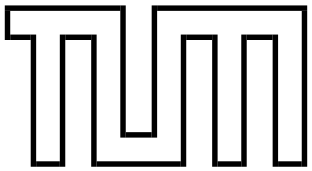

FAKULTÄT FÜR
WIRTSCHAFTSWISSENSCHAFTEN

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Dissertation

Replica Placement in Content Delivery Networks: Model and Methods

Dipl.-Inf. Univ. André Dahlmann

TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Technische Dienstleistungen und Operations Management

Replica Placement in Content Delivery Networks: Model and Methods

Dipl.-Inf. Univ. André Dahlmann

Vollständiger Abdruck der von der Fakultät für Wirtschaftswissenschaften der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Wirtschaftswissenschaften (Dr. rer. pol.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Florian von Wangenheim
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. Rainer Kolisch
	2.	Univ.-Prof. Dr. Martin Bichler

Die Dissertation wurde am 16.02.2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Wirtschaftswissenschaften am 11.05.2011 angenommen.

Contents

List of Tables

List of Figures

Chapter 1

The Problem of Replica Placement in Content Delivery Networks

Content delivery networks (CDN) (see Bartolini et al. [9], Dilley et al. [18], Pallis and Vakali [53], Pathan et al. [54], Peng [56], Rabinovich and Spatschek [60], Vakali and Pallis [72], Verma [73]) replicate data in the Internet to reduce the latencies for users requesting the data. They are of enormous importance for the usability of websites. Companies have to ensure that their web services are always reachable and performing well. Especially for large companies with high traffic on their websites this typically can not be achieved with a single server run by the company itself. The huge amount of requests originating from all over the world has to be processed by several servers which should not be concentrated on one location but widely distributed. Doing this has several effects. The requests can be handled by near servers so that the distance each data package has to pass and thus the time delay is short. Furthermore, the requests are split among several servers to avoid overloaded servers with high response times and they are split among different routes such that congestion on data links in the Internet can be reduced.

A content delivery network provider offers exactly this service. He hosts rich and thus bandwidth consuming content or even the complete website of a customer, processes the requests and delivers the content for him. That is why CDN customers, typically companies, are sometimes referred to as

content providers for the CDN. An end–user visiting the website and thus generating the requests is called client. A CDN has a large amount of servers widely spread over the world. These servers and their connections through physical or virtual links can be viewed as an overlay network on the Internet (see Andersen et al. [7], Jannotti et al. [28], Jeon et al. [29], Lazar and Terrill [45], Pathan et al. [54], Rahul et al. [62], Savage et al. [65], Shi and Turner [66, 67], Verma [73], Wang et al. [76]).

Let us assume the network in Figure 1.1 to be a small part of the Internet. The circles represent the servers of the CDN, the edges represent their interconnections and together they form the overlay network. The dotted edges are connections to nodes outside the CDN. The square node 0 represents the origin server, i. e. the server of the customer and content provider. The two laptops represent clients. If the dark nodes in the figure, i. e. servers 2, 3 and 4, additionally store replicas of the original data from node 0, requests from the two clients can be answered faster. A request for a file is typically routed to the nearest server which belongs to the CDN. Thus, we need to consider just the overlay network of the CDN plus the origin server. Requests from clients can be aggregated at the nearest CDN server, i. e. at server 5 and 2 in our example. Inside the CDN each request is then redirected to the nearest server which stores a replica of the data and is available, i. e. from server 5 to server 3 or 4 (see Bartolini et al. [9], Dilley et al. [18], Pallis and Vakali [53], Vakali and Pallis [72]). A lot of work has been done in this field of request redirection and load balancing (e. g. Cardellini et al. [11, 12], Guyton and Schwartz [22], Kangasharju et al. [32], Plaxton et al. [57], Rabinovich and Spatschek [60], Rangarajan et al. [63], Ranjan [64], Verma [73]) and we do not consider it in our work.

Regarding the transfer technique inside the CDN we distinguish between the content delivery and the placement. As a server handles the requests independently and each request has to be served immediately, the content delivery is done via traditional point–to–point connections. With this so–called ‘unicast’ approach all answers are sent independently from each other and the individual transfers can not be combined. This still holds if two or more requests from the same client arrive at the same server in the same period – even if the period is infinitesimal short (see Cidon et al. [16]). As an example, consider Figure 1.1 with the assumption that all edges have a weight of 1. If only the origin server 0 stores the data requested by the two

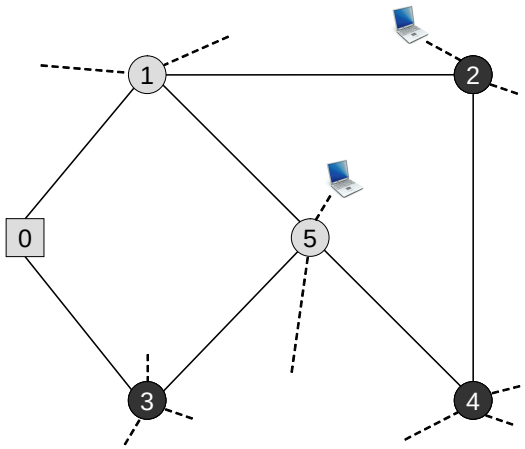

Figure 1.1: Simplified structure of an overlay CDN

clients, we get delivery cost of 4. The client whose request is aggregated on server 2 gets the file via node 1 at cost 2. The other client, aggregated on server 5, gets the file at cost 2 either via node 3 or also via node 1. If the dark nodes in the figure, i. e. servers 2, 3 and 4, store replicas of the object we only have cost of 1 to deliver the object from server 3 or 4 to node 5.

In contrast, the placement, i. e. the transfer of the data to all designated servers, is done simultaneously. Hence, point–to–multipoint connections can be used, the so–called 'multicast' transfers (see Frank et al. [21], Paul [55], Wittmann and Zitterbart [78]). This leads to considerably reduced cost and is by far more efficient (see Wolfson and Milo [79]). If our figure represents the first period of a planning horizon and the data is new to the CDN, the origin server 0 is the only host in the beginning of the period. Replicas of the object should be transferred to the dark colored servers 2, 3 and 4. In a unicast approach three transfers using the shortest paths from the origin server to each designated replica server would be used. With edge weights of 1 we would use, e. g., the unicast paths shown in Figure 1.2 with placement cost of 6.

In a multicast approach an edge induces cost only once, no matter how often it is used in a unicast sense. The object can be copied on every passed replica server and then be forwarded several times. Thus, it is often superior to bundle several unicast transfers to one multicast edge. In our example the optimal multicast route could be just the multicast path shown in Figure 1.2

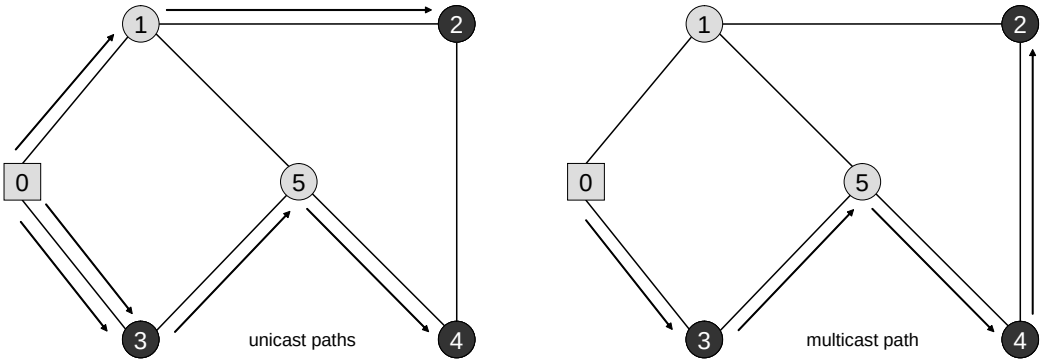

Figure 1.2: Unicast and multicast placement paths

with cost of 4. With a multicast approach the data flow during the placement uses edges in a subtree of the network. As in our example, this tree is typically different from the set of individual shortest paths.

A contract between a company and a CDN provider typically contains a service level agreement (SLA). It defines for example that a given percentage of all requests has to be handled within a given timespan, the permitted latency. Such a guaranteed quality of service (QoS) can not be achieved with a caching approach which is often used to reduce latencies and traffic in the Internet. Web caches basically store copies of files for a certain timespan when they are delivered. Succeeding requests for a cached file are not forwarded to the origin server again but directly served by the cache. As a cache stores the files not until they are requested, these approaches are not suitable if SLAs are in place. Hence, CDNs and replica placement algorithms are needed (see Karlsson and Mahalingam [36]).

As there is not enough storage space to store all files of all customers on all servers, the CDN provider has to decide which files are stored on which locations. Of course, this decision depends on the frequency of the requests and their geographical distribution. And as the request rate, i. e. the number of requests per period, changes over time, the optimal assignment of the replicas is dynamic. An optimal assignment is one with minimum cost which fulfills the service level agreements.

In this thesis we consider the aforementioned problem with three cost types and a single object to be placed. This object can be seen, for example,

as the complete website of one customer of the CDN provider. The first type of cost are the storage cost. They occur for each server and period if a replica of the object is stored on the server in that period. The second cost type is for the placement, the cost for transferring copies of the object to the designated replica servers in each period. The last type are the cost for delivering the data from the replica servers to the points of requests. They occur for each individual transfer of data to a client

This thesis is organized as follows. In Chapter 2 we provide an overview of the work done so far in the field of replica placement and content delivery networks. We classify the models and solution procedures regarding the assumptions made and approaches chosen. We finally put the model presented in this thesis in perspective to existing approaches.

In Chapter 3 we present a new MIP formulation which allows a more realistic and flexible modeling of the problem than the modeling approaches which have been proposed so far. We show the complexity of the problem and prove that the latter is NP–hard. Finally, we present a relaxation and an improved lower bound which is used for evaluating the proposed solution approaches.

In Chapter 4 we first describe the framework we developed to incorporate the different algorithms, especially the solution representation and solution evaluation. Afterwards we propose several heuristic solution approaches for the problem since optimal solutions can not be obtained in reasonable time with a standard MIP–solver such as CPLEX. We propose an efficient simulated annealing heuristic, several variants of a variable neighborhood search and various benchmark heuristics from the literature.

Chapter 5 presents the computational study. We describe our experimental setup and the design. We compare the results of our algorithms with our lower bound as well as the benchmark heuristics. We show promising results regarding solution time and quality of our algorithms, especially for an application in the real world. We then analyze the impact of the parameters on the solutions.

Finally, we conclude with a summary and an outlook in Chapter 6.

Chapter 2

Literature Review

2.1 Previous approaches

The very first treatment in the literature of a basically similar problem was the classical file allocation problem (FAP) (see Chu [15], Dowdy and Foster [19]), dealing with the allocation of file copies in a distributed filesystem or their assignment to storage nodes in a computer network.

Later research focused on web proxies and web caches which can be seen as the origin of content delivery networks (see Baentsch et al. [8], Pathan et al. [54], Rabinovich and Spatschek [60]). Typical problems are the placement of web proxies or web caches in the Internet to reduce overall traffic in the network or the access latency (see Korupolu et al. [40], Krishnan et al. [43], Li et al. [46]). In these first approaches the authors assumed very simple network architectures like line, ring, tree or other hierarchical topologies.

In a different context Jamin et al. [26] tackled a similar placement problem with general topologies. Later the work was generalized to the placement of web server replicas in content delivery networks by Qiu et al. [59] and Jamin et al. [27]. Radoslavov et al. [61] did a slightly refined study based on the work of Jamin et al. [27].

Cidon et al. [16] addressed the problem with a hierarchy tree instead of a general topology. They made an important step towards a realistic model: Up to their approach the problems were typically modeled as some sort of

minimum k-median [43, 46, 59] or minimum k-center problems [26, 27]. These approaches minimize latency or bandwidth consumption. They constrain the maximum number of replicas of the object stored in the CDN or assume that the number of replicas is known in advance. Hence, the number of replicas is obtained separately from the actual assignment and is not optimized. In contrast, Cidon et al. [16] minimize the sum of storage and communication (bandwidth) cost for the delivery. They do not bound the number of replicas and optimize it implicitly.

Kangasharju et al. [33] do not bound the number of replicas as well and they consider general topologies. The drawback here is that they neither minimize the number of replicas nor incorporate replication or maintenance cost. But they were one of the first to consider multiple objects. Therefore, they constrain the storage space on each node. They minimize the average distance a request must traverse.

One of the first approaches towards quality of service (QoS) oriented content delivery is by Chen et al. [13]. While not considering general topologies, they use a more realistic objective function in another sense: They minimize the number of replicas while meeting capacity constraints (load, bandwidth or storage) for the servers and latency constraints for the clients. Hence, a maximum latency for the clients is assured. Although they term their approach dynamic they formulate the problem for independent periods.

To the best of our knowledge Karlsson and Karamanolis [34] are the first to model a real dynamic, i. e. multi–period version of the problem. Their model considers multiple objects, general topologies and specific guaranteed QoS goals. The QoS goals are modeled to meet realistic service level agreements (SLA). They minimize the sum of storage and replica creation cost.

The drawback of their approach is the assumption of unit costs of 1 for the storage and the replica creation. Although they state that the replica creation cost represent the cost of network resources used to create a replica, in their model these cost do not depend on the distance to the origin of the copy or the bandwidth used. Moreover, the delivery cost are not considered. In a later paper, Karlsson and Karamanolis [35] expanded their work to average performance constraints. They force the average latency to be less or equal to a specific threshold. However, because of the averaging this approach can not provide performance guarantees as they are required by SLAs.

Tang and Xu [69] also concentrate on QoS requirements and add performance guarantees. They minimize storage cost and maintenance cost for keeping the replicas consistent with changes of the original data. They neither consider the multi–period nor the multi–object case.

Aioffi et al. [5] provide another contribution with a dynamic model, optimizing several periods at once. Their objective function minimizes the total traffic over the network, generated by delivery to the client, replica creation and replica maintenance. They do not consider storage cost and service levels.

2.2 Comparison to our approach

In contrast to the work done so far we present an integrated model for the simultaneous solution of the replica assignment, placement and content delivery. Our model was inspired by Sickinger and Kolisch [68]. We consider service level agreements, the multi–period case and incorporate multicast transfers. The approach differs from what has been presented in the literature in several ways.

First, our approach differs from others with respect to the underlying topology. In contrast to [13, 16, 43, 46] we do not require a special topology but consider a general topology as suggested as an important issue by Cidon et al. [16]. Furthermore, we do not set or bound the number of replicas as it is done by [26, 27, 43, 46, 59].

Second, we do not consider just one period or a static problem like all of the referenced papers except [34, 35]. Cidon et al. [16] find that additional work is needed to take into account the dynamic characteristic of the Internet. As the demand changes over time the optimal replica assignment can also be different in each period. The optimal multi–period assignment can not be found by solving the single–period problems independently because of the cost for the placement process we need to incorporate. To illustrate this assume that a specific replica server needs to store a replica in the periods t and $t + 2$. Depending on the cost coefficients it might be better to store an additional replica in period $t + 1$ instead of deleting it and replicating it to the server again because the latter alternative has higher placement cost.

Similar assumptions of multicast transfers can be found in Jia et al. [30], Kalpakis et al. [31], Unger and Cidon [71], Xu et al. [80] for content updates or write operations. But to the best of our knowledge we are the first to model realistic multicast transfers. We assume that the content of the origin server does not change over the entire planning horizon. Hence, we use multicast transfers for the placement, i. e. updating the assignment. While, e. g., Tang and Xu [69] do consider multicast transfers for their content updates, they assume a fixed update distribution tree for the multicast to be given. In our model, the multicast tree is implicitly optimized for each placement.

We assume the placement to be done in the very beginning of each period, as modeled by Karlsson and Karamanolis [34, 35]. In contrast to Karlsson and Karamanolis [34] we address the problem with cost for the use of the edges that depend on a distance metric. In addition to the placement cost we consider delivery cost.

Yang and Fei [81] state that the traffic for distributing objects to the designated replica servers can not be ignored, especially because of the huge number of files and the more and more common multimedia files of huge sizes. They conclude that the storage cost as well as the delivery cost and the cost for updating the assignment have to be considered, which is what we do with our model. Nguyen et al. [50] incorporate storage and delivery cost but they do only consider a static problem and therefore do not consider cost for updating the assignment. Furthermore, they just set a bound on the maximum latency instead of modeling SLAs.

Only little other work has been done with respect to QoS, e. g. [13, 29, 34, 69]. Except [34] all these papers only constrain maximum latencies like Nguyen et al. [50]. They do not model realistic SLAs as we do and furthermore they consider just the static case. Karlsson et al. [37] as well as Cidon et al. [16] state that the consideration of systems with constrained resources and guaranteed QoS properties is an important area for further research.

When it comes to solution methods optimal algorithms have only been proposed for special topologies (see [16, 43, 46]). Most of the authors propose simple heuristics (see [5, 27, 33, 59, 61, 69]), especially for more general topologies. Karlsson and Karamanolis [34, 35] put the most effort into a realistic model for the different aspects but they do not deal with new solution methods for the problem. To the best of our knowledge we are the first to tackle such a problem with metaheuristics.

Chapter 3

Model

3.1 Model assumptions

Before we present our MIP formulation of the problem we make some basic assumptions.

First, we consider only one object, as it is done in most of the previous work, e. g. [13, 16, 26, 27, 29, 43, 46, 59, 69]. This can be seen as one single file or the complete web page of one customer. Some companies out–house only bandwidth intensive files like videos or other multimedia content. Others hand their complete Internet presence over to the CDN provider. As we consider only one object we do not incorporate the size of the object. Thus, we do not need to model the bandwidth of the edges and storage capacities on the servers. Consequently, our storage cost have to be mainly seen as opportunity cost, besides the real cost for storing a replica on a server. There are two reasons why storage cost need to be considered. On the one hand, they reflect the fact that in the real world we cannot store all files of all customers on every replica server due to limited storage capacity. On the other hand, the more replicas of one object exist, the more complex and expensive gets the maintenance of the files (see Aggarwal and Rabinovich [2], Chen et al. [14], Jamin et al. [27], Jeon et al. [29], Shi and Turner [66], Tang and Xu [69]). For example if the original file changes, all the replicas need to be refreshed in order to retain consistency.

The placement cost are also partly opportunity cost. There are real cost for using data links in the Internet for the transfer. But we also want to model the fact that in reality the bandwidth of the edges is limited and valuable. The bandwidth should not be used excessively for shifting replicas in each period. We assume that the placement is done in the beginning of each period and takes no time.

Second, we assume a deterministic time–varying demand per period.

Third, we have assumptions concerning routing and transfer techniques. We do not consider the routing outside the overlay network of the CDN as it can not be easily affected by the CDN provider. Hence, we assume that a request of a client is directed into the CDN to the nearest server using the shortest path. Likewise we assume that each delivery of an object from a replica server to a client is done via the shortest path in the network. These assumptions of shortest paths are prevalent (see [5, 14, 26, 27, 29, 36, 40, 43, 50, 69]).

To ease the model and reduce the amount of data needed we assume that the cost for the transfers is proportional to the length of the used path. We weight the network distances with cost coefficients for placement and delivery to get the corresponding cost values. In reality the cost for the data transmission depend mainly on the used bandwidth as the data links have limited capacities. Hence, they depend on the amount of data transferred. As we do not consider the size of the object we cannot use this criterion. A second important factor is the passed distance. In reality this could be the number of autonomous systems traversed (AS–hops, see Kangasharju et al. [33]). The distance as employed in our model correlates highly with the number of AS–hops. In general, our model works with any metric or can at least be easily adapted. If more realistic values for the edge weights are available, they can be used right away.

Finally we assume that deleting an object from a replica server can be done at zero cost (see, e. g., Karlsson and Karamanolis [35]).

3.2 Model description

Network topology. We consider a general network topology representing the CDN overlay network. Data flow is possible in both directions of an edge in the network. As we need directed edges to model the problem we substitute each undirected edge with two arcs directed opposite each other and both having the same weight as the edge. Let $\mathcal{V}$ be the set of nodes, $V := |\mathcal{V}|$, and let $\mathcal{A}$ be the set of arcs in a topology. $\delta_{i,j}$ is the weight of arc $(i, j) \in \mathcal{A}$, e. g. the distance between the nodes.

Replica storage. One node $v_0 \in \mathcal{V}$ is the origin server, the one and only server which stores the original object initially in the dummy starting period $t = 0$. $\mathcal{V}_s \subseteq \mathcal{V} \setminus \{v_0\}$ is the set of replica servers. They are part of the content delivery network and can store a replica of the considered object. Each replica server $s \in \mathcal{V}_s$ has a cost coefficient α_s which represents the cost for storing a replica in one period.

For an easier notation of the model we require that the replica servers of the CDN, i. e. the nodes $s \in \mathcal{V}_s$, form a connected subgraph in the topology. This typically holds for real CDN topologies because of the overlay network characteristic we already mentioned.

Placement. As the optimal assignment of the replicas changes over time, we consider a placement being processed at the beginning of each period $t \in \mathcal{T}$. A replica server can receive a replica of the object in period t only from a replica server which stored a replica in period $t-1$ or from the origin server v_0. Normally, at the beginning of period $t = 1$ the only source of the object is v_0. In the following periods all replica servers which store a replica can distribute a copy of it.

The placement is done via multicast. Hence, the optimal routes are not obvious and we need to model the placement flow. In order to ease the notation of the model we define two additional sets of nodes.

$$\mathcal{V}^{out}(s) := \{i \in \mathcal{V}_s \cup \{v_0\} \mid (i, s) \in \mathcal{A}\}$$

$$\mathcal{V}^{in}(s) := \{j \in \mathcal{V}_s \cup \{v_0\} \mid (s, j) \in \mathcal{A}\}$$

$\mathcal{V}^{out}(s)$ is the set of nodes $i \in \mathcal{V}_s \cup \{v_0\}$ with an outgoing arc $(i, s) \in \mathcal{A}$ to

server $s \in \mathcal{V}_s \cup \{v_0\}$ while $\mathcal{V}^{in}(s)$ denotes the set of nodes $j \in \mathcal{V}_s \cup \{v_0\}$ with an incoming arc $(s, j) \in \mathcal{A}$ from server $s \in \mathcal{V}_s \cup \{v_0\}$.

Content delivery. Let $\mathcal{V}_c \subseteq \mathcal{V} \setminus \{v_0\}$ be the set of clients from which the object can be requested. The number of requests of a client $c \in \mathcal{V}_c$ in period $t \in \mathcal{T}$ is denoted as $r_{c,t}$. Each replica server $s \in \mathcal{V}_s$ has a load capacity C_s^L which is the maximum number of requests the server can process per period.

The service level agreement is defined by the tuple (λ, q). In each period, all requests for the object have to be served. With q we denote the maximum latency within which a request has to be served. λ defines the fraction of all requests which has to be served within q. Each single request of a client can be served by a different replica server. Hence, the total number of requests of one client in one period can be split among several replica servers. A single request can not be split.

For the corresponding constraints we denote with $\mathcal{V}_s(c, q) \subseteq \mathcal{V}_s$ the set of replica servers where each server $s \in \mathcal{V}_s(c, q)$ can serve a request of client $c \in \mathcal{V}_c$ within latency q. Hence, in the topology there is a shortest path between client c and server s of length $d_{c,s} \leq q$ which would be used for the delivery.

Variables. We employ the following decision variables. For the replica assignment $x_{s,t} \in \{0, 1\}$ equals 1 if server $s \in \mathcal{V}_s$ stores a replica of the object in period $t \in \mathcal{T}$ and 0 otherwise.

To model the flow of the placements we denote with $0 \leq w_{s,t} \leq 1$ the variable which indicates for $w_{s,t} = 1$ that replica server $s \in \mathcal{V}_s$ receives a replica in period $t \in \mathcal{T}$. Note that $w_{s,t}$, although modeled as continuous variable, will always be forced to an integer through the other constraints. We employ $z_{i,j,t} \in \{0, 1\}$ for the actual routes of the placements. $z_{i,j,t} = 1$ indicates that arc $(i, j) \in \mathcal{A}$ is used for the placement in the beginning of period $t \in \mathcal{T}$. With $\tilde{z}_{i,j,t} \geq 0$ we denote the variable which counts how many times arc $(i, j) \in \mathcal{A}$ is used for the placement in period $t \in \mathcal{T}$. We need $\tilde{z}_{i,j,t}$ to model the multicast flow.

Finally, for modeling the delivery to the clients we employ $y_{c,s,t} \geq 0$ which is the fraction of all requests from client $c \in \mathcal{V}_c$ in period $t \in \mathcal{T}$ which is served by replica server $s \in \mathcal{V}_s$. Note that although $y_{c,s,t}$ is continuous the resulting number of requests served by each server is integer. Table A.1 in the Appendix provides a summary of the notation.

3.3 MIP formulation

We can now model the following mixed–binary linear program for the dynamic replica placement with service levels (DRPSL):

$$\begin{aligned}\text{Min } & \sum_{s\in\mathcal{V}_s}\sum_{t\in\mathcal{T}} \alpha_s \cdot x_{s,t} + \beta \cdot \sum_{(i,j)\in\mathcal{A}}\sum_{t\in T} \delta_{i,j} \cdot z_{i,j,t} \\ & + \gamma \cdot \sum_{c\in\mathcal{V}_c}\sum_{s\in\mathcal{V}_s}\sum_{t\in\mathcal{T}} d_{c,s} \cdot r_{c,t} \cdot y_{c,s,t}\end{aligned} \tag{1}$$

subject to

$$\sum_{c\in\mathcal{V}_c} r_{c,t} \cdot y_{c,s,t} \leq C_s^L \cdot x_{s,t} \qquad \begin{matrix}\forall s \in \mathcal{V}_s \\ \forall t \in \mathcal{T}\end{matrix} \tag{2}$$

$$\sum_{s\in\mathcal{V}_s} y_{c,s,t} = 1 \qquad \begin{matrix}\forall c \in \mathcal{V}_c \\ \forall t \in \mathcal{T}\end{matrix} \tag{3}$$

$$\sum_{c\in\mathcal{V}_c}\sum_{s\in\mathcal{V}_s(c,q)} r_{c,t} \cdot y_{c,s,t} \geq \lambda \cdot \sum_{c\in\mathcal{V}_c} r_{c,t} \qquad \forall t \in \mathcal{T} \tag{4}$$

$$w_{s,t} \geq x_{s,t} - x_{s,t-1} \qquad \begin{matrix}\forall s \in \mathcal{V}_s \\ \forall t \in \mathcal{T}\end{matrix} \tag{5}$$

$$\sum_{j\in\mathcal{V}^{in}(s)} \tilde{z}_{s,j,t} \geq \sum_{i\in\mathcal{V}^{out}(s)} \tilde{z}_{i,s,t} - w_{s,t} \qquad \begin{matrix}\forall s \in \mathcal{V}_s \\ \forall t \in \mathcal{T}\end{matrix} \tag{6}$$

$$\sum_{j\in\mathcal{V}^{in}(s)} \tilde{z}_{s,j,t} \leq \sum_{i\in\mathcal{V}^{out}(s)} \tilde{z}_{i,s,t} - w_{s,t} + M \cdot x_{s,t-1} \qquad \begin{matrix}\forall s \in \mathcal{V}_s \cup \{v_0\} \\ \forall t \in \mathcal{T}\end{matrix} \tag{7}$$

$$M \cdot z_{i,j,t} - \tilde{z}_{i,j,t} \geq 0 \qquad \begin{matrix}\forall (i,j) \in \mathcal{A} \\ \forall t \in \mathcal{T}\end{matrix} \tag{8}$$

$$x_{s,0} \leq 0 \qquad \forall s \in \mathcal{V}_s \tag{9}$$

$$x_{v_0,0} \geq 1 \tag{10}$$

$$x_{v_0,t} \geq 1 \qquad \forall t \in \mathcal{T} \tag{11}$$

$$w_{v_0,t} \leq 0 \qquad \forall t \in \mathcal{T} \tag{12}$$

$$x_{s,t} \in \{0,1\} \qquad \begin{array}{l} \forall s \in \mathcal{V}_s \cup \{v_0\} \\ \forall t \in \mathcal{T} \end{array} \tag{13}$$

$$w_{s,t} \in [0,1] \qquad \begin{array}{l} \forall s \in \mathcal{V}_s \cup \{v_0\} \\ \forall t \in \mathcal{T} \end{array} \tag{14}$$

$$z_{i,j,t} \in \{0,1\} \qquad \begin{array}{l} \forall (i,j) \in \mathcal{A} \\ \forall t \in \mathcal{T} \end{array} \tag{15}$$

$$\tilde{z}_{i,j,t} \geq 0 \qquad \begin{array}{l} \forall (i,j) \in \mathcal{A} \\ \forall t \in \mathcal{T} \end{array} \tag{16}$$

$$y_{c,s,t} \geq 0 \qquad \begin{array}{l} \forall c \in \mathcal{V}_c \\ \forall s \in \mathcal{V}_s \\ \forall t \in \mathcal{T} \end{array} \tag{17}$$

The objective function (1) minimizes the sum of three cost terms. First, the storage cost α_s for each replica server and period if the server stores a replica of the object. Second, the cost for the placement, i. e. the sum of the lengths $\delta_{i,j}$ of all arcs (i,j) used in the multicast placement processes in all periods multiplied with cost coefficient β. Third, the cost for the delivery to the clients. The cost coefficient γ is multiplied with the sum of the distances of all paths used for the delivery in all periods. As mentioned earlier we consider the potential split of the requests of a client among several replica servers with $y_{c,s,t}$. Note that for the delivery cost single arcs or even whole identical paths are added as often as they are used according to the unicast transfer.

Constraints (2) ensure that each replica server in each period can only serve requests if it stores a replica of the object. Additionally, the load capacity C_s^L is defined as the maximum number of requests the server s can handle per period. Due to constraints (3) all requests of each client in each period have to be served. Constraints (4) ensure that in each period the service level agreements are fulfilled, i. e. that at least the fraction λ of all requests is handled by replica servers in the range of the maximum latency q.

The variable $w_{s,t}$ is set through constraints (5). We need it to model the multicast data flow of the placement process in each period. The latter is done in constraints (6) and (7) which cover the following three cases.

First, if the considered server $s \in \mathcal{V}_s$ did not store a replica in period $t-1$ (i. e. $x_{s,t-1} = 0$) and does not store a new replica in the actual period t (i. e. $x_{s,t} = 0$) it follows through constraints (5) that $w_{s,t} = 0$. In this case (6) reduces to

$$\sum_{j \in \mathcal{V}^{in}(s)} \tilde{z}_{s,j,t} \geq \sum_{i \in \mathcal{V}^{out}(s)} \tilde{z}_{i,s,t} \,,$$

(7) reduces to

$$\sum_{j \in \mathcal{V}^{in}(s)} \tilde{z}_{s,j,t} \leq \sum_{i \in \mathcal{V}^{out}(s)} \tilde{z}_{i,s,t}$$

and thus we have

$$\sum_{j \in \mathcal{V}^{in}(s)} \tilde{z}_{s,j,t} = \sum_{i \in \mathcal{V}^{out}(s)} \tilde{z}_{i,s,t} \,.$$

For server s the outflow is set equal to the inflow, i. e. s is just a transshipment node. The server s did not store a replica in the preceding period and therefore can not copy the object and send it to other servers on its own. As the server s also does not store a replica in period t, all objects it receives are meant for other servers.

Second, if the considered server $s \in \mathcal{V}_s$ receives a new replica in period t we have $x_{s,t-1} = 0$, $x_{s,t} = 1$ and due to constraints (5) $w_{s,t} = 1$. In this case the constraints (6) and (7) set the outbound flow equal to the inbound flow minus 1 unit which is the unit stored at the server.

Third, the case if $x_{s,t-1} = 1$ and through constraints (5) $w_{s,t} = 0$. The considered server has a replica in period $t-1$ and thus can send copies to other servers in period t. Independent of $x_{s,t}$, constraints (6) force the outbound flow greater than or equal to the inbound flow while constraints (7) do not restrict the outbound flow in this case. Instead, the additional summand M allows the server to send copies of the object even without an inbound flow. Note that constraints (7) hold not only for the replica servers $s \in \mathcal{V}_s$ but do also hold for the origin server v_0. Hence, constraints (7) allow to send copies from the origin server to replicate the object. It is easy to see that w. l. o. g. it is sufficient to set $M = |\mathcal{V}_s|$ in constraints (7) as this is an

upper bound for the amount of outgoing copies. This could be the case if only the origin server v_0 has the object and all replica servers receive a replica.

Finally the placements which are so far modeled as unicast transfers in constraints (6) and (7) need to be reduced to a multicast flow. This is done by constraints (8) which set the binary decision variables $z_{i,j,t}$ to 1 if there is a flow on (i, j) for the placement in period t regardless of the number of units which are transferred.

The decision variables are defined in equations (9)–(17).

3.4 Discussion

We now want to discuss some special cases of the DRPSL. If we set $\beta = 0$ and $\gamma = 0$ the objective function (1) considers the storage cost only. The goal is then to store as few replicas as possible while still being capable of serving the demand under the service level constraints (4). In this case the model decomposes into $|\mathcal{T}|$ independent submodels. Hence, the model for each period t can be considered separately. If we additionally set $\lambda = 1$, i. e. the maximum latency holds for all requests, the problem can be formulated as a set covering problem (see Krarup and Pruzan [42], Shi and Turner [66]).

By setting $\alpha = 0$ and $\beta = 0$ we are minimizing the delivery cost only. In this case the optimal solution is trivial. We place a replica on every replica server in each period.

The combination of the storage cost and delivery cost, i. e. $\alpha > 0$, $\beta = 0$ and $\gamma > 0$, leads to a generalized facility location problem with service level constraints. Constraints (5)–(8) are dropped. Constraints (2) and (3) are analog to the constraints in a capacitated facility location problem (CFLP) that deal with the capacity of a facility and with serving all the demand (see Aikens [4]). Constraints (4) can be seen as the maximum distance between each facility and its assigned customers which has to be met for a certain percentage of all customers. As there are no cost for deleting a replica DRPSL also decomposes into one independent subproblem for each period.

By just minimizing storage and placement cost, i. e. setting $\gamma = 0$, we search for a 'stable' assignment with as few replicas as possible while serving

the demand and fulfilling the service levels. A 'stable' replica assignment means in this context an assignment with as few changes as possible over time. This is due to the second summand of the objective function, the placement cost. Through the placement cost the periods are connected and consecutive periods depend on each other.

3.5 Model complexity

In what follows we show that the DRPSL can be restricted to two well known NP–hard optimization problems, the uncapacitated facility location problem and the Steiner tree problem in graphs.

The uncapacitated facility location problem. The problem can be restricted to an uncapacitated facility location problem by setting $|\mathcal{T}| = 1$, $\beta = 0$, $\gamma = 1$, $C_s^L \equiv M$ and $\lambda = 0$. The set $\mathcal{V}_s$ corresponds to the set of possible locations for the facilities. Storing a replica on a replica server then corresponds to opening a facility.

The Steiner tree problem in networks. The Steiner tree problem in networks is defined as follows (see Hwang and Richards [25]).

Given: An undirected network $\mathcal{N} = (\mathcal{V}, \mathcal{E}, w)$ with nodes $v \in \mathcal{V}$, edges $e \in \mathcal{E}$ and edge weights w.
A non–empty set of terminal nodes $\mathcal{K} \subseteq \mathcal{V}$.

Find: A subnetwork $\mathcal{S}_\mathcal{N}(\mathcal{K})$ of $\mathcal{N}$ such that:
– there is a path between every pair of terminal nodes
– the total length $|\mathcal{S}_\mathcal{N}(\mathcal{K})| = \sum_{e \in \mathcal{S}_\mathcal{N}(\mathcal{K})} w(e)$ is minimized

The vertices in $\mathcal{V} \backslash \mathcal{K}$ are called non-terminals. Non–terminals that end up in the Steiner tree $\mathcal{S}_\mathcal{N}(\mathcal{K})$ are called Steiner vertices.

We restrict the DRPSL to the Steiner tree problem using the flow formulation of Voss [74] by allowing only instances with $|\mathcal{T}| = 1$, $\alpha_s \equiv 0$, $\gamma = 0$, $r_{c,t} \equiv 0$ and $\lambda = 0$. Then we set $x_{s,t} = 1$ for all terminal nodes. For one of these terminal nodes we set $x_{s,t-1} = 1$ as the "origin" of the flow in the preceding period. For the solution of the Steiner tree problem in graphs it

does not matter which node is chosen as the resulting Steiner tree will be the same in each case.

Proposition 1. *The DRPSL is NP–hard.*

Proof. The uncapacitated facility location problem (see Krarup and Pruzan [42]) and the Steiner tree problem in graphs (see Karp [38]) are both known to be NP–hard. Since the uncapacitated facility location problem and the Steiner tree problem in graphs are special cases of the DRPSL the latter has to be NP–hard. □

3.6 Lower bounds

Since the DRPSL is an NP–hard optimization problem we are interested in lower bounds. The latter can be used to determine the solution gap when evaluating heuristic algorithms.

LP–relaxation. A straight forward LP–relaxation relaxes the binary decision variables $x_{s,t}$ and $z_{i,j,t}$ to be in $[0, 1]$. However, this relaxation provides bad lower bounds because the assignment variable $x_{s,t}$ is very fragmented, i. e. many servers store a small fraction of the object. This has two effects. First, a very low flow is sufficient for the placement, especially as $z_{i,j,t}$ is also relaxed. Second, there is almost no flow for the delivery. Thus, the absolute cost values for network usage in the objective function, i. e. the placement cost and delivery cost, are very low and far from realistic.

Improved lower bound. We improve the standard LP–relaxation in two respects. First, by extending the model to the disaggregated or 'strong' formulation which is known from the facility location problem (see Krarup and Pruzan [42]). In order to do so, we add constraints (18) to DRPSL.

$$y_{c,s,t} \leq x_{s,t} \qquad \forall c \in \mathcal{V}_c, s \in \mathcal{V}_s, t \in \mathcal{T} \tag{18}$$

Constraints (18) state for each period t and each client c that the client can only be supplied by a replica server s if the latter stores the object, i. e. $x_{s,t} = 1$. This is basically the same as in Constraints (2) except for the load

capacity C_s^L. The second difference is that Constraints (2) aggregate over the clients c. Note that with M instead of C_s^L in Constraints (2), i. e. without the capacity restriction, Constraints (2) could be omitted in the disaggregated model. We denote the tightened formulation (1)–(17) and (18) as DRPSL2.

Second, we calculate an improved lower bound for the placement cost $\beta \cdot \sum_{(i,j)\in\mathcal{A}} \sum_{t\in T} \delta_{i,j} \cdot z_{i,j,t}$ in the objective function (1). $\underline{c^p}(n)$ gives the lower bound for the placement cost if $n \in \{1, 2, \ldots, |\mathcal{V}_s| \cdot T\}$ replicas are stored on all servers over the entire planning horizon. Before we explain our approach to calculate $\underline{c^p}(n)$ we want to note how to get the number of replicas n from a relaxed solution. As the binary assignment variable x is relaxed we have to calculate n as a lower bound for the number of replicas given the storage cost component c^s of the objective function value. Therefore we define:

$$n := \min\{|\mathcal{R}| \quad | \quad \mathcal{R} \subseteq \mathcal{V}_s : \sum_{r\in\mathcal{R}} \alpha_r \geq c^s\} \tag{19}$$

If $\alpha_r \equiv \alpha \ \forall r \in \mathcal{V}_s$ 19 reduces to $n = \left\lceil \frac{1}{\alpha} \cdot c^s \right\rceil$.

To calculate $\underline{c^p}(n)$ for a given n, we pursue the following approach. Considering only the placement cost we construct a tree which we call minimum update tree (MUT) as it is used to update the replica assignment. Before giving the general notation we want to give a short example. We define $T := |\mathcal{T}|$ for an easier notation.

The optimal assignment over all periods of up to T replicas at minimum placement cost is to start in period $t = 1$ and store the object on the replica server nearest to the origin server. We leave the replica there for the following periods $t = 1, \ldots, T$ as this does not lead to additional placement cost. The arc between the origin server and the nearest replica server is the first arc in the MUT. We denote the length of this arc with $\hat{\delta}_1 := \min\{\delta_{v_0,s} \mid s \in \mathcal{V}_s : (v_0, s) \in \mathcal{A}\}$. This leads to cost of $\beta \cdot \hat{\delta}_1$. In Figure 3.1 the first server of the MUT is server 3, the first arc is $(0, 3)$.

For storing a total amount of replicas in the range of $[T + 1, 2 \cdot T]$ it is optimal to store the additional replicas in consecutive periods on one server. For this purpose we choose the server with the lowest distance to the set of two servers which already store replicas. In Figure 3.1 this is server 5, reached through arc $(3, 5)$. We call the length of this second arc in our MUT

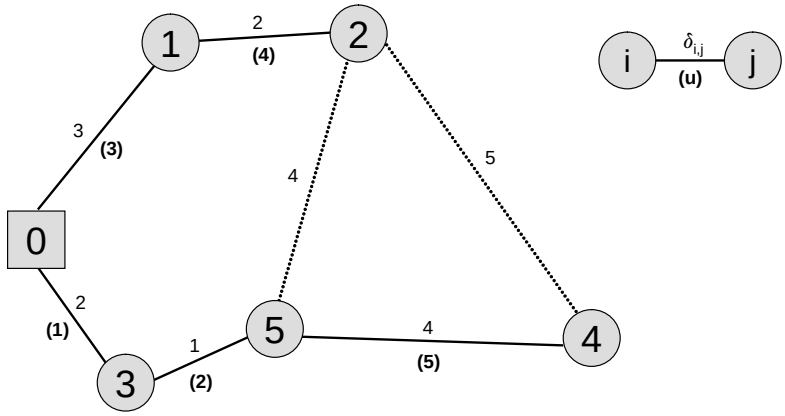

Figure 3.1: Example for a minimum update tree (MUT)

$\hat{\delta}_2 := \min\{\delta_{i,j} \mid i \in \{v_0, s\}, j \in \mathcal{V}_s \setminus \{s\} : (i,j) \in \mathcal{A} \wedge s \in \mathcal{V}_s : \delta_{v_0,s} = \hat{\delta}_1\}$. It follows that in this case the minimum possible placement cost are $\beta \cdot (\hat{\delta}_1 + \hat{\delta}_2)$.

For the general case we denote with $\mathcal{V}^{MUT}(u) \subseteq \mathcal{V}_s$ the set of the $u = 1, \ldots, |\mathcal{V}_s|$ replica servers in our MUT, each one storing T or less of the n replicas. It follows that the MUT to store n replicas consists of the nodes $\mathcal{V}^{MUT}(\lceil n \,/\, T \rceil)$. We define $\mathcal{V}^{MUT}(0) = \{v_0\}$. In our example the sequence in which the nodes are added to build the complete MUT is $(3, 5, 1, 2, 4)$.

The u-th arc in the MUT, i. e. the arc used to place the replicas $(u-1) \cdot T + 1, \ldots, u \cdot T$ has a length of $\hat{\delta}_u$. In Figure 3.1 all $u = 1, \ldots, 5$ arcs of the complete MUT are given. We can now define:

$$\hat{\delta}_u := \min\{\delta_{i,j} \mid i \in \mathcal{V}^{MUT}(u-1), j \in \mathcal{V}_s \setminus \mathcal{V}^{MUT}(u-1) : (i,j) \in \mathcal{A}\}$$

$$\mathcal{V}^{MUT}(u) := \mathcal{V}^{MUT}(u-1) \cup \{s^{mindist}\} \quad , s^{mindist} \in \mathcal{V}^{mindist}(u)$$

$$\mathcal{V}^{mindist}(u) := \{s \in \mathcal{V}_s \setminus \mathcal{V}^{MUT}(u-1) \mid \delta_{i,s} = \hat{\delta}_u : i \in \mathcal{V}^{MUT}(u-1)\}$$

Finally, the minimum placement cost given the number of replicas n are equal to the weighted length of the MUT:

$$\underline{c}^p(n) = \sum_{u=1,\ldots,\lceil n \,/\, T \rceil} \hat{\delta}_u \ , \forall n = 1, \ldots, |\mathcal{V}_s| \cdot T \tag{20}$$

Chapter 4

Methods

Even for small problem instances an exact solution of the proposed mixed–binary linear program DRPSL might not be possible as it is NP–hard and has a vast amount of binary variables. To solve the problem in reasonable time we propose two different efficient metaheuristic approaches, i. e. a simulated annealing algorithm (SA) and several variants of variable neighborhood search (VNS). We develop various problem–specific neighborhoods to optimize the different components of the problem efficiently. For comparison we implement common heuristics from the literature and build refined versions of them to better suit our problem.

For all methods, we consider a topology which consists of the origin server and the replica servers of the CDN. As we already stated, this can be done w. l. o. g. as a request from a client for a replicated object is routed to the nearest CDN server and can be added to the latter in the topology. Hence, each server can store a replica of the object and can have a demand at the same time, i. e. $\mathcal{V} \equiv \mathcal{V}_s \cup v_0$ and $\mathcal{V}_s \equiv \mathcal{V}_c$ (see Karlsson and Karamanolis [34], Tang and Xu [69]). Thereby we neglect the so–called "last mile", i. e. the distance between a client and his nearest replica server (see Candan et al. [10]).

4.1 Framework

All implemented algorithms are embedded in a modular framework. The framework incorporates a representation of network topologies and all the data necessary for the solution methods. Among other interfaces the framework provides methods for the efficient evaluation of solutions. We will now describe the solution representation and the solution evaluation of our framework.

4.1.1 Solution representation

As solution representation for our algorithms we choose a binary $|\mathcal{V}| \times |\mathcal{T}|$ matrix. An entry $x_{s,t}$ equals 1 if replica server $s \in \mathcal{V}_s$ stores a replica of the object in period $t \in \mathcal{T}$ and 0 otherwise. Hence, each entry in the matrix corresponds to a decision variable $x_{s,t}$ of the DRPSL, each row represents a replica server and each column a period. With this intuitive representation solutions can easily be modified and special neighborhoods can be implemented straightforward. The variable $z_{i,j,t}$ for the flow of the placement in our model is not part of the solution representation for our algorithms. We consider the placement process in the evaluation of a solution x. Hence, in the context of our algorithms by saying solution x we mean the solution represented by the replica assignment in matrix x.

In our initial solution we set $x_{s,t} = 1 \ \forall s \in \mathcal{V}_s, t \in \mathcal{T}$, i. e. every replica server stores a replica in every period. The advantage of this initial solution is that it is feasible.

4.1.2 Solution evaluation

For the evaluation of solution x we need to compute the objective function value of it, i. e. the values of the three cost components.

Storage cost. From the solution matrix x we calculate the storage cost according to the first term of the objective function (1) to $\sum_{s \in \mathcal{V}_s} \sum_{t \in \mathcal{T}} \alpha_s \cdot x_{s,t}$.

Placement cost. To calculate the placement cost for a given solution x we need to know for each period the length of the multicast tree in the network used for the placement. As we already mentioned our problem can be restricted to a Steiner tree problem in graphs (STP) by just considering the placement cost. I. e. the problem of finding the optimal multicast tree in one period for a given replica assignment can be formulated as a Steiner tree problem in graphs (see Oliveira et al. [51], Voss [75]).

To transform the placement problem of a period t into a Steiner tree problem we need to know the replica assignment in period t and $t-1$. The reason is that the servers which stored a replica in the preceding period $t-1$ are those which can send out copies in the placement process of period t. We also have to define the set of terminal nodes in the network. The Steiner tree connects all the terminal nodes and can use the other nodes to do so. Hence, the terminal nodes represent the replica servers which receive a replica in period t. Additionally, a replica server which stores a replica in period $t-1$ needs to be in the set of terminal nodes to send out replicas using the Steiner tree as multicast placement tree.

If there are multiple servers with replicas in the preceding period, all of them need to be considered as each one could be used to send a replica. We cannot add each of them individually to the set of terminal nodes as then their connection would be enforced. The resulting placement cost would be higher than necessary. In fact, using a server to send a replica is optional. Accordingly, the necessary modification is to replace all replica servers which store a replica in the preceding period by one virtual node. An example is given in Figure 4.1 where the two replica servers 2 and 5 are aggregated to the virtual node v. The virtual node is then added to the set of terminal nodes as the replica origin for the placement. The virtual node gets connected by all arcs of the aggregated nodes. As in Figure 4.1, two special cases have to be considered. First, arcs between the aggregated nodes are ignored as these arcs would be located within the virtual node. Second, if there would be more than one arc between one external node and the virtual node, only the arc with the minimum weight is used.

With this preprocessing, the placement cost of each period can be calculated by solving a Steiner tree problem. Such a STP might not be solvable in reasonable time as it is NP–hard (see Section 3.5). To compute the placement cost of one solution x, $|\mathcal{T}|$ Steiner tree problems have to be solved.

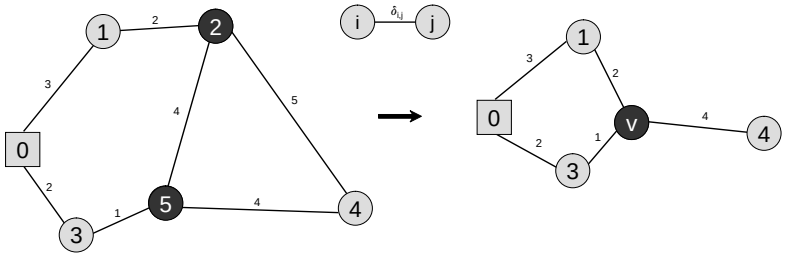

Figure 4.1: Aggregation of two replica servers to a virtual node

And as there is a considerable amount of neighbor solutions that needs to be evaluated for the metaheuristics, we cannot compute the optimal solution of this cost component. Hence, we use a heuristic in order to solve the STP.

To compute each Steiner tree we use the well–known KMB heuristic by Kou et al. [41] which is based on minimum spanning trees. We adapt the scheme presented by Prömel and Steger [58]. The procedure is outlined in Algorithm 1. Figure 4.2 provides an example for the five steps of the algorithm for a simple network in which the square node represents the origin server and the dark circles represent the terminal nodes. The KMB heuristic is considered to be among the Steiner tree heuristics with the best solution quality. It guarantees a length of the generated tree no more than 2 times the length of the optimal tree. Furthermore, it is shown that this heuristic usually achieves a gap of at most 5% to the optimal solution (see Oliveira and Pardalos [52]).

At the beginning of our algorithms we have to compute the Steiner trees for all periods to get the complete placement cost. For the evaluation of following solutions we take advantage of our neighborhood structure. Our neighborhoods change only one period of the prior solution at a time. Hence, it is sufficient to recompute only the Steiner trees of the changed period t and the subsequent period $t + 1$ to evaluate the change. Both periods need to be considered as the replicas in period t are potential receivers for the placement in period t and potential sources for the placement in period $t+1$.

Algorithm 1 KMB heuristic for the Steiner tree problem

Input: Connected network $\mathcal{N} = (\mathcal{V}, \mathcal{E}, w)$ with the set of nodes $\mathcal{V}$, the set of edges $\mathcal{E}$ and edge weights w
Input: Set of terminal nodes $\mathcal{K} \subseteq \mathcal{V}$

1: Compute the distance network $\mathcal{D}_\mathcal{N}(\mathcal{K})$
$\mathcal{D}_N(\mathcal{K})$ = the complete network $(\mathcal{K}, \mathcal{E}_D, w_D)$ in $\mathcal{N}$; $|\mathcal{E}_D| = \frac{|\mathcal{K}| \cdot (|\mathcal{K}|-1)}{2}$ and $w_D(i,j) = d_{i,j}$, the length of the shortest path between nodes $i, j \in \mathcal{V}$
2: Compute a minimum spanning tree $\mathcal{T}_D$ in $\mathcal{D}_\mathcal{N}(\mathcal{K})$
3: Transform $\mathcal{T}_D$ into a subnetwork $\mathcal{N}[\mathcal{T}_D]$ by replacing every edge of $\mathcal{T}_D$ with the corresponding shortest path in $\mathcal{N}$
4: Compute a minimum spanning tree $\mathcal{T}$ for the subnetwork $\mathcal{N}[\mathcal{T}_D]$
5: Transform $\mathcal{T}$ into a Steiner tree $\mathcal{S}_\mathcal{N}(\mathcal{K})$ of $\mathcal{N}$ by successively deleting the leaves which are no terminal nodes and the corresponding edges

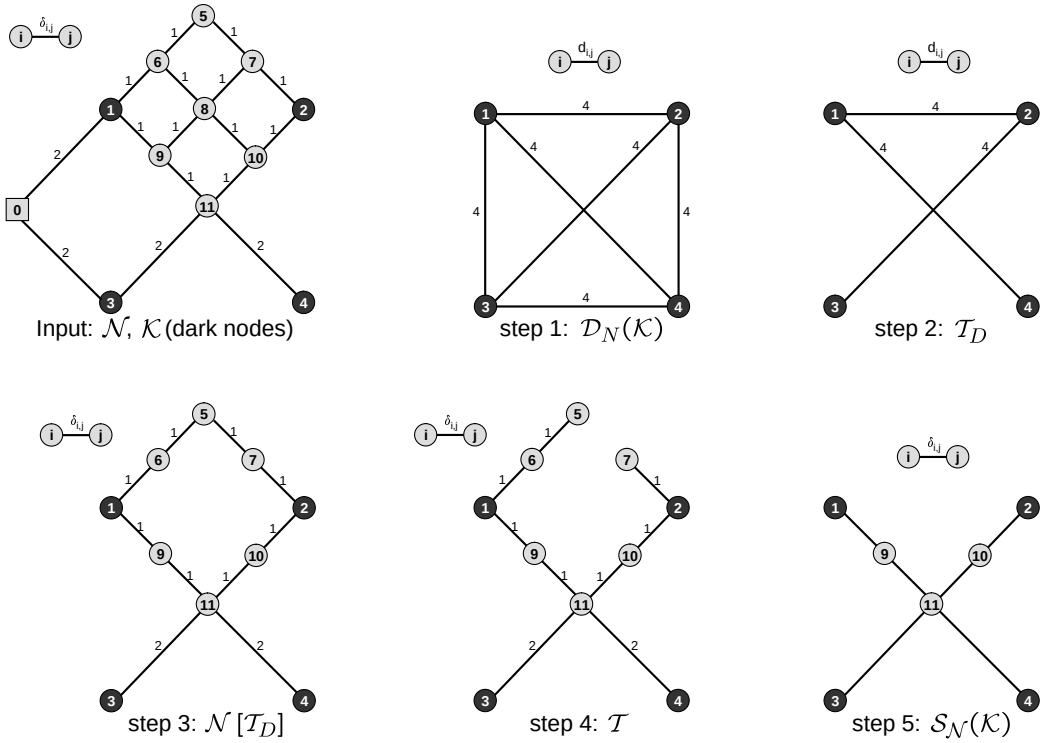

Figure 4.2: Steps of the KMB heuristic

Content delivery cost. As the number of requests $r_{c,t}$ $\forall c \in \mathcal{V}_c, t \in \mathcal{T}$ is given we need to determine $y_{c,s,t}$, the fraction of all requests from client $c \in \mathcal{V}_c$ in period $t \in \mathcal{T}$ which is served by replica server $s \in \mathcal{V}_s$. This is done by solving a transportation problem. For this purpose we implement the efficient algorithm of Ahrens [3]. This algorithm is completely based on simple list data structures and has a very low memory usage.

Averaged over 38 problem instances with our considered problem size of 50 nodes and 12 periods, between 14,000 and 238,000 transportation problems (99,160 on average) are solved by our different metaheuristics to evaluate the content delivery cost. CPLEX is not suitable for this task as it takes too much time. We compared an evaluation of the delivery cost using CPLEX against our implementation. To this end we used random assignments with every possible amount of replicas in the network. On average it took 372 ms to evaluate one solution using CPLEX. Hence, each of the twelve transportation problems in a solution took on average 31 ms to solve. To some extend this is due to the necessary transformation of the input and output data between the data structures of our framework and the special objects that the CPLEX API requires. But even if we only consider the runtime of the CPLEX solver we get 136.32 ms for a complete solution or 11.36 ms per transportation problem. Our implementation takes just 8.89 ms to evaluate a complete solution or 0.74 ms to solve one transportation problem on average – including all pre– and post processing.

As with the placement cost the runtime can be further reduced. Our implementation is suited for repeatedly evaluating similar solutions. We save the results of every period of the actual solution. To evaluate a new neighbor solution we recompute only those periods in which the assignment changed.

4.2 Simulated Annealing

Simulated annealing (SA) (see Kirkpatrick et al. [39]) is a well–known metaheuristic inspired by the physical annealing process of solids. Starting with an initial solution x, the SA generates a neighbor solution using a neighborhood $N(x)$. If the generated solution is better in terms of the objective function value it is accepted and replaces the current solution. A worse solution is only accepted with a certain probability. This is done in order to escape from local minima. The acceptance probability depends on the difference between the objective function values of the previous and the current solution and a parameter called temperature. Basically, the temperature is decreased after each iteration of the algorithm using a predefined cooling schedule. Hence, the acceptance probability of worse solutions decreases during the runtime. Based on the new or unchanged old solution a new neighbor is generated and evaluated in each iteration until a stopping criterion is reached. We adapt the standard simulated annealing scheme presented in Henderson et al. [24] (see Algorithm 2).

4.2.1 Neighborhoods

The construction of a neighbor solution in neighborhood $N(x)$ works according to Algorithm 3. The changes are always made in just one period of the replica storage matrix. Hence, a period has to be selected at first.

As we start the algorithm with a feasible solution and change the assignment in only one period in each step, the first infeasible neighbor solution can only be infeasible due to exactly one period. Thus, a neighbor of an infeasible solution can be feasible again. We try to repair an infeasible solution as soon as possible. To this end we select the infeasible period t_{infeas}, if there is one, and search in this period for a server where an additional replica can be placed.

Otherwise, if the actual solution is feasible, we randomly select a period and remove a replica from one server in this period with probability p. With probability $1 - p$ we add a replica to a server in the selected period. The change of the assignment is done by negation of the boolean value $x_{s,t}$.

Algorithm 2 Simulated annealing scheme

Input: Initial temperature $t_0 \geq 0$
Input: Temperature cooling schedule t_k
Input: Repetition schedule M_k that defines the number of iterations executed at each temperature t_k

```
1: Select an initial solution x
2: Set the temperature change counter: k ← 0
3: Set the currently best solution: x* ← x
4: while k < stopping criterion do
5:     for repetition counter m ← 0 to M_k do
6:         Generate a solution x' ∈ N(x)
7:         Calculate Δ_{x,x'} ← cost(x') − cost(x)
8:         if Δ_{x,x'} ≤ 0 then
9:             Accept new solution x' (x ← x')
10:            if cost(x') < cost(x*) then
11:                x* ← x'
12:                k ← 0
13:            end if
14:        else
15:            Accept new solution x' (x ← x') with
16:            probability exp(−Δ_{x,x'}/t_k)
17:        end if
18:    end for
19:    k ← k + 1
20: end while

21: return x*
```

For the choice of a server s to change the assignment in period t several aspects have to be considered. In an earlier version of the neighborhood we arbitrarily selected a server. But the resulting solutions had high placement cost. For good solutions in this respect it is essential that the servers store replicas in consecutive periods. An example for such a solution matrix is given in Figure 4.3. In this solution the placed replicas mostly remain on the servers for several periods. Some servers do never store a replica over the planning horizon. Their demand is met by nearby servers. It can be seen that

Algorithm 3 Generate a neighbor of solution x in neighborhood $N(x)$

Input: Probability p to remove a server
Input: Solution x

1: **if** $t_{infeas} \neq NULL$ **then**
2: $t \leftarrow t_{infeas}$
3: $s \leftarrow$ chooseServerToAdd(x, t)
4: **else**
5: $t \leftarrow random(\mathcal{T})$
6: **if** $random[0, 1[< p$ **then**
7: $s \leftarrow$ chooseServerToRemove(x, t)
8: **else**
9: $s \leftarrow$ chooseServerToAdd(x, t)
10: **end if**
11: **end if**
12: $x_{s,t} \leftarrow \neg x_{s,t}$ (▷ negation of the boolean value to change the assignment)
13: **return** x

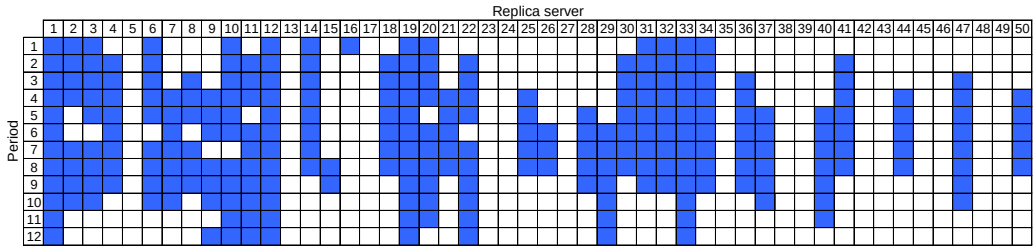

Figure 4.3: Possible solution structure

the periods 4–9 have the highest overall demand as several additional servers store replicas in these periods. To minimize the overall cost, while serving all demand and fulfilling the service levels, the amount of replicas should be low, the distances to the clients should be small and the replica assignment should rarely change. The SA scheme with the earlier neighborhood did not find these structural blocks in the solution matrix.

Our proposed neighborhood is improved in this respect as it selects a replica server in the chosen period t not randomly but based on the change

of the placement cost. We calculate a value for each server according to the change of the placement cost if the assignment on the server switches. We do not calculate the real change as this is too time consuming considering how many of the placement cost evaluations have to be calculated. Instead we categorize the servers. As adding and removing a replica are analog cases we explain the valuation by means of searching a server to remove a replica from, i. e. the method CHOOSESERVERTOREMOVE(x, t) (see Algorithm 4).

Algorithm 4 Choice of a server to remove a replica from

```
1: function CHOOSESERVERTOREMOVE(solution x, period t)
2:     for all s ∈ V_s do
3:         if x_{s,t} = 0 then
4:             r_s ← −1
5:         else
6:             r_s ← 0
7:             if t > 1 and x_{s,t−1} = 0 then
8:                 r_s ← r_s + 1
9:             end if
10:            if t < |T| and x_{s,t+1} = 0 then
11:                r_s ← r_s + 1
12:            end if
13:        end if
14:    end for
15:    return a randomly chosen replica server s with the highest value r_s
16: end function
```

If a server does not store a replica in period t, we assign the value -1. The server will not be selected as it is impossible to remove a replica. There are the following three cases if a server stores a replica in period t. First, if a server stores a replica in the preceding period $t-1$ and in the succeeding period $t+1$, we assign the value 0. In this case, by removing the replica a block of consecutive periods would be split and the placement cost would increase. Second, a value of 1 is assigned to a server which stores a replica in period t and in one adjacent period, i. e. $t-1$ or $t+1$. Placement cost typically change only a little or not at all in this case. Third, if the server stores a replica in period t but does not store a replica in the adjacent periods. Since the placement cost can be reduced this is the best case and we assign the value 2. After we have valued all servers in the chosen period t, we select one server with the highest value randomly.

As a given solution differs from an arbitrary solution of the neighborhood in exactly one element, the size of the neighborhood is $|\mathcal{V}_s| \times |\mathcal{T}|$.

4.2.2 Infeasible solutions

So far we did not address the problem of infeasible solutions. For simulated annealing we do not want to exclude infeasible solutions categorically. The acceptance of worse solutions is the key to escape from local minima in the SA scheme. And as already stated we try to repair infeasible solutions explicitly in our neighborhood definition. But we do not want to accept all of those worse solutions.

In the SA scheme a new neighbor solution is accepted with a probability which decreases over time if its objective function value is higher than that of the prior solution. The probability of accepting a worse solution depends on the degree of worsening. Therefore, infeasible solutions should have a higher objective function value than feasible solutions. We achieve this by adding penalty cost to the objective function value if a solution is infeasible. As this is not necessary for the other algorithms we implemented it only in the SA algorithm as an addition to the general solution evaluation provided by the framework.

The penalty cost value should be chosen as the maximum possible change of the objective function value between any two neighbor solutions. As our neighborhood basically adds or removes exactly one replica, the maximum change can be estimated as

$$\Delta^{max} = \max(\alpha_s) + \beta \cdot \max(d_{c,s}) + \gamma \cdot \overline{d}_{c,s} \cdot \max(r_{c,t}) \tag{21}$$

where $\overline{d}_{c,s}$ denotes the average distance between a client and a server. The three terms of Δ^{max} are as follows.

First, the maximum storage cost coefficient. As only one replica is added or removed the maximum storage cost coefficient is an upper bound for the change in storage cost.

Second, the length $d_{c,s}$ of the longest shortest path in the network weighted with the placement cost coefficient β. This is an upper bound for the change

in placement cost. It occurs if the very first replica is placed as far away as possible from the origin server. If it is an additional replica the change in placement cost will be less because of the multicast transfers.

Third, an estimate for the change in delivery cost. If a replica is removed from a server the demand of this server needs to be served by another one. If a replica is added on a server it will serve its own demand and potentially that of other clients. We multiply the maximum number of requests of a client in one period $r_{c,t}$ by the average distance between a server and a client in the topology $\overline{d}_{c,s}$ and the delivery cost coefficient γ. This is no upper bound as higher values are possible. A real upper bound, e. g. calculated with the maximum distance between a server and a client, would be unrealistic high and would lead to too low acceptance probabilities for infeasible solutions. This problem arises only at this third term due to the multiplication with the maximum amount of requests and the maximum distance between a server and a client which is very unlikely for the delivery because of the SLAs. By using the average distance we aim at penalty cost values such that infeasible solutions get accepted, but with a lower probability than feasible solutions that are worse than their predecessor.

4.2.3 Cooling schedule

We implement the widely used geometric schedule (see Aarts et al. [1]). It is a typical and reasonable static cooling scheduling and we use it with $t_k = 0.9 \cdot t_{k-1}$ as proposed by Aarts et al. [1].

The initial temperature t_0 for the SA algorithm needs to be chosen carefully. Selecting a too low temperature leads to less acceptions of inferior solutions. This increases the risk of being stuck in local minima. In contrast, a too high initial temperature is inefficient because of the resulting longer runtime. It follows that a suitable initial temperature depends on the parameters of the problem. For the geometric schedule, Aarts et al. [1] propose the maximal difference in cost between any two neighbor solutions as an adequate initial temperature. We estimated the maximal difference Δ^{max} in Equation (21). As this value is used as penalty cost for infeasible solutions and a neighbor solution can be infeasible we get $t_0 = 2 \cdot \Delta^{max}$.

The number of evaluated neighbor solutions at each temperature level t_k is in our case $M_k \equiv |\mathcal{V}| \cdot |\mathcal{T}|$ which is basically the size of the neighborhood as also proposed by Aarts et al. [1].

The stop criterion is normally defined as the amount of consecutive temperature levels processed without finding a better solution. In our case we use 6 consecutive temperature levels without a new best solution. In preliminary tests the value of 6 turned out to be a good choice for our problem. As we will show later this leads to a runtime long enough to reach a stable state regarding solution quality.

4.3 Variable Neighborhood Search

Variable neighborhood search (VNS) is a metaheuristic recently introduced by Mladenovic and Hansen [49]. The basic idea is a local search with systematic usage of different neighborhoods. It explores increasingly distant neighborhoods and accepts a new solution only if it is an improvement. The VNS does not follow a trajectory and can jump to more distant solutions if necessary. VNS is well–suited to find local minima as well as to escape from potentially surrounding valleys.

We adapt the basic variable neighborhood search scheme presented in Hansen and Mladenovic [23] (see Algorithm 5). After selecting the set of k_{max} neighborhood structures $\mathcal{N}_k$ ($k = 1, \ldots, k_{max}$) and choosing a stopping criterion the algorithm starts with an initial solution x. In each iteration until the stopping criterion is met the algorithm begins with the first neighborhood, i. e. $k = 1$. In the step called *shaking* a neighbor solution x' is selected at random within the actual neighborhood $\mathcal{N}_k(x)$. x' is used as initial solution for a local search to find the local optimum x''. If the local optimum x'' is better than x it is accepted and replaces the incumbent solution. The search starts over with the new solution and the first neighborhood $\mathcal{N}_1$. This case is called *move*. If the local optimum x'' is not better than x the algorithm switches to the next neighborhood $k = k + 1$ to find a better solution starting from x again. This is repeated until $k = k_{max}$. Should the last neighborhood k_{max} be reached without a better solution being found, a new iteration starts with $k = 1$.

The increasing size of successive neighborhoods is typically achieved by nested neighborhoods where $\mathcal{N}_k \subset \mathcal{N}_{k+1}$. Consequently, the first neighborhoods which are smaller and close to the incumbent solution will be explored more thoroughly than the farther ones.

Algorithm 5 Basic variable neighborhood search scheme

Input: Set of neighborhoods $\mathcal{N}_k$ $(k = 1, \ldots, k_{max})$

1: Select an initial solution x
2: **while** stopping criterion is not met **do**
3: **for** neighborhood $k \leftarrow 1$ **to** k_{max} **do**
4: *shaking:* Generate a random solution x' from the k-th neighborhood of x $(x' \in \mathcal{N}_k(x))$
5: *local search:* Apply a local search method with x' as initial solution to obtain local optimum x''
6: **if** local optimum x'' is better than the incumbent solution x **then**
7: *move:* Move to the new local optimum $(x \leftarrow x'')$ and continue the search with $\mathcal{N}_1$ $(k \leftarrow 1)$.
8: **end if**
9: **end for**
10: **end while**

11: **return** x

In the following section we explain all our developed neighborhoods and local search schemes. As we do not use all procedures at once this is a kind of construction kit to compose different variants of the VNS.

4.3.1 Neighborhoods and local search schemes

Through the VNS scheme we have the opportunity to use several different neighborhoods and well–matched local search procedures. Thus, we can address different aspects of the objective function with dedicated and specialized neighborhoods. The key is the selection of appropriate neighborhoods and their meaningful application in the different stages of the algorithm. In what follows we introduce the components of our VNS and afterwards the implemented VNS variants.

Neighborhoods. As the successive neighborhoods should be of increasing size, nested neighborhoods are a common approach. We define our primary

neighborhoods $\mathcal{N}_k$ for $k = 1, \ldots, k_{max}$ as follows. There are two possible actions to change a solution x. The action ADD places a replica in a period t on a server s which did not store a replica in the incumbent solution. By contrast, the action DROP removes an existing replica from the incumbent solution. The construction of a neighbor solution in neighborhood $\mathcal{N}_k(x)$ works according to Algorithm 6.

Algorithm 6 Generate a neighbor of solution x in neighborhood $\mathcal{N}_k(x)$

Input: Set $\mathcal{V}_r(t)$ of servers $s \in \mathcal{V}_s$ that store a replica in period t
Input: Solution x

$t \leftarrow random(\mathcal{T})$
$k_{ADD} \leftarrow random[0, k]$
$k_{DROP} \leftarrow k - k_{ADD}$
if $k_{ADD} \leq |\mathcal{V}_s \backslash \mathcal{V}_r(t)|$ **and** $k_{DROP} \leq |\mathcal{V}_r(t)|$ **then**
    $\mathcal{A} \leftarrow \emptyset$ (▷ temporary set for the servers that receive a replica)
    for $i \leftarrow 0$ **to** k_{ADD} **do**
        $r \leftarrow random(\mathcal{V}_s \backslash \mathcal{V}_r(t))$
        $\mathcal{V}_r(t) \leftarrow \mathcal{V}_r(t) \cup \{r\}$
        $\mathcal{A} \leftarrow \mathcal{A} \cup \{r\}$
        $x_{r,t} \leftarrow 1$
    end for
    for $i \leftarrow 0$ **to** k_{DROP} **do**
        $r \leftarrow random(\mathcal{V}_r(t) \backslash \mathcal{A})$
        $\mathcal{V}_r(t) \leftarrow \mathcal{V}_r(t) \backslash \{r\}$
        $x_{r,t} \leftarrow 0$
    end for
end if

return x

The neighborhoods $\mathcal{N}_k$ choose a random period at first. In this period k actions ADD or DROP are executed. The number of ADD actions k_{ADD} and DROP actions k_{DROP} with $k = k_{\text{ADD}} + k_{\text{DROP}}$ are chosen randomly. The selection of a feasible server for each action is also random. With these nested neighborhoods we aim at a reduction of the storage and delivery cost. They do not address the placement cost component of the objective function specifically.

As stated in 3.4 the problem is basically a combination of an uncapacitated facility location problem and a Steiner tree problem in graphs. The Steiner tree component is due to the placement cost. Thus, we search for a stable assignment over time with as few changes as possible. Therefore we developed an additional neighborhood called *fill*. The latter is also intended to be a fast and problem–oriented alternative to the time–consuming local search component of the basic VNS scheme (see Algorithm 5). The construction of a neighbor solution in neighborhood *fill* works according to Algorithm 7.

Algorithm 7 Generate a neighbor of solution x in neighborhood *fill*(x)

Input: Set $\mathcal{V}_r(t)$ of servers $s \in \mathcal{V}_s$ that store a replica in period t
Input: Solution x

Set the currently best solution: $x^* \leftarrow x$
Set the currently lowest cost: $cost^* \leftarrow cost(x)$
$s \leftarrow random(\mathcal{V}_s)$
for $t \leftarrow 2$ **to** $T-1$ **do**
    if $x_{s,t-1} = 1$ **and** $x_{s,t} = 0$ **and** $x_{s,t+1} = 1$ **then**
        $x' \leftarrow x$
        $x'_{s,t} \leftarrow 1$
        for all $r \in \mathcal{V}_r(t)$ **do**
            $x'_{r,t} \leftarrow 0$
            $cost' \leftarrow cost(x)$
            if x' feasible **and** $cost' < cost^*$ **then**
                $x^* \leftarrow x'$
                $cost^* \leftarrow cost'$
            end if
            $x'_{r,t} \leftarrow 1$
        end for
        $x \leftarrow x*$
    end if
end for

return x

The *fill* neighborhood starts with the random selection of a server $s \in \mathcal{V}_s$ in the incumbent solution x. On server s we search for assignment gaps, i. e. periods $t \in \mathcal{T}$ where $x_{s,t} = 0$ and $x_{s,t-1} = x_{s,t+1} = 1$ holds. We try to fill the

gaps by placing an additional replica, i. e. $x_{s,t} = 1$. Since we do not want to change the total number of replicas for each period we remove another replica in period t. We evaluate all possibilities to fill the gap $x_{s,t} = 0$ with an existing replica $x_{i,t} = 1$ from a server $i \in \mathcal{V}_s$ in period t. We choose the move with the highest decrease of the objective function value, if there is one. Otherwise we do not fill this gap. To get to the neighbor solution we try to fill all the gaps in the timeline of server s in the incumbent solution x according to this principle.

Local search schemes. We developed two different local search schemes. The first one (l_{swap}) searches in a given period t for the best of all possible swaps of one replica from a server which stores a replica to a server which does not. The local search can be applied to all periods or just to the period that was changed by the last shaking step.

As an alternative we implement the local search denoted by l_{remove} with less computational effort. It evaluates all possible removals of one replica in a period and executes the best. Again, this can be done in just the period most recently changed by a neighborhood $\mathcal{N}_k$ or in all periods.

Both local search variants are best improvement or steepest descent strategies. But in each case we execute only one iteration. This is justified by two aspects. First, the local search is very time consuming. Although we execute only one iteration the local search leads to very long runtimes if applied after every shaking step. Second, each change of the assignment has a huge impact on all three components of the objective function. Besides the storage cost it also affects the delivery to the clients and the placement in two periods. In our problem, a local search which continues until the local optimum has been found would change the solution much more than it is reasonable for the VNS scheme.

4.3.2 Implemented variants of the VNS

An overview of the developed VNS variants is given in Table 4.1.

All VNS variants use the primary neighborhoods $\mathcal{N}_k$ with $k_{max} = 3$. The basic VNS utilizes a local search after each shaking step, i. e. every new solution starts a new local search (see Algorithm 5). Therefore it is

		neighborhood	
		$\mathcal{N}_k(k_{max} = 3)$	$\mathcal{N}_k$ and *fill*
local search	no local search	RVNS	RVNS_{fill}
	l_{remove} every 100 steps $\forall\ t$	VNS-$\text{L}_{100remove}$	VNS_{fill}-$\text{L}_{100remove}$
	l_{swap} every 100 steps $\forall\ t$	VNS-$\text{L}_{100swap}$	VNS_{fill}-$\text{L}_{100swap}$
	l_{remove} every step for current t	VNS_{remove}	-

Table 4.1: Composition of the implemented VNS variants

sufficient to limit the local search to the changed period. Although our local search schemes run just one iteration the basic VNS is very time–consuming. Preliminary tests showed runtimes of one to two hours for the basic VNS with l_{swap} due to the complexity of l_{swap} and its application after each shaking step. Runtimes of this length are inappropriate for the solution of our problem. Hence, we do not consider VNS_{swap} in our experimental investigation. As basic VNS we consider the VNS_{remove} which utilizes the faster l_{remove} local search.

The VNS variant with the shortest runtime is the reduced variable neighborhood search (RVNS) which omits the local search.

To strike a balance between runtime and solution quality we developed several other variants. Through the different combinations of the VNS components we can analyze the contribution of each of them and deduce reasonable approaches.

One approach is to apply the local search not after each shaking step but only every 100 iterations. To compensate for the solutions without a following local search we apply the local search to every period in this variant. We implemented this approach with both local search variants and call them VNS-$\text{L}_{100remove}$ and VNS-$\text{L}_{100swap}$.

To further compensate for the restricted local search and to reduce in particular the placement cost we extend all but the basic VNS with the additional *fill* neighborhood as the last neighborhood to apply, i. e. as neighborhood $\mathcal{N}_4$. We denote these variants as RVNS_{fill}, VNS_{fill}-$\text{L}_{100remove}$ and VNS_{fill}-$\text{L}_{100swap}$.

4.4 Benchmark heuristics

To evaluate our approach we compare it with several algorithms from the literature. To the best of our knowledge almost all work done so far solves replica placement problems with simple heuristics, especially greedy algorithms. We select some simple and the most established and promising approaches which showed good results in the literature as benchmarks for our VNS variants. As we consider a new model which addresses additional aspects compared to those in the literature, we have to adapt the algorithms to fit our problem.

Random algorithms. We implemented three random algorithms. First, the most simple and straight–forward approach *random-add* (Algorithm 8) that randomly adds replicas to servers in the first period until the demand in this period is satisfied and the SLAs in this period are fulfilled. It processes all periods successively in the same way.

Second, *random-delete* (Algorithm 9) which starts with a full placement matrix and removes replicas in the first period from randomly chosen servers as long as the demand is satisfied and the SLAs are fulfilled. Each period is processed like this successively.

Third, *random-delete-all* (Algorithm 10) is identical to *random-delete* except that it does not switch to the next period as soon as a removal would lead to an unsatisfied demand or violated SLAs. Instead it tries to remove a replica from another randomly chosen server. It only continues with the next period if no removal leads to a feasible solution any more. Thus it removes as many replicas as possible.

HotSpot algorithms. An approach introduced by Qiu et al. [59] are the so–called HotSpot algorithms which are based on a ranking. Basically, they place replicas in the regions with the highest demand. We implemented three variants of them and also adapted them to better suit our problem. In preliminary tests all variants showed bad results, even worse than all the random algorithms. Hence, we do not consider them in this work.

Algorithm 8 *random-add*

Create starting solution x with $x_{s,t} \leftarrow 0 \; \forall s \in \mathcal{V}_s, t \in \mathcal{T}$
for all $t \in \mathcal{T}$ **do**
    repeat
        $s \leftarrow random(\mathcal{V}_s \text{ with } x_{s,t} = 0)$
        $x_{s,t} \leftarrow 1$
    until x is feasible in period t
end for
return x

Algorithm 9 *random-delete*

Create starting solution x with $x_{s,t} \leftarrow 1 \; \forall s \in \mathcal{V}_s, t \in \mathcal{T}$
for all $t \in \mathcal{T}$ **do**
    $x' \leftarrow x$
    while x' is feasible **do**
        $x \leftarrow x'$
        $s \leftarrow random(\mathcal{V}_s \text{ with } x_{s,t} = 1)$
        $x'_{s,t} \leftarrow 0$
    end while
end for
return x

Algorithm 10 *random-delete-all*

Create starting solution x with $x_{s,t} \leftarrow 1 \; \forall s \in \mathcal{V}_s, t \in \mathcal{T}$
for all $t \in \mathcal{T}$ **do**
    $\mathcal{V}_{temp} \leftarrow \mathcal{V}_s$
    while $\mathcal{V}_{temp} \neq \emptyset$ **do**
        $s \leftarrow random(\mathcal{V}_{temp})$
        $\mathcal{V}_{temp} \leftarrow \mathcal{V}_{temp} \backslash \{s\}$
        $x' \leftarrow x$
        $x'_{s,t} \leftarrow 0$
        if x' is feasible **then**
            $x \leftarrow x'$
        end if
    end while
end for
return x

Greedy algorithms. For replica placement problems the most frequently applied heuristics in the literature are greedy algorithms (e. g. Cronin et al. [17], Jamin et al. [27], Kangasharju et al. [33], Krishnan et al. [43], Qiu et al. [59], Tang and Xu [69]) which show good results. Thus we implement meaningful variants to put our VNS approach into perspective. The greedy algorithms by Kangasharju et al. [33], Krishnan et al. [43], Qiu et al. [59] successively add replicas based on their respective contribution to the objective function value. This is similar to the common ADD heuristics for facility location problems (see Kuehn and Hamburger [44]). These greedy approaches were generalized and improved by Cronin et al. [17]. The resulting *l-greedy* algorithms add a backtracking of l steps. That means that in order to add 1 replica all combinations of removing l replicas and adding $l + 1$ replicas are evaluated. This is done in each but the first iteration of the algorithm to find the next greedy step. Thus, for $l = 0$ this algorithm is identical to the simple greedy approaches. For $l = 1$ the algorithm is considerably more complex and time–consuming but is expected to have much better results.

Tang and Xu [69] call all those approaches *l-greedy-insert*. They are not the best choice for our problem as they start without any replicas and add them successively. For our model all solutions in the beginning of the algorithm would be infeasible. It is complicated to calculate an useful objective function value for the selection of the next greedy step. The similar ADD heuristics for FLPs are mostly used for uncapacitated problems where there are no infeasibilities. For capacitated problems there is typically inserted a dummy client with zero transportation cost for the excess supply and a dummy supplier with transportation cost M for the evaluation of the otherwise infeasible solutions. For our model the SLAs are an additional difficulty. They can not be easily evaluated for infeasible solutions and different infeasible solutions are hard to differentiate. Tang and Xu [69] propose *l-greedy-delete* algorithms as an alternative which is much easier to adapt to our problem. As this approach led to results of comparable quality and is more suitable for our problem we select it for our comparison. *l-greedy-delete* algorithms start with the maximum amount of replicas and remove the replicas with the maximum cost reduction successively as long as there is a cost reduction. Again, for $l = 0$ this is similar to DROP heuristics for FLPs (see Feldman et al. [20]).

The pseudocode for *0-greedy-delete* as we implement it is given in Algorithm 11. One adaption with respect to our problem is that we apply the

Algorithm 11 *0-greedy-delete*

```
1: Create starting solution x with x_{s,t} ← 1 ∀s ∈ V_s, t ∈ T
2: Set currently best solution: x* ← x
3: Set currently lowest cost: cost* ← cost(x)
4: for all t ∈ T do
5:     repeat
6:         betterSolutionFound ← false
7:         for all s ∈ V_s with x_{s,t} = 1 do
8:             x' ← x
9:             x'_{s,t} ← 0
10:            cost' ← cost(x')
11:            if x' is feasible and cost' < cost* then
12:                x* ← x'
13:                cost* ← cost'
14:                betterSolutionFound ← true
15:            end if
16:        end for
17:        x ← x*
18:    until betterSolutionFound = false
19: end for
20: return x
```

greedy delete steps to each period of the solution matrix individually. The backtracking of *l-greedy-delete* if $l > 0$ works analogous to the *l-greedy-insert*, i. e. the best combination of adding l replicas and removing $l + 1$ replicas is used in each but the first iteration in which just $l + 1$ replicas are removed. Our implementation of *1-greedy-delete* is shown in Algorithm 12.

As our model incorporates new aspects of the problem we have some possibilities regarding the server selection for the delete and insert steps.

First, we implement the straight–forward selection *l-greedy-delete* as shown in Algorithms 11 and 12. We select the server on which the removal of a replica leads to the maximum cost reduction while maintaining feasibility.

Second, we implement a selection scheme which pays more attention to the SLAs. Preliminary tests showed that the *l-greedy-delete* algorithms mostly stop processing a period because of an infeasibility regarding the

SLAs. We want to avoid this as long as possible with the *l-greedy-delete*$_{QoS}$ algorithms. The pseudocode for *0-greedy-delete*$_{QoS}$ is given in Algorithm 13. We process each period individually and in each period we iterate until no improvement can be found. In each iteration we remove a replica from the current period such that the achieved QoS of the solution is as high as possible. From several possibilities with equally high QoS we choose the one with the lowest objective function value. Hence, we try to reduce the cost without lowering the QoS as long as possible. Only if there is no other possibility we accept the minimum reduction of the QoS and search on this QoS level for removals with lower objective function values. *1-greedy-delete*$_{QoS}$ works analog to *0-greedy-delete*.

Algorithm 12 *1-greedy-delete*

Create starting solution x with $x_{s,t} \leftarrow 1 \; \forall s \in \mathcal{V}_s, t \in \mathcal{T}$
Set currently best solution: $x^* \leftarrow x$
Set currently lowest cost: $cost^* \leftarrow cost(x)$
for all $t \in \mathcal{T}$ **do**
    $betterSolutionFound \leftarrow$ `false`
    for all $j \in \mathcal{V}_s$ with $x_{j,t} = 1$ **do**
        for all $k \in \mathcal{V}_s$ with $x_{k,t} = 1$ and $k \neq j$ **do**
            $x' \leftarrow x$
            $x'_{j,t} \leftarrow 0$
            $x'_{k,t} \leftarrow 0$
            $cost' \leftarrow cost(x')$
            if x' is feasible **and** $cost' < cost^*$ **then**
                $x^* \leftarrow x'$
                $cost^* \leftarrow cost'$
                $betterSolutionFound \leftarrow$ `true`
            end if
        end for
    end for
    $x \leftarrow x^*$

    while $betterSolutionFound =$ `true` **do**
        $betterSolutionFound \leftarrow$ `false`
        for all $i \in \mathcal{V}_s$ with $x_{i,t} = 0$ **do**
            for all $j \in \mathcal{V}_s$ with $x_{j,t} = 1$ **do**
                for all $k \in \mathcal{V}_s$ with $x_{k,t} = 1$ and $k \neq j$ **do**
                    $x' \leftarrow x$
                    $x'_{i,t} \leftarrow 1$
                    $x'_{j,t} \leftarrow 0$
                    $x'_{k,t} \leftarrow 0$
                    $cost' \leftarrow cost(x')$
                    if x' is feasible **and** $cost' < cost^*$ **then**
                        $x^* \leftarrow x'$
                        $cost^* \leftarrow cost'$
                        $betterSolutionFound \leftarrow$ `true`
                    end if
                end for
            end for
        end for
        $x \leftarrow x^*$
    end while
end for
return x

Algorithm 13 *0-greedy-delete*$_{QoS}$

Create starting solution x with $x_{s,t} \leftarrow 1 \; \forall s \in \mathcal{V}_s, t \in \mathcal{T}$
Set currently best solution: $x^* \leftarrow x$
Set currently lowest cost: $cost^* \leftarrow cost(x)$
for all $t \in \mathcal{T}$ **do**
    repeat
        $betterSolutionFound \leftarrow$ `false`
        $qos^* \leftarrow 0$
        for all $s \in \mathcal{V}_s$ with $x_{s,t} = 1$ **do**
            $x' \leftarrow x$
            $x'_{s,t} \leftarrow 0$
            $qos' \leftarrow qos(x')$
            $cost' \leftarrow cost(x')$
            if x' is feasible **and** $cost' < cost^*$ **then**
                if $qos' > qos^*$ **then**
                    $x^* \leftarrow x'$
                    $qos^* \leftarrow qos'$
                    $cost^{qos^*} \leftarrow cost'$
                    $betterSolutionFound \leftarrow$ `true`
                else if $qos' = qos^*$ **and** $cost' < cost^{qos^*}$ **then**
                    $x^* \leftarrow x'$
                    $cost^{qos^*} \leftarrow cost'$
                    $betterSolutionFound \leftarrow$ `true`
                end if
            end if
        end for
        $x \leftarrow x^*$
        $cost^* \leftarrow cost^{qos^*}$
    until $betterSolutionFound =$ `false`
end for
return x

Chapter 5

Experimental Investigation

5.1 Experimental setup

5.1.1 Implementation

All algorithms are implemented in Java and use object–oriented programming. They are embedded in a modular Java framework which enables the use of future solution approaches. The framework is also capable of solving the problem instances with CPLEX. Therefore, all the input like the topology and the request data as well as the parameters are converted to match the application programming interface (API) of the CPLEX library.

Independent from the solution method the user can choose whether the parameters and configuration data are loaded from an XML file, from a Microsoft Excel Spreadsheet or from a database. The input via XML is mainly intended for the solution of single instances. When using the Excel or the database connection the framework allows batch processing, i. e. fully automatic execution of any number of runs with arbitrary parameter combinations and solution methods. The solutions and all relevant data are saved back to the Excel file or the database respectively.

5.1.2 Test instances

As it is very hard to get real data of CDNs we obtained data from the website of Akamai [6], one of the biggest commercial CDN providers. To compensate for some unreliable data we vary the parameters of a base case across a broad range of values using a factorial ceteris paribus design.

A fundamental part of a test instance is the network topology. We use general topologies created with the network generator BRITE (see Medina et al. [48]). We configured BRITE to use the well–known Waxman model for topology creation (see Waxman [77]). This is one of the most common approaches for generating graphs algorithmically (see Jamin et al. [26], Jeon et al. [29], Qiu et al. [59], Shi and Turner [66], Tang and Xu [69], Zhang et al. [82]) and was shown to generate realistic networks (see Medina et al. [47], Tangmunarunkit et al. [70]).

As stated earlier, we can limit the topologies to the nodes $s \in \mathcal{V}_s$ that are part of the content delivery network and the one origin server v_0. The millions of clients do not need to be modeled explicitly. According to Akamai [6], there are over 55,000 servers worldwide to date. The servers are clustered in data centers with several servers in each data center. Several data centers are typically deployed in the wider area of a city as the CDN provider tries to have servers as near as possible to the clients. This is done by using data centers in the networks of different Internet service providers (ISP) instead of just one. Hence, on each geographic location like a city several data centers with a number of servers in each of them are ready to serve requests. We use this to aggregate the number of servers for our model. According to Akamai [6] we can confine a realistic worldwide topology to about 600 nodes, equal to the number of cities or, more generally, locations with one or more data centers in the real CDN.

From the provided information we can conclude that there are about 300 locations in the USA, about 50 are located in the European Union. We choose the latter case of 50 nodes for our study. While solving the model for a complete CDN is desirable, considering regional problems is still helpful as, for the sake of the SLAs, replicas can not be placed too far away from the requests. The main reason for the regional consideration is the size of the resulting problem, especially because we consider the dynamic case. For our study we consider a planning horizon of a whole day with 12 periods of 2

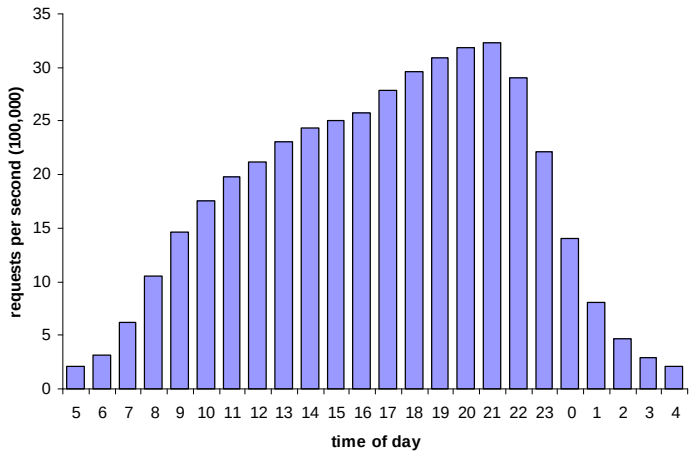

Figure 5.1: Example for a request rate profile of Europe

hours length each. With this configuration we obtain test instances where the DRPSL has 36,172 variables, 3,111 of them binary, and 35,535 constraints.

For the request rates on the nodes we analyzed real–time data on the homepage of Akamai [6]. We extracted the raw data provided by a web service which is used for a graphical live visualization of the actual traffic on the homepage. The most useful information is the actual number of raw requests received at the servers, separated by continental regions like North America or Europe. Additionally, the history of the last 24 hours of this data is provided. Therefore, we have time–varying request profiles given. An example for a request rate profile of Europe can be seen in Figure 5.1. The profiles for the different regions are structurally very similar. They vary mainly in the number of requests but not in the distribution.

These values are scaled to reasonable request rates of the European Union which could occur for the files of one customer. We do this as we consider a complete homepage of one customer as the single object in our model. Besides the request rates varying over time, we further take into account the different time zones in the geographic area represented by the topology. We do so by shifting the applied request profiles according to the time zones, i. e. each server is assigned a request rate according to its local time. In each timezone all servers have the same request rates.

We alter a base case by varying the following three types of factors for which real and reliable data is hard to gather.

First, we vary the load capacity C_s^L of the servers as a performance related value. Although our model can handle different load capacities for the servers, we assume all of them to be identical for our experimental investigation. As we consider an abstraction with several servers on each node in the topology and periods with a length of two hours, absolute values for the number of requests a node can process per period are not very meaningful. Therefore, we vary the parameter ρ^{max} and derive C_s^L from it. With

$$\rho^{max} = \frac{\max\limits_{t \in \mathcal{T}} \left(\sum\limits_{c \in \mathcal{V}_c} r_{c,t} \right)}{\sum\limits_{s \in \mathcal{V}_s} C_s^L} \tag{22}$$

we denote the average workload in the period with the maximum number of requests if all servers store a replica, i. e. the ratio of the total amount of requests in the period to the sum of all load capacities.

Second, we vary SLA related parameters, i. e. the percentage λ of the requests which have to be served within the maximum latency q and the maximum latency q itself. q is varied indirectly by a factor l. As q depends heavily on the topology, we derive q from the average length of the edges in the topology $\bar{\delta}_{i,j}$ multiplied by the factor l.

$$q = \bar{\delta}_{i,j} \cdot l \qquad i, j \in \mathcal{V} \tag{23}$$

Third, the cost coefficients α_s, β and γ are varied. As we do not expect any special insights from varying the storage cost of the nodes, we assume $\alpha_s \equiv \alpha$ and consider only α in our study.

The parameters of our test instances are summarized in Table 5.1. The values were chosen in preliminary tests to fit the specifics of each parameter and to cover typical and important values. Hence the number of values for each parameter differ as well as their margins. In total we have 38 different problem instances. The value of the base case is set in bold. A detailed overview of the different test instances can be found in Table B.1 in the Appendix.

Param.	experimental levels
ρ^{max}	0.9, 0.8, 0.75, 0.7, **0.66**, 0.6, 0.5, 0.4, 0.33, 0.3, 0.25, 0.2, 0.1
λ	0.75, 0.9, 0.95, **0.99**, 1.0
l	0.5, 0.75, 1.0, 1.5, **2.0**, 3.0
α	10, 50, 100, 500, **1000**, 2000
β	0, 0.1, **0.2**, 0.5, 1
γ	0, **0.01**, 0.05, 0.1, 0.5, 1, 2, 5

Table 5.1: Parameters and levels for the experimental test design (see also Appendix Table B.1)

5.1.3 Solution methods

We apply variants of five different solution approaches to the problem. An overview of the solution methods that were applied to all instances is given in Table 5.2 at the end of this section.

First, for a general analysis of the problem and to gain some insights into the solution space we generate and evaluate random solutions (*eval-random*) for some of the instances. As we do not know the optimal number of replicas we proceed as follows. For each number of replicas in the interval $[1, T \cdot |\mathcal{V}_s|]$ we generate 1,000 random replica assignments. As the test instances have 50 nodes and a planning horizon of 12 periods we generate 600,000 random solutions for the selected instances. We evaluate these solutions with our framework, i. e. we calculate the corresponding storage, placement and delivery cost as described in Section 4.1.2.

From the 1,000 samples for each number of replicas we compute the mean, the minimum and maximum values as well as the 25% and 75% quantiles for every recorded value. With this information we can put the results of the other approaches into perspective. Given the number of replicas in a specific solution, we can compare this solution to the according benchmark values. As *eval-random* does not optimize the objective function value and is not applied to all instances it is not shown in Table 5.2.

Second, we employ CPLEX 10.1 with default parameters to solve the strong LP–relaxation of DRPSL2 (CPLEX LP-s). We tighten the resulting solution with our lower bound for the placement cost, outlined in Section 3.6. The improved lower bound (CPLEX LP-i) obtained this way is used throughout the

following analysis. For a comparison we also solve the weak LP–relaxation of DRPSL (CPLEX LP-w). Next, we use CPLEX with a time limit of 3,600 seconds (CPLEX 3,600s) to solve DRPSL2. By this we want to see if CPLEX can find optimal or near–optimal solutions within a relatively long runtime. In a real CDN the assignment of the replicas is changed several times per hour, even several times per minute are realistic. Hence, our model would most likely be used with much shorter time intervals and perhaps a shorter planning horizon than in our experimental study. Thus, a solution method with a runtime of one hour is not applicable in the real world. Therefore, we additionally solve DRPSL2 with CPLEX and time limits of 60 seconds (CPLEX 60s), 300 (CPLEX 300s) and 600 seconds (CPLEX 600s). To get further insights into the impact of runtime on the solution quality we finally employ CPLEX with a relative gap limit of 0.1 (CPLEX 10%).

Third, we apply the benchmark heuristics, i. e. *random-add*, *random-delete*, *random-delete-all*, *0-greedy-delete*, *0-greedy-delete*$_{QoS}$, *1-greedy-delete* and *1-greedy-delete*$_{QoS}$. The three random algorithms introduced in Section 4.4 are applied 10 times to each instance as they are not deterministic. The results are averaged. The greedy algorithms are deterministic, thus they are executed only once for each instance. All seven benchmark algorithms have no hard time or iteration limit. They run without stopping criterion until no improvement can be found.

Fourth, we apply our SA algorithm. Again, we solve each instance 10 times independently. In the following analysis we always provide the average values of these 10 runs (SA) if no notable differences occurred.

Fifth, the VNS algorithms described in Section 4.3 are applied 10 times independently on each instance and we provide the average values for these runs. The stopping criterion for the VNS algorithms is selected based on preliminary tests. It is defined as the maximum number of iterations. The value 4,000 turned out in preliminary tests to be a good compromise between runtime and solution quality.

All problem instances are solved using one Intel Xeon core with 1.6 GHz on a workstation with 6 GB RAM. The whole framework, all metaheuristics, the benchmark heuristics and all components of the algorithms are implemented in Java 6.0 and without the use of CPLEX. CPLEX is only used for the benchmark solutions.

Solution method	stopping criterion
CPLEX (LP-w)	none
CPLEX (LP-s)	none
CPLEX (LP-i)	none
CPLEX (60s)	60 sec runtime
CPLEX (300s)	300 sec runtime
CPLEX (600s)	600 sec runtime
CPLEX (3,600s)	3,600 sec runtime
CPLEX (10%)	0.1 relative gap
random-add	none
random-delete	none
random-delete-all	none
0-greedy-delete	none
0-greedy-delete$_{QoS}$	none
1-greedy-delete	none
1-greedy-delete$_{QoS}$	none
SA	6 temperature levels without new local optimum
RVNS	4,000 iterations
RVNS$_{fill}$	4,000 iterations
VNS-L$_{100remove}$	4,000 iterations
VNS$_{fill}$-L$_{100remove}$	4,000 iterations
VNS-L$_{100swap}$	4,000 iterations
VNS$_{fill}$-L$_{100swap}$	4,000 iterations
VNS$_{remove}$	4,000 iterations

Table 5.2: Solution methods applied

5.2 Computation times

The average runtime of each algorithm is shown in Table 5.3. We average over all 380 runs, i. e. 10 runs for each of the 38 instances, for the algorithms that are not deterministic (SA, VNS and random variants). For the deterministic algorithms, i. e. CPLEX and greedy variants, we average over 38 runs.

When applied to DRPSL2, CPLEX could not solve any of the instances to optimality within the time limit of 3,600 seconds. For two instances CPLEX could not find any feasible solution within 60 seconds. Running CPLEX with a gap limit of 10% leads to an average runtime of 2.4 hours and a maximum runtime of 44.9 hours.

Solution method	min. runtime	avg. runtime	max. runtime
CPLEX (LP-w)	0	< 1	3
CPLEX (LP-s)/(LP-i)	0	11	35
CPLEX (10%)	0	8,629	161,795
random-add	0	0	0
random-delete	0	0	0
random-delete-all	0	0	0
0-greedy-delete	0	4	6
0-greedy-delete$_{QoS}$	0	4	6
1-greedy-delete	87	1,237	1,585
1-greedy-delete$_{QoS}$	86	926	1,098
SA	3	21	32
RVNS	7	17	26
RVNS$_{fill}$	9	25	63
VNS-L$_{100remove}$	13	24	41
VNS$_{fill}$-L$_{100remove}$	14	28	63
VNS-L$_{100swap}$	20	321	510
VNS$_{fill}$-L$_{100swap}$	23	322	512
VNS$_{remove}$	47	272	416

Table 5.3: Average runtimes of the algorithms [s]

Compared to the solution of the unrelaxed problem with CPLEX, the LP–relaxation is solved fast. This is consistent with the bad quality of the lower bounds (see Section 3.6). Solving the weak LP–relaxation (CPLEX LP-w) took for all but one instance less than one second, the outlier took 3 seconds. As expected, the strong LP–relaxation (CPLEX LP-s) results in lower bounds which are much better. Compared to the objective function value of CPLEX LP-w they are improved by approx. 27% on average. To compute the strong LP–relaxation CPLEX needed between 0 and 35 seconds, 11 seconds on average.

It can be clearly seen that the random and greedy algorithms run very fast, except the complex greedy variants with a backtracking of 1 step. These two *1-greedy-delete* algorithms are probably not suitable for solving real problems due to their long runtime of 15 to 20 minutes. They also run longer than all metaheuristics.

Our simulated annealing algorithm took on average between 3 and 32 seconds for the different instances. The average runtime over all 380 runs was 21 seconds.

When comparing the SA to the VNS variants we can see that only the less complex VNS variants have runtimes similar to those of the SA. The VNS without local search (RVNS and RVNS$_{fill}$) and the VNS with the faster local search l_{remove} applied every 100 steps (VNS-L$_{100remove}$ and VNS$_{fill}$-L$_{100remove}$) are comparable to the SA. They run between 17 and 28 seconds.

The additional neighborhood *fill* has almost no influence on the runtime of the VNS variants. Only for the RVNS there is a difference which is most likely due to the more frequent use of the *fill* neighborhood in RVNS$_{fill}$. The *fill* neighborhood takes more time than the other neighborhoods because of its evaluation of several possibilities to fill each gap (see Section 4.3.1). Since *fill* is implemented as the last neighborhood it is only used if no other neighborhood in the preceding steps found a better solution than the incumbent. A local search increases the probability to find a better solution than the incumbent, in which case the algorithm starts over with the first neighborhood without using neighborhood *fill*. As there is no local search in the RVNS the *fill* neighborhood is used more often. In contrast, all VNS variants except the two RVNS algorithms use some kind of local search, thus the *fill* neighborhood is used less frequently and the runtime does not increase much.

As expected, the type of local search has a big influence on the runtime. The more complex and thorough the local search is, the longer is the runtime of the algorithm. Considering the local search l_{remove} it takes on average 24 seconds to apply it every 100 iterations (VNS-L$_{100remove}$) in contrast to 272 seconds if it is applied after every shaking step (VNS$_{remove}$). The comparison of the variants VNS-L$_{100remove}$ (24 seconds) and VNS-L$_{100swap}$ (321 seconds) as well as VNS$_{fill}$-L$_{100remove}$ (28 seconds) and VNS$_{fill}$-L$_{100swap}$ (322 seconds) clearly shows the higher complexity and time consumption of the l_{swap} local search compared to l_{remove}. Note that applying l_{swap} every 100 steps takes even longer than applying l_{remove} after every step.

Considering just the runtime we conclude that CPLEX is presumably not suitable to solve real world problems whereas most of the implemented algorithms are in general applicable. Only the runtime of the *1-greedy-delete*

algorithms is certainly too long for an efficient real–world use. It depends on the specific purpose if a runtime of 5 minutes for the complex VNS variants is acceptable. If the replica assignment is to be updated very frequently like every minute for example, the two variants with l_{swap} local search and the VNS_{remove} are also not suited.

5.3 Solution quality

5.3.1 Lower bound

First we want to look at the quality of the lower bounds, especially when it comes to the placement cost. If we look at the results of our general problem analysis (*eval-random*) we can see evidence for the bad lower bound of the placement cost CPLEX (LP-s) provides. Figure 5.2 shows the results regarding the placement cost for instance $i30$ which has a higher placement cost coefficient β than the base case. We can see the lower bound CPLEX (LP-s) provides without our improvement. Like in every other instance it is nearly 0. The lower bound improved by our MUT approach (see Section 3.6) is given by the bold line for each number of replicas so that we can evaluate the placement cost of a solution no matter how many replicas it stores.

We can see that our improvement of the CPLEX (LP-s) lower bound is very helpful and leads to a much better lower bound for the placement cost. If we take a look at the placement cost of solutions from our algorithms we can already see some basic results. The more complex the heuristic or metaheuristic is, the lower are the achieved placement cost. The random based algorithms, even the most complex *random-delete-all*, find solutions that are still within the 0.75 quantile of our randomly generated solutions (*eval-random*). We see that this is one of the instances for which CPLEX (600s) can not find solutions anywhere near the optimum. With high placement cost and by far too many replicas the solution z of CPLEX (600s) has a solution gap $\Delta = \frac{z-LB}{z}$ of 48.4%. SA and the best VNS variants perform well with total gaps of around 4–5%.

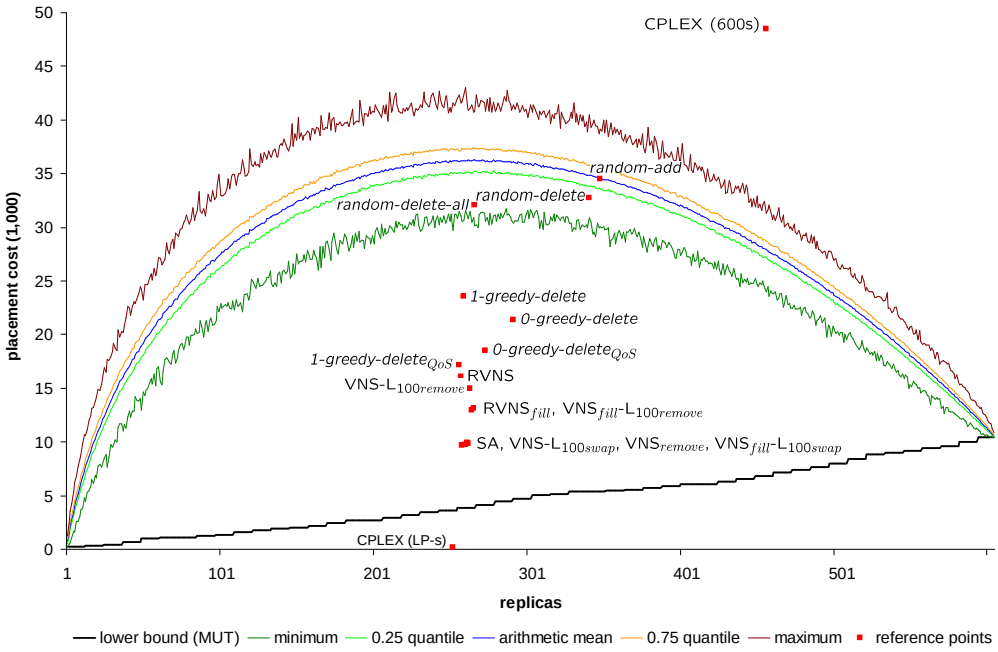

Figure 5.2: Analysis of the placement cost (*eval-random*) where for each amount of replicas 1,000 random solutions are evaluated

5.3.2 Progress over runtime

Now we look at the values of the objective function during runtime for two of the metaheuristics.

Figure 5.3 shows the progress of the objective function value when applying SA to instance $i30$. The typical behavior of a well configured SA can be clearly seen, i. e. the frequent rise of objective function values due to the acceptance of worse solutions to overcome local minima. In the example the SA evaluated almost 21,000 solutions, about 9,000 solutions were accepted and thus are shown in the figure. On this instance, 23% or about 4,800 of the evaluated solutions are improved solutions, about 470 are new best solutions. Over all instances of the test set SA evaluated on average 16,536 solutions with a maximum of 40,200 evaluations.

The VNS variants behave quite different. As defined by the stopping criterion they run 4,000 iterations. But the number of solutions they evaluate

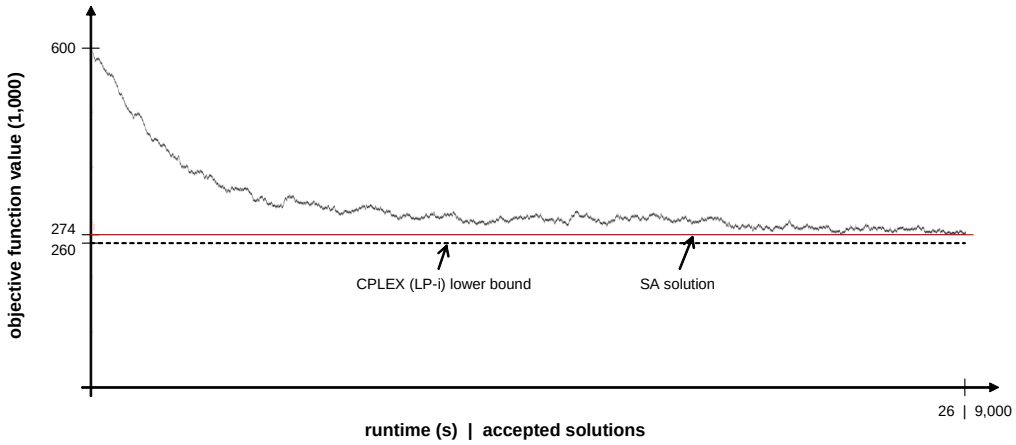

Figure 5.3: Typical progress of the objective function value during SA (instance $i30$)

Solution method	# evaluated solutions
SA	16,537
RVNS	14,255
RVNS$_{fill}$	19,700
VNS-L$_{100remove}$	20,064
VNS$_{fill}$-L$_{100remove}$	24,945
VNS-L$_{100swap}$	234,029
VNS$_{fill}$-L$_{100swap}$	237,829
VNS$_{remove}$	225,835

Table 5.4: Number of evaluated solutions

depends heavily on whether a local search is used, how often the local search is used, the complexity of the local search and whether the *fill* neighborhood is used or not. Table 5.4 provides for each metaheuristic the average number of evaluated solutions over all instances.

Figure 5.4 gives an example for the sequence of the neighborhoods during VNS. It shows the first 100 neighborhoods used by VNS$_{fill}$-L$_{100remove}$ during a run on instance $i30$. Note that one neighborhood in the figure is not equal to one iteration of the VNS. In each iteration the VNS runs through as many of the defined neighborhoods as necessary to find an improved solution (see Algorithm 5). In the later stages of VNS the 'bigger' neighborhoods are used more and more often as improved solutions get harder to find. In the

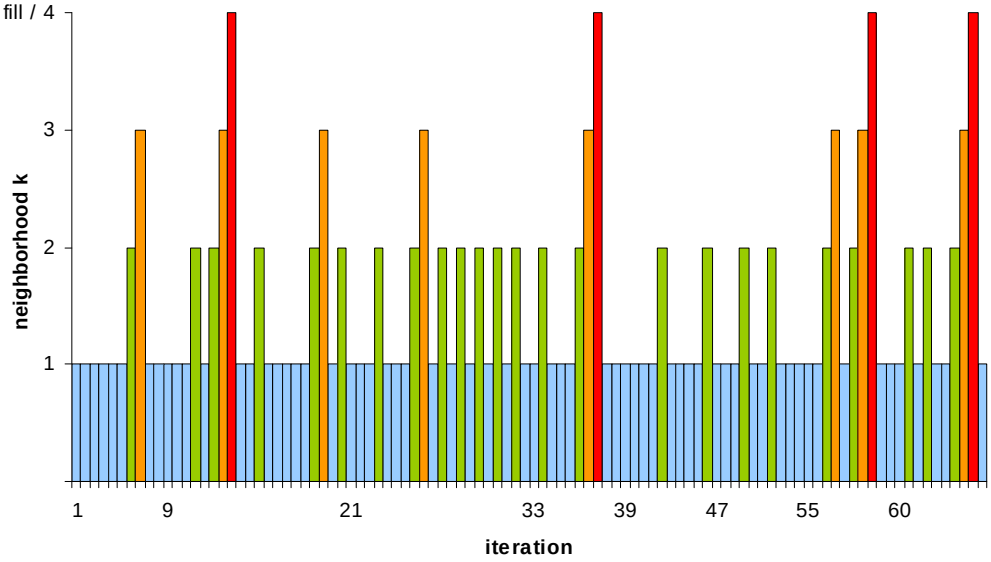

Figure 5.4: Typical sequence of neighborhoods in the beginning of VNS (VNS_{fill}-$\text{L}_{100remove}$, instance $i30$)

final stages each iteration typically uses all neighborhoods.

We already showed that the runtimes of the more simple VNS variants are on the same level as the SA. Adding complex local search components leads to much longer runtimes. However, the progress of the objective function value is very similar for all VNS variants. A typical example and one with a runtime comparable to the SA is shown in Figure 5.5. This is again for instance $i30$, solved with VNS_{fill}-$\text{L}_{100remove}$. For this instance about 10% or 400 of the iterations lead to solutions which are improved over the previous solution, i. e. to new best solutions as worse solutions are never accepted.

We can conclude that the SA finds good solutions faster than the VNS variants. In the beginning of the SA algorithm the objective function value decreases steeper than during the VNS. Hence, if the available runtime is very limited the SA is the better choice of the metaheuristics. We will now compare the solution quality of the algorithms in more detail.

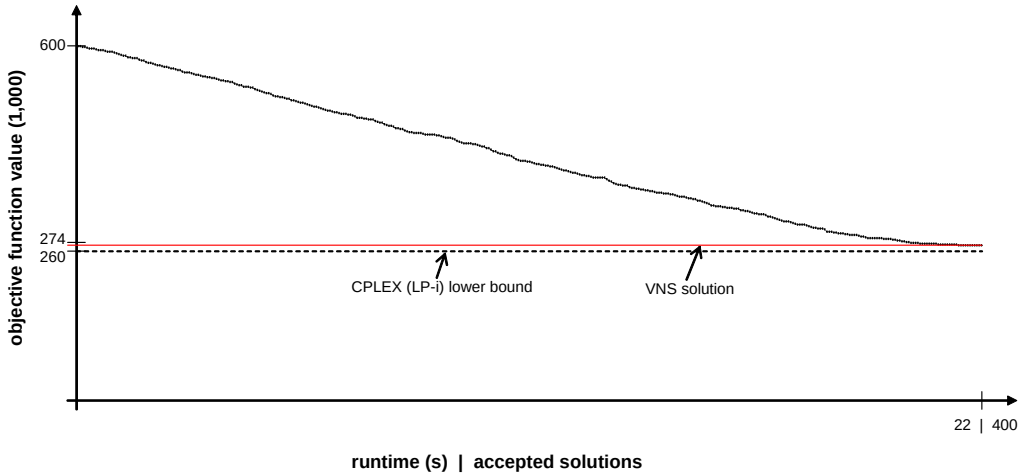

Figure 5.5: Typical progress of the objective function value during VNS (VNS_{fill}-$\text{L}_{100remove}$, instance $i30$)

5.3.3 Evaluation of the algorithms

To get an overview of the solution quality of the algorithms we use two different measures.

Ranking for each instance. For the first measure we rank the objective function values of the algorithms for each of the 38 instances separately. The algorithm with the lowest objective function value for an instance gets rank 1 while the algorithm with the worst performance gets rank 19. All ranks are shown in Table 5.5.

Table 5.6 provides the number of instances each algorithm solves with the best solution. If we just consider the implemented algorithms and leave CPLEX out, for $\frac{2}{3}$ of the instances the basic VNS with l_{remove} local search after each step (VNS_{remove}) provides the best solution. Using l_{swap} local search but only every 100 steps (VNS-$\text{L}_{100swap}$) leads to the best solution for 5 instances. The same variant with the additional neighborhood *fill* in VNS_{fill}-$\text{L}_{100swap}$ finds the best solution for 7 of the 38 instances. When also considering the solutions obtained by CPLEX, the picture changes only with respect to CPLEX (3,600s). CPLEX (3,600s) provides better solutions than the other algorithms for 14 of the 38 instances. It improves the solution for 11 of the 25 instances where VNS_{remove} has generated the so far best solution. Within 600 seconds CPLEX does not find the best solution for any instance.

Instance	$i1$	$i2$	$i3$	$i4$	$i5$	$i6$	$i7$	$i8$	$i9$	$i10$	$i11$	$i12$	$i13$
CPLEX (60s)	19	19	19	18	18	18	19	19	19	18	19	19	19
CPLEX (300s)	11	12	18	18	18	18	18	18	18	18	6	11	4
CPLEX (600s)	9	10	11	11	10	17	10	11	10	11	5	4	3
CPLEX (3,600s)	10	4	4	4	5	10	8	4	1	3	1	1	1
random-add	17	17	16	16	16	15	16	16	17	17	17	17	18
random-delete	18	18	17	17	17	16	17	17	16	16	18	18	17
random-delete-all	14	14	14	13	13	12	14	14	14	14	16	16	16
0-greedy-delete	16	16	15	15	15	14	15	15	15	15	15	15	14
0-greedy-delete$_{QoS}$	15	15	13	14	14	13	13	13	13	13	14	14	14
1-greedy-delete	13	13	12	12	12	11	12	12	12	12	13	13	13
1-greedy-delete$_{QoS}$	12	11	9	10	11	9	11	10	11	10	12	12	12
SA	8	9	10	7	7	6	6	9	9	8	11	8	10
RVNS	7	8	8	9	9	8	9	8	7	7	10	10	11
RVNS$_{fill}$	5	5	6	6	6	5	5	5	5	6	7	6	7
VNS-L$_{100remove}$	6	7	7	8	8	7	7	7	8	9	9	9	8
VNS$_{fill}$-L$_{100remove}$	4	6	5	5	4	4	4	6	6	5	8	7	9
VNS-L$_{100swap}$	2	3	3	3	3	3	3	3	3	4	3	5	6
VNS$_{fill}$-L$_{100swap}$	3	2	2	1	1	2	1	2	4	2	4	3	5
VNS$_{remove}$	1	1	1	2	2	1	2	1	2	1	2	2	2

Instance	$i14$	$i15$	$i16$	$i17$	$i18$	$i19$	$i20$	$i21$	$i22$	$i23$	$i24$	$i25$	$i26$
CPLEX (60s)	19	19	19	19	5	17	17	19	18	12	19	19	19
CPLEX (300s)	18	18	18	18	3	6	3	18	18	11	12	12	18
CPLEX (600s)	11	12	17	11	2	4	2	3	10	10	9	10	11
CPLEX (3,600s)	6	9	6	1	1	1	1	1	6	8	4	4	4
random-add	14	16	15	17	18	19	18	16	15	16	18	18	16
random-delete	15	15	16	16	19	18	19	17	16	17	17	17	17
random-delete-all	13	14	13	13	16	16	16	14	12	18	16	16	14
0-greedy-delete	17	17	14	14	17	14	10	15	17	15	14	14	15
0-greedy-delete$_{QoS}$	12	11	11	14	14	7	7	13	14	14	13	13	13
1-greedy-delete	16	13	12	12	11	5	6	11	13	19	15	15	12
1-greedy-delete$_{QoS}$	10	10	10	9	4	3	5	5	11	13	11	11	10
SA	9	8	9	10	15	15	15	12	9	9	10	9	6
RVNS	7	6	8	8	12	11	13	8	8	7	8	7	9
RVNS$_{fill}$	4	5	4	5	13	9	12	7	5	6	7	6	5
VNS-L$_{100remove}$	8	7	7	7	9	10	11	9	7	5	6	8	8
VNS$_{fill}$-L$_{100remove}$	5	4	5	6	8	13	14	10	4	2	5	5	7
VNS-L$_{100swap}$	2	1	3	2	7	8	9	6	3	3	2	3	2
VNS$_{fill}$-L$_{100swap}$	3	3	1	4	10	12	8	4	1	4	3	1	1
VNS$_{remove}$	1	2	2	3	6	2	4	2	2	1	1	2	3

Instance	$i27$	$i28$	$i29$	$i30$	$i31$	$i32$	$i33$	$i34$	$i35$	$i36$	$i37$	$i38$	$\sum$
CPLEX (60s)	18	4	19	18	19	19	19	19	11	11	9	10	640
CPLEX (300s)	18	3	12	18	18	17	11	9	6	6	4	11	494
CPLEX (600s)	11	2	8	17	10	17	9	10	5	5	3	7	338
CPLEX (3,600s)	5	1	6	5	4	1	4	5	1	1	1	2	144
random-add	16	18	18	16	16	15	17	17	14	14	14	14	620
random-delete	17	19	17	15	17	16	18	18	13	13	13	13	630
random-delete-all	13	15	14	13	14	14	13	13	15	15	15	15	544
0-greedy-delete	15	17	16	14	15	13	16	16	17	17	17	17	578
0-greedy-delete$_{QoS}$	14	16	15	12	13	12	15	14	18	18	18	18	517
1-greedy-delete	12	14	13	11	12	7	14	15	19	19	19	19	494
1-greedy-delete$_{QoS}$	9	12	11	10	11	10	10	12	16	16	16	16	401
SA	10	13	10	4	5	5	12	11	12	12	12	12	362
RVNS	7	11	9	9	9	11	8	8	10	10	11	8	334
RVNS$_{fill}$	4	10	5	7	7	6	5	6	8	9	10	9	248
VNS-L$_{100remove}$	8	8	7	8	8	9	7	7	9	8	8	3	292
VNS$_{fill}$-L$_{100remove}$	6	9	4	6	6	8	6	4	7	7	7	6	237
VNS-L$_{100swap}$	2	5	1	2	3	3	1	2	3	3	6	5	131
VNS$_{fill}$-L$_{100swap}$	3	7	3	3	2	4	2	3	4	4	5	4	131
VNS$_{remove}$	1	6	2	1	1	2	3	1	2	2	2	1	75

Table 5.5: Ranks of the algorithms for each instance based on the objective function values

Solution method	# best solutions	# best solutions (incl. CPLEX)
CPLEX (60s)		0
CPLEX (300s)		0
CPLEX (600s)		0
CPLEX (3,600s)		14
random-add	0	0
random-delete	0	0
random-delete-all	0	0
0-greedy-delete	0	0
0-greedy-delete$_{QoS}$	0	0
1-greedy-delete	0	0
1-greedy-delete$_{QoS}$	1	0
SA	0	0
RVNS	0	0
RVNS$_{fill}$	0	0
VNS-L$_{100remove}$	0	0
VNS$_{fill}$-L$_{100remove}$	0	0
VNS-L$_{100swap}$	5	3
VNS$_{fill}$-L$_{100swap}$	7	7
VNS$_{remove}$	25	14

Table 5.6: Number of instances solved best

If we add up all 38 ranks of each algorithm we obtain the overall ranking shown in Table 5.7. Note that both VNS$_{fill}$-L$_{100swap}$ and VNS-L$_{100swap}$ have the same sum. Hence, they are both on rank 2 and we omit rank 3. The same holds for CPLEX (300s) and *1-greedy-delete* on rank 12.

All our metaheuristics outperform all simple heuristics which are prevalent in the literature. All variants of the variable neighborhood search also outperform our simulated annealing approach. This is due to the great flexibility of the VNS scheme with its different neighborhoods and local search schemes. As we already showed, the different components of our VNS variants are specialized on different components of the objective function. With the SA approach we had to combine all aspects of the problem in one neighborhood. This could be a reason why in our experimental study the SA is almost always faster than the VNS variants (see Section 5.2) but not as good when it comes to solution quality.

Rank	Solution method	Rank	Solution method
1	VNS_{remove}	11	*1-greedy-delete*$_{QoS}$
2a	VNS_{fill}-$\text{L}_{100swap}$	12a	CPLEX (300s)
2b	VNS-$\text{L}_{100swap}$	12b	*1-greedy-delete*
4	CPLEX (3,600s)	14	*0-greedy-delete*$_{QoS}$
5	VNS_{fill}-$\text{L}_{100remove}$	15	*random-delete-all*
6	RVNS_{fill}	16	*0-greedy-delete*
7	VNS-$\text{L}_{100remove}$	17	*random-add*
8	RVNS	18	*random-delete*
9	CPLEX (600s)	19	CPLEX (60s)
10	SA		

Table 5.7: Ranking of the algorithms based on their ranks for each instance

We now want to analyze the influence of the different VNS components on the ranking of the algorithms (see Table 5.7). Obviously, there is a benefit of using a local search after each shaking step (VNS_{remove}). We can confirm that the l_{swap} local search is more efficient than l_{remove} (rank 7 and 2b, 5 and 2a) but at the cost of a longer runtime (see Section 5.2).

We also see a benefit in using the additional *fill* neighborhood (rank 8 and 6, 7 and 5). It seems that it is an even better addition to the reduced VNS (RVNS) than the local search l_{remove} applied every 100 steps (rank 8, 7, 6).

Note that all VNS variants outperform CPLEX (600s) and have runtimes of only 17 seconds (RVNS) to at most 322 seconds (VNS_{fill}-$\text{L}_{100swap}$). The VNS variants with longer runtimes of about 300 seconds, i. e. VNS_{remove}, VNS_{fill}-$\text{L}_{100swap}$ and VNS-$\text{L}_{100swap}$ (rank 1, 2a, 2b) even outperform CPLEX with a runtime of 3,600s.

Considering the greedy algorithms, the positive influences of the backtracking (rank 16 and 12b, 14 and 11) as well as of the adaption to QoS metrics (rank 16 and 14, 12b and 11) can be seen. But as we will see next, the solution quality of all the greedy algorithms is considerably inferior than the solution quality of the metaheuristics.

Rank	Solution method	$\overline{\Delta}$	Rank	Solution method	$\overline{\Delta}$
1	VNS_{remove}	3.78	11	CPLEX (300s)	40.38
2	CPLEX (3,600s)	3.80	12	*1-greedy-delete*$_{QoS}$	42.13
3	VNS-$\text{L}_{100swap}$	4.07	13	*random-delete-all*	42.90
4	VNS_{fill}-$\text{L}_{100swap}$	4.10	14	*random-add*	44.26
5	VNS_{fill}-$\text{L}_{100remove}$	4.88	15	*random-delete*	44.36
6	RVNS_{fill}	4.88	16	*0-greedy-delete*$_{QoS}$	46.79
7	VNS-$\text{L}_{100remove}$	5.14	17	*0-greedy-delete*	47.18
8	RVNS	5.25	18	*1-greedy-delete*	51.46
9	SA	7.03	19	CPLEX (60s)	53.51
10	CPLEX (600s)	13.66			

Table 5.8: Ranking of the algorithms based on the gap $\overline{\Delta}$ [%]

Average objective function values. With the second measure we want to provide an insight into the differences in solution quality of the algorithms and how they compare with each other.

Therefore we take a look at the average objective function values over all instances. Based on the average objective function value $\overline{z}$ of each algorithm as well as the average $\overline{LB}$ of the improved lower bounds obtained by CPLEX (LP-i) we provide the gap $\overline{\Delta}$ as follows.

$$\overline{\Delta} = \frac{\overline{z} - \overline{LB}}{\overline{z}} \tag{24}$$

We use this definition instead of an average over the gaps for each instance as we want to penalize solution methods with low quality results according to their objective function values. As the gaps for each instance are $\in [0, 1]$ solutions with a very low quality have a gap near 1. But this gap value does not clearly reflect how bad the solution actually is compared to other low quality solutions. Building an average of these gaps results in lower values that are harder to compare and distortet. With our definition, low quality solutions lead to higher gaps $\overline{\Delta}$. The resulting ranking of the algorithms and the gaps are given in Table 5.8.

We can see that the ranking according to $\overline{\Delta}$ differs from the first ranking, but mainly for the greedy and random algorithms. VNS_{remove} is still on rank 1. CPLEX (3,600s) is ranked second best. Considering rank 3 (VNS-$\text{L}_{100swap}$) and 4 (VNS_{fill}-$\text{L}_{100swap}$) the order of these two algorithms is

not as expected but the difference of the gap is only 0.03%. It seems that the *fill* neighborhood has no positive influence on the VNS with l_{swap} local search. Note that the addition of the *fill* neighborhood to the VNS-$L_{100remove}$ (rank 7 and 5) reduces the gap by 0.26%. Adding the *fill* neighborhood to the RVNS without any local search (rank 8 and 6) reduces the gap by 0.37%. We conclude that the *fill* neighborhood is more useful the less effort is done in the local search. This is consistent with our findings regarding the runtimes (see Section 5.2). We also get confirmation of the benefit of a local search after each step (VNS_{remove}) when compared to a local search every 100 steps (VNS-$L_{100remove}$) and confirmation of the l_{swap} local search being superior to l_{remove} if both are used every 100 steps.

Considering the gap $\overline{\Delta}$ not only all VNS variants but also our simulated annealing approach outperform CPLEX (600s) clearly and as we saw in Section 5.2 with a distinct advantage in runtime. All greedy and random algorithms outperform CPLEX (60s), most of them with neglectable runtimes but all of them with very high gaps of 42% to 52%. Even CPLEX (300s) is not much better with a gap of 40%. Only our SA and VNS metaheuristics as well as CPLEX (3,600s) have gaps smaller than 10%.

Note that the order of the greedy and random algorithms changes in this ranking based on the gaps when compared to the first ranking. Only the most sophisticated *1-greedy-delete*$_{QoS}$ is in both rankings the best heuristic. In contrast, the second best *1-greedy-delete* is on the last rank with $\overline{\Delta}$ as a measure. Even the random algorithms are ranked better than all greedy algorithms but *1-greedy-delete*$_{QoS}$.

One possible explanation for the bad performance of the greedy heuristics is as follows. All the simple heuristics perform particularly bad in the 4 problem instances $i35$–$i38$. These instances do have very high delivery cost coefficients. Hence, good solutions have a large number of replicas to reduce the delivery cost. Table 5.9 illustrates this with the number of replicas in the solutions of all algorithms for instances $i35$–$i38$. The greedy algorithms do not find good solutions in this respect. They have too few replicas and thus high delivery cost in the objective function value. Even all the random algorithms perform better on these instances, while with gaps of 26.78–87.15% they are still far from optimal. With the difference in the objective function values being very huge, these 4 instances are the only reason for the change

Instance	$i35$	$i36$	$i37$	$i38$
CPLEX (60s)	400	473	542	588
CPLEX (300s)	379	469	542	588
CPLEX (600s)	377	470	540	588
CPLEX (3,600s)	375	469	539	588
random-add	341	334	338	340
random-delete	351	344	346	343
random-delete-all	265	265	265	264
0-greedy-delete	289	289	289	289
0-greedy-delete$_{QoS}$	271	271	271	271
1-greedy-delete	258	259	260	264
1-greedy-delete$_{QoS}$	254	254	254	253
SA	375	591	600	600
RVNS	381	471	542	588
RVNS$_{fill}$	384	473	542	589
VNS-L$_{100remove}$	380	470	542	588
VNS$_{fill}$-L$_{100remove}$	382	472	542	588
VNS-L$_{100swap}$	383	470	542	588
VNS$_{fill}$-L$_{100swap}$	384	471	542	588
VNS$_{remove}$	379	470	541	588

Table 5.9: Average number of replicas for the instances $i35$–$i38$ with high delivery cost coefficients

in the second ranking. In these instances the VNS algorithms all perform very well with gaps of 0.04–3.03%. The SA algorithm lies in between with gaps of 0.56–16.31%. For a complete overview of the gaps $\Delta = \frac{z-LB}{z}$ for each algorithm and problem instance see Tables 5.10 and 5.11 in Section 5.4.

5.3.4 Solution examples

Finally, we discuss some typical solutions of selected algorithms. The structure of the binary assignment matrix provides information on the characteristics of the algorithms, especially how much attention they pay to the placement cost component. For a better and objective comparison of the latter we define a normalized measure ς for the fragmentation of a solution. Therefore, ς^* counts the number of changes to the replica assignment on the

servers over the planning horizon as follows.

$$\varsigma^* = \sum_{s \in \mathcal{V}_s} \sum_{t \in \mathcal{T}} |x_{s,t} - x_{s,t-1}| \quad \text{with } x_{s,0} = 0 \tag{25}$$

For our problem instances the maximum possible value $\varsigma^{max} = 600$ which occurs if 300 replicas are placed alternately over the periods beginning with a replica in period 1, i. e. if $x_{s,t} = 1 \; \forall s \in \mathcal{V}_s, t \in \mathcal{T} : t \equiv 1 \pmod 2$. Hence, $\varsigma = \frac{\varsigma^*}{\varsigma^{max}}$ and $\varsigma \in [0, 1]$.

The solution matrices of *random-add* and *random-delete* are very similar, hence we provide only one example in Figure 5.6 with $\varsigma = 0.47$. The random solution structure can also be seen in the solution of *random-delete-all* with $\varsigma \approx 0.46$ (see Figure 5.7). Note that the solutions of *random-delete-all* have less replicas because the search for removable replicas is more exhaustive in this algorithm.

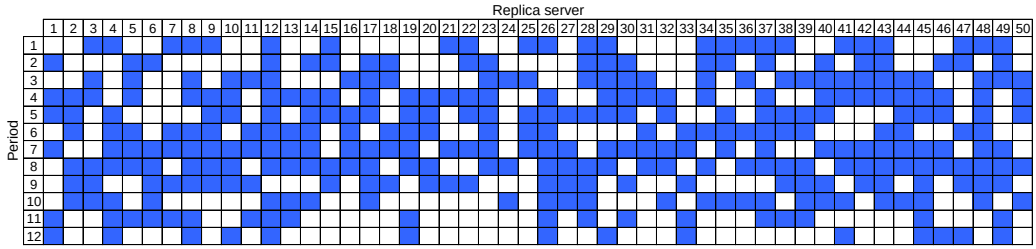

Figure 5.6: Exemplary solution from *random-add* with $\varsigma = 0.47$

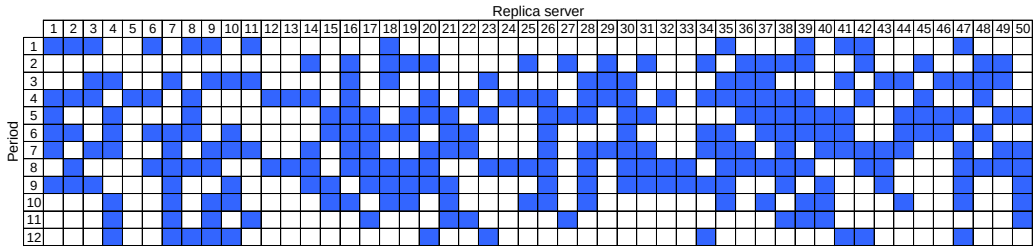

Figure 5.7: Exemplary solution from *random-delete-all* with $\varsigma \approx 0.46$

The solutions of the greedy algorithms also have very similar solutions. Basically they are a little bit more structured than solutions of the random algorithms. A solution of *0-greedy-delete* with $\varsigma \approx 0.33$ is given in Figure 5.8.

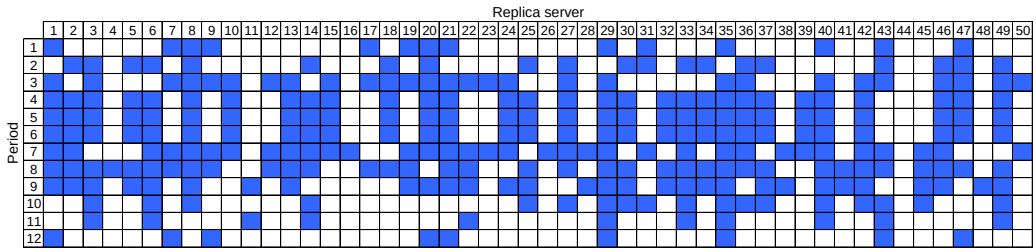

Figure 5.8: Exemplary solution from *0-greedy-delete* with $\varsigma \approx 0.33$

For the sample solution of the simulated annealing algorithm in Figure 5.9 we get $\varsigma = 0.1$, the lowest value of all our algorithms. We can see the distinct and very clear structure in the assignment matrix. It is formed by our neighborhood definition which pays particular attention to the placement cost. The blocks of replicas in consecutive periods can be clearly seen.

Also when looking at the solutions provided by the VNS variants we can see some typical characteristics (see Figures 5.10–5.16).

The addition of the *fill* neighborhood leads to more and longer blocks of consecutive periods with replicas. This effect can be seen most clearly when comparing the solutions of RVNS ($\varsigma \approx 0.29$) and RVNS$_{fill}$ ($\varsigma \approx 0.19$) as well as VNS-L$_{100remove}$ ($\varsigma = 0.26$) and VNS$_{fill}$-L$_{100remove}$ ($\varsigma \approx 0.16$).

As we already concluded, a local search in the VNS scheme has a similar effect as the *fill* neighborhood. Especially the algorithms with the more complex local search l_{swap} (VNS-L$_{100swap}$, $\varsigma = 0.14$) and with l_{remove} local search after each step (VNS$_{remove}$, $\varsigma = 0.14$) obtain very structured solutions with replica–blocks although they do not use the *fill* neighborhood. The impact of the *fill* neighborhood decreases with increasing complexity and frequency of the local search. In our example solution of VNS$_{fill}$-L$_{100swap}$ the value $\varsigma = 0.14$ is the same as without the *fill* neighborhood. Note that the solutions of the top three metaheuristics in our rankings (VNS-L$_{100swap}$, VNS$_{fill}$-L$_{100swap}$, VNS$_{remove}$) all have the same value $\varsigma = 0.14$ although this is not the lowest possible value (see SA with $\varsigma = 0.1$). Obviously the placement cost component is very important for the problem but not the only aspect that needs to be considered for near–optimal solutions.

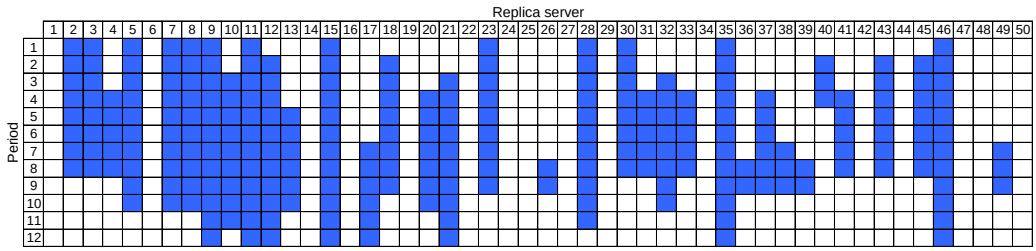

Figure 5.9: Exemplary solution from SA with $\varsigma = 0.1$

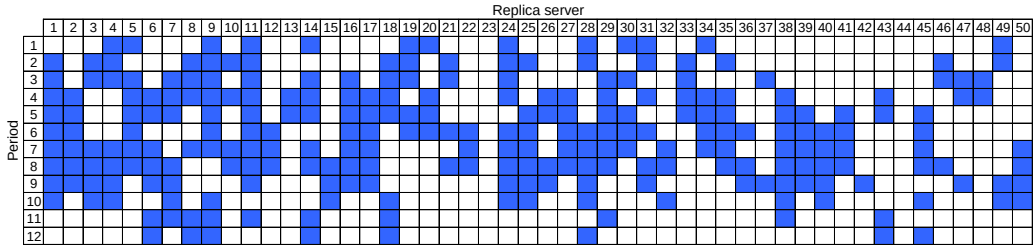

Figure 5.10: Exemplary solution from RVNS with $\varsigma \approx 0.29$

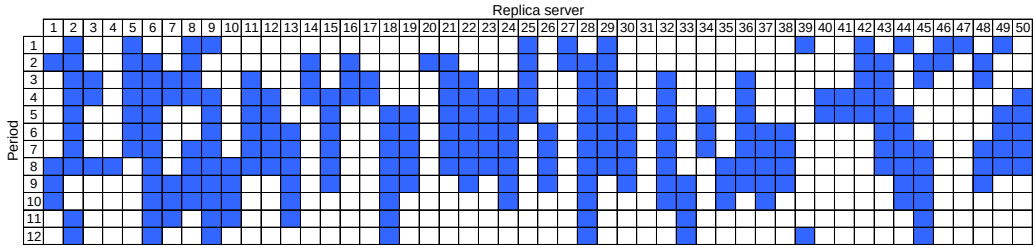

Figure 5.11: Exemplary solution from RVNS$_{fill}$ with $\varsigma \approx 0.19$

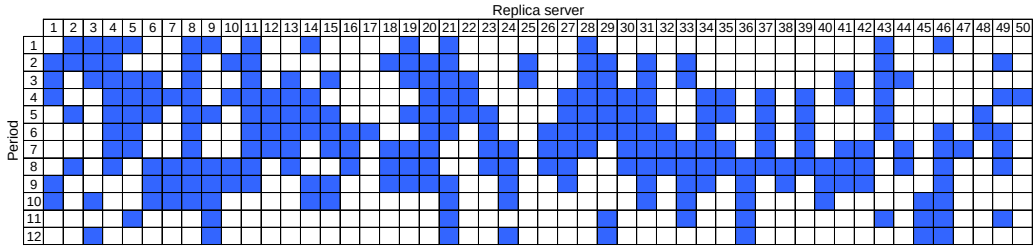

Figure 5.12: Exemplary solution from VNS-L$_{100remove}$ with $\varsigma = 0.26$

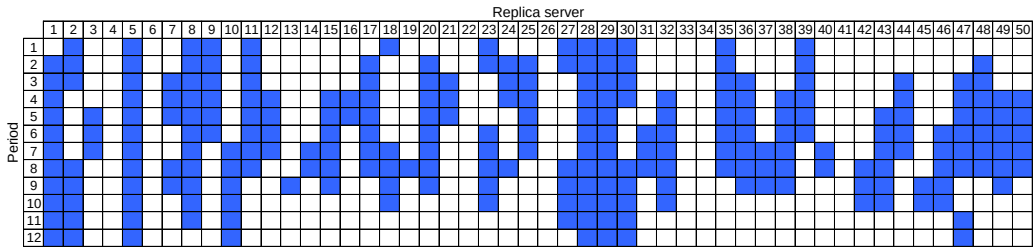

Figure 5.13: Exemplary solution from VNS$_{fill}$-L$_{100remove}$ with $\varsigma \approx 0.16$

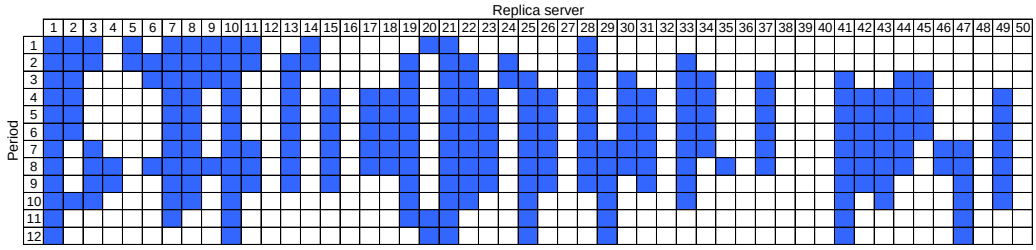

Figure 5.14: Exemplary solution from VNS-L$_{100swap}$ with $\varsigma = 0.14$

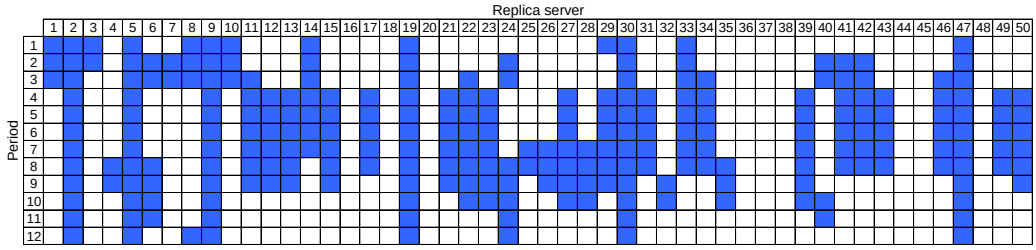

Figure 5.15: Exemplary solution from VNS$_{fill}$-L$_{100swap}$ with $\varsigma = 0.14$

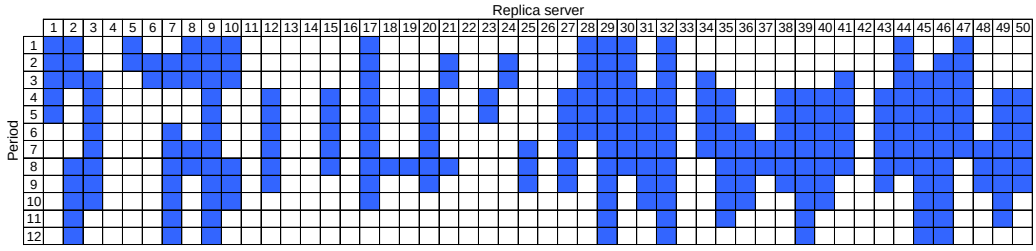

Figure 5.16: Exemplary solution from VNS$_{remove}$ with $\varsigma = 0.14$

Instance	CPLEX (60s)	CPLEX (300s)	CPLEX (600s)	CPLEX (3,600s)	*random-add*	*random-delete*	*random-delete-all*	*0-greedy-delete*	*0-greedy-delete*$_{QoS}$	*1-greedy-delete*	*1-greedy-delete*$_{QoS}$
*i*1	44.10	3.36	3.25	3.30	17.78	18.90	6.29	9.81	6.33	4.96	3.43
*i*2	50.61	4.51	4.24	2.95	22.80	22.89	7.90	13.90	8.81	5.51	4.25
*i*3	53.89	45.49	5.97	3.47	25.09	25.46	8.35	10.51	7.50	6.24	4.67
*i*4	56.45	56.45	5.54	3.81	24.82	25.66	9.28	16.34	9.29	6.37	4.75
*i*5	59.15	59.15	4.18	3.65	27.71	28.92	9.97	16.25	10.82	7.44	4.64
*i*6	63.37	63.37	55.84	6.25	29.90	32.48	10.75	14.66	10.99	8.20	6.07
*i*7	68.54	62.73	7.51	6.23	35.51	39.17	13.85	14.23	13.05	10.19	7.61
*i*8	74.43	70.40	11.28	6.05	43.37	45.40	18.15	19.61	16.19	12.76	9.54
*i*9	78.24	74.47	10.34	5.72	51.67	50.69	21.64	25.03	20.74	14.60	13.14
*i*10	79.89	79.89	13.67	7.84	53.88	52.92	23.67	24.48	20.50	16.88	11.37
*i*11	82.72	11.02	10.01	6.39	57.68	60.83	25.74	23.65	21.58	19.92	15.03
*i*12	85.30	15.11	10.47	8.71	63.04	63.87	29.57	24.12	23.66	21.30	16.82
*i*13	89.46	13.68	12.90	9.62	74.08	71.71	37.91	35.09	35.09	26.69	24.37
*i*14	59.51	43.70	4.96	3.10	7.25	7.35	7.05	9.52	5.05	7.54	4.39
*i*15	59.44	51.81	5.42	3.70	11.05	11.00	7.55	11.97	5.05	6.93	4.39
*i*16	59.20	47.23	44.66	3.23	14.28	16.08	8.48	12.92	5.40	7.04	4.39
*i*17	59.76	44.22	6.04	2.97	32.28	32.01	11.97	16.35	16.35	9.30	4.62
*i*18	7.54	7.14	7.12	6.83	25.00	25.33	10.30	10.83	9.05	8.69	7.38
*i*19	39.26	10.78	10.23	6.71	42.06	41.46	15.43	12.73	11.17	10.46	10.15
*i*20	44.65	6.23	5.59	5.13	45.08	45.62	17.19	12.18	11.99	11.00	9.79
*i*21	59.68	46.23	5.84	4.10	39.56	40.83	16.41	17.05	12.30	8.85	6.33
*i*22	59.13	59.13	4.15	3.18	8.23	8.89	7.06	9.03	8.22	7.82	4.21
*i*23	61.72	47.19	46.48	44.31	75.62	75.98	77.63	74.55	73.42	78.27	70.96
*i*24	73.88	36.79	23.00	20.31	48.23	48.14	46.61	43.01	40.04	46.34	34.66
*i*25	67.29	26.55	21.20	16.50	40.99	40.91	34.80	33.25	29.48	33.81	23.65
*i*26	59.88	52.81	11.38	5.16	30.74	30.76	14.11	19.02	13.87	12.12	7.72
*i*27	59.02	59.02	4.75	2.37	26.95	27.89	7.18	14.71	9.11	4.76	2.92
*i*28	1.58	1.50	1.48	1.47	24.50	26.83	6.54	14.35	8.99	4.73	2.71
*i*29	58.25	3.72	3.13	2.92	28.78	27.61	7.84	15.31	9.92	6.11	3.68
*i*30	62.61	62.61	48.40	5.31	32.61	30.98	14.77	18.92	13.40	11.16	7.36
*i*31	–	57.53	10.66	7.13	34.71	35.89	21.93	22.92	17.29	16.63	11.46
*i*32	–	56.53	56.53	2.22	27.77	29.50	9.49	6.72	5.11	3.40	3.79
*i*33	55.13	7.69	6.00	3.96	25.09	25.74	11.62	18.31	14.06	13.84	7.51
*i*34	50.09	6.87	7.08	4.79	23.38	24.11	13.35	20.30	17.13	19.50	10.26
*i*35	4.28	2.53	2.21	1.93	27.46	26.78	33.70	40.30	40.33	47.77	34.04
*i*36	2.12	1.56	1.53	1.39	44.85	42.95	53.22	58.52	59.15	65.46	54.61
*i*37	0.61	0.48	0.48	0.42	62.66	62.09	71.33	74.74	75.38	79.64	72.46
*i*38	0.06	0.07	0.05	0.04	81.77	81.73	87.15	88.66	89.03	90.92	87.69

Table 5.10: Gap Δ [%] for all algorithms applied to all instances (part 1)

5.4 Impact of the parameters

5.4.1 Impact on solution quality

With the detailed results in Tables 5.10 and 5.11 we can see a clear impact of the parameters on the solution quality. Note that this impact holds for all evaluated algorithms. Thus, higher gaps on some instances can also be due to the quality of the lower bound. The lower bound is most likely not equally tight for all instances.

Instance	SA	RVNS	RVNS$_{fill}$	VNS-L$_{100remove}$	VNS$_{fill}$-L$_{100remove}$	VNS-L$_{100swap}$	VNS$_{fill}$-L$_{100swap}$	VNS$_{remove}$
$i1$	2.98	2.86	2.61	2.74	2.50	2.24	2.25	2.23
$i2$	4.15	3.56	3.32	3.52	3.34	2.75	2.75	2.60
$i3$	5.29	3.89	3.69	3.84	3.55	3.04	3.02	3.01
$i4$	4.11	4.45	3.91	4.32	3.87	3.49	3.20	3.30
$i5$	3.73	4.07	3.71	4.01	3.48	2.97	2.93	2.96
$i6$	4.67	4.84	4.40	4.82	4.23	3.54	3.54	3.44
$i7$	5.82	6.28	5.42	5.97	5.19	4.16	4.04	4.04
$i8$	8.45	8.04	7.24	7.88	7.80	5.42	5.38	5.25
$i9$	9.63	9.35	8.58	9.52	8.78	6.72	6.98	6.33
$i10$	10.64	10.32	9.88	10.74	9.73	7.84	7.74	6.90
$i11$	13.00	12.54	11.40	11.95	11.91	9.42	9.72	9.33
$i12$	14.11	14.46	13.32	14.40	13.71	10.55	10.43	10.07
$i13$	18.65	18.88	17.99	18.03	18.21	14.34	13.69	12.86
$i14$	3.42	3.28	3.07	3.30	3.08	2.60	2.62	2.59
$i15$	3.40	3.26	3.08	3.33	3.04	2.57	2.59	2.58
$i16$	3.38	3.34	3.06	3.33	3.10	2.57	2.57	2.57
$i17$	4.81	4.55	4.11	4.47	4.12	3.03	3.20	3.15
$i18$	9.76	8.72	9.04	8.51	8.45	8.28	8.61	7.54
$i19$	13.88	12.42	12.34	12.35	12.61	11.81	12.44	9.90
$i20$	14.44	13.17	12.93	12.87	13.28	12.07	12.05	9.42
$i21$	9.82	8.35	8.25	8.40	8.50	6.52	6.32	5.68
$i22$	3.67	3.41	3.10	3.31	3.09	2.65	2.57	2.58
$i23$	46.39	44.14	44.07	44.02	43.97	43.99	44.02	43.96
$i24$	23.88	21.86	21.28	21.22	20.77	19.92	20.23	19.87
$i25$	17.88	17.03	16.59	17.12	16.53	14.64	14.15	14.27
$i26$	5.84	6.86	5.79	6.64	5.89	4.56	4.55	4.68
$i27$	3.39	2.56	2.29	2.57	2.49	2.06	2.07	2.01
$i28$	3.61	2.02	1.95	1.89	1.95	1.72	1.79	1.75
$i29$	3.58	3.25	2.89	3.09	2.83	2.37	2.47	2.44
$i30$	5.23	6.65	5.69	6.36	5.57	4.25	4.28	4.24
$i31$	7.53	9.98	8.54	9.90	8.46	6.37	6.26	6.01
$i32$	3.20	3.94	3.33	3.63	3.54	3.14	3.18	3.00
$i33$	8.17	5.26	4.38	5.13	4.47	3.34	3.34	3.35
$i34$	9.32	5.71	5.06	5.52	4.71	3.40	3.41	3.38
$i35$	16.31	3.03	2.73	2.85	2.70	2.01	2.01	1.98
$i36$	10.71	2.01	1.90	1.88	1.80	1.42	1.45	1.41
$i37$	3.92	0.66	0.61	0.59	0.59	0.53	0.51	0.47
$i38$	0.56	0.05	0.06	0.04	0.05	0.05	0.04	0.04

Table 5.11: Gap Δ [%] for all algorithms applied to all instances (part 2)

Server capacity. If the capacity of the servers rises (instances $i1$–$i13$) the gaps of all solution procedures rise as well. The gaps just for these instances are shown in Table 5.12. With down to only 10% average workload if $x_{s,t} = 1 \; \forall s \in \mathcal{V}_s, t \in \mathcal{T}$ on instance $i13$, few servers are sufficient to obtain feasible solutions. This can be also seen in Table 5.13 which provides the number of replicas in the solutions of all algorithms for instances $i1$–$i13$.

We expect that for the high capacity instances the higher gaps are not due to the quality of the lower bounds but due to the low quality of the upper bounds. Because of the sufficiency of few replicas these instances have a high degree of freedom in the solution space, i. e. there are more feasible replica

Instance	$i1$	$i2$	$i3$	$i4$	$i5$	$i6$	$i7$	$i8$	$i9$	$i10$	$i11$	$i12$	$i13$
ρ^{max}	0.9	0.8	0.75	0.7	0.66	0.6	0.5	0.4	0.33	0.3	0.25	0.2	0.1
CPLEX (60s)	44.1	50.6	53.9	56.4	59.2	63.4	68.5	74.4	78.2	79.9	82.7	85.3	89.5
CPLEX (300s)	3.4	4.5	45.5	56.4	59.2	63.4	62.7	70.4	74.5	79.9	11.0	15.1	13.7
CPLEX (600s)	3.3	4.2	6.0	5.5	4.2	55.8	7.5	11.3	10.3	13.7	10.0	10.5	12.9
CPLEX (3,600s)	3.3	2.9	3.5	3.8	3.6	6.3	6.2	6.0	5.7	7.8	6.4	8.7	9.6
random-add	17.8	22.8	25.1	24.8	27.7	29.9	35.5	43.4	51.7	53.9	57.7	63.0	74.1
random-delete	18.9	22.9	25.5	25.7	28.9	32.5	39.2	45.4	50.7	52.9	60.8	63.9	71.7
random-delete-all	6.3	7.9	8.4	9.3	10.0	10.8	13.8	18.1	21.6	23.7	25.7	29.6	37.9
0-greedy-delete	9.8	13.9	10.5	16.3	16.3	14.7	14.2	19.6	25.0	24.5	23.6	24.1	35.1
0-greedy-delete$_{QoS}$	6.3	8.8	7.5	9.3	10.8	11.0	13.0	16.2	20.7	20.5	21.6	23.7	35.1
1-greedy-delete	5.0	5.5	6.2	6.4	7.4	8.2	10.2	12.8	14.6	16.9	19.9	21.3	26.7
1-greedy-delete$_{QoS}$	3.4	4.3	4.7	4.8	4.6	6.1	7.6	9.5	13.1	11.4	15.0	16.8	24.4
SA	3.0	4.2	5.3	4.1	3.7	4.7	5.8	8.5	9.6	10.6	13.0	14.1	18.6
RVNS	2.9	3.6	3.9	4.5	4.1	4.8	6.3	8.0	9.3	10.3	12.5	14.5	18.9
RVNS$_{fill}$	2.6	3.3	3.7	3.9	3.7	4.4	5.4	7.2	8.6	9.9	11.4	13.3	18.0
VNS-L$_{100rem.}$	2.7	3.5	3.8	4.3	4.0	4.8	6.0	7.9	9.5	10.7	12.0	14.4	18.0
VNS$_{fill}$-L$_{100rem.}$	2.5	3.3	3.5	3.9	3.5	4.2	5.2	7.8	8.8	9.7	11.9	13.7	18.2
VNS-L$_{100swap}$	2.2	2.8	3.0	3.5	3.0	3.5	4.2	5.4	6.7	7.8	9.4	10.6	14.3
VNS$_{fill}$-L$_{100swap}$	2.2	2.7	3.0	3.2	2.9	3.5	4.0	5.4	7.0	7.7	9.7	10.4	13.7
VNS$_{remove}$	2.2	2.6	3.0	3.3	3.0	3.4	4.0	5.2	6.3	6.9	9.3	10.1	12.9

Table 5.12: Gap Δ [%] when varying the server capacities

Instance	$i1$	$i2$	$i3$	$i4$	$i5$	$i6$	$i7$	$i8$	$i9$	$i10$	$i11$	$i12$	$i13$
ρ^{max}	0.9	0.8	0.75	0.7	0.66	0.6	0.5	0.4	0.33	0.3	0.25	0.2	0.1
CPLEX (60s)	600	600	600	600	600	600	600	600	600	600	600	600	600
CPLEX (300s)	347	309	515	600	600	600	514	529	517	600	106	89	55
CPLEX (600s)	347	309	291	273	253	517	195	158	132	125	105	87	55
CPLEX (3,600s)	347	309	291	272	253	234	195	158	131	122	102	86	53
random-add	409	387	373	345	337	318	288	265	261	251	234	226	228
random-delete	415	387	375	350	343	331	307	276	256	246	255	232	208
random-delete-all	355	319	299	281	264	243	208	174	150	140	122	106	80
0-greedy-delete	372	345	310	311	289	258	213	181	163	147	122	102	80
0-greedy-delete$_{QoS}$	357	324	298	284	271	247	209	173	152	139	118	101	80
1-greedy-delete	349	310	291	272	256	236	198	162	136	126	110	91	64
1-greedy-delete$_{QoS}$	347	310	291	272	254	235	198	161	137	124	108	91	66
SA	348	313	297	273	254	235	198	163	136	126	109	92	62
RVNS	347	310	291	273	254	234	197	160	134	124	108	90	61
RVNS$_{fill}$	347	310	291	272	254	234	197	160	134	124	107	90	61
VNS-L$_{100remove}$	347	310	291	273	254	234	196	160	135	125	107	90	60
VNS$_{fill}$-L$_{100remove}$	347	310	291	273	254	234	197	162	135	124	108	91	62
VNS-L$_{100swap}$	347	310	291	273	254	234	197	160	134	124	106	88	59
VNS$_{fill}$-L$_{100swap}$	347	310	291	272	254	234	197	160	134	124	107	89	58
VNS$_{remove}$	347	309	291	272	254	234	196	159	133	122	106	88	57

Table 5.13: Average number of replicas when varying the server capacities

assignments. Additionally, the heuristics tend to use too many replicas when compared to the solutions of CPLEX (3,600s) (see Table 5.13). This seems to be due to the SLA constraints, especially the maximum allowed latency. With less replicas our neighborhoods supposably can not find the few changes to the assignment that still lead to feasible solutions and further reduce the amount of servers. Note that these high capacities are unusual for web servers in data centers.

Instance	$i14$	$i15$	$i16$	$i5$	$i17$
λ	0.75	0.9	0.95	0.99	1
CPLEX (60s)	59.5	59.4	59.2	59.2	59.8
CPLEX (300s)	43.7	51.8	47.2	59.2	44.2
CPLEX (600s)	5.0	5.4	44.7	4.2	6.0
CPLEX (3,600s)	3.1	3.7	3.2	3.6	3.0
random-add	7.2	11.1	14.3	27.7	32.3
random-delete	7.3	11.0	16.1	28.9	32.0
random-delete-all	7.1	7.6	8.5	10.0	12.0
0-greedy-delete	9.5	12.0	12.9	16.3	16.4
0-greedy-delete$_{QoS}$	5.0	5.0	5.4	10.8	16.4
1-greedy-delete	7.5	6.9	7.0	7.4	9.3
1-greedy-delete$_{QoS}$	4.4	4.4	4.4	4.6	4.6
SA	3.4	3.4	3.4	3.7	4.8
RVNS	3.3	3.3	3.3	4.1	4.6
RVNS$_{fill}$	3.1	3.1	3.1	3.7	4.1
VNS-L$_{100rem.}$	3.3	3.3	3.3	4.0	4.5
VNS$_{fill}$-L$_{100rem.}$	3.1	3.0	3.1	3.5	4.1
VNS-L$_{100swap}$	2.6	2.6	2.6	3.0	3.0
VNS$_{fill}$-L$_{100swap}$	2.6	2.6	2.6	2.9	3.2
VNS$_{remove}$	2.6	2.6	2.6	3.0	3.2

Table 5.14: Gap Δ [%] when varying the service level λ

Service level. The fraction λ of all requests that needs to be handled within the maximum latency (instances $i14$–$i17$) seems to have only very little influence on the solution quality. The gaps for these instances are shown in Table 5.14. Only the simple random and greedy heuristics are an exception here. The simpler the heuristic is, the higher is the gap for high values of λ (e. g. instance $i17$). If the maximum allowed latency is a hard restriction which can not be exceeded for some of the requests, then the simple heuristics lead to inferior results. Only the most complex of the simple heuristics (*1-greedy-delete*$_{QoS}$) does not show this behavior and handles high values of λ equally well.

	Gap Δ [%]						average number of replicas					
Instance	$i18$	$i19$	$i20$	$i21$	$i5$	$i22$	$i18$	$i19$	$i20$	$i21$	$i5$	$i22$
l	0.5	0.75	1	1.5	2	3	0.5	0.75	1	1.5	2	3
CPLEX (60s)	7.5	39.3	44.6	59.7	59.2	59.1	475	537	529	600	600	600
CPLEX (300s)	7.1	10.8	6.2	46.2	59.2	59.1	473	358	309	456	600	600
CPLEX (600s)	7.1	10.2	5.6	5.8	4.2	4.2	473	356	308	255	253	253
CPLEX (3,600s)	6.8	6.7	5.1	4.1	3.6	3.2	472	349	308	255	253	253
random-add	25.0	42.1	45.1	39.6	27.7	8.2	589	571	541	411	337	257
random-delete	25.3	41.5	45.6	40.8	28.9	8.9	591	564	547	421	343	259
random-delete-all	10.3	15.4	17.2	16.4	10.0	7.1	489	383	350	289	264	253
0-greedy-delete	10.8	12.7	12.2	17.1	16.3	9.0	493	374	333	297	289	257
0-greedy-delete$_{QoS}$	9.1	11.2	12.0	12.3	10.8	8.2	483	367	332	280	271	255
1-greedy-delete	8.7	10.5	11.0	8.9	7.4	7.8	481	362	326	264	256	253
1-greedy-delete$_{QoS}$	7.4	10.2	9.8	6.3	4.6	4.2	474	362	323	259	254	253
SA	9.8	13.9	14.4	9.8	3.7	3.7	488	381	344	275	254	253
RVNS	8.7	12.4	13.2	8.3	4.1	3.4	481	372	336	266	254	253
RVNS$_{fill}$	9.0	12.3	12.9	8.2	3.7	3.1	483	373	336	268	254	253
VNS-L$_{100remove}$	8.5	12.3	12.9	8.4	4.0	3.3	480	372	335	266	254	253
VNS$_{fill}$-L$_{100remove}$	8.5	12.6	13.3	8.5	3.5	3.1	480	374	338	269	254	253
VNS-L$_{100swap}$	8.3	11.8	12.1	6.5	3.0	2.7	480	371	334	264	254	253
VNS$_{fill}$-L$_{100swap}$	8.6	12.4	12.0	6.3	2.9	2.6	481	374	334	264	254	253
VNS$_{remove}$	7.5	9.9	9.4	5.7	3.0	2.6	476	363	323	261	254	253

Table 5.15: Gap Δ [%] and average number of replicas when varying the maximum allowed latency

Maximum allowed latency. The parameter l which increases the maximum allowed latency (instances $i18$–$i22$), i. e. the maximum allowed distance between server and client, has a specific influence on the problem. The corresponding data is provided in Table 5.15.

On the one side, low values of l lead to a high amount of replica servers needed for feasible solutions. With this lower degree of freedom the gaps are smaller, especially when using the greedy heuristics and metaheuristics. On instance $i18$ with the lowest allowed latency, even CPLEX (60s) performs well. On the other side, high values of l also lead to low gaps for the heuristics and metaheuristics. The SLAs are almost not restrictive in this case and do not impose any additional difficulty to the problem. The gaps are higher in the middle of our tested range. It seems that here are the most conflicts between the different components of the objective function and the realization of the claimed service levels.

Note that the results of CPLEX do not show this influence. The gaps achieved by CPLEX behave very differently when varying l depending on the time limits. We suppose that this is due to the underlying algorithms of CPLEX.

	Gap Δ [%]						average number of replicas					
Instance	$i23$	$i24$	$i25$	$i26$	$i5$	$i27$	$i23$	$i24$	$i25$	$i26$	$i5$	$i27$
α	10	50	100	500	1,000	2,000	10	50	100	500	1,000	2,000
CPLEX (60s)	61.7	73.9	67.3	59.9	59.2	59.0	584	600	600	600	600	600
CPLEX (300s)	47.2	36.8	26.6	52.8	59.2	59.0	476	280	257	521	600	600
CPLEX (600s)	46.5	23.0	21.2	11.4	4.2	4.8	484	297	263	253	253	256
CPLEX (3,600s)	44.3	20.3	16.5	5.2	3.6	2.4	491	281	256	254	253	254
random-add	75.6	48.2	41.0	30.7	27.7	27.0	345	337	345	342	337	339
random-delete	76.0	48.1	40.9	30.8	28.9	27.9	339	342	344	344	343	343
random-delete-all	77.6	46.6	34.8	14.1	10.0	7.2	266	265	265	264	264	263
0-greedy-delete	74.6	43.0	33.2	19.0	16.3	14.7	289	289	289	289	289	289
0-greedy-delete$_{QoS}$	73.4	40.0	29.5	13.9	10.8	9.1	271	271	271	271	271	271
1-greedy-delete	78.3	46.3	33.8	12.1	7.4	4.8	259	256	256	256	256	256
1-greedy-delete$_{QoS}$	71.0	34.7	23.6	7.7	4.6	2.9	254	254	254	254	254	254
SA	46.4	23.9	17.9	5.8	3.7	3.4	600	291	272	255	254	256
RVNS	44.1	21.9	17.0	6.9	4.1	2.6	506	293	257	254	254	254
RVNS$_{fill}$	44.1	21.3	16.6	5.8	3.7	2.3	504	292	257	254	254	254
VNS-L$_{100remove}$	44.0	21.2	17.1	6.6	4.0	2.6	500	291	256	254	254	254
VNS$_{fill}$-L$_{100remove}$	44.0	20.8	16.5	5.9	3.5	2.5	500	291	257	254	254	254
VNS-L$_{100swap}$	44.0	19.9	14.6	4.6	3.0	2.1	504	295	258	254	254	254
VNS$_{fill}$-L$_{100swap}$	44.0	20.2	14.2	4.5	2.9	2.1	505	299	259	254	254	254
VNS$_{remove}$	44.0	19.9	14.3	4.7	3.0	2.0	500	288	255	254	254	254

Table 5.16: Gap Δ [%] and average number of replicas when varying the storage cost coefficient α

Storage cost coefficient. Now we look at the cost coefficients. With an increasing storage cost coefficient α (instances $i23$–$i27$), the gaps typically decrease (see Table 5.16). The instances with very low storage cost coefficients (instances $i23$, $i24$) show gaps which are among the highest gaps of all instances for all considered algorithms.

We suppose that the high gaps are mainly due to weak lower bounds. A low storage cost coefficient leads to solutions with a large amount of replicas because this reduces the delivery cost (see Table 5.16). The remaining placement cost involve the multi–period Steiner tree problem. This problem is hard to solve and we already stated that the lower bounds are not tight. The contribution of the placement cost to the total cost gets proportionally bigger as the storage cost decrease. The placement cost become more important in determining the gap, thus the gap rises even if the placement and delivery cost components stay constant. Hence, we assume that the best algorithms provide solutions of good quality although there are high gaps on the instances with low storage cost coefficients.

Instance	$i32$	$i5$	$i33$	$i34$	$i35$	$i36$	$i37$	$i38$
γ	0	0.01	0.05	0.1	0.5	1	2	5
CPLEX (60s)	–	59.15	55.13	50.09	4.28	2.12	0.61	0.06
CPLEX (300s)	56.53	59.15	7.69	6.87	2.53	1.56	0.48	0.07
CPLEX (600s)	56.53	4.18	6.00	7.08	2.21	1.53	0.48	0.05
CPLEX (3,600s)	2.22	3.65	3.96	4.79	1.93	1.39	0.42	0.04
random-add	27.77	27.71	25.09	23.38	27.46	44.85	62.66	81.77
random-delete	29.50	28.92	25.74	24.11	26.78	42.95	62.09	81.73
random-delete-all	9.49	9.97	11.62	13.35	33.70	53.22	71.33	87.15
0-greedy-delete	6.72	16.25	18.31	20.30	40.30	58.52	74.74	88.66
0-greedy-delete$_{QoS}$	5.11	10.28	14.06	17.13	40.33	59.15	75.38	89.03
1-greedy-delete	3.40	7.44	13.84	19.50	47.77	65.46	79.64	90.92
1-greedy-delete$_{QoS}$	3.79	4.64	7.51	10.26	34.04	54.61	72.46	87.69
SA	3.20	3.73	8.17	9.32	16.31	10.71	3.92	0.56
RVNS	3.94	4.07	5.26	5.71	3.03	2.01	0.66	0.05
RVNS$_{fill}$	3.33	3.71	4.38	5.06	2.73	1.90	0.61	0.06
VNS-L$_{100remove}$	3.63	4.01	5.13	5.52	2.85	1.88	0.59	0.04
VNS$_{fill}$-L$_{100remove}$	3.54	3.48	4.47	4.71	2.70	1.80	0.59	0.05
VNS-L$_{100swap}$	3.14	2.97	3.34	3.40	2.01	1.42	0.53	0.05
VNS$_{fill}$-L$_{100swap}$	3.18	2.93	3.34	3.41	2.01	1.45	0.51	0.04
VNS$_{remove}$	3.00	2.96	3.35	3.38	1.98	1.41	0.47	0.04

Table 5.17: Gap Δ [%] when varying the delivery cost coefficient γ

Delivery cost coefficient. Increasing the delivery cost coefficient γ (instances $i32$–$i38$) results in mostly decreasing gaps when using CPLEX or the metaheuristics (see Table 5.17). Note that on the instances with the highest delivery cost coefficients CPLEX and all metaheuristics also have the lowest gaps of all instances. This is because the delivery cost dominate the other two cost components of the objective function. As the contribution of the storage cost to the total cost gets proportionally smaller, placing a lot more replicas does not have a big influence on the objective function value but reduces the delivery cost of the solution significantly. Thus, with high delivery cost coefficients good solutions have a large amount of replicas (see Table 5.18).

A high delivery cost coefficient helps CPLEX to find near–optimal solutions in short time. The CPLEX gaps are remarkably small even after 60 seconds and go down to 0.04% after 3,600 seconds. Hence, together with the very short computation times we have a strong hint that near–optimal solutions are trivial in this case. The random and greedy heuristics are an exception which we already addressed and explained in Section 5.3.3.

Instance	$i32$	$i5$	$i33$	$i34$	$i35$	$i36$	$i37$	$i38$
γ	0	0.01	0.05	0.1	0.5	1	2	5
CPLEX (60s)	–	600	600	600	400	473	542	588
CPLEX (300s)	546	600	254	257	379	469	542	588
CPLEX (600s)	546	253	253	256	377	470	540	588
CPLEX (3,600s)	253	253	253	254	375	469	539	588
random-add	334	337	335	343	341	334	338	340
random-delete	343	343	339	348	351	344	346	343
random-delete-all	265	264	264	265	265	265	265	264
0-greedy-delete	263	289	289	289	289	289	289	289
0-greedy-delete$_{QoS}$	258	271	271	271	271	271	271	271
1-greedy-delete	253	256	256	256	258	259	260	264
1-greedy-delete$_{QoS}$	254	254	254	254	254	254	254	253
SA	255	254	267	274	375	591	600	600
RVNS	254	254	254	254	381	471	542	588
RVNS$_{fill}$	254	254	254	254	384	473	542	589
VNS-L$_{100remove}$	254	254	254	254	380	470	542	588
VNS$_{fill}$-L$_{100remove}$	255	254	254	254	382	472	542	588
VNS-L$_{100swap}$	254	254	254	254	383	470	542	588
VNS$_{fill}$-L$_{100swap}$	254	254	254	254	384	471	542	588
VNS$_{remove}$	254	254	254	254	379	470	541	588

Table 5.18: Average number of replicas when varying the delivery cost coefficient γ

Placement cost coefficient. Finally, we look at the impact of the placement cost coefficient β. Increasing β (instances $i28$–$i31$) does not have a big influence on the degree of freedom as the number of feasible assignments does not change and there is no obvious reason for significantly more or less replicas as seen when varying some of the other parameters. The placement cost could be omitted if $x_{s,t} = 1\ \forall s \in \mathcal{V}_s, t \in \mathcal{T}$ but this is most likely no good solution. Of the tested algorithms only the short–running CPLEX (60s) and CPLEX (300s) place more replicas with increasing β and that results in low quality solutions with gaps of about 60%. Nonetheless, the gaps rise for all algorithms with increasing β as shown in Table 5.19. The reason is that the difficult Steiner tree problem gets more prominent. Its solution gets a higher proportion of the objective function value. In addition, the problem of the weak lower bound for the placement cost has a stronger effect here, i. e. the lower and the upper bounds diverge with an increasing placement cost coefficient.

Instance	$i28$	$i29$	$i5$	$i30$	$i31$
β	0	0.1	0.2	0.5	1
CPLEX (60s)	1.6	58.2	59.2	62.6	–
CPLEX (300s)	1.5	3.7	59.2	62.6	57.5
CPLEX (600s)	1.5	3.1	4.2	48.4	10.7
CPLEX (3,600s)	1.5	2.9	3.6	5.3	7.1
random-add	24.5	28.8	27.7	32.6	34.7
random-delete	26.8	27.6	28.9	31.0	35.9
random-delete-all	6.5	7.8	10.0	14.8	21.9
0-greedy-delete	14.4	15.3	16.3	18.9	22.9
0-greedy-delete$_{QoS}$	9.0	9.9	10.8	13.4	17.3
1-greedy-delete	4.7	6.1	7.4	11.2	16.6
1-greedy-delete$_{QoS}$	2.7	3.7	4.6	7.4	11.5
SA	3.6	3.6	3.7	5.2	7.5
RVNS	2.0	3.3	4.1	6.6	10.0
RVNS$_{fill}$	2.0	2.9	3.7	5.7	8.5
VNS-L$_{100remove}$	1.9	3.1	4.0	6.4	9.9
VNS$_{fill}$-L$_{100remove}$	1.9	2.8	3.5	5.6	8.5
VNS-L$_{100swap}$	1.7	2.4	3.0	4.2	6.4
VNS$_{fill}$-L$_{100swap}$	1.8	2.5	2.9	4.3	6.3
VNS$_{remove}$	1.8	2.4	3.0	4.2	6.0

Table 5.19: Gap Δ [%] when varying the placement cost coefficient β

5.4.2 Impact on runtimes

We now want to highlight the influences of parameters on the runtimes. In Table 5.20 the runtimes of all algorithms applied to all instances are shown. All but two parameter configurations have no influence on the runtimes of the greedy algorithms. Hence, we will not discuss them any further. We will concentrate on the metaheuristics and CPLEX (10%) as the LP relaxation CPLEX (LP-i) does not generate feasible solutions.

Server capacity. The influence of rising server capacities (instances $i1$–$i13$) seems to be different for CPLEX (10%) and the metaheuristics. On instances with higher capacities and hence a higher degree of freedom CPLEX (10%) basically has longer runtimes. The metaheuristics have slightly decreasing runtimes in these cases. Especially the more complex VNS variants VNS-L$_{100swap}$, VNS$_{fill}$-L$_{100swap}$ and VNS$_{remove}$ show runtimes which are shorter than the average on instance $i13$ with the highest server capacities.

Instance	CPLEX (10%)	CPLEX (LP-i)	*random-add*	*random-delete*	*random-delete-all*	*0-greedy-delete*	*0-greedy-delete*$_{QoS}$	*1-greedy-delete*	*1-greedy-delete*$_{QoS}$	SA	RVNS	RVNS$_{fill}$	VNS-L$_{100remove}$	VNS$_{fill}$-L$_{100remove}$	VNS-L$_{100swap}$	VNS$_{fill}$-L$_{100swap}$	VNS$_{remove}$
$i1$	159	5	0	0	0	4	4	984	743	27	19	26	21	24	355	348	362
$i2$	267	7	0	0	0	4	4	1173	850	24	20	27	23	27	384	386	335
$i3$	530	6	0	0	0	5	4	1244	961	24	19	27	21	25	405	404	327
$i4$	330	5	0	0	0	4	4	1335	928	28	19	27	23	25	425	421	311
$i5$	505	13	0	0	0	5	5	1432	1053	32	21	28	24	27	425	435	294
$i6$	878	14	0	0	0	4	5	1327	960	28	18	23	21	22	368	369	232
$i7$	428	13	0	0	0	5	5	1412	1041	28	20	25	23	23	345	349	205
$i8$	911	14	0	0	0	4	4	1316	928	22	18	22	21	23	293	291	150
$i9$	770	27	0	0	0	4	4	1199	869	23	19	23	23	24	272	273	138
$i10$	3527	35	0	0	0	4	3	1089	756	21	17	21	21	22	234	231	114
$i11$	2686	27	0	0	0	3	3	919	718	19	18	21	22	22	221	218	106
$i12$	855	28	0	0	0	2	2	671	587	17	16	18	19	20	180	172	82
$i13$	892	14	0	0	0	2	2	456	416	9	13	15	17	17	95	97	47
$i14$	530	14	0	0	0	6	5	1550	1052	22	18	23	20	22	506	511	352
$i15$	483	14	0	0	0	6	5	1585	1063	26	18	24	20	22	504	512	347
$i16$	3456	11	0	0	0	6	5	1546	1052	26	19	24	21	22	494	488	346
$i17$	551	12	0	0	0	4	4	1360	1054	30	22	33	28	34	397	412	262
$i18$	22	1	0	0	0	0	0	87	86	14	9	11	13	14	41	43	78
$i19$	2118	1	0	0	0	1	1	421	392	16	14	19	22	25	124	127	138
$i20$	138	2	0	0	0	2	2	634	628	17	19	26	30	34	179	184	161
$i21$	355	7	0	0	0	3	3	1164	1058	22	26	38	41	48	341	352	224
$i22$	395	15	0	0	0	6	6	1532	1098	19	18	23	20	23	510	494	352
$i23$	41351	0	0	0	0	4	5	1433	1056	3	8	9	27	25	74	74	393
$i24$	161795	1	0	0	0	4	5	1432	1052	20	15	19	33	32	246	243	356
$i25$	97282	4	0	0	0	4	5	1435	1058	21	22	26	29	30	404	404	311
$i26$	2659	13	0	0	0	4	5	1436	1055	32	21	27	24	26	434	427	302
$i27$	607	13	0	0	0	4	5	1437	1058	26	20	28	24	28	429	431	304
$i28$	55	12	0	0	0	4	5	1436	1053	27	20	63	24	63	458	508	332
$i29$	293	14	0	0	0	4	5	1444	1053	30	20	27	24	28	435	427	306
$i30$	783	13	0	0	0	4	5	1445	1051	26	21	28	25	26	432	425	302
$i31$	1012	14	0	0	0	4	5	1438	1056	25	21	28	25	27	435	430	291
$i32$	770	25	0	0	0	4	4	1067	1055	26	23	31	27	32	459	454	300
$i33$	289	8	0	0	0	4	5	1434	1056	23	20	28	24	29	447	429	310
$i34$	182	3	0	0	0	4	5	1440	1052	19	20	34	24	33	446	455	325
$i35$	27	0	0	0	0	4	5	1434	1054	9	11	25	27	33	184	196	357
$i36$	22	0	0	0	0	4	5	1436	1054	3	10	27	27	37	124	138	400
$i37$	4	0	0	0	0	4	5	1433	1055	3	9	19	27	36	67	72	416
$i38$	0	0	0	0	0	4	5	1395	1063	3	7	11	25	28	20	23	358
avg.	8629	11	0	0	0	4	4	1237	926	21	17	25	24	28	321	322	272

Table 5.20: Runtimes of all algorithms applied to all instances

The explanation is very simple. As good solutions for these instance need only few replicas, the local search procedures which swap or remove existing replicas need less comparisons to find the best action. That is why the effect is most obvious on the algorithms with a complex local search.

Service level. The fraction λ of all requests that needs to be handled within the maximum latency (instances $i14$–$i17$) has almost no impact on the runtimes. This is consistent with the result of Section 5.4.1 that it also has no significant impact on the solution quality.

Maximum allowed latency. The maximum allowed latency, controlled by parameter l (instances $i18$–$i22$), leads to short runtimes if it is very low (instance $i18$). This is also true for CPLEX (10%) and the greedy algorithms and is caused by the low degree of freedom with the necessity of a high amount of replicas which we already explained in Section 5.4.1.

Storage cost coefficient. An increasing storage cost coefficient α (instances $i23$–$i27$) has mainly an influence on the runtime of CPLEX (10%). The runtime is varying but generally we observe longer runtimes for low values of α. On instances $i23$–$i25$ CPLEX (10%) has the longest runtimes of all instances. It seems that CPLEX hardly finds tight lower bounds. In these instances the storage cost are of little importance while placement and delivery cost have normal levels. Herein we see evidence that the problem is hard to solve for CPLEX especially due to the placement cost. This is consistent with the importance of the placement cost for the problem and the complexity they bring along (see Section 3.4).

The metaheuristics except VNS_{remove} are primarily faster on instance $i23$ with the lowest storage costs. The other cost components dominate in this case. It follows that SA can not escape permanently from the local minimum with the maximum amount of replicas (see Table 5.16) and hence stops soon due to the stopping criterion (see Section 4.2). When it comes to the VNS variants the reduction of the runtimes is stronger with l_{swap} local search than with l_{remove}. The reason is that with a very low storage cost coefficient good solutions have many replicas to reduce the other cost components. Hence, there are not so many possibilities to swap replicas but still many possibilities to remove a replica. Thus, l_{remove} has to evaluate much more solutions. This is also the reason why VNS_{remove} with the most frequent l_{remove} local search does not have a reduced runtime at all in this case.

Delivery cost coefficient. With an increasing delivery cost coefficient γ (instances $i32$–$i38$) the runtimes of CPLEX (10%) and of the metaheuristics except VNS_{remove} decrease. As we already mentioned, these high delivery cost coefficients lead to solutions with replicas on almost all servers and very little potential for optimization (see Section 5.4.1). On instance $i38$ for example, even CPLEX (10%) finishes in under one second. On the other hand, VNS_{remove} has the longest runtimes of all instances as the local search l_{remove} has to do the most comparisons here.

Placement cost coefficient. Finally, we look at the placement cost coefficient β (instances $i28$–$i31$). While most of the runtimes are similar for each algorithm, we can see two interesting effects here.

First, the impact of the NP–hard Steiner tree problem on CPLEX. The runtime of CPLEX (10%) is 55 seconds without the placement cost component (instance $i28$, $\beta = 0$) and 1,012 seconds for instance $i31$ with $\beta = 1$. This is a further indication of the complexity of the Steiner tree problem we already saw on the instances $i23$–$i25$ with low storage cost coefficients.

Second, with $\beta = 0$ the VNS variants with additional *fill* neighborhood have longer runtimes than the others. This is most likely due to the characteristics of the *fill* neighborhood. There is no need to fill holes in the solution matrix if there are no placement cost. The *fill* neighborhood tries to fill all holes it finds on a selected server with a replica from another server in the same period. All these possibilities are evaluated but without placement cost there is no benefit. Thus, most of the holes are not filled. While with placement cost coefficient $\beta > 0$ the *fill* neighborhood has less work to do over time, with $\beta = 0$ each call of the *fill* neighborhood is similar complex and time consuming.

Note that generally, in contrast to the runtimes of CPLEX, the runtimes of all implemented algorithms are mostly robust and very similar over the different parameter sets. Especially for the SA algorithm this is remarkable as we do neither impose a time limit nor a hard iteration limit.

Chapter 6

Conclusion and Outlook

In this thesis we addressed the problem of dynamic replica placement in content delivery networks, consisting of the storage, the placement and the delivery of the content. We consider three cost types, take account of service level agreements, consider the multi–period case and therefore incorporate multicast transfers. We proposed a mixed–binary linear program which incorporates all the aspects of the problem and identifies the optimal amount and location of replicas of a single content over several periods and the transfer paths for the placement and the delivery. The considered problem is typical for content delivery networks and emerges in different scopes and sizes at the service providers.

We developed several new solution approaches for the problem. We proposed a simulated annealing metaheuristic as well as several variable neighborhood search variants to tackle the problem. We put effort on the fast and efficient solution evaluation as well as on sophisticated neighborhoods for the metaheuristics.

To evaluate our metaheuristics we adapted well known heuristics from the literature to solve our problem. We also refined them to better fit the specifics of our model. We tested all the algorithms using a factorial ceteris paribus design with 38 different problem instances across a broad range of values. We analyzed the runtime and solution quality of the different algorithms as well as how these characteristics change with the different parameter sets.

In the experimental study we demonstrated that all our metaheuristics can solve the problem in a much faster and more reliable manner than CPLEX as the solution quality of the metaheuristics is more constant with varying parameters. This in particular holds true as CPLEX fails to solve the MIP–formulation to optimality for all our problem instances when restricted to 3,600 seconds runtime. In reasonable time CPLEX can only find solutions with high gaps or no feasible solutions at all. Even within 10 minutes CPLEX can not constantly provide solutions of good quality.

The simulated annealing performs clearly better than CPLEX. Throughout the tested problem instances SA provides feasible and mostly good results within a remarkably short time. In the vast majority of cases our SA approach finds in less than a minute better solutions than CPLEX with a runtime of up to 600 seconds. Even when compared to CPLEX (3,600s), SA finds better solutions in 13% of the instances.

We found that the variable neighborhood search VNS is even superior than the SA. VNS outperforms all simple heuristics, simulated annealing and CPLEX with a runtime of up to 600 seconds. The best VNS variant (VNS_{remove}) even outperforms CPLEX (3,600s) in 60% of all instances and has a runtime of 272 seconds on average. If a solution needs to be found very fast the SA approach is the better choice as SA outperforms the VNS variants initially. If the solution quality is more important than the runtime then the VNS variants should be preferred. The choice of one VNS variant depends mainly on the time available for solving such a problem.

We see several possibilities to carry on the work in the field of replica placement in content delivery networks. First and foremost the model could be further extended. In our opinion there are three important factors which would make the model more realistic.

First, the consideration of several content providers or different files from one content provider, i. e. the multi–object case. This would not change the formulation of the model very much but it gets a lot more decision variables and constraints. Second, content updates of the stored files can be included. Consequently, they bring along new time–critical multicast transfers. As we already mentioned, a file that changes on the origin server should be updated on the replica servers as soon as possible. Third, the transfers and service levels can be made more realistic by adding bandwidths to the topologies.

The demand for bandwidth intensive multimedia files, especially videos, is increasing in the Internet. Therefore, the bandwidth of the connections gets more and more important. The transfer cost as well as the service levels can be defined based on the bandwidth. As a result, file sizes are necessary and capacities on the edges of the network have to be considered.

Last but not least, for a more realistic model unexpected events can be considered. If, e. g., an important news emerges, the demand for Internet news sites can increase dramatically. It is a real challenge to handle such highly stochastic demand. In classical logistics, at least the available capacity, e. g. of a warehouse, can often be used even though it is not sufficient. By contrast, if the replica placement is not done right almost no client is served as the servers slow down or do not respond at all under huge load. The monitoring of the Internet and the fast and correct reaction to stochastic events is probably one of the most interesting and challenging areas for further research in the scope of replica placement in content delivery networks.

Appendix A

Notation

Sets:	
$\mathcal{T}$	sequence of periods
$\mathcal{V}$	set of all nodes in the network
$\mathcal{V}_s$	set of replica servers in the network
$\mathcal{V}_c$	set of clients in the network
$\mathcal{V}_s(c,q)$	set of replica servers which can serve client $c \in \mathcal{V}_c$ within the maximum latency q
$\mathcal{A}$	set of arcs in the network
$\mathcal{V}^{out}(s)$	set of nodes $i \in \mathcal{V}$ in the network with an outbound arc $(i,s) \in \mathcal{A}$ to server $s \in \mathcal{V}$
$\mathcal{V}^{in}(s)$	set of nodes $j \in \mathcal{V}$ in the network with an inbound arc $(s,j) \in \mathcal{A}$ from server $s \in \mathcal{V}$
Parameter:	
V	number of nodes $n \in \mathcal{V}$
$\delta_{i,j}$	weight of arc $(i,j) \in \mathcal{A}$ (e.g. distance)
$d_{c,s}$	distance if a request of client $c \in \mathcal{V}_c$ is served by replica server $s \in \mathcal{V}_s$ (i.e. the length of the shortest path)
C_s^L	load capacity of replica server $s \in \mathcal{V}_s$ per period
$r_{c,t}$	number of requests from client $c \in \mathcal{V}_c$ for the object in period $t \in \mathcal{T}$
q	maximum allowed latency due to service level agreements
λ	fraction of all requests which has to be served within the maximum latency q
α_s	cost coefficient for the storage of a replica on server $s \in V_s$ per period
β	cost coefficient for the placement per distance unit
γ	cost coefficient for the delivery per distance unit
Decision variables:	
$x_{s,t}$	indicates if server $s \in \mathcal{V}_s$ stores a replica in period $t \in \mathcal{T}$
$y_{c,s,t}$	fraction of all requests from client $c \in \mathcal{V}_c$ in period $t \in \mathcal{T}$ which is served by replica server $s \in \mathcal{V}_s$
$w_{s,t}$	indicates if a replica is newly stored on replica server $s \in \mathcal{V}_s$ in period $t \in \mathcal{T}$
$\tilde{z}_{i,j,t}$	number of uses of arc $(i,j) \in \mathcal{A}$ during the placement in period $t \in \mathcal{T}$
$z_{i,j,t}$	indicates if arc $(i,j) \in \mathcal{A}$ is used for the placement in period $t \in \mathcal{T}$

Table A.1: Notation for DRPSL and DRPSL2

Appendix B

Test Instances

#	ρ^{max}	λ	l	α	β	γ
$i1$	0.9					
$i2$	0.8					
$i3$	0.75					
$i4$	0.7					
$i5$	**0.66**	**0.99**	**2**	**1,000**	**0.2**	**0.01**
$i6$	0.6					
$i7$	0.5					
$i8$	0.4					
$i9$	0.33					
$i10$	0.3					
$i11$	0.25					
$i12$	0.2					
$i13$	0.1					
$i14$		0.75				
$i15$		0.9				
$i16$		0.95				
$i17$		1				
$i18$			0.5			
$i19$			0.75			
$i20$			1			
$i21$			1.5			
$i22$			3			
$i23$				10		
$i24$				50		
$i25$				100		
$i26$				500		
$i27$				2,000		
$i28$					0	
$i29$					0.1	
$i30$					0.5	
$i31$					1	
$i32$						0
$i33$						0.05
$i34$						0.1
$i35$						0.5
$i36$						1
$i37$						2
$i38$						5

Table B.1: Test instances: variations of the base case (instance $i5$)

Bibliography

[1] E. Aarts, J. Korst, and M. Wil. Simulated annealing. In E. K. Burke and G. Kendall, editors, *Search Methodologies*. Springer, 2005.

[2] A. Aggarwal and M. Rabinovich. Performance of dynamic replication schemes for an internet hosting service. Technical report, AT&T Labs, 1998.

[3] J. Ahrens. A code for the transportation problem. Universitaet Kiel, 1977.

[4] C. H. Aikens. Facility location models for distribution planning. *European Journal of Operational Research*, 22(3):263–279, 1985.

[5] W. M. Aioffi, G. R. Mateus, J. M. Almeida, and R. C. Melo. Dynamic content placement for mobile content distribution networks. In *Web Content Caching and Distribution*. Springer, 2004.

[6] Akamai. Akamai technologies inc., 2009.

[7] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. *ACM SIGOPS Operating Systems Review*, 35(5):131–145, 2001.

[8] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm. Enhancing the web's infrastructure: from caching to replication. *Internet Computing, IEEE*, 1(2):18–27, 1997.

[9] N. Bartolini, F. Presti, and C. Petrioli. Optimal dynamic replica placement in content delivery networks. In *The 11th IEEE International Conference on Networks, 2003. ICON2003*, pages 125–130, 2003.

[10] K. S. Candan, W.-S. Li, and D. Agrawal. Improving user response times in application delivery networks through reduction of network and server latencies. *Journal of Interconnection Networks*, 8(3):181–208, 2007.

[11] V. Cardellini, M. Colajanni, and P. Yu. Dynamic load balancing on web-server systems. *Internet Computing, IEEE*, 3(3):28–39, 1999.

[12] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The state of the art in locally distributed web-server systems. *ACM Computing Surveys*, 34(2):263–311, 2002.

[13] Y. Chen, R. Katz, and J. Kubiatowicz. Dynamic replica placement for scalable content delivery. In *Peer-to-Peer Systems*. Springer, 2002.

[14] Y. Chen, L. Qiu, W. Chen, L. Nguyen, and R. Katz. Efficient and adaptive web replication using content clustering. *IEEE Journal on Selected Areas in Communications*, 21(6):979–994, 2003.

[15] W. Chu. Optimal file allocation in a multiple computer system. *IEEE Transactions on Computers*, C-18(10):885–889, 1969.

[16] I. Cidon, S. Kutten, and R. Soffer. Optimal allocation of electronic content. *Computer Networks*, 40(2):205–218, 2002.

[17] E. Cronin, S. Jamin, C. Jin, A. Kurc, D. Raz, and Y. Shavitt. Constrained mirror placement on the internet. *Selected Areas in Communications, IEEE Journal on*, 20(7):1369–1382, 2002.

[18] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *Internet Computing, IEEE*, 6(5): 50–58, 2002.

[19] L. W. Dowdy and D. V. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313, 1982.

[20] E. Feldman, F. A. Lehrer, and T. L. Ray. Warehouse location under continuous economics of scale. *Management Science*, 12(9):670–684, May 1966.

[21] A. Frank, L. Wittie, and A. Bernstein. Multicast communication on network computers. *Software, IEEE*, 2(3):49–61, 1985.

[22] J. D. Guyton and M. F. Schwartz. Locating nearby copies of replicated internet servers. *ACM SIGCOMM Computer Communication Review*, 25(4):288–298, 1995.

[23] P. Hansen and N. Mladenovic. Variable neighborhood search. In E. K. Burke and G. Kendall, editors, *Search Methodologies*. Springer, 2005.

[24] D. Henderson, S. Jacobson, and A. Johnson. The theory and practice of simulated annealing. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics (International Series in Operations Research & Management Science)*. Springer, 2003.

[25] F. K. Hwang and D. S. Richards. Steiner tree problems. *Networks*, 22 (1):55–89, 1992.

[26] S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. On the placement of internet instrumentation. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 295–304, 2000.

[27] S. Jamin, C. Jin, A. Kurc, D. Raz, and Y. Shavitt. Constrained mirror placement on the internet. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 31–40, 2001.

[28] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, J. W. O'Toole, Jr., M. Frans, and K. James. Overcast: Reliable multicasting with an overlay network. In *Operating Systems Design and Implementation (Proceedings of the 4th conference on Symposium on Operating System Design & Implementation)*, volume 4, pages 197–212, 2000.

[29] W. J. Jeon, I. Gupta, and K. Nahrstedt. Mmc01-6: Qos-aware object replication in overlay networks. In *Global Telecommunications Conference, 2006. GLOBECOM '06. IEEE*, pages 1–5, 2006.

[30] X. Jia, D. Li, X. Hu, W. Wu, and D. Du. Placement of web-server proxies with consideration of read and update operations on the internet. *The Computer Journal*, 46(4):378–390, 2003.

[31] K. Kalpakis, K. Dasgupta, and O. Wolfson. Optimal placement of replicas in trees with read, write, and storage costs. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):628–637, 2001.

[32] J. Kangasharju, K. W. Ross, and J. W. Roberts. Performance evaluation of redirection schemes in content distribution networks. *Computer Communications*, 24(2):207–214, 2001.

[33] J. Kangasharju, J. Roberts, and K. W. Ross. Object replication strategies in content distribution networks. *Computer Communications*, 25 (4):376–383, 2002.

[34] M. Karlsson and C. Karamanolis. Bounds on the replication cost for QoS. Technical report, Hewlett Packard Labs, July 2003.

[35] M. Karlsson and C. Karamanolis. Choosing replica placement heuristics for wide-area systems. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 350–359, 2004.

[36] M. Karlsson and M. Mahalingam. Do we need replica placement algorithms in content delivery networks. In *7th International Workshop on Web Content Caching and Distribution (WCW)*, 2002.

[37] M. Karlsson, C. Karamanolis, and M. Mahalingam. A framework for evaluating replica placement algorithms. Technical report, Hewlett Packard Labs, 2002.

[38] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*. Plenum Press, 1972.

[39] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[40] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Placement algorithms for hierarchical cooperative caching. *Journal of Algorithms*, 38 (1):260–302, 2001.

[41] L. Kou, G. Markowsky, and L. Berman. A fast algorithm for steiner trees. *Acta Informatica*, 15(2):141–145, 1981.

[42] J. Krarup and P. M. Pruzan. The simple plant location problem: Survey and synthesis. *European Journal of Operational Research*, 12(1):36–81, 1983.

[43] P. Krishnan, D. Raz, and Y. Shavitt. The cache location problem. *IEEE/ACM Transactions on Networking*, 8(5):568–582, 2000.

[44] A. A. Kuehn and M. J. Hamburger. A heuristic program for locating warehouses. *Management Science*, 9(4):643–666, July 1963.

[45] I. Lazar and W. Terrill. Exploring content delivery networking. *IT Professional*, 3(4):47–49, 2001.

[46] B. Li, M. Golin, G. Italiano, X. Deng, and K. Sohraby. On the optimal placement of web proxies in the internet. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1282–1290, 1999.

[47] A. Medina, A. Lakhina, I. Matta, and J. Byers. Brite: An approach to universal topology generation. In *Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, page 346. IEEE Computer Society, 2001.

[48] A. Medina, A. Lakhina, I. Matta, and J. Byers. Brite: Universal topology generation from a user's perspective. Technical report, Boston University, Boston, MA, 2001.

[49] N. Mladenovic and P. Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997.

[50] T. V. Nguyen, F. Safaei, P. Boustead, and C. Tung Chou. Provisioning overlay distribution networks. *Computer Networks*, 49(1):103–118, 2005.

[51] C. A. Oliveira, P. M. Pardalos, and M. G. Resende. Optimization problems in multicast tree construction. In M. G. Resende and P. M. Pardalos, editors, *Handbook of Optimization in Telecommunications*. Springer, 2006.

[52] C. A. S. Oliveira and P. M. Pardalos. A survey of combinatorial optimization problems in multicast routing. *Computers & Operations Research*, 32(8):1953–1981, 2005.

[53] G. Pallis and A. Vakali. Insight and perspectives for content delivery networks. *Communications of the ACM*, 49(1):101–106, 2006.

[54] M. Pathan, R. Buyya, and A. Vakali. Content delivery networks: State of the art, insights, and imperatives. In R. Buyya, M. Pathan, and A. Vakali, editors, *Content Delivery Networks*. Springer, 2008.

[55] S. Paul. *Multicasting on the Internet and Its Applications*. Kluwer Academic Publishers, 1998.

[56] G. Peng. CDN: Content distribution network. Technical report, Experimental Computer Systems Lab, Department of Computer Science, State University of New York at Stony Brook, 2008.

[57] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 311–320, Newport, Rhode Island, United States, 1997. ACM.

[58] H.-J. Prömel and A. Steger. *The Steiner Tree Problem*. 2002.

[59] L. Qiu, V. Padmanabhan, and G. Voelker. On the placement of web server replicas. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1587–1596, 2001.

[60] M. Rabinovich and O. Spatschek. *Web caching and replication*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[61] P. Radoslavov, R. Govindan, and D. Estrin. Topology-informed internet replica placement. *Computer Communications*, 25(4):384–392, 2002.

[62] H. S. Rahul, M. Kasbekar, R. K. Sitaraman, and A. W. Berger. Performance and availability benefits of global overlay routing. In *Content Delivery Networks*. Springer, 2008.

[63] S. Rangarajan, S. Mukherjee, and P. Rodriguez. User specific request redirection in a content delivery network. In *Web content caching and distribution: proceedings of the 8th international workshop*, pages 223–232. Kluwer Academic Publishers, 2004.

[64] S. Ranjan. Request redirection for dynamic content. In *Content Delivery Networks*. Springer, 2008.

[65] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The end-to-end effects of internet path selection. *ACM SIGCOMM Computer Communication Review*, 29(4):289–299, 1999.

[66] S. Shi and J. Turner. Placing servers in overlay networks. In *In Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPETS)*, 2002.

[67] S. Shi and J. Turner. Routing in overlay multicast networks. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1200–1208, 2002.

[68] C. Sickinger and R. Kolisch. Experimental investigation of heuristics for single-object replica placement. Paper presented at the International Conference Operations Research 2004, Tilburg, 01.-03.09.2004, 2004.

[69] X. Tang and J. Xu. Qos-aware replica placement for content distribution. *IEEE Transactions on Parallel and Distributed Systems*, 16(10):921–932, 2005.

[70] H. Tangmunarunkit, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. Network topology generators: degree-based vs. structural. In *ACM SIGCOMM Computer Communication Review (Proceedings of the 2002 SIGCOMM conference)*, volume 32, pages 147–159. ACM, 2002.

[71] O. Unger and I. Cidon. Optimal content location in multicast based overlay networks with content updates. *World Wide Web*, 7(3):315–336, 2004.

[72] A. Vakali and G. Pallis. Content delivery networks: status and trends. *Internet Computing, IEEE*, 7(6):68–74, 2003.

[73] D. C. Verma. *Content Distribution Networks: An Engineering Approach*. John Wiley & Sons, Inc., 2002.

[74] S. Voss. *Steiner-Probleme in Graphen*. Anton Hain, 1990.

[75] S. Voss. Steiner tree problems in telecommunications. In M. G. Resende and P. M. Pardalos, editors, *Handbook of Optimization in Telecommunications*. Springer, 2006.

[76] Y. Wang, Z. Wang, and L. Zhang. Internet traffic engineering without full mesh overlaying. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings*, volume 1, pages 565–571, 2001.

[77] B. Waxman. Routing of multipoint connections. *IEEE Journal on Selected Areas in Communications*, 6(9):1617–1622, 1988.

[78] R. Wittmann and M. Zitterbart. *Multicast communication: protocols and applications*. Morgan Kaufmann Publishers Inc., 1999.

[79] O. Wolfson and A. Milo. The multicast policy and its relationship to replicated data placement. *ACM Transactions on Database Systems*, 16 (1):181–205, 1991.

[80] J. Xu, B. Li, and D. L. Lee. Placement problems for transparent data replication proxy services. *IEEE Journal on Selected Areas in Communications*, 20(7):1383–1398, 2002.

[81] M. Yang and Z. Fei. A model for replica placement in content distribution networks for multimedia applications. In *IEEE International Conference on, Communications, 2003. ICC '03*, volume 1, pages 557–561, 2003.

[82] X. Zhang, W. Wang, X. Tan, and Y. Zhu. Data replication at web proxies in content distribution network. In *Web Technologies and Applications*. Springer, 2003.