.

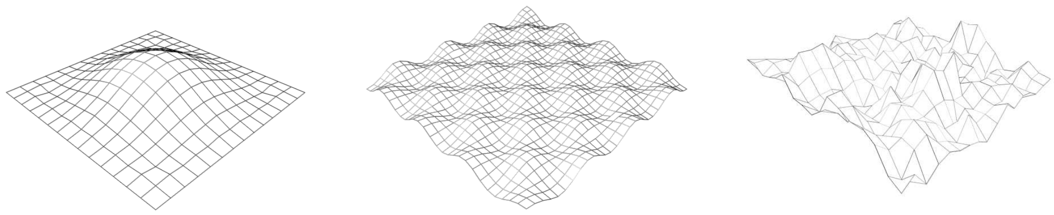

Figure 2.3: An easy search space for maximizing an objective function (left), one with many local optima (middle), and a rugged one (right). The grids are a simplification, assuming that all nodes have only 2-4 neighbours.

Solution encoding

Most algorithms do not work on the solutions directly, rather, they use an *encoding* or *representation* of the solution that better suits their needs. For example, algorithms will usually not work on network topologies directly, instead they encode the relevant attributes of those topologies.

Definition 2.7: Encoding, Decoding

An encoding of an element s of the search space S is a mapping $S \mapsto S^*$ to an alternative, semantically equivalent representation s^* of the element in the representation space S^*. The decoding is the reverse mapping $S^* \mapsto S$ that restores the solution from the encoding.

It is desirable that the encoding/decoding mapping be bijective. An injective mapping $S \mapsto S^*$ produces a single solution with multiple encodings, which, although not necessarily incorrect, will result in a loss of detail and/or performance in the calculation. Using a surjective mapping $S \mapsto S^*$, multiple solutions would have the same encoding, which likely is incorrect. It is important to notice that the performance of algorithms depends critically on the meaningfulness of the encoding and the computational efforts necessary to do the en-/decoding.

Some of the most frequently used meta-heuristics are explained below.

2.2 Hill Climbing

Hill climbing is one of the simplest techniques of optimization. From a given starting solution s_0, the procedure iteratively examines the entire neighbourhood $\mathcal{N}(s)$ of the current solution s and makes a transition to the best neighbour found, until no further improvement is possible.

The algorithm then has found a local optimum[2], which is not necessarily the global optimum.

The main disadvantage of hill climbing is its limited horizon. From the starting solution, it moves straight to the closest local optimum and has no way to escape it afterwards. This is why it is also called a *greedy algorithm* or *steepest descent method*. Referring to Fig. 2.3, the algorithm will always find the optimal solution in the search space depicted on the left, while the probability that it will reach the global optimum in the right search space is small. A possible approach to this problem is running multiple iterations of the algorithm from multiple starting solutions.

Greedy algorithm

Algorithm 2.1: Hill Climbing

```
best_objective = -INFINITY;
s = some starting solution;

while (objective(s) > best_objective)
  {
     N_s = the neighbourhood of s;

     for all solutions s_temp in N_s
       {
          if (objective(s_temp) > objective(s))
            {
               s = s_temp;
            }
       }

     best_objective = objective(s);
  }

return s;
```

2.3 Tabu Search

Tabu Search (TS) [Glo86][Glo97], like other heuristic algorithms, is imitating a concept found in nature: it tries to mimic the way in which the brain memorizes events. It makes use of memory in two basic ways, using *adaptive* and *responsive* memory. Adaptiveness refers to the ability of memorizing key events during the search (such as good solutions), thus guiding the direction of local search. Responsiveness, in contrast, means the ability of making *strategic* decisions on how to proceed (i.e. accepting bad solutions on purpose when they might lead to overall better solutions).

Adaptive & responsive memory

There are four primary metrics that Tabu Search uses, namely *recency*, *frequency*, *quality* and *influence*. Recency is quantified in *short term* memory and captures when a certain solution was last encountered. Tabu Search uses a *tabu tenure* which defines the duration (in terms of number of transitions) for which a solution is tabu, i.e., cannot be visited any more. Such a solution is said to be *tabu-active*. Oftentimes a special override criterion (the *aspiration criterion*) is used under specific conditions. In contrast to recency, frequency is used in *long term* memory, counting the number of times that solutions were encountered. The quality attribute is used to store particularly good solutions in a *candidate list*. Influence can be seen as a special type of quality aspect and captures structural changes in

Recency, frequency, quality, influence

[2] It does not matter whether the goal is to find the overall maximum or minimum of the search space, since they can easily be converted into each other. In the following it will be assumed that the goal is to maximize the objective function.

the search path rather than qualitative ones. As an example, influence could be used to analyze the average quality encountered after a certain solution.

Tabu Search also differentiates between the nature of the information it remembers: *explicit* memory stores actual solutions, whereas *attributive* memory concentrates on how attributes of the solutions change over the course of the search. In context of the network topology design problem, every move modifies edges (remember Fig. 2.2). Which edges actually get changed during a transition is stored in attributive memory.

Tabu Search tries to benefit from those techniques in order to overcome the limitations of pure local search as described above. In particular, it uses explicit & attributive memory and candidate lists for *intensification*[3] of the search, while also encouraging the visit of solutions that differ significantly from solutions encountered so far, promoting *diversification*.

Because of the variety in available metrics, Tabu Search implementations can vary greatly in their degree of sophistication. However [Glo97] mentions that even the simpler ones using e.g. only short-term memory can reach good quality solutions. Tabu Search qualifies as a meta-heuristic, since it uses high-level techniques (the various memory schemes) to guide a basic heuristic, the local search itself.

Algorithm 2.2: Tabu Search

```
s = some starting solution;
m_s = short term memory structure;
m_l = long term memory structure;
c = empty candidate list;

while (stopping criterium == false)
  {
     N_s = the neighbourhood of s, based on m_s and m_l;
     s   = move to the next solution based on adaptive and responsive strategy;

     update m_s and m_l;
     insert s into c if it qualifies;
  }

return s;
```

[GLR91] describes an application of Tabu Search to a network topology design problem: Tabu Search is used in a two-level manner, first in order to find locations for service centers, then in order to find access designs to those servers.

2.4 Simulated Annealing

Simulated Annealing (SA) [KJV83] mimics the behaviour of atoms in heated solids when cooling them down. From an initial point in the search space, random jumps are made with an initially large operating range (like atoms in the hot solid), which is gradually reduced to smaller jumps (like atoms in cooler solids) using a *cooling scheme*. While improvements to the current solution are accepted immediately, worse solutions are accepted with probability $p_{accept} = e^{-\Delta C/T}$ where ΔC is the (positive) difference between the result

[3] Attributive memory can be used for intensification e.g. by making certain solution attributes a precondition.

of the objective function for the potential solution and that of the current one, and T is the temperature. The intention of Simulated Annealing is to use a cooling scheme that prevents the search from getting stuck in local optima.

A number of extensions can be integrated with the algorithm, such as re-heating the solid, thus again increasing the acceptance radius. As the temperature approaches zero, the algorithm behaves more like hill climbing.

Possible stopping criteria for the algorithm are e.g. temperature thresholds, number of iterations or number of iterations without improvements. A simple cooling scheme would be a *linear cooldown* that reduces the temperature by a constant amount ΔT: $T_{t+1} = T_t - \Delta T$ in each iteration. Another option is *geometric cooldown*: $T_{t+1} = \alpha \cdot T_t$ for some $0 < \alpha < 1$.

Algorithm 2.3: Simulated Annealing

```
s = some starting solution;
t = initial temperature;

while (stopping criterium == false)
  {
     s'    = a random element in N(s);
     rand  = random number in [0;1];

     if (objective(s') < objective(s))
       {
          s = s';
       }
     else if (exp((objective(s) - objective(s')) / t) > rand)
       {
          s = s';
       }

     t = modification of t according to cooldown scheme;
  }

return s;
```

Another influence is the general cooldown strategy, i.e. whether cooldown is monotonous, or whether the temperature can increase as well, in order to give the algorithm enough time to escape bad local optima.

2.5 Genetic Algorithms

Genetic Algorithms (GAs)[GC97] imitate the process of evolution and Darwin's principle of survival of the fittest, as they are found in nature. Instead of working with the solutions directly, the algorithm encodes them in a fashion similar to *chromosomes*, i.e. the solution is represented using a string of scalar information[4] (see Fig 2.4).

In contrast to the algorithms described so far, it uses a *population*-based approach: Genetic Algorithms do not investigate a single solution at a time, they keep a pool of solutions. In each iteration of the algorithm, the population gets modified, producing different *generations*. *Population* *Generations*

Modification of generations happens by means of *crossover-*, *mutation-* and *selection* operators. A crossover operation consists of combining two strings by splitting them at identical

[4] Just as the human chromosomes and their genes are a representation of the human being itself.

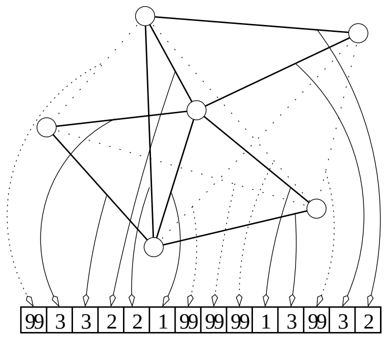

Figure 2.4: Encoding of an OSPF network, using OSPF lengths as genes (adapted from [Fis00]).

points and recombining the swapped substrings. Mutation refers to randomly modifying single elements of the string. The selection process controls how parents are selected and how offspring replaces the ancestors.

Using an initially random population and by triggering the influences of intensification (via crossovers) and diversification (using mutation), Genetic Algorithms can effectively avoid getting stuck in local optima and explore wide areas in the search space. Genetic Algorithms are also well-suited to incorporate problem-specific local search heuristics, making it a versatile meta-heuristic.

Algorithm 2.4: Genetic Algorithm

```
P = population of n solutions (individuals)

initialize P with random individuals;

while (stopping criterium == false)
  {
     select parents (p1, p2) from P according to reproductive scheme;
     (o1, o2) = create offspring resulting from crossover operator on (p1, p2);

     mutate o1 and o2 according to mutation scheme;

     replace individuals with worst fitness value in P with offspring;
  }

return best individual;
```

[ES96] illustrates an application of Genetic Algorithms to LAN/MAN topology design, using different encodings for clustering and spanning tree variations.

2.6 Other Techniques

There are many other heuristic algorithms that can be applied to network design problems, such as Ant Colony Optimization (ACO) [CDG99], which is another domain that mimics nature by simulating the way ants agree on using the shortest path between two points after a short time, using pheromones. Other examples include Artificial Neural Networks (ANNs) which model the way neurons work in the central nervous system [Ree93], or

Immune System Methods which mimic the way the human immune systems learns to defend itself against virii and bacteria [CDG99].

It should be mentioned that the methodologies explained in this chapter are by no means fixed and only applicable selectively. As Fig. 2.1 illustrated, they belong to the class of adaptive memory programming techniques and thus share basic principles. For instance, [Fox92] describes a heuristic algorithm based on Simulated Annealing with added concepts of short-term memory to prevent cycling in the search space, taken from Tabu Search, and a population-based approach to foster diversification, inspired by Genetic Algorithms.

3

Analyzing the Search Space of a Network Design Problem

This chapter introduces the network model used in this thesis, illustrates the difficulty of finding the overall optimal solution to the network design problem and presents the methodology used.

3.1 The Network Model

In this section, the network model that will serve as the basis for analyzing the solution space is defined. Some of the assumptions were already stated in Sec. 1.1.

Definition 3.1: Network Model

- The network is represented as an attributed graph $g(V, E)$.
  Network nodes are represented as nodes in a graph, and are attributed with geographical coordinates in order to be able to determine distances between nodes.

  A network link is represented as edge e in a graph g, and attributed with an integer i_e^g that represents an OSPF length as a vector index (see below). Furthermore, the network dimensioning process attributes every edge e with a traffic capacity cap_e^g that defines the upper bound on the amount of traffic the link can carry.

  Thus, the solution space S consists of a set of attributed graphs.

- The network is connected.
  Traffic emerging from any node must be able to reach any other node in the network, i.e. there is at least one path p between all pairs of nodes $(n_1, n_2) \in V \times V$:
  $$\forall (n_1, n_2) \in V \times V : |P_{(n_1,n_2)}| > 0$$

- The routing model is based on OSPF routing.
  The available OSPF lengths are defined as a real-valued vector $\vec{o} \in \vec{\mathbb{R}}$. The edge attributes i_e^g are used as indices into $\vec{o}$, $1 \leq w_e \leq |\vec{o}|$, $w_e \in \mathbb{N}$.

Routing takes place along shortest paths based on these edge attributes, i.e. (recall the definitions in Section 1.1)

$$\forall (n_1, n_2) \in V \times V : \quad w_{p(n_1,n_2)} = \min_{p \in P_{(n_1,n_2)}} [w_p]$$

- Traffic is directed and non-bifurcated.
  Traffic flowing from node a to b is using edge $(a \triangleright b)$, traffic flowing from b to a uses edge $(b \triangleright a)$, so that asymmetric traffic flows can be modeled adequately. In contrast to OSPF's abilities, traffic from node a to node b exclusively occupies a single path. Also, there are no multiple traffic types or quality-of-service classes.

- The problem encoding uses OSPF lengths as elements of a vector.
  Edges are numbered and OSPF lengths are put in a vector whose elements are indexed by the edge numbers. Unimplemented edges are penalized by a large OSPF length representing infinity (see Fig. 3.1). This string is a complete representation of the network topology.

  Formally, there is a bijective mapping function $Idx : E \leftrightarrow \mathbb{N}$ of the set of edges E in g into the non-negative integer domain, i.e.

  $$\forall e \in E \; \exists \, i \in \mathbb{N} \qquad \text{such that} \qquad Idx(e) = i$$

  Now the transformation $Enc : S \mapsto S^*$ of the solution space S to the encoding space S^* is a mapping of graphs $g(V, E)$ onto vectors of integer indices $\vec{\mathbb{N}}_{|E|}$ into the OSPF lengths vector $\vec{o}$ as follows: Let s be a graph in S and s^* a vector in S^*, then

  $$Enc(s) = s^* \qquad \text{such that} \qquad \forall e \in E_s : s^*_{Idx(e)} = i^s_e$$

  Encodings are either given as a vector of integer indices, or as a vector of OSPF values resulting from substituting the integer indices with the OSPF lengths in $\vec{o}$ that the index points to (see Fig. 3.1). The former will be called *index encoding*, the latter *OSPF encoding*. Both encodings are logically equivalent.

  Notice that multiple encodings can represent the same network (see Fig. 3.2). Also, encodings exist that lead to link capacity overloads or invalid topologies, assuming that flow over a an edge with a symbolical infinite OSPF length is invalid.

- A neighbour of an encoding is an encoding with a single change in the vector assignment.
  More specifically, let s be a solution, then

  $$\forall s' \in \mathcal{N}(s) : s^* \text{ and } (s')^* \text{ differ only in a single index } i \text{ , and } |s^*_i - (s')^*_i| = 1$$

- Edge costs are distance-time-based with economy-of-scale effects.
  Edge costs are calculated from a fixed setup cost $c_{setup}(e)$, a fix cost per data unit per distance $c_{unit}(e)$ and an economy-of-scale ratio α defining the relative growth of cost versus bandwidth. A base cost $c_r(e)$ is calculated for each edge as

  $$c_r(e) = w_e \left(cap_e \cdot c_{unit}(e)\right)^{\alpha}$$

This cost is used as input for a *discounted cash flow* calculation yielding $c_{reoc}(e)$, which takes into account the overall decrease in money value over time that is important with regard to reoccurring payments (for instance, lease fees). See Appendix C of [Fis00] for details. The complete edge cost c_e then becomes

$$c(e) = c_{setup}(e) + c_{reoc}(e)$$

Given the topology (recall Def. 1.4), the final objective function of a solution s in our optimization then is

$$\mathcal{O}(s) = \sum_{e \in E_{imp}} c(e)$$

- Node costs and capacities are ignored.
  Technical details like number of ports, throughput rate etc. are dropped from the model. Furthermore, during cost evaluation, node setup and running costs are also ignored, as they only contribute to the complexity of the cost function.

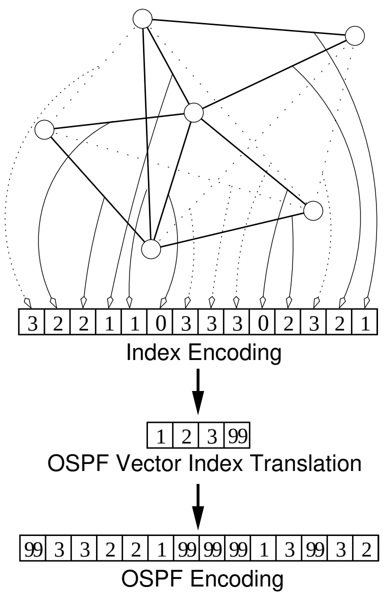

Figure 3.1: Encoding of an OSPF network topology by building a vector of OSPF lengths.

3.2 Hardness of the Problem

Finding the optimal solution solution, i.e, the OSPF assignment yielding a valid topology with smallest cost, is by no means trivial, in fact, recall from Sect. 1.4 that it is NP-complete. Assuming that it is possible to evaluate the objective function of a network in 10 ms and given a set of four different OSPF lengths, Table 3.1 shows the amount of

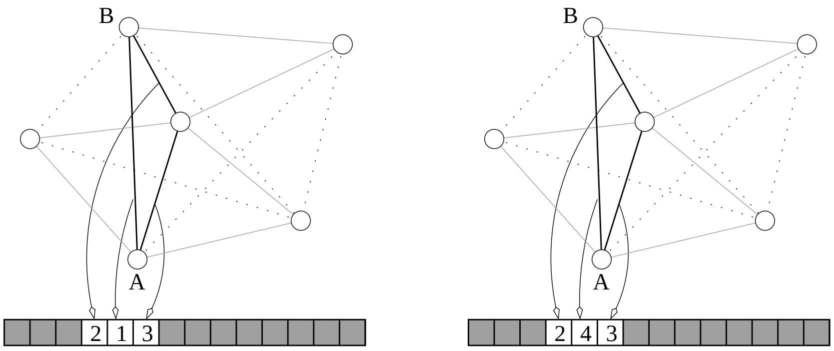

Figure 3.2: Two OSPF encodings that yield the same network topology. Notice that although the OSPF length from node A to node B has changed, the direct link stays in the topology. To be removed, it would need to be at least 5 (assuming the other edges have no impact on the decision).

time necessary for a full enumeration of all possible networks[1]. This calculation does not include the time needed for the iteration and the generation of candidate solutions.

Number of nodes	3	4	5	6
Appr. time to compute	40.96 s	46.6 hours	342 years	$3.587 \cdot 10^8$ years [2]

Table 3.1: Computational complexity of full enumerations for various network sizes.

Obviously, full enumeration is not feasible, not even for small numbers of nodes.

3.3 Approaching the Problem

As full analysis of the solution space is prohibitively complex for any realistic problem size, the following restrictions are imposed on the analysis:

- Use of meta-heuristics and randomized search.
  Instead of a brute-force full enumeration, multiple runs of Tabu Search, a Genetic Algorithm and Simulated Annealing implementations as well as randomized searches are used to acquire as much information as possible about the networks under analysis. The log files of those runs are used for comparing the relative effectiveness of the algorithms as well as for analyzing statistical properties of the solutions found by each of the the employed methods or a combination thereof.

- Focus on best solutions.
  Instead of focusing on the entire solution space, the analysis concentrates on the best solutions found. This includes logging of the values of the optimization function, the associated topologies and (possibly multiple, equivalent) OSPF assignments.

[1] The size of the solution space A for a complete graph consisting of n nodes with m OSPF lengths is $|A| = m^{n(n-1)}$.

[2] The age of the universe is about 10^{10} years.

The software used was implemented in C++ using the FooNet framework [Fis00], the calculations were performed on various UNIX machines, including Solaris and Linux operating systems. In detail, the algorithms used include:

- Genetic Algorithm (GA).
  A GA was implemented that randomly initializes a population of a given size, removes a number of individuals with bad fitness value, selects one parent on a fitness basis (the better an individual the more likely it is to be selected) and the other randomly, uses the crossover operator as means to produce offspring from the parents and employs mutation.

- Simulated Annealing (SA).
  A variation of SA supporting estimation of adequate starting temperatures and reheating was implemented as a specialization of a FooNet network design algorithm. The starting temperature is increased until a given acceptance ratio is satisfied, reheating occurs after a given number of non-improving solutions is encountered. The number of re-heats is also parameterized.

- Tabu Search (TS).
  The TS algorithm class implemented in FooNet was re-used and improved to facilitate various needed logging schemes.

- Fully randomized search (RND).
  A simple, fully randomized search was implemented that randomly assigns OSPF lengths to each edge, taken from the set of available OSPF lengths. Thus there may or may not be any similarities between two consecutive solutions.

- Randomized neighbourhood search (RNS).
  A randomized neighbourhood search was implemented in which a transition is performed in each iteration from the current solution to a random solution in the neighbourhood as defined in Definition 3.1. In contrast to TS, no tabu list prevents the algorithm from immediately going back to the previous solution.

The optimization procedure is shown in Fig. 3.3 and basically works as follows: Given a set of network nodes and a traffic requirements matrix, a decision is made on which edges are potential implementation candidates and which edges are excluded from the design. This decision is at the center of heuristic link-reduction approaches, which are presented in Chapter 4.

The edges are assigned OSPF lengths according to some solver-dependent scheme. An all-pairs shortest paths algorithm is then employed to determine the shortest paths between each source and destination of network traffic. When the shortest paths are established, the network topology is loaded and dimensioned as described in Section 1.5. Finally, the objective function is calculated and the current solution is logged for later analysis. If the (solver-dependent) end criterion is reached, the process aborts, otherwise the OSPF assignment is updated in some way which may or may not lead to better results, and the process is repeated.

The following information is gathered during the search:

- Solution pool.
  A solution pool was implemented that can store encountered solutions as equivalence

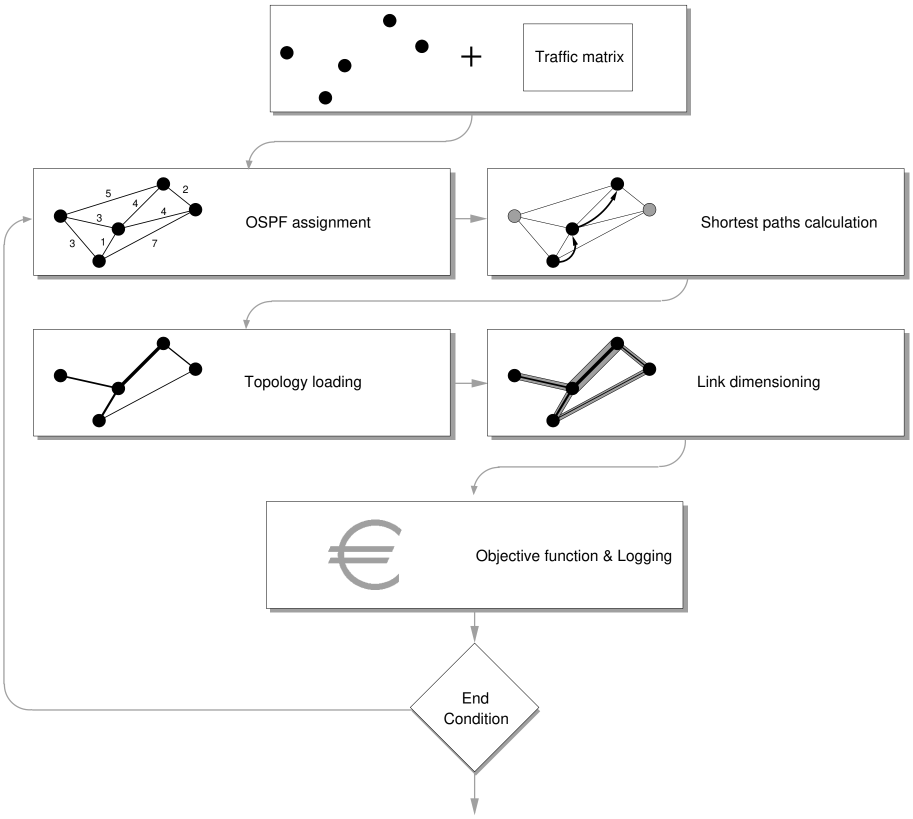

Figure 3.3: Flow chart of the optimization procedure.

classes, i.e. the pool offers a fixed number of slots, where each slot collects all solutions with equal objective function value that lead to identical shortest path structures (see Fig. 3.4). In particular, each slot stores

- The cost of the topology,
- The traffic loads on each link (including zero load for unused links),
- A set of OPSF index encodings that yield identical traffic loads on the links.

As illustrated above, different OSPF encodings can yield the same topology. Collecting data this way eases analysis of equivalent solutions' properties. Slots in the pool are sorted by decreasing quality of objective function results. Insertions into the pool are arranged in such a way that bad solutions are progressively replaced by better solutions, resulting in a collection of the best n solutions and their equivalent OSPF assignments. The pool is serializable, i.e. can be logged to permanent storage and re-read at any time.

- Objective function convergence.
  Convergence plots show the quality of solutions found during the search by opposing iteration numbers to objective function results.

  Two different modes of convergence logging were implemented: *complete logging*, which saves the cost of every encountered solution together with the current iteration number and the currently best objective function value. While this type of logging

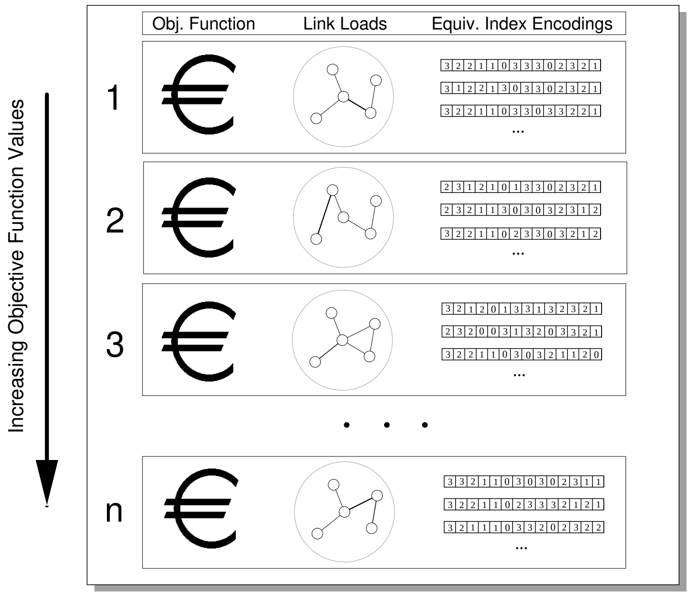

Figure 3.4: Solution Pool Organization.

shows the full characteristics of the search process (see Fig. 3.5 left), log-files can grow to large file sizes (many hundred megabytes for long-running searches). To address this problem, a second logging strategy is used, which only saves the iteration number of points in the search where improvements happen, along with the new best objective function value, called *convergence logging*. While the actual shape of objective value functions is lost (see Fig. 3.5 right), log-files stay small in size.

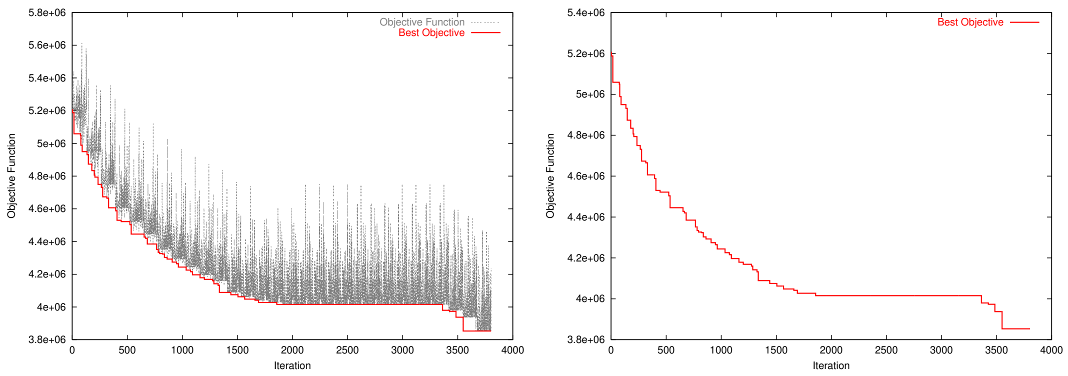

Figure 3.5: Complete (left) and improvement-only (right) convergence plots, generated by a short run (appr. 20s) of TS on a 10-node network.

Appendix A shows an UML chart of the class layout. See the source code and its documentation available at `http://www.informatik.tu-muenchen.de/~kreibich/da-software.tar.gz` for further details on the code.

4

Runtime and Performance Improvements

The optimization process as illustrated in the previous chapter offers various possibilities for performance improvement, which are discussed in this chapter before experimental results will be described in Chapter 5.

4.1 Incremental Processing

Assuming that a solver makes only few changes to the OSPF assignment of a network during each iteration, there are four obvious performance bottlenecks in the optimization process (recall Fig. 3.3): the necessary work which needs to be performed by shortest path calculation, topology loading, link dimensioning and the calculation of the objective function is likely rather small compared to the work performed by a full recalculation of all shortest paths, all edge loads and all link capacities.

This assumption is valid for most of the solvers used: Tabu Search, Simulated Annealing and Randomized Neighbourhood Search change the OSPF assignment considering only the neighbourhood (recall Defs. 2.5 and 3.1) of the current assignment as potential move candidates.

Therefore a lot of performance is to be gained if the full calculations illustrated above are replaced by *incremental* procedures, that ideally only perform exactly the amount of work necessary, but no more.

Definition 4.1: Incremental Algorithm

Let $A(I) \to J$ be an algorithm that repeatedly transforms input of type I in output J for iterations $1, \ldots, n, n+1, \ldots, n_{max}$. Let the difference between two instances n, $n+1$ of I be of type Δ, and denoted as $\Delta_{n,n+1}$. Then an algorithm $A(J, \Delta) \to J$ that produces J_{n+1} from J_n and $\Delta_{n,n+1}$ is called *incremental* algorithm. See Fig. 4.1 as illustration.

In the given problem setting the all-pairs shortest-path calculation is the basis for the following topology loading, link dimensioning and objective function evaluation. An all-pairs shortest-path algorithm that works incrementally can update only the necessary parts of the shortest path data structures and gather information about which paths are modified during the process. Instead of re-calculating every link in the topology, topology loading and link dimensioning can then use this information in order to update only the

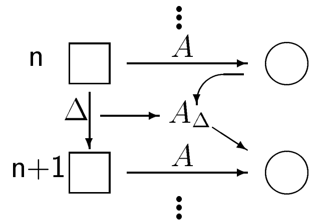

Figure 4.1: A_Δ, an incremental algorithm.

necessary parts. An incremental objective function evaluation then only needs to update the last result by again inspecting only the affected links.

The following sections will illustrate the strategies and difficulties of implementing this approach.

Implementation of an Incremental All-Pairs Shortest-Path (IAPSP) Algorithm

The implementation described here is based on an algorithm suggested in [Roh85]. However there are subtle issues with that algorithm which unexpectedly caused a lot of time being spent on its implementation, so the description here will be rather detailed.

The goal of the algorithm is as follows:

> **After a single change (increase or decrease) in length of a link in the network, the algorithm incrementally updates the shortest-path and distance data structures for the affected node pairs to reflect the change.**

Note that this approach also covers the cases of adding/deleting links from a network by setting the link length to a symbolic very large value ∞ representing infinity, or decreasing the length from that value, respectively.

Predecessor Matrix

The data structures used for storing shortest-path information are *predecessor node matrices*. Assuming that nodes are numbered[1], entry (m, n) in a predecessor matrix contains the index of the next-to-last node on the path from node m to n. Diagonal entries (i, i) (for which there is no previous node) contain either null entries or index i. Iteration over path $p(a, b)$ is thus possible by looking up entry (a, b) in the predecessor matrix, then $(a, (a, b))$, etc until entry (a, a) is reached and iteration is complete. The shortest path p between nodes is therefore available in $O(|p|)$. In the IAPSP algorithm, a similar *successor matrix* that works just as a predecessor matrix but for next-to-first nodes is also used. The predecessor matrix will be called $Pred$, the successor matrix $Succ$.

Successor Matrix

Distance Matrix

The distance data structure is also a matrix $Dist$, in which element (m, n) contains the length of the path from node m to node n. Accessing is possible in $O(1)$.

The most important data structure is the one used for storing the parts of the graph which are affected by a link length update. Explicit, centralized storage of, say, all node pairs whose connecting path is affected by changing a certain link decreases performance too

[1] LEDA actually allows indexing using node objects directly, instead of using node indices. Since the meaning is clear, in the following indexing a matrix using nodes means the equivalent of indexing by the indices of nodes.

much, because detouring a path would necessitate a lot of updates. Instead, the approach used works *implicitly* and data is stored in a *distributed* fashion:

For all links, every node maintains information about which of its adjacent edges are affected by changes in some link length. To further differentiate start- and end nodes, one such data structure is maintained for in-edges into a node, and one for out-edges. This data structure implicitly defines a tree of affected edges which can be used to iterate over. The tree resulting from the in-edges data structure will be called *in-tree*, the one for the out-edges *out-tree* (see Fig. 4.2).

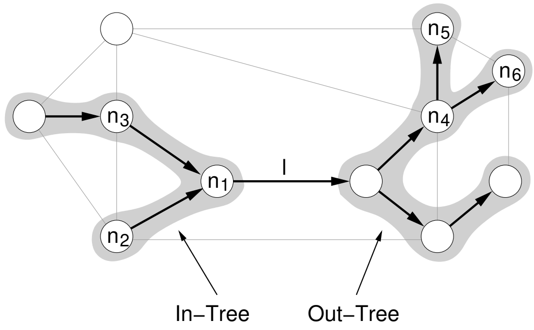

Figure 4.2: Implicit tree structure. Querying for link l in the in-tree structure at node n_1 will report nodes n_2 and n_3, querying the out-tree structure at node n_4 will report nodes n_5 and n_6.

In the implementation, the data structure looks as follows (the `leda_` prefix is omitted):

```
typedef node_array < edge_array < h_array < edge, bool> > >  tree_info;
```

So the code for performing an operation on all adjacent edges of node n which are affected by a change of link l looks as follows:

Algorithm 4.1: Affected Edges Iteration

```
node n;          // Node which is investigated
edge l;          // Link which is modified
tree_info t;     // Properly initialized tree structure;

edge e;          // Edge for iteration

forall_defined(e, t[n][l])
  {
     // perform something on edge e;
  }
```

The following sections will explain the way the algorithm works. Like all incremental algorithms, the IAPSP algorithm first performs an initialization:

Initialization

In the initialization, predecessor and successor matrices as well as the distance matrix are set up using Floyd-Warshall's algorithm. Also, the in- and out-tree structures need to be initialized, which is done by iterating over all shortest paths

- forward, marking in the in-tree at every node in the path its outgoing link towards the path destination as affected by changes of the last link on the path and
- backward, marking in the out-tree at every node in the path its incoming link from the path source as affected by changes of the first link on the path.

The initialization procedure is computed in $O(|V|^3)$. For all following link length updates, assume the changed link is called l, with nodes $(n_{l_s} \triangleright n_{l_t})$. There are two distinct cases the algorithm addresses differently:

Edge Length Increase

The primary strategy in case of a length increase is:

- A shortest path that does not go through link l is not affected by the change, since it cannot possibly get shorter by using l.
- A shortest path that goes through link l either remains unchanged and is only increased in length, or the length increase is sufficient to cause a detour of the path.

This can be exploited by only examining paths that used link l before the update. In order to do so, both the in-tree and out-tree rooted at the modified edge are traversed in a depth-first manner. Let V_s be the set of nodes in the in-tree and V_t the set of nodes in the out-tree. For every step in the in-tree, the entire out-tree is traversed as long as the paths are still routed inside the tree[2]. Let the currently visited node in the in-tree be n_i, the one in the out-tree be n_o. Notice that the fact that n_i is member of the in-tree and node n_o is member of the out-tree does not necessarily imply that the shortest path $p(n_i, n_o)$ uses link l — the path starting in n_i with *some* destination node in the out-tree uses link l, not *every* destination node.

When the edge length increase is large enough to cause a detour, the new shortest path does not have to be calculated completely — it must be using the shortest path up to a neighbour of the destination node, plus the link that connects the neighbour to the destination node itself. Therefore, in every step of the traversation a check is made whether neighbour nodes n_n of the current node in the out-tree can improve the path length: $w_{p(n_i,n_n)} + w_{(n_n \triangleright n_o)} < w_{p(n_i,n_o)}$? If so, the neighbour with the greatest improvement is a candidate for a path detour. Fig. 4.3 illustrates the process.

The problem at this point is that it cannot be said whether path $p(n_i, n_o)$ must be detoured via that greatest-improvement neighbour, because some sub-path of $p(n_i, n_o)$ may be detoured and decreased in length by a greater value than the greatest-improvement neighbour can come up with, effectively eliminating the need for detouring the path. The problem of finding the correct sequence for updating the shortest-path and distance structures is solved using a priority queue. During every traversation of the out-tree, this priority queue is set up by inserting the detour candidates with the improvement they can offer as priority. When the traversation is complete, nodes with the best improvement are extracted from the priority queue, shortest-path and distance structures are updated,

[2] This can be verified by comparing the last link in the path to the edge in the out-tree that led the traversation of the out-tree to the current target node. If they are not identical, the shortest path does not use link l and the subtree of the out-tree below the current node can be pruned.

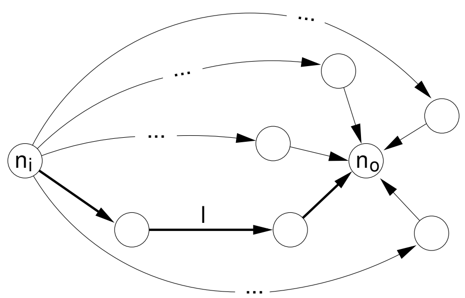

Figure 4.3: Candidate inspection — link l gets longer, the in-edge neighbours of n_o are checked for path length improvement by detouring through them the path starting at n_i.

and the nodes that are affected by the change (those in the out-tree subtree rooted at the extracted node) are decreased in priority accordingly.

Let path $p(n_s, n_t)$ be the path to detour, and let n_n be the neighbour of n_t to detour via. A path detour is accomplished as follows:

- Set the predecessor matrix entry $P(n_s, n_t) = n_n$, i.e., set the predecessor node of the target node to the neighbour node.
- Set the successor matrix entry $S(n_s, n_t) = S(n_s, n_n)$, i.e., set the successor node of n_s to the successor node on the path to the neighbour node.
- Let link $l_{last} = (n_{t1} \triangleright n_t)$ be the last link on the old path from n_s to n_t. Let link $l_{first} = (n_s \triangleright n_{s1})$ be the first link on the old path from n_s to n_t.
  - In the in-tree, delete l_{first} from the links being affected by a change of link l_{last} in node n_{s2}.
  - In the out-tree, delete l_{last} from the links being affected by a change of link l_{first} in node n_{t2}.
- Let link $l_{last} = (n_{t1} \triangleright n_t)$ be the last link on the new path from n_s to n_t. Let link $l_{first} = (n_s \triangleright n_{s1})$ be the first link on the new path from n_s to n_t.
  - In the in-tree, add l_{first} to the links being affected by a change of link l_{last} in node n_{s2}.
  - In the out-tree, add l_{last} to the links being affected by a change of link l_{first} in node n_{t2}.
- Set $Dist(n_s, n_t) = Dist(n_s, n_n) + w_{(n_n \triangleright n_t)}$.

Notice that an update involves only manipulations of the first and last links in the old and new paths, respectively. The remaining path is fixed implicitly during the process[3].

[3]Which can make the debugging process quite painful.

Edge Length Decrease

The primary strategy in case of a length decrease is:

- A shortest path that does not go through link l may be affected by the change and may need to be detoured.
- A shortest path that goes through link l remains unchanged and is only decreased in length.

The algorithm must consider all node pairs $(n_s, n_t) \in (V \times V)$ for which there is a path $p(n_s, n_{l_s}) \circ l \circ p(n_{l_t}, n_t)$ from n_s to n_t running via l. In context of the network design problem, this will address all node pairs, because our network is assumed to be connected, i.e. there is a path from every node to every node. If connectedness is not a prerequisite, consider instead all node pairs $(n_s, n_t) \in (V_s \times V_t)$, where V_s and V_t are the sets of reachable nodes defined through $V_s : \{n \in V \text{ such that } \exists\ p_{(n,n_{l_s})}\}$ and $V_t : \{n \in V \text{ such that } \exists\ p_{(n_{l_r},n)}\}$.

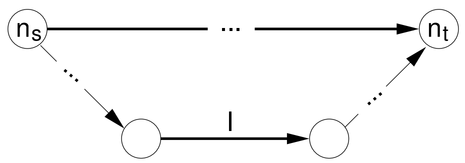

Figure 4.4: Candidate inspection — link l gets shorter, node pairs are checked for improvement by detouring via l.

In any case, the algorithm checks for all those node pairs whether detouring the path via the shortened link improves the overall path length (see Fig. 4.4): $w_{p(n_s,n_{l_s})} + w_l + w_{p(n_{l_t},n_t)} < w_{p(n_s,n_t)}$? If this condition is true, the path is detoured in a similar fashion as described above:

- Set the predecessor matrix entry $P(n_s, n_t) = P(n_{l_t}, n_t)$, i.e., set the predecessor node of the target node to the predecessor node of the path from the shortened link to the destination node.
- Set the successor matrix entry $S(n_s, n_t) = S(n_s, n_{l_s})$, i.e., set the successor node of n_s to the successor node on the path to the shortened link.
- Let link $l_{last} = (n_{t1} \rhd n_t)$ be the last link on the old path from n_s to n_t. Let link $l_{first} = (n_s \rhd n_{s1})$ be the first link on the old path from n_s to n_t.
  - In the in-tree, delete l_{first} from the links being affected by a change of link l_{last} in node n_{s2}.
  - In the out-tree, delete l_{last} from the links being affected by a change of link l_{first} in node n_{t2}.
- Let link $l_{last} = (n_{t1} \rhd n_t)$ be the last link on the new path from n_s to n_t. Let link $l_{first} = (n_s \rhd n_{s1})$ be the first link on the new path from n_s to n_t.
  - In the in-tree, add l_{first} to the links being affected by a change of link l_{last} in node n_{s2}.

– In the out-tree, add l_{last} to the links being affected by a change of link l_{first} in node n_{t2}.

- Set $Dist(n_s, n_t) = Dist(n_s, n_{l_s}) + w_l + Dist(n_{l_t}, n_t)$.

For paths which are already routed via l, the path detour does not cause any changes, only the distance matrix entry is updated.

The Problem of Equal Path Lengths

A point not addressed in [Roh85] is the problem of multiple paths of equal lengths. While the problem is unlikely to occur when using geographical distances, it will almost certainly be encountered when using e.g. integer-valued OSPF lengths.

The problem is that when there are multiple alternatives for choosing a path, the priority queue will not necessarily return the correct one in each iteration. This leads to two phenomena:

- Inconsistent predecessor and successor arrays.
  When there are multiple nodes of equal minimum priority in the priority queue, inconsistent predecessor and successor paths can result from different nodes returned for different source- and/or target nodes.

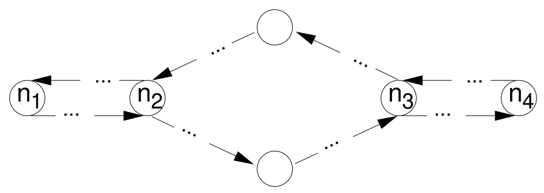

Figure 4.5: Inconsistent predecessor and successor paths.

- Inconsistent sub-paths through rounding errors.
  This means that when considering a path $p(n_1, n_2) \circ p(n_2, n_3) \circ p(n_3, n_4)$, iterating over path $p(n_1, n_4)$ using the predecessor matrix will produce a different path than iterating over $p(n_2, n_4)$, see Fig. 4.6. This is a variation of the problem illustrated above.

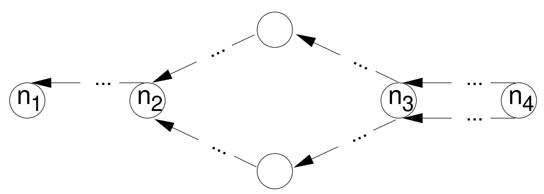

Figure 4.6: Inconsistent sub paths.

This is a side-effect not controlled by the in- and out-tree structures. Inconsistent sub paths can cause errors in the incremental topology loading algorithm (see below) because link loads can become negative.

There are two possible ways of addressing these issues:

- Improving the algorithm so that it knows how to avoid inconsistencies and choose correct paths at all times.
- Avoiding paths of equal length from the beginning.

The former option is difficult to implement and involves a lot of overhead. A priority queue is needed that arranges nodes of equal priority in a way that does not only depend on the way the priority queue's internal data structure happens to arrange them, but rather uses one that returns the correct node at all times. Furthermore, keeping predecessor and successor paths synchronized is much more difficult than just detouring a path by manipulating the first and last links as illustrated above, due to the fact that the original context of the path iteration (i.e. the original start and destination nodes) are lost during iteration when using predecessor/successor matrices.

The latter option is much easier. Even when equal length paths are likely to occur, there is a way to avoid them by introducing "fudge factors", small offsets of length for each link in the network. The idea is to add these offsets to the real link length so that two different paths which would otherwise be of equal length deviate in length by the difference of the sums of the offsets (see Fig. 4.7). The path length including the fudge factors will be called *virtual path length*, as opposed to the *real path length*.

Virtual, Real Path Length

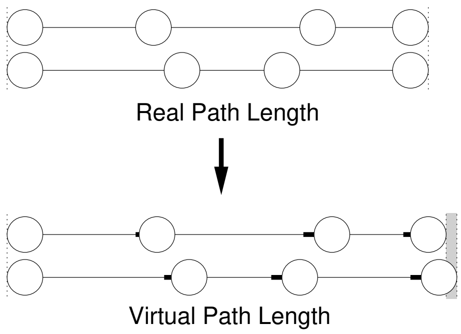

Figure 4.7: Avoiding equal length paths using fudge factors (thick black lines).

The offsets need to be small enough so that their sum can never be greater than the smallest (original) difference Δ in length that two paths can have. In the case of integer OSPF lengths, this means that summing up the offsets of the longest conceivable path must result in a number smaller than 1. Since paths are assumed to be loopless, the maximum number of hops in a path can be $|V|-1$. A possible way of calculating the fudge factors is therefore the generation of a random number in the range $]0,\ \Delta/(|V|-1)[$. Also, no two sums of offsets must be of equal size, which is achieved by assigning the reciprocal value of prime numbers to edges and then scaling the values into the desired range.

In the implementation, all internal computations of the IAPSP algorithm are transparently based on virtual path lengths.

IAPSP Algorithm Performance

This section will give an overview of performance characteristics of the IAPSP algorithm, based on both analytical and measurement results.

While one might assume that the average number of changed paths per edge update increases with a growing number of edges in a graph, the opposite is the case. Fig. 4.8 (left) illustrates the behaviour using various degrees of edge density, where the minimum edge density is 0 edges and the maximum density is $|V| \cdot (|V| - 1)$ edges, i.e. a complete graph. Fig. 4.8 (right) shows the behaviour with growing number of nodes and fixed edge density.

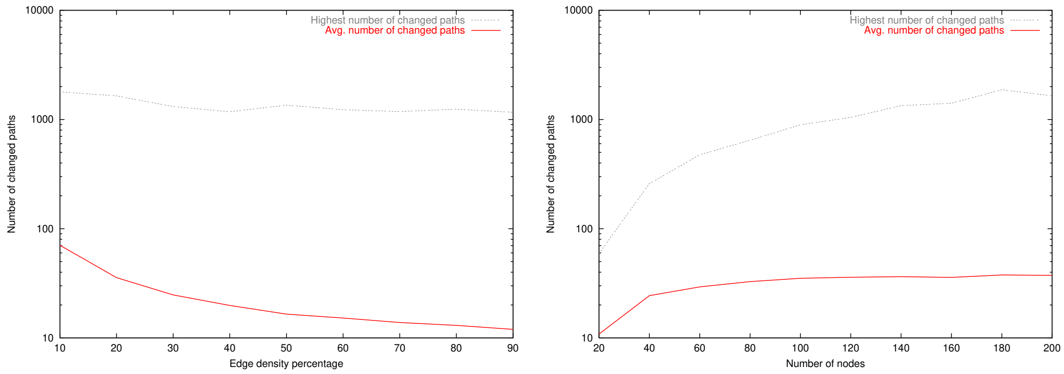

Figure 4.8: Total average and average maximum number of paths changed incrementally in graphs with 100 nodes and varying edge density (left), and graphs with varying numbers of nodes and 10% edge density (right). 10 randomly generated graphs were exposed to 10.000 random link length updates each, at each density (left) and node numbers ranging from 20 to 200 (right). Note that the y axis is in logscale on both plots, i.e. the average maximum number of changes is orders of magnitude larger than the average. Also, in every run there were link updates that caused 0 paths to be changed, which is why the minimum number is not shown.

[Ber92] states that no incremental algorithm for the all-pairs shortest-path problem in a directed graph can have a better worst-case run time than full recomputation[4]. The straightforward proof is given by an example of a graph in which there is one edge which is part of all shortest paths and a subsequent length increase of that edge.

However it is important to note that this is a worst-case statement. Fig. 4.9 compares the performance of Floyd-Warshall's non-incremental all-pairs shortest-path algorithm to the IAPSP algorithm when applied to randomly generated graphs[5], for varying degrees of edge density and node numbers. Each test set comprised 10 graphs, each of which was subjected to 10.000 random link length updates. For each graph, the maximum, minimum and average run time was recorded.

The figures were computed by collecting statistics during each update performed by the IAPSP algorithm. As can be seen, the performance improvement is impressive: for varying degrees of edge density, the IAPSP algorithm was on average 94 times faster than Floyd-Warshall. For graphs varying in node numbers and 10% edge density the speedup increases

[4] The statement is actually a quote of [EG85], however that paper could not be found in the available libraries or on the Internet.

[5] The graphs were generated using LEDA's graph generation facilities.

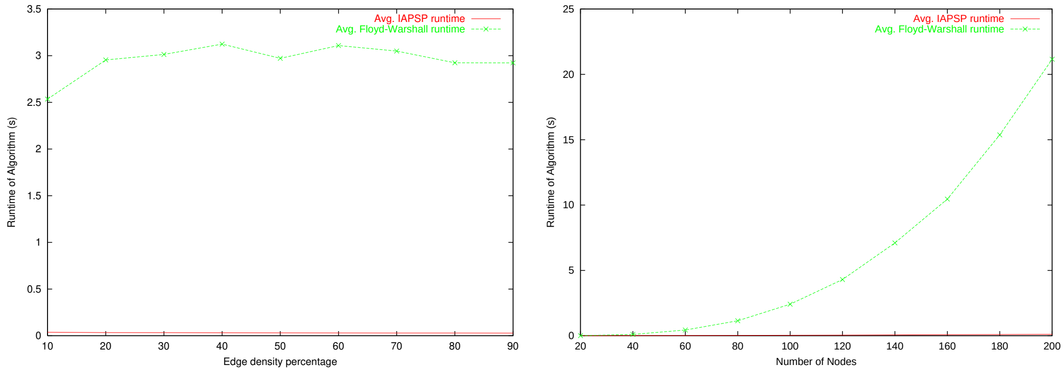

Figure 4.9: Runtime comparison of IAPSP and Floyd-Warshall algorithms for varying degrees of edge density in a 100-nodes graph (left) and varying numbers of nodes with 10% edge density (right).

from about 64 times Floyd-Warshall's performance at 100 node size to almost 188 times Floyd-Warshall's run time at 200 nodes.

While the average run time continually stays low, the average maximum run times encountered start to grow strongly with both varying edge density and number of nodes, as shown in Fig. 4.10. However, for varying edge densities, the run time actually decreases with increasing density, reflecting the number of changed paths in Fig. 4.8. While for varying edge densities the minimum run time was continually too low to be measurable, an average minimum run time above zero starts to show at graphs with 140 nodes.

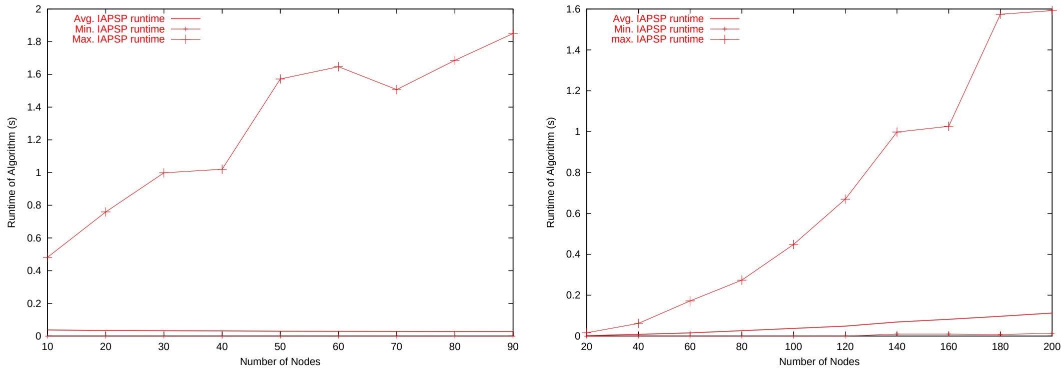

Figure 4.10: Average, maximum and minimum run times of the IAPSP algorithm for varying degrees of edge density in a 100-nodes graph (left) and varying numbers of nodes with 10% density (right). At each step, 10 randomly generated graphs were exposed to 10.000 random link length updates each, and average minimum, maximum and overall run times recorded.

The reader is referred to [Roh85] for details on analytical upper and lower bounds of the IAPSP algorithm. With regard to Definition 4.1, J symbolizes the predecessor/successor and distance matrices, while Δ is the link that is modified plus its new length.

Implementation of an Incremental Topology-Loading and Link-Dimensioning Algorithm

With the information on which paths are to be changed available, an incremental version of the topology-loading and link-dimensioning algorithm is simple, see Algorithm 4.2. The traffic requirements of the affected paths are simply reduced on the old paths and increased on the new ones. In the incremental context, J here is the matrix containing current link loads, and Δ is the set of paths changed by the IAPSP algorithm.

Algorithm 4.2: Incremental Loading & Dimensioning

```
L = list of node pairs whose connecting paths have changed

for all node pairs (n1, n2) in L
  {
     p_old = old path from n1 to n2 before link length update
     p_new = new path from n1 to n2 after link length update

     for all links in path (p_old)
       Load(n1, n2) -= Req(n1, n2);
     for all links in path (p_new)
       Load(n1, n2) += Req(n1, n2);
  }

for all node pairs (n1, n2) in L
  {
     p_old = old path from n1 to n2 before link length update
     p_new = new path from n1 to n2 after link length update

     for all links l in path (p_old)
       set capacity on link l to smallest capacity > Load on link
     for all links l in path (p_new)
       set capacity on link l to smallest capacity > Load on link
  }
```

Implementation of an Incremental Objective Function Calculation

Incremental objective function calculation is also straightforward — the current cost is updated by the difference in cost that results from the differences in loads on the old and new paths as reported by the IAPSP algorithm, as illustrated in Algorithm 4.3. J here is the topology cost, while Δ is again the set of paths changed by the IAPSP algorithm.

Algorithm 4.3: Incremental Objective Function Evaluation

```
C        = Result of last objective function evaluation (cost).
L        = list of node pairs whose connecting paths have changed
Load_old = old loads on the links before last link length update

for all node pairs (n1, n2) in L
  {
     p_old = old path from n1 to n2 before link length update
     p_new = new path from n1 to n2 after link length update

     for all links in path (p_old)
       C += (Load(n1, n2) -  Load_old(n1, n2))
     for all links in path (p_new)
       C += (Load(n1, n2) -  Load_old(n1, n2))
  }
```

4.2 Heuristic Link Reduction

Using incremental algorithms is a general approach to obtain performance improvements. In contrast, *heuristic link reduction*(HLR) approaches are problem-specific. The basic goal is to delete links from the network which are unlikely to be implemented in the resulting network. How to determine that likeliness is up to the heuristic. Several approaches are conceivable, such as:

- Pure distance-based
  Direct links between two nodes are likely unjustified when there is a disjunct path running comparatively parallel to it. Distance-based heuristics use a length ratio $\alpha > 1$ which causes removal of direct links $l = (n_s \triangleright n_t)$ whenever $w_p < \alpha \cdot w_l$ for some path $p(n_s, n_t)$ that does not use l.
- Pure traffic-based
  This approach simply drops direct links when the capacity requirement falls below a parameterized threshold value.
- Distance/traffic-based
  Drops a link when both of the above criteria would drop it.
- Load/length-based
  Drop links for which the ratio of load/length on the link falls below a threshold value, i.e. long links need to carry more traffic to stay in the topology.

All of these approaches require careful adjustment of the parameters in order to avoid elimination of important links, but can substantially reduce the search space size. It may be desirable to have multiple paths between source and destination in order to increase network reliability. An algorithm like given in [Suu74],[HC98] for finding k shortest paths can be used to verify that enough alternative paths exist after the direct link is removed.

4.3 Further Potential

There is other room for optimization. As explained in Sect. 3.1, multiple OSPF assignments can result in the same network. Being able to recognize these equivalence classes during the search would greatly reduce the amount of computation necessary, as it would remove large amounts of redundancy from the search space.

However, from an incremental point of view, this would at least necessitate a means to recognize the threshold values for link length increase/decrease that causes path detours, which is not a trivial problem if the IAPSP algorithm is to remain efficient. This is an interesting issue for further research.

5

Meta-Heuristics Comparison and Search Space Visualization

This chapter presents the results of analyzing the behaviour and performance of the solvers. The analysis covers solver runtimes, relative performance comparison of the solvers in terms of solution quality and convergence speed, characteristics of equivalent solutions, solution convergence progress and HLR effectiveness.

Fig. 5.1 shows the node layout of four networks under consideration, with 5 (**net_S**), 10 (**net_M**), 15 (**net_L**) and 20 nodes (**net_XL**), respectively. Larger nodes represent nodes with higher weight (recall Def. 3). The networks and traffic matrices were generated using the **tiers** network generator [Doa96].

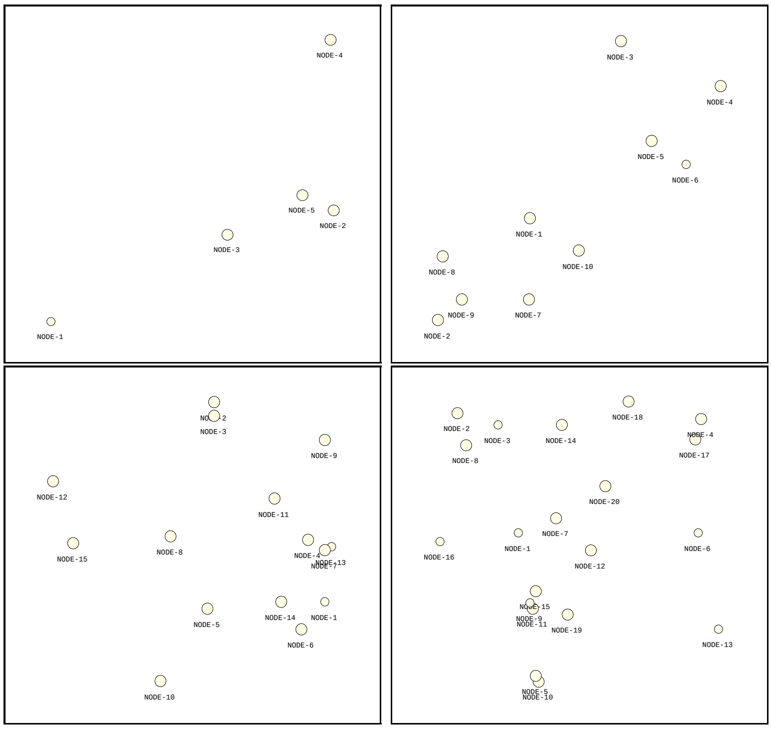

Figure 5.1: Node layout and weights of four networks, top left to bottom right: net_S, net_M, net_L and net_XL.

The stepped cost function used is shown in Fig. 5.2.

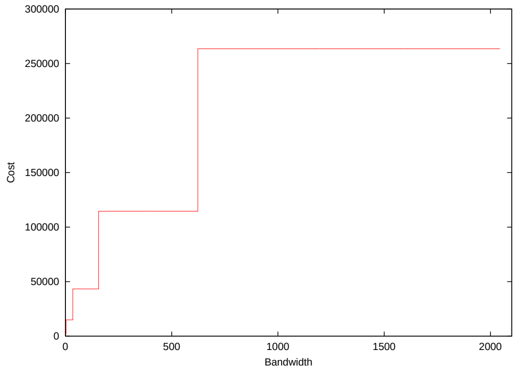

Figure 5.2: The cost function used for evaluating topologies.

5.1 Influence of OSPF link lengths

Obviously, the choice of available OSPF link lengths $\vec{o}$ has direct influence on the search space structure. Recall that a complete graph with n nodes has $|\vec{o}|^{n(n-1)}$ different OSPF assignments, therefore, the size of the search space depends heavily on the number of different lengths. The interesting point is to find a set of OSPF lengths that is both as small as possible and at the same time leads to quick heuristics' convergence.

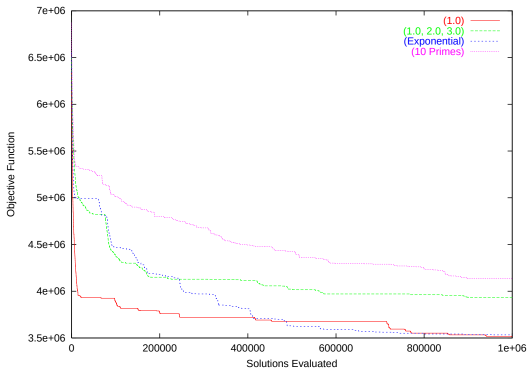

Figure 5.3: Convergence of TS on `net_M` for various sets of OSPF lengths.

Fig 5.3 shows the behaviour of TS on `net_L` with the following sets of OSPF lengths (∞ is used to symbolize a value that is large enough so that traffic is never routed via a link with that length, effectively removing it from the topology), averaged over six runs each:

- A A single length and ∞.
- B 1.0, 2.0., 3.0 and ∞.
- C 1.0, 10.0, 100.0, 1000.0, 10000.0 and ∞.
- D The first 10 prime numbers and ∞.
- E The first 100 prime numbers and ∞.

The reasoning behind these sets is an attempt to cover some of the obvious length patterns and has no sophisticated background. There is clearly some influence of the choice of OSPF lengths. Set D (the first ten prime numbers) produced the worst convergence speed and final result. For the E (the first 100 primes) the result gets so bad that it is not shown in the plot. It's final objective function value is two orders of magnitude larger than that of the others. C and A, the exponential set and that with only a single OSPF length (except for ∞), produce similar results, with the latter converging somewhat faster.

This is, however, only the behaviour for TS. In fact, set A drastically increases the memory consumption of the solvers, the cause of which is likely to be found in the data structures of the IAPSP algorithm, since a change in link length here effectively means adding/removing links from the topology. Also, SA needed much larger initial temperatures to satisfy a given acceptance rate.

Set B offered a good balance between search stability and solution space size, which is the reason why it was used in all of the following investigations. Nevertheless the topic would clearly deserve more thorough investigation.

5.2 Metaheuristics parameterization

Table 5.1 restates the solvers used and their basic characteristics (recall Sect. 3.3).

Solver	Incremental	Comment
GA		Population-based
RND		Zero context between iterations
RNS	✓	Zero statekeeping
SA	✓	Almost like RNS
TS	✓	Statekeeping via short-, medium- and long-term memory

Table 5.1: Employed solvers and their basic characteristics.

Note that in principle the GA solver could be implemented using an incremental approach, which evaluates the offspring by changing link lengths in the encoding one at a time, updating the objective function. Evaluating under what circumstances this would be beneficial was unfortunately not possible in the scope of this thesis due to lack of time.

The parameter setting for the solvers were as follows:

- GA
  The GAs used had an initial population size of 150 individuals, a working population of 100, 20 individuals were replaced with offspring in each iteration, where the offspring was exposed to a 0.3 probability of mutation, which, if applied, affected at most 3, 13, 31 and 57 link lengths for `net_S`, `net_M`, `net_L` and `net_XL`, respectively.

- RND
  The fully randomized search did not involve any parameterization.

- RNS
  As for RND, randomized neighbourhood search did not involve any parameterization.

- SA
  Simulated Annealing increased the starting temperature until 95% of all moves were accepted, using reheats during the search, which were initiated after 400, 900, 2100 and 3800 nonaccepted solutions for **net_S**, **net_M**, **net_Land** **net_XL**, respectively, reflecting the increasing neighbourhood sizes. The cooling scheme used was $T' = \frac{T}{1+C\cdot T}$ with a cooling factor $C = 2^{-9}$.

- TS
  Tabu Search used a tabu list length of 7, updating the medium-term memory every 150 iterations, iterating 50 times at each entry in the intensification phase. Long term memory was used to facilitate diversification by penalizing solutions produced by changing link lengths that were already frequently updated.

All solvers were attacking complete graphs, i.e. for a test graph with n nodes the topology consists of potentially up to $n \cdot (n-1)$ links. Recall from Def. 3.1 that a link is left out of the design when it is assigned a large weight representing infinite length, so that no traffic will ever be routed via that link.

All results are derived using averaged results of 5 runs of each solver on each network, unless otherwise noted. All solvers were run until each had evaluated one million solutions.

5.3 Metaheuristics runtime

Even though the heuristic approach immensely reduces the time required to achieve good-quality solutions compared to extensive search, the computational resources needed for a thorough search may become quite costly. If the time available to complete the design task is limited, it may be important to be aware of the relative run times of different solvers. Even though a solver may not produce the best solutions compared to other techniques, it may be a valuable option for e.g. finding promising layouts to use as base solutions for other optimization techniques. Fig 5.4 shows the runtime behaviour of the solvers for the four different networks. During each solver's search, every evaluated solution was counted as an iteration, and the time measured until 100.000 iterations were completed.

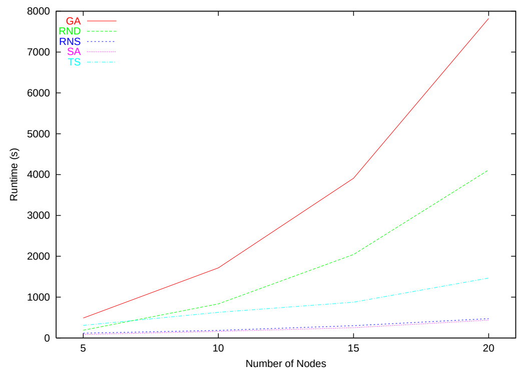

Figure 5.4: Relative runtime behaviour of TS, GA, SA, RNS and RND solvers on the four networks, with 5, 10, 15 and 20 nodes.

The differences in run times are impressive, but hardly surprising. In the used implementation, GA is clearly an expensive technique, as it not only non-incremental, but also requires a good deal of resources to re-sort the population and to prepare and apply the crossover and mutation operators. RND comes second, as it is also non-incremental in nature, but since it requires no additional overhead, runs faster than GA. While being slower than RND for very small networks, TS quickly wins over GA and RND solvers with growing network sizes due to its incremental implementation. However, its complex structure and administrative tasks in order to keep the memory structures up-to-date make it between 2.6 and 3.3 times slower than simple RNS in the networks tested. SA is quite similar in the code structure compared to RNS, which explains the almost identical runtimes.

5.4 Metaheuristics solution quality and convergence speed

The relative performance of the five solvers in terms of resulting solution quality is illustrated in this section. It is important to notice that in the network design problem, the difficulty is not to find a valid solution at all, as in many other discrete optimization problems, but rather to find a promising path through the search space that does not get stuck in local minima or plateaus in the search space. Therefore it is important that the search technique has a way of memorizing characteristics of the regions of the search space it investigated so far, and exploiting that knowledge in order to move the search to promising regions and keeping it away from poor ones.

Fig 5.5 shows the average performance of the TS, GA, SA, RNS and RND solvers for the four networks. Average performance was derived from the convergence logs of 5 runs of each solver. Notice that the x axis does not show the number of iterations, but rather the number of solutions evaluated, since it is solver-dependent how many solutions are actually encountered during each "iteration" (e.g. TS evaluates the entire neighbourhood before deciding on a transition whereas RND simply randomly selects a single solution).

Clearly, the more sophisticated heuristics, namely TS and GA, win. With growing problem sizes, SA's and RNS's performance becomes increasingly similar. Surprisingly, RND search can maintain a lead over SA at all network sizes, which is an indicator for too simple an SA implementation. Furthermore, at all network sizes, GA at first converges faster than TS, but in each case TS manages to catch up and improve on the GA's results. However, with growing problem sizes it takes increasingly longer for this to happen. Interestingly, TS shows an step-like behaviour (especially in `net_XL`), which is apparently caused by the diversification through long-term memory and the tabu list. The GA solver shows a more continuous convergence behaviour. Fig. 5.6 shows the minimum, average and maximum convergence plots for GA and TS for the larger networks.

The best result was in all cases produced by the TS solver. Still, the best solutions found likely represent topologies that are too dense. The main reason for this is the large size of the solution space caused by the fact that all searches start from a complete graph topology. Sect. 5.7 will investigate the effectiveness of HLRs to overcome this problem.

5.5 Characteristics of Equivalent Solutions

During the search process, all of the solvers encounter a number of solutions that yield identical topologies (recall Sect. 3.1). It is of great interest to find out how many of those

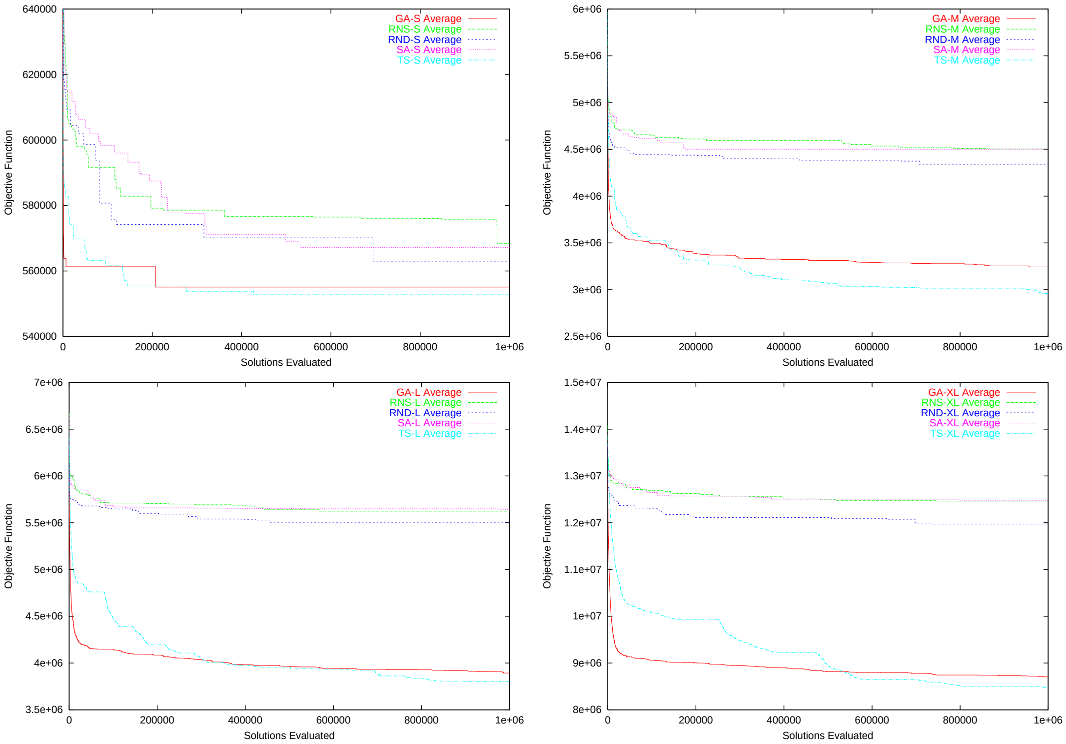

Figure 5.5: Average meta-heuristics convergence for TS, GA, SA, RNS and RND solvers for the four networks.

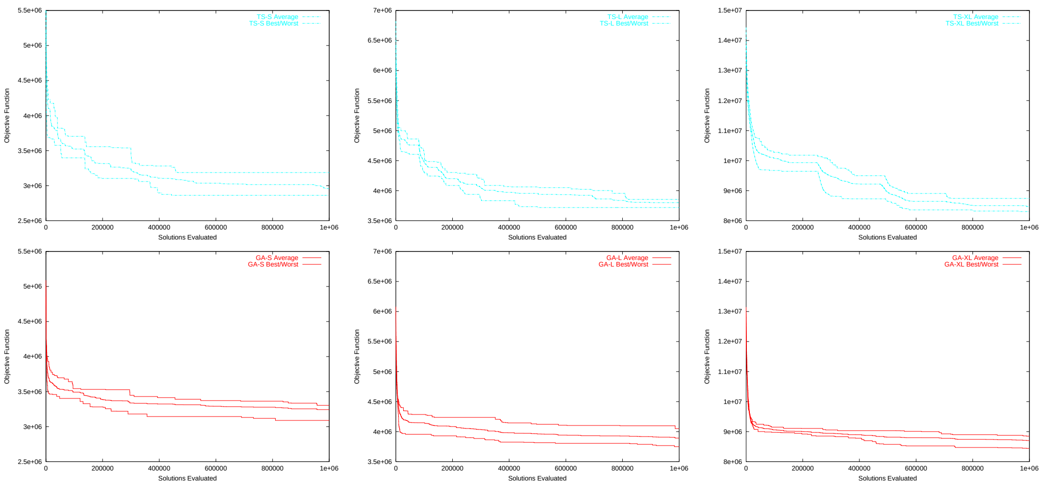

Figure 5.6: Best/Average/Worst performance for TS and GA for `net_M`, `net_L` and `net_XL`.

equivalent solutions exist, and how they are distributed throughout the search space.

Fig. 5.7 illustrates the number of equivalent topologies for the 1000 best solution classes stored in the solution pool (recall Fig. 3.4), again averaged over 5 runs. Again TS and GA easily outperform SA, RND and RNS. Not only do they find better solutions (see above), but also much larger numbers of them. TS is more pervasive for `net_S`, while GA was more successful on the larger networks. While SA could not produce better solutions

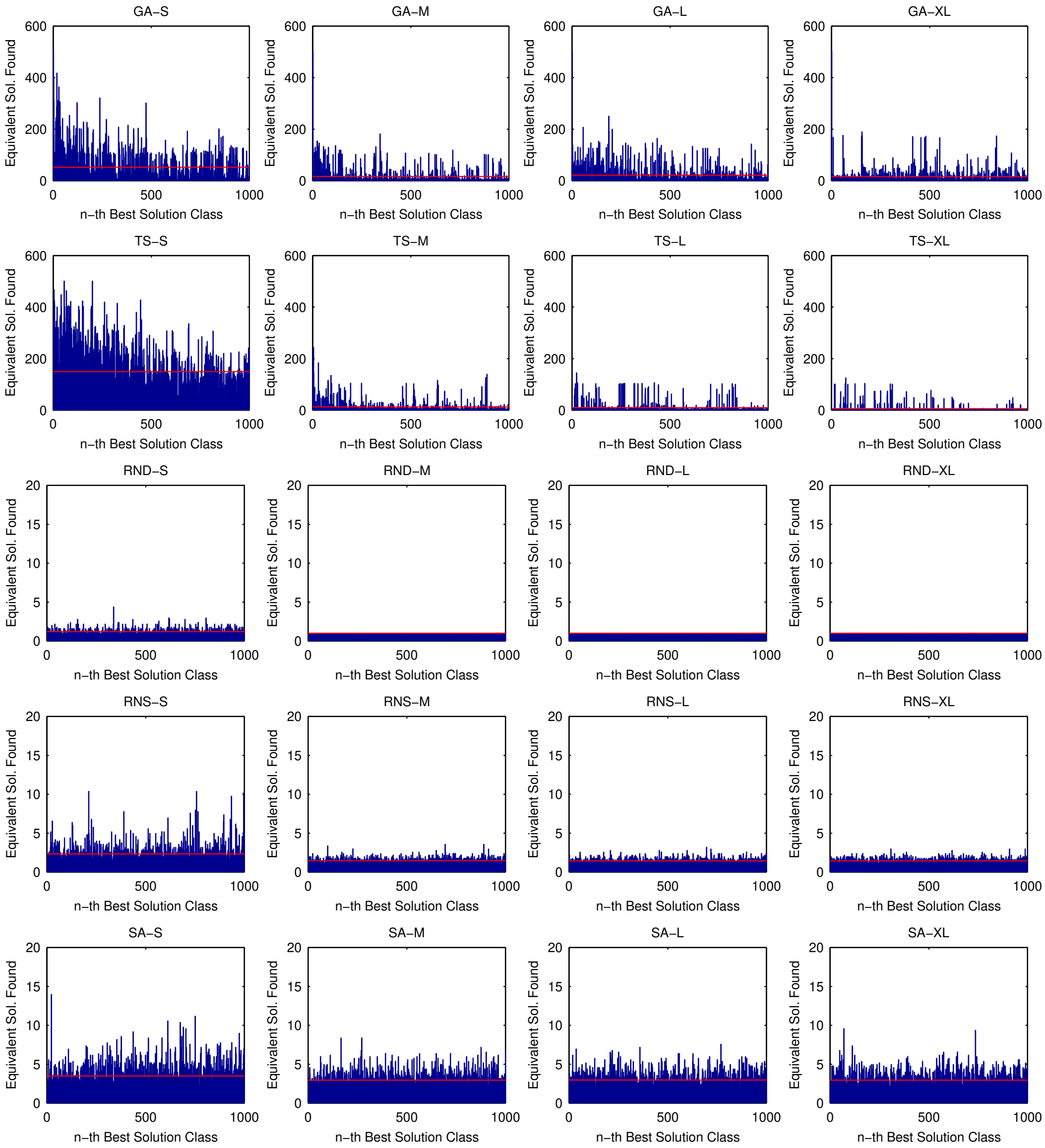

Figure 5.7: Number of equivalent solutions found by the solvers, with a red line showing the average. Notice the different ranges on the y-axis for GA, TS and RND, RNS, SA, respectively. An upper limit of 500 equivalent topologies was used in the solution pool code to avoid memory exhaustion.

than RND and RNS, Fig. 5.7 shows that it does, however, find more equivalent solutions. RNS is slightly superior to RND, which for `net_M`, `net_L` and `net_XL` only found a single topology for each solution.

Sammon Mapping, Search Space Matrices

While the bar chart illustrated the number of equivalent topologies, search space visualization techniques can give a good impression of what that distribution looks like. Those techniques try to map the high-dimensional search space to low-dimensional spaces (usually two- or three-dimensional), while keeping relative distance metrics among points in both spaces as equal as possible. Two techniques need to be mentioned: *Sammon Mapping*[Sam69] and *Search Space Matrices*(SSM)[Col98].

The former is an optimization process itself and does not map the entire high-dimensional space, but rather tries to keep the relative distances of a given set of evaluated solutions equal in the low-dimensional space. In contrast, the latter can potentially map the entire search space on-line, using a high-resolution matrix, providing a formula to directly map items of the high-dimensional space onto their low-dimensional equivalents.

Based on absolute and mean error analysis, [Col98] votes the SSM method to be superior to Sammon Mapping, hence SSM will be used here. There are many different mapping functions conceivable, for a comparison see again [Col98]. In this thesis, a Gray-coded mapping function will be used, which is introduced now.

Gray-coded Mapping

Let s^* be the index encoding of a solution s that is to be mapped to two dimensions. Let $\vec{o}$ be the vector of available OSPF lengths, and $|\vec{o}|$ its length. Let s^*_{even} and s^*_{odd} be the subvectors of s^* produced by eliminating the odd indices and even indices in s^*, respectively[1]. s^*_{even} is used for the x coordinate, s^*_{odd} for the y coordinate, which are calculated by interpreting the subvectors as integer Gray-codes and mapping those Gray-codes to their integer equivalents.

As an example, assume a network has only four links, whose lengths can be chosen from 3 different lengths, i.e. $|\vec{o}| = 3$. Let $s^* = [2201]$, hence $s^*_{even} = [21]$ and $s^*_{odd} = [20]$. The Gray-codes for two indices and 3 integers are shown in Table 5.2.

Integer	0	1	2	3	4	5	6	7	8
Gray-code	00	01	02	12	11	10	20	21	22

Table 5.2: Integer Gray-codes for two indices and three integers.

Therefore, s^* gets mapped to $(x, y) = (\text{Integer}([21]), \text{Integer}([20])) = (7, 6)$. Since entries in a matrix can only have four (non-diagonal) neighbours, the Gray-coded mapping can only be exact for four neighbouring OSPF encodings. Appendix B gives an example.

Fig. 5.8 shows the search space matrices of the solution pools. Clearly, the more sophisticated GA and TS do a better job of focusing the search on specific regions of the search space. For `net_S` TS is especially pervasive. For the larger networks the equivalence classes are so focused that they only show up as a single point in the plots, however, recall from Fig. 5.7 that TS and GA find the largest number of equivalent solutions. Also, notice the size of the matrices necessary to visualize the entire search space. The more randomized searches RND, RNS and SA can only slightly focus in `net_S`, and entirely lose that ability in the larger networks, were the solutions are found mostly at random points in the search space.

[1] Indices start from the right at index zero. Note that when $|E|$ is odd, s^*_{even} and s^*_{odd} do not have the same length.

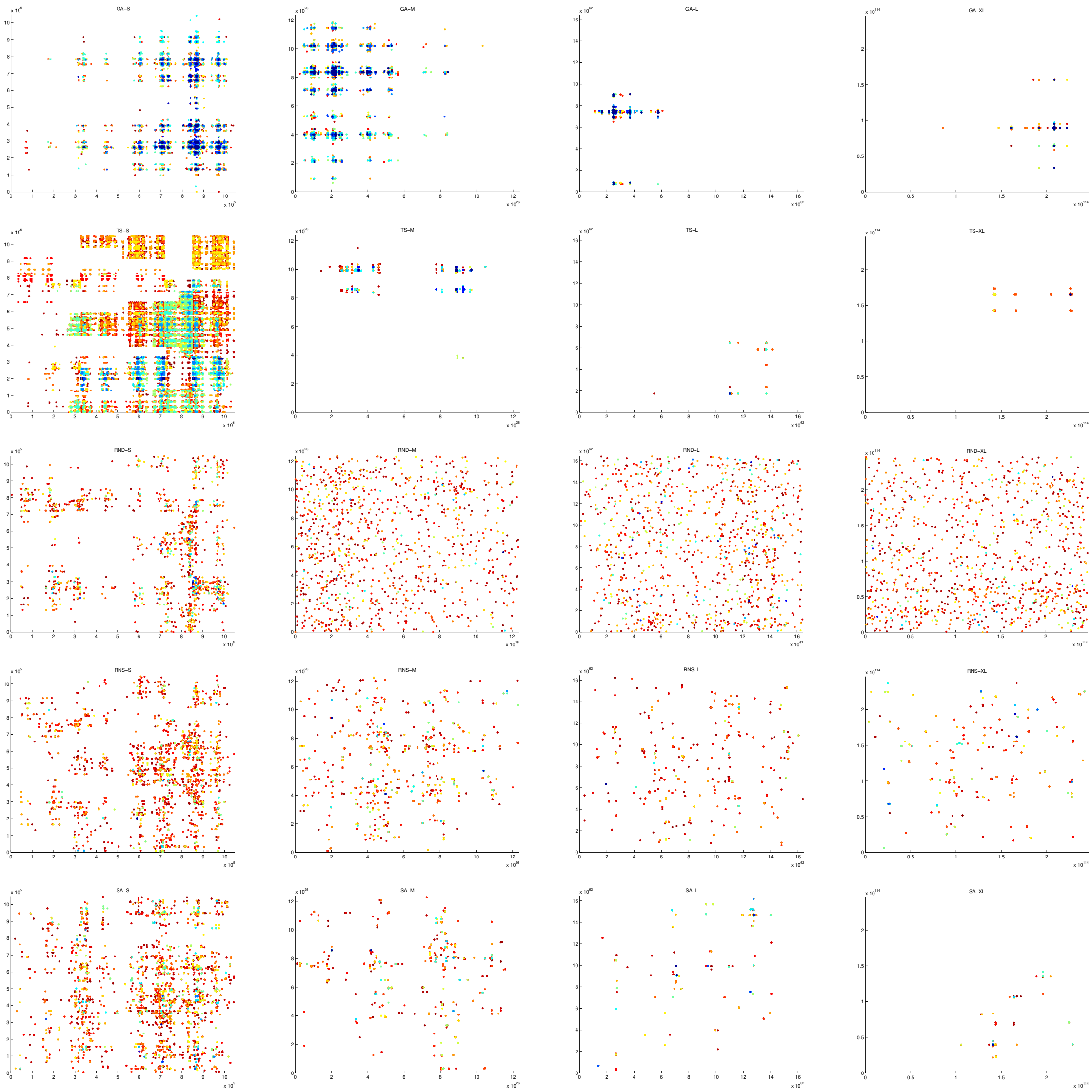

Figure 5.8: Search space matrices for the solution pools of the best runs of GA, TS and RND, RNS, SA on the four networks. The color indicates the value of the objective function, with dark red being the cost of the 1000th-best solution class in each pool and dark blue being the best one. Therefore, all equivalent solutions are shown in the same color.

5.6 Solution Convergence Progress in Search Space

Sect. 5.4 has illustrated the relative performance of the various solvers in terms of solution quality and convergence speed, and Sect. 5.5 has tried to analyze the characteristics of equivalent solutions. What was not shown in those sections is the structure of the path the solvers take through the search space during the convergence process. Using the Gray-code mapping technique introduced in the previous section, this will be investigated now.

To show the path of a convergence process, every solver logged the index encoding of every solution it evaluated. By using a Gray-code mapping to transform these encodings into two-dimensional coordinates, it is then possible to plot each solver's path through the search space. Fig 5.9 shows the results for GA, TS, RND, RNS and SA on `net_M`. The color here indicates the time scale: dark red means the beginning of the search, dark blue the end[2]. Recall from Sect. 5.2 that each heuristic evaluated one million solutions, therefore every plot contains one million dots, some or most of which are hidden under other dots, depending on the heuristic used.

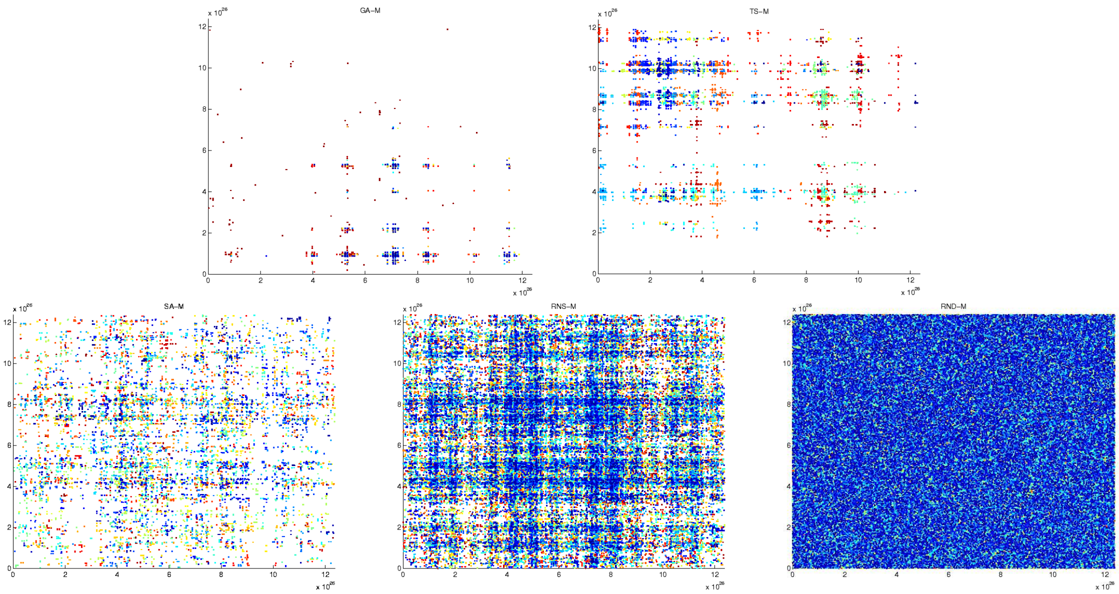

Figure 5.9: Search space matrices for the solution convergence for (top left to bottom right) GA, TS and SA, RNS, RND, on `net_M`.

The quick convergence of GA is clearly visible. There is a large number of red and blue dots visible, whereas the middle of the spectrum is almost entirely missing, as it is in fact covered by the blue dots. For TS the situation is similar, though not quite as distinctive. RNS obviously traverses the entire search space more randomly, remaining a little more focus due to its localizing search structure. The similarity of SA and RNS is also recognizable. As expected, RNS traverses the search space entirely randomly.

5.7 Influence of HLRs

A major issue of the results produced in the previous sections is the fact that all calculations were based on complete graphs, making it hard for the solvers to remove enough links from the topology. While this is arguably mainly a matter of giving the searches enough resources in terms of time and memory, HLRs contain great potential for reducing the resources required. Therefore HLR techniques will be investigated in this section in order to see how they affect the obtained solution quality.

A pure distance-based HLR was employed to remove links from the network until the topology appeared to be a good starting point. Recalling Sect. 4.2, pure distance-based

[2]The plots were generated from different runs than those shown in Fig 5.8.

HLRs delete a link $l = (n_s \triangleright n_t)$ whenever $w_p < \alpha \cdot w_l$ for some path $p(n_s, n_t)$ that does not use l, for some $\alpha > 1$. Fig. 5.10 shows the starting topologies, Table 5.3 shows the values used for α.

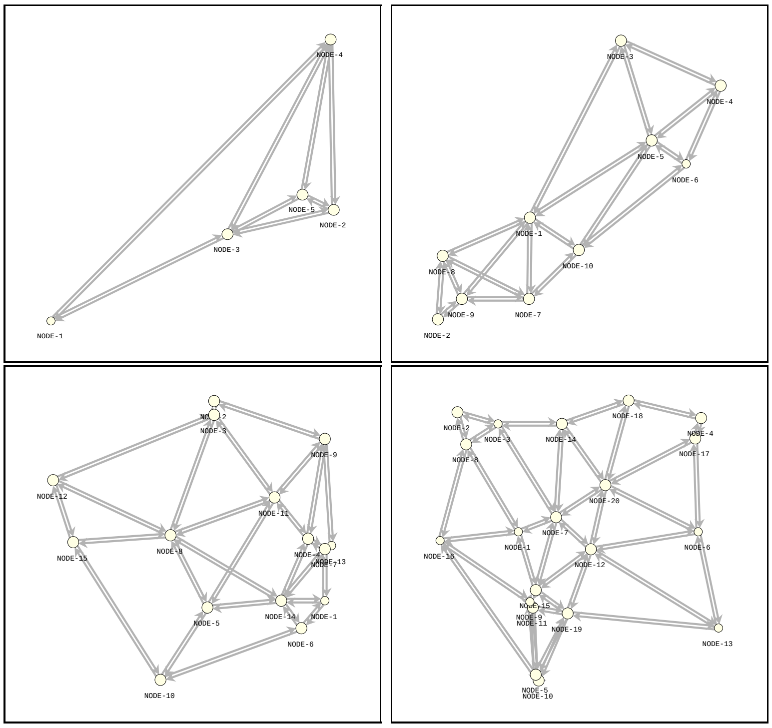

Figure 5.10: The four initial starting topologies after pure-distance HLR application.

Network	`net_S`	`net_M`	`net_L`	`net_XL`
α	1.01	1.15	1.2	1.3

Table 5.3: Parameterization of a distance-based HLR.

Using those initial topologies, all solvers were run again five times for each network. The result is clearly positive: for growing network sizes, HLR application provides substantially better objective function values, compared to the results obtained on the complete graphs. Fig. 5.11 shows the best topologies found. Thicker links mean higher capacity, warmer colors higher utilization. Fig. 5.13 shows the average performance of the TS, GA, SA, RNS and RND solvers for the four HLR-reduced networks.

The performances are similar to those for the full networks, TS again produces the best results. The fact that all solvers found the same class of solutions for `net_S` is a strong indication that it is the best solution for this network, under the given HLR. Table 5.4 shows the performance improvement obtained.

However, one must be careful to not overdo it — once links are deleted from the topology they cannot be considered in the search, which can cause a counterproductive effect when important links are removed. As an example, if `net_S` is reduced to the initial topology

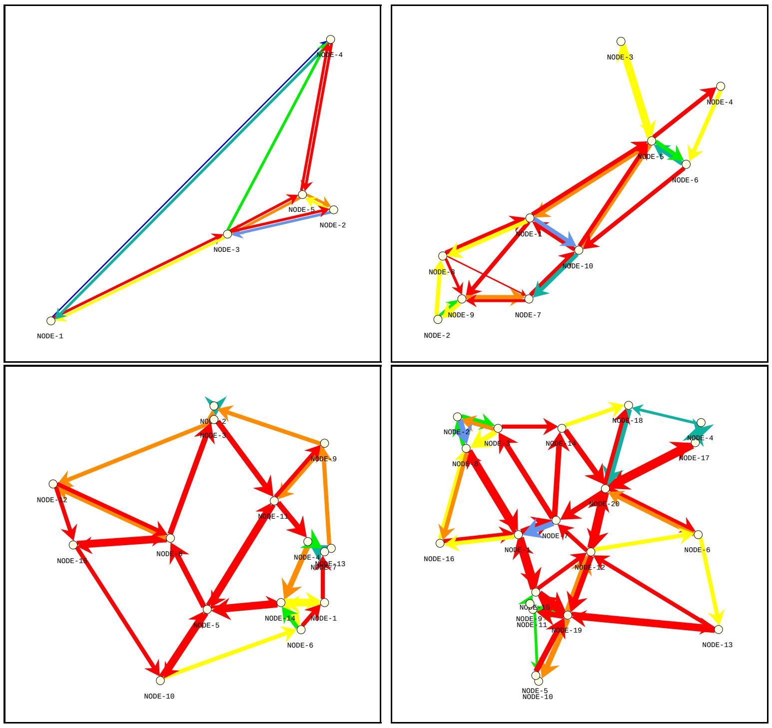

Figure 5.11: The four best topologies found after pure-distance HLR application.

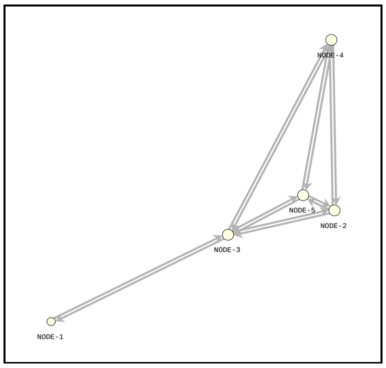

Figure 5.12: An example of a topology in which too many links are deleted.

Network	`net_S`	`net_M`	`net_L`	`net_XL`
Orig. best cost	552264	$2.86223 \cdot 10^6$	$3.71942 \cdot 10^6$	$8.42332 \cdot 10^6$
HLR best cost	547103	$2.22915 \cdot 10^6$	$2.32732 \cdot 10^6$	$4.37980 \cdot 10^6$
Improvement in %	0.93	22.12	37.43	48.00

Table 5.4: Performance improvement gained using a distance-based HLR.

shown in Fig. 5.12, all results are in fact worse than those calculated on the full topology.

In order to obtain better results, an iterative approach seems appropriate. Links which are not used in the best solutions found (compare Fig 5.10 to Fig 5.11) would be deleted, thus further reducing the search space size, and the solvers run again. The need for manual

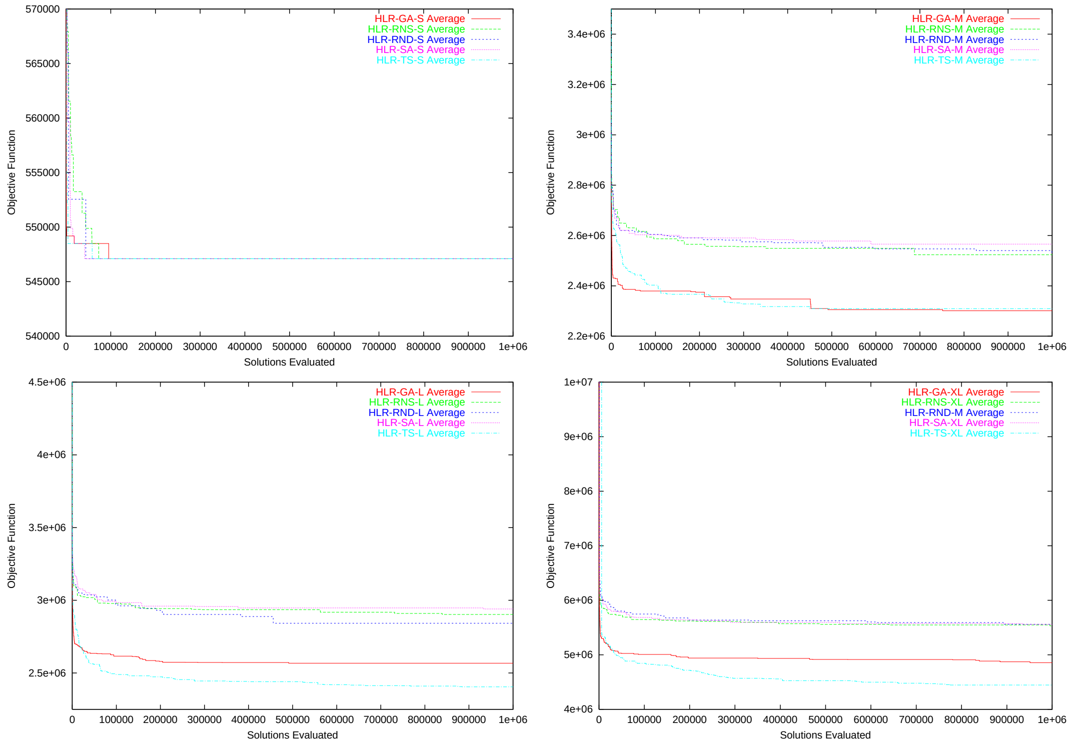

Figure 5.13: Average meta-heuristics convergence for TS, GA, SA, RNS and RND solvers for the four networks, after HLR application.

interaction becomes clear here – the designer will want to be able to test the outcome of experimenting with specific links, which might be hard to be eliminated using a general HLR.

6

Conclusion & Outlook

The thesis gave an overview of the problem setting in the network topology design domain, and compared the relative performance of a Genetic Algorithm (GA), Tabu Search (TS), Simulated Annealing (SA), Randomized Neighbourhood Search (RNS) and Random Search (RND) applied to this field, in terms of runtime, solution quality, convergence speed and search space analysis.

To improve the runtime performance, an incremental all-pairs shortest path algorithm as suggested in [Roh85] was implemented, however, that algorithm is unstable when multiple paths in a network can have the same length, a problem which often occurs in an OSPF-routed network, where link lengths are defined by administrators. Therefore the concept of *virtual paths lengths* was introduced and implemented to overcome this problem. The performance improvements the algorithm provides over non-incremental algorithms is impressive – the IAPSP algorithm was on average about 94 times faster than Floyd-Warshall's algorithm for varying degrees of edge density, and about 64 times faster on a network with 100 nodes, increasing to 188 times for one with 200 nodes.

The IAPSP algorithm was used for those heuristics who easily comply with the incremental paradigm, namely TS, SA and RNS, which obviously has impact on the runtime performances. The GA was found to be the most time-consuming, since it was implemented not only non-incrementally but also needs to create multiple individuals in each iteration. RND is second, since it is also non-incremental. TS is clearly faster as it uses the IAPSP algorithm, but requires some overhead in maintaining the various memory structures, so it is not quite as fast as RNS and SA.

In terms of solution quality, TS performed best on all networks, with the GA being close second. RND is third, RNS and SA are last, performing almost identically. Both TS and the GA produced solutions that are by far superior to those of the remaining solvers, which is understandable, as the former maintain a much higher degree of state-keeping that controls the way the search progresses through the search space. Concerning convergence speed, the GA was found to be superior to TS, which needs increasing amounts of time with growing network sizes to catch up with the GA.

In order to find out more about the distribution of solutions in the search space, a Gray-coded search space matrix as introduced in [Col98] was used. With the exception of the smallest network, the GA was able to find the largest number of solutions that result in identical topologies, closely followed by TS. SA is third here, RND and RNS follow. Concerning the way the searches take through the search space, the search space matrices clearly reflect the quick convergence of the GA, which focuses tightly on a single area. TS's search path is more continuous, but also focuses on a particular area in the end. For the other solvers, the randomized character of the search is the dominant factor.

Table 6.1 summarizes the results.

Feature	GA	RND	RNS	SA	TS
Incremental processing			✓	✓	✓
Runtime	$\ominus$	$\pm$	$\oplus\oplus$	$\oplus\oplus$	$\oplus$
Solution Quality	$\oplus$	$\pm$	$\ominus$	$\ominus$	$\oplus\oplus$
Convergence Speed	$\oplus\oplus$	$\pm$	$\ominus$	$\ominus$	$\oplus$
Number of equiv. Solutions	$\oplus\oplus$	$\ominus\ominus$	$\ominus$	$\pm$	$\oplus$

Table 6.1: Summary of the Meta-heuristics' behaviour.

With regard to the credibility of the resulting topologies, all solvers have difficulty to find good solutions in reasonable time when the search is based on a complete graph. A simple Heuristic Link Reduction (HLR) algorithm was used to remove links from the topology that are unlikely to be beneficial, which increased the quality of the resulting topology in the same timeframe by up to 48%, with an increasing tendency for growing networks.

There are a number of open problems for future research:

- Influence of the set of OSPF lengths.

  The experiments with various sets of OSPF lengths as presented clearly suggest that this choice has great influence on the outcome of the solution. Analyzing OSPF length distributions and their influence on various meta-heuristics could produce interesting results.

- Influence of different cost functions.

  The influence of different cost functions was not addressed in this thesis. The impact of varying the influence of e.g. socket costs and lease costs on the resulting topologies is clearly an important issue, for example regarding the topology of Virtual Private Networks, where socket costs are negligible.

- Other search space visualization techniques.

  Search space matrices are the only technique used in this thesis to visualize search space characteristics. Other techniques, particularly Sammon Mapping, would be an interesting field of study.

- Extending IAPSP applications.

  The use of the IAPSP algorithm could be extended to e.g. GA by performing multiple incremental updates instead of a single, non-incremental change that modifies a set of link lengths. Investigating various implementations using e.g. threshold values for the number of changes up to which the IAPSP is used could yield substantially faster algorithms.

- Reducing search space redundancy.

  If an efficient way is found to make the IAPSP algorithm aware of the amount by which link lengths need to be changed in order to cause path detours, the size of the search space would be reduced drastically since a lot of redundancy would be eliminated.

A

UML Class Diagram

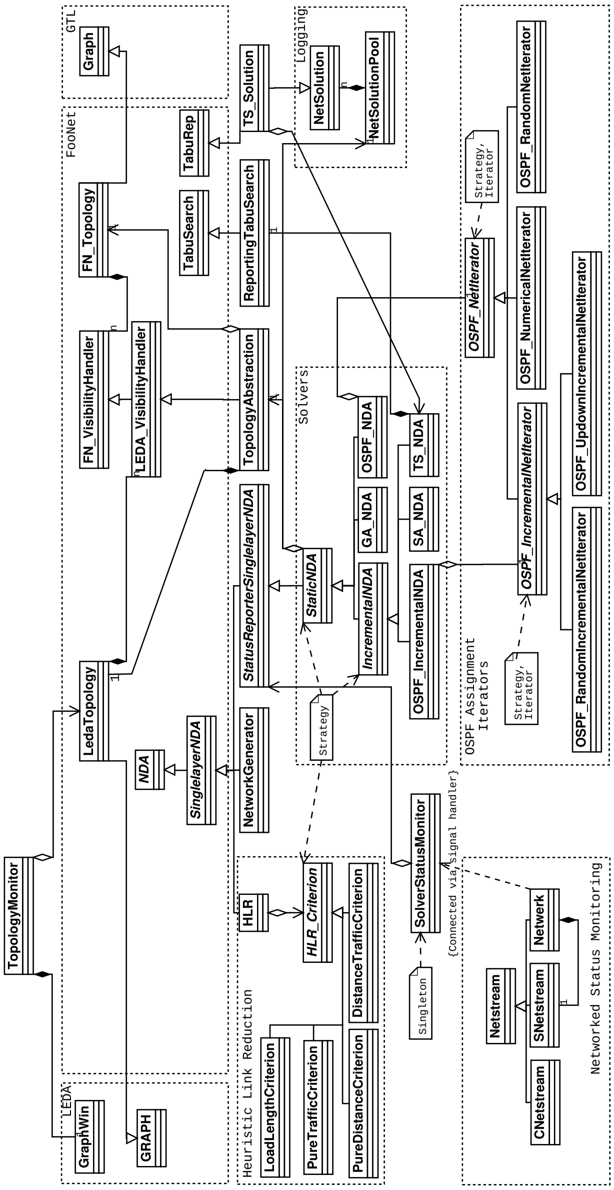

Figure A.1: UML class chart of the source code package.

B

Search Space Matrix Example

0000	0010	0020	0030	1030	1020	1010	1000	2000	2010	2020	2030	3030	3020	3010	3000
0001	0011	0021	0031	1031	1021	1011	1001	2001	2011	2021	2031	3031	3021	3011	3001
0002	0012	0022	0032	1032	1022	1012	1002	2002	2012	2022	2032	3032	3022	3012	3002
0003	0013	0023	0033	1033	1023	1013	1003	2003	2013	2023	2033	3033	3023	3013	3003
0103	0113	0123	0133	1133	1123	1113	1103	2103	2113	2123	2133	3133	3123	3113	3103
0102	0112	0122	0132	1132	1122	1112	1102	2102	2112	**2122**	2132	3132	3122	3112	3102
0101	0111	0121	0131	1131	1121	1111	1101	2101	2111	2121	2131	3131	3121	3111	3101
0100	0110	0120	0130	1130	1120	1110	1100	2100	2110	2120	2130	3130	3120	3110	3100
0200	0210	0220	0230	1230	1220	1210	1200	2200	2210	2220	2230	3230	3220	3210	3200
0201	0211	0221	0231	1231	1221	1211	1201	2201	2211	**2221**	2231	3231	3221	3211	3201
0202	0212	0222	0232	1232	**1222**	1212	1202	2202	**2212**	**2222**	**2232**	3232	**3222**	3212	3202
0203	0213	0223	0233	1233	1223	1213	1203	2203	2213	**2223**	2233	3233	3223	3213	3203
0303	0313	0323	0333	1333	1323	1313	1303	2303	2313	2323	2333	3333	3323	3313	3303
0302	0312	0322	0332	1332	1322	1312	1302	2302	2312	**2322**	2332	3332	3322	3312	3302
0301	0311	0321	0331	1331	1321	1311	1301	2301	2311	2321	2331	3331	3321	3311	3301
0300	0310	0320	0330	1330	1320	1310	1300	2300	2310	2320	2330	3330	3320	3310	3300

Figure B.1: A search space matrix for 4 different OSPF lengths and 4-digit strings. The neighbourhood of 2222 is in boldface.

Bibliography

[Bel58] R.E. Bellman. On a Routing Problem. *Quart. Appl. Math.*, 16:87–90, 1958.

[Ber92] Arthur M. Berman. *Lower and Upper Bounds for Incremental Algorithms.* PhD thesis, Rutgers State University of New Jersey, 1992.

[BG92] Dimitri Bertsekas and Robert Gallanger. *Data Networks.* Prentice Hall, 1992.

[Cah98] Robert S. Cahn. *Wide Area Network Design - Concepts and Tools for Optimization.* Morgan Kaufmann Publishers Inc., 1998. The Morgan Kaufmann Series in Networking.

[Cay89] A. Cayley. A Theorem On Trees. *Quarterly Journal of Mathematics*, 23:376–378, 1889.

[CDG99] D. Corne, M. Dorigo, and F. Glover. *New Ideas in Optimization.* McGraw-Hill, 1999.

[Cisa] Cisco Inc., `http://www.cisco.com/cpress/cc/td/cpress/design/ospf/on0407.htm`. *Designing & Implementing an OSPF Network*, white paper edition.

[Cisb] Cisco Inc., `http://www.cisco.com/cpress/cc/td/cpress/fund/ith2nd/it2441.htm`. *Open Systems Interconnection (OSI) Routing Protocol*, white paper edition.

[Cis91] Cisco Inc., `http://www.cisco.com/warp/public/103/5.html`. *An Introduction to IGRP*, white paper edition, 1991.

[Col98] Trevor D. Collins. *The Application of Software Visualization Technology to Evolutionary Computation: A Case Study in Genetic Algorithms.* PhD thesis, Knowledge Media Institute, The Open University, UK, 1998.

[CSI] CSIC, `http://www.rediris.es/red/index.en.html`. *RedIRIS.*

[Dij59] E.W. Dijkstra. A Note on two Problems in Connection with Graphs. *Numerische Mathematik*, 1, 1959.

[Doa96] Matthew B. Doar. A Better Model for Generating Test Networks. In *IEEE Global Telecommunications Conference/GLOBECOM '96*, 1996.

[ED85] P. Ein-Dor. Grosch's Law re-visited: CPU Power and the Cost of Computation. *Comm. ACM*, 28(2):142–151, 1985.

[EG85] S. Even and H. Gazit. Updating Distances in Dynamic Graphs. *Methods of Operations Research*, 49:371–387, 1985.

[ES96] Reuven Elbaum and Moshe Sidi. Topological Design Of Local-Area Networks Using Genetic Algorithms. *IEEE/ACM Trans. Networking*, 4(5):766 – 778, 1996.

[EW66] L. Esau and K. Williams. On Teleprocessing System Design: A Method for Approximating the Optimal Network. *IBM System Journal*, 5, 1966.

[Fis00] Volker G. Fischer. *Evolutionary Design of Corporate Networks under Uncertainty*. PhD thesis, Technische Universität München, Institut für Informatik, 2000.

[Flo62] F.W. Floyd. Algorithm 97: Shortest Paths. *Communications of the ACM*, 5:345–FIXME, 1962.

[Fox92] Bennet L. Fox. Uniting Probabilistic Methods For Optimization. In *Proceedings of the Conference on Winter simulation*, pages 500 – 505, 1992.

[GC97] M. Gen and R. Cheng. *Genetic Algorithms and Engineering Design*. John Wiley & Sons, 1997.

[Glo86] F. Glover. Future Paths for Integer Programming and Links to Artificial Intelligence. *Computer and Operations Research*, 13:156–166, 1986.

[Glo97] Fred Glover. *Tabu Search*. Kluwer Academic Publishers, 1997.

[GLR91] F. Glover, M. Lee, and J. Ryan. Least-cost Network Topology Design for a new Service: An Application of Tabu Search. *Annals of Operations Research*, 33(1-4):351–362, 1991.

[Gro53] H.A. Grosch. High Speed Arithmetic, The Digital Computer as a Research Tool. *J. Optical Society of America*, 4(4):306–310, 1953.

[HC98] E. Hadjiconstantinou and N. Christofides. An Efficient Implementation of an Algorithm for Finding K Shortest Simple Paths. *Networks*, 34:88–101, 1998.

[Hed88] C. Hedrick. RFC 1058: Routing Information Protocol. Technical report, Rutgers University, 1988.

[Hoc95] Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, 1995.

[Ker93] Aaron Kershenbaum. *Telecommunications Network Design Algorithms*. McGraw-Hill, 1993.

[KJV83] S. Kirkpatrick, C.D. Jelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.

[KKG91] A. Kershenbaum, P. Kermani, and G. Grover. Mentor: An Algorithm for Mesh Network Topological Optimization and Routing. *IEEE transactions on communication*, COM-39, 4:503–513, 1991.

[Kru56] J.R. Kruskal. On the Shortest Spanning Subtree of a Graph and the Travelling Salesman Problem. *Proceedings of the American Mathematical Society*, 7(1), 1956.

[Mil83] D. Mills. RFC 891: DCN Local Network Protocols. Technical report, DARPA Network Information Center, 1983.

[Moy89] J. Moy. RFC 1131: Open Shortest Path First. Technical report, Proteon, Inc., 1989.

[Moy98] J. Moy. RFC 2328: OSPF Version 2. Technical report, Ascend Communications, Inc., 1998.

[MS81] Andranik Mirzaian and Kenneth Steiglitz. A Note on the Complexity of the Star-Star Concentrator Problem. *IEEE Transactions on Communications*, COM-29(10):1549–1552, October 1981.

[Nar89] T. Narten. Internet Routing. In *Symposium proceedings on Communications architectures and protocols*, volume 16, pages 271–282, 1989.

[Pri57] R.C. Prim. Shortest Connection Networks and some Generalizations. *Bell System Technical Journal*, 36, 1957.

[Ree93] Colin R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems.* Blackwell Scientific Publications, 1993.

[RK87] D. A. Reed and C. Kim. Packet Routing Algorithms for Integrated Switching Networks. In *Proceedings of the ACM conference on Measurement and modeling of computer systems*, pages 7–15, 1987.

[Roh85] Hans Rohnert. *A Dynamization of the All Pairs Least Cost Path Problem*, volume 182 of *Lecture Notes in Computer Science*, pages 279–286. Springer-Verlag Berlin Heidelberg New York, 1985.

[Sam69] John W. Sammon. A Nonlinear Mapping for Data Structure Analysis. *IEEE Transactions on Computers*, C-18:401–409, May 1969.

[Suu74] J. W. Suurballe. Disjoint Paths in a Network. *Networks*, 4:125–145, 1974.

[Tan97] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall International, 1997.

Index