*Combinatorial Online Optimization in Real Time

Martin Grötschel[1], Sven O. Krumke[1], Jörg Rambau[1], Thomas Winter[2], and Uwe T. Zimmermann[3]

[1] Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB), Germany
[2] Information and Communication Mobile Networks, Siemens AG, Germany
[3] Abteilung Mathematische Optimierung, Technische Universität Braunschweig, Germany

Abstract Optimization is the task of finding a best solution to a given problem. When the decision variables are discrete we speak of a combinatorial optimization problem. Such a problem is online when decisions have to be made before all data of the problem are known. And we speak of a real-time online problem when online decisions have to be computed within very tight time bounds. This paper surveys the art of combinatorial online and real-time optimization, it discusses, in particular, the concepts with which online and real-time algorithms can be analyzed.

1 Introduction

Models and methods from Combinatorial Optimization provide powerful tools for solving highly complex problems from a broad spectrum of industrial and other applications. The traditional optimization techniques assume, in general, knowledge of all data of a problem instance. There are many cases in practice, however, where decisions have to be made before complete information about the data is available. In fact, it may be necessary to produce a part of the problem solution as soon as a new piece of information becomes known. We call this an *online situation*, and we say that an algorithm *runs online* if it makes a decision (computes a partial solution) whenever a new piece of data requests an action.

Practice may be even more demanding. The online algorithm may indeed be required to deliver the next piece of the solution within a very tight time bound. In this case, we speak of a *real-time problem* (or real-time system), i.e., a problem where an online algorithm is required to react in real-time.

How tight do time bounds have to be in order to turn an online problem into a real-time problem? There is no general rule. A standard answer is: The required reaction time of the algorithm must be short compared to the "time frame of the system", i.e., the definition depends on problem-specific settings. For example, we all expect telecommunication and computer systems to react within a few seconds or faster. Thus, real-time algorithms that, e.g., decide about routing, switching, capacity, or paging must answer within milliseconds. Real-time algorithms controlling chemical reactions or other production processes may be given a few seconds for the computation of a solution, while in transportation or traffic a few minutes lead time could be acceptable. In fact, what could be considered real-time or not may also depend on the complexity of the mathematical model applied, the importance of the decision, and other problem-specific items.

Online and real-time problems have been around in continuous optimization (e.g., control of airplanes, re-entry of a spacecraft) for quite a long time, while combinatorial optimizers have neglected this issue to a large extent. With a few exceptions, systematic investigation of combinatorial online problems started only about 15 years ago. Initially, research was mainly driven by applications in computing and communication machinery. The emergence of new paradigms for the analysis of online algorithms particularly fostered this "combinatorial online research". Interesting and important additional applications broadened its scope.

Why is new theory necessary? Isn't it possible to transfer online results from continuous optimization to combinatorial optimization? The (unfortunate) truth is that continuous and discrete optimization are very different in nature. Combinatorial decision making is, in general, non-convex and non-continuous. Continuous techniques rarely apply to discrete models.

In this paper we will discuss many of the models that have been proposed in the recent years for the analysis of online algorithms. These models usually differ in the way information becomes available to the online algorithm. We will describe the by far most common online paradigms, the *sequence model* and the *time-stamp model*, in greater detail.

Despite significant research efforts in recent years, combinatorial online optimization is not in a mature state yet. Compared to this, combinatorial real-time optimization is even still in its infancy. No commonly accepted tools and concepts for the analysis of combinatorial real-time algorithms that take both, solution quality and time requirements, into account have been established yet. We will address this topic in Section 3.

It is, however, important to note that practical applications have become a driving force in this area. And, thus, we may hope to see new success stories on both, the theoretical and the practical side, in the near future.

1.1 The Sequence Model

An online problem in the *sequence model* can be described as follows. An algorithm ALG, we call it the *online algorithm*, is confronted with a finite *request sequence* $\sigma = r_1, r_2, \ldots$. The requests must be served in the order of their occurrence. More precisely, when serving request r_i, the online algorithm ALG does not have any knowledge of requests r_j with $j > i$. When request r_i is presented to ALG it must be served by ALG according to the specific rules of the problem. The action taken by ALG to serve r_i incurs a cost and the overall goal is to minimize the total service cost.[1] The decision by ALG of how to serve r_i is irrevocable. Only after r_i has been served, the next request r_{i+1} becomes known to ALG. In some cases the appearance of the last request is announced, in some not.

We begin with sketching a very basic decision problem that occurs in various forms frequently in everyday life. We phrase it as a ski rental problem. Despite its

[1] It is also possible to define online profit-maximization problems. For those problems, the serving of each request yields a profit and the goal is to maximize the total profit obtained.

simplicity the ski rental problem will enable us to point out some of the subtleties in the modeling and analysis of online algorithms.

Example 1 (Ski Rental Problem). Suppose that a woman goes skiing for the first time in her life. She is faced with the question of whether to buy skis for $B \gg 1$ Euro or to rent skis at the cost of 1 Euro per day. Of course, if the woman knew how many times she would go skiing in the future, her decision would be easy. But unfortunately, she is in an online situation where the number of skiing days only becomes known at the very last day. □

The above situation can be modeled as an online problem in the sequence model. In the *Ski Rental Problem* each request r_i is a day the woman goes skiing. Each request can be "served" in three different ways: (i) rent skis at the cost of 1 Euro, (ii) buy skis at the cost of B Euro, (iii) use the skis that she already owns at the cost of 0 Euro (where of course this option is only available in case she already bought skis when serving some request r_j with $j < i$). Request r_{i+1} (that is, the next skiing day, if there is any) only becomes known to the woman after r_i has been served. The overall goal is to minimize the total rental/buying cost.

Some comments apply to the ski rental problem. We have formulated the problem in such a way that the skiing woman does neither have any lookahead (that is knowledge about a certain number of subsequent requests) nor any statistical information about the future. This is in accordance with the basic sequence model. If we want to incorporate any of these additional information into the problem then the sequence model must be augmented.

Example 2 (Paging Problem). Consider a two-level memory system (e.g., of a computer) that consists of a small fast memory (the cache) with k pages and a large slow memory consisting of a total of N pages. Each request specifies a page in the slow memory, that is, $r_i \in \{1, \ldots, N\}$. In order to serve the request, the corresponding page must be brought into the cache. If a requested page is already in the cache, then the cost of serving the request is zero. Otherwise one page must be evicted from the cache and replaced by the requested page at a cost of 1. A paging algorithm specifies which page to evict. An online algorithm must base its decisions when serving r_i only on the requests $r_1, \ldots, r_i$ without any knowledge of future requests. The objective is to minimize the total cost of processing the sequence of page requests. □

1.2 The Time Stamp Model

In the *time stamp model* requests become available over time at their *arrival* or *release dates*. The release date $t_i \geq 0$ is a nonnegative real number and specifies the time at which request r_i is released (becomes known). An online algorithm ALG must determine its behavior at a certain moment t in time as a function of all the requests released up to time t. Again, we are in the situation that an online algorithm ALG is confronted with an input sequence $\sigma = r_1, \ldots, r_n$ of requests which is given in order of non-decreasing release times and the service of each request incurs a cost

for ALG. The difference to the sequence model is that the online algorithm is allowed to wait and to revoke decisions. Waiting incurs additional costs, typically depending on the elapsed time. Previously made decisions may, of course, only be revoked as long as they have not been executed.

Example 3 (Online Machine Scheduling with Jobs Arriving Over Time).
In scheduling one is concerned with the distribution of jobs (activities) to a number of machines (the resources). In our example, one is given m identical machines and is faced with the task of scheduling independent jobs on these machines. The jobs become available at their release dates, specifying their processing times. An online algorithm learns the existence of a job only at its release date. Once a job has been started on a machine the job may not be preempted and has to run until completion. However, jobs that have been scheduled but not yet started may be rescheduled. The objective is to minimize the average flow time of a job, where the *flow time* of a job is defined to be the difference between the completion time of the job and its release date. □

The above problem can be modeled as an online problem in the time stamp model. Request r_i (corresponding to job i) is a pair $r_i = (t_i, p_i)$, where t_i is the release time of job i and p_i is the processing time. An online algorithm must make its decisions at point t in time only based on the jobs released up to time t. The online algorithm may leave some of its machines idle for some time even if unprocessed jobs that have already been released exist. (Using a small amount of idle time can actually be beneficial in order to gather information about potential new jobs).

Example 4 (Online Traveling Salesman Problem). An instance of the *Online Traveling Salesman Problem* consists of a metric space $M = (X, d)$ with a distinguished origin $o \in M$ and a sequence $\sigma = r_1, \ldots, r_n$ of requests. Each request is a pair $r_i = (t_i, x_i)$, where t_i is the time at which request r_i is released (becomes known), and $x_i \in X$ is the point in the metric space requested to be visited. A server is located at the origin o at time 0 and can move at unit speed. A feasible online/offline solution is a route for the server which serves all requested points, where each request is served not earlier than the time it is released, and which starts and ends in the origin o. The cost of such a route is the time when the server has served the last request and has returned to the origin (if the server does not return to the origin at all, then the cost of such a route is defined to be infinity). This objective function is also called the *makespan* in scheduling.

It is assumed here that an online algorithm does neither have information about the time when the last request is released nor about the total number of requests. An online algorithm must determine the behavior of the server at a certain moment t of time as a function of all the requests released until time t. □

Notice that the Online Traveling Salesman Problem differs from its famous relative, the Traveling Salesman Problem (see Example 17 in Section 3.2), in certain aspects: First, the cost of a feasible solution is not the length of the tour but the total travel-time needed by the server. The total travel time is obtained from the tour

length plus the time during which the server remains idle. Second, due to the online nature of the problem it may be unavoidable that a server reaches a certain point in the metric space more than once.

A delicate issue arises when designing an online algorithm for the Online Traveling Salesman Problem: Suppose that at some moment in time all known requests have been served. If the algorithm wants to produce a solution with finite cost, then its server must return to the origin after a finite amount of waiting time. But how long should this waiting time be? If the server returns immediately, then a new request might become known and all the traveling to the origin has been in vain. However, a too large waiting time before returning to the origin increases the cost of the solution unnecessarily.

2 Competitive Analysis

Combinatorial online problems and algorithms had been studied in the sixties to eighties rather sporadically. Broad systematic investigation started when Sleator and Tarjan [46] suggested comparing an online algorithm to an *optimal offline algorithm*, thus laying the foundations of *competitive analysis*. The term "competitive analysis" was coined in the paper [33].

We call an algorithm *deterministic* if its actions are uniquely determined by the input. A *randomized* algorithm may, in contrast, execute random moves, i.e., one and the same input given to such an algorithm twice may result in two different outputs. For the analysis of deterministic and randomized algorithms, of course, different tools are needed.

2.1 Deterministic Algorithms

Let ALG be a deterministic online algorithm. Given a request sequence σ denote by $\mathsf{ALG}(\sigma)$ the cost incurred by ALG when serving σ and denote by $\mathsf{OPT}(\sigma)$ the optimal offline cost (the optimal offline algorithm OPT knows the entire request sequence in advance and hence can serve it with minimum cost).

Definition 5 (Competitive Algorithm, Deterministic Case). Let $c \geq 1$ be a real number. A deterministic online-algorithm ALG is called c-*competitive* if

$$\mathsf{ALG}(\sigma) \leq c\,\mathsf{OPT}(\sigma) \tag{1}$$

holds for any request sequence σ. The *competitive ratio* of ALG is the infimum over all c such that ALG is c-competitive. □

We want to remark here that the definition of c-competitiveness varies in the literature. Often, an online algorithm is called c-competitive if there exists a constant b such that

$$\mathsf{ALG}(\sigma) \leq c\,\mathsf{OPT}(\sigma) + b$$

holds for any request sequence. Some authors even allow b to depend on some problem or instance specific parameters. Thus, whenever c-competitiveness is addressed

one should check which definition is applied. We will stick to the definition given above since, in the examples we consider, requiring $b = 0$ is the natural choice.

Observe that, in the above definition, there is no restriction on the computational resources of an online algorithm. The only scarce resource in competitive analysis is information. In many practical applications, severe restrictions on the computation time of an online algorithm apply. We address this issue in Section 3.

Competitive analysis of online-algorithms can be imagined as a game between an *online player* and a malicious *offline adversary*. The online player uses an online algorithm to process an input which is generated by the adversary. If the adversary knows the (deterministic) strategy of the online player, he can construct a request sequence which maximizes the ratio between the player's cost and the optimal offline cost.

We illustrate competitive analysis of deterministic online algorithms on two examples.

Example 6 (A Competitive Algorithm for the Ski Rental Problem). Due to the simplicity of the Ski Rental Problem *all possible* deterministic online algorithms can be specified. A generic online algorithm ALG_k rents skis until the woman has skied $k-1$ times for some $k \geq 1$ and then buys skis on day k. The value $k = \infty$ is allowed and means that the algorithm never buys. Clearly, each such algorithm is online. Notice that on a specific request sequence σ algorithm ALG_k might not get to the point that it actually buys skis, since σ might specify less than k skiing days. We claim that ALG_k for $k = B$ is c-competitive with $c = 2 - 1/B$.

Let σ be any request sequence specifying n skiing days. Then our algorithm has cost $\mathsf{ALG}_B(\sigma) = n$ if $n \leq B-1$ and cost $\mathsf{ALG}_B(\sigma) = B-1+B = 2B-1$ if $j \geq B$. Since the optimum offline cost is given by $\mathsf{OPT}(\sigma) = \min\{n, B\}$, it follows that our algorithm is $(2-1/B)$-competitive. □

Example 7 (A Bad Algorithm for the Paging Problem). The algorithm LFU – least frequently used – for the Paging Problem given in Example 2 works as follows: For each page p from the main memory, LFU maintains a counter on the number of times that p has been requested so far. Upon a request r_i which is currently not in the cache, LFU evicts the page from the fast memory which has been requested least frequently in the past.

The algorithm LFU is not competitive. Indeed: suppose that $X = \{p_1, \ldots, p_k\}$ is the initial cache contents and p_{k+1} is one additional page from the slow memory. Let $\ell \geq 1$, and consider the sequence $\sigma = p_1^\ell, p_2^\ell, \ldots, p_{k-1}^\ell, (p_{k+1}, p_k)^\ell$. Here p_i^ℓ means that page p_i is requested ℓ times in a row and $(p_{k+1}, p_k)^\ell$ states that p_{k+1} and p_k are requested alternatingly ℓ times. Starting with the $\ell(k-1)+1$st request, LFU has cost 1 for every subsequent request, which gives $\mathsf{LFU}(\sigma) = 2\ell$. On the other hand, OPT can process the sequence at cost 1 by evicting page p_1 upon the first request to p_{k+1}. Since ℓ can be chosen arbitrarily large, it follows that LFU is not competitive. □

Example 8 (Negative Result in Machine Scheduling). The online scheduling problem in Example 3 is notoriously difficult. It can be shown that even in the case of a

single machine any deterministic online algorithm has a competitive ratio that grows with the number of jobs presented in the input sequence. More precisely, any deterministic online algorithm has a competitive ratio of at least $n - 1$, where n is the number of jobs. (see [24, Chapter 9]). □

Example 9 (Competitive Algorithms for the Online TSP). Probably the most obvious algorithm for the Online TSP (see Example 4 for the definition) is given by the following "REPLAN"-strategy: If a new request becomes known, plan a shortest route starting at the current position, serving all yet unserved requests and ending in the origin. It can be shown that this algorithm is $5/2$-competitive (see [8,9]). However, there are more complicated algorithms which achieve a competitive ratio of 2 (see [6, 8, 9]) in general metric spaces. For the special case that the metric space is the real line, a $7/4$-competitive algorithm is presented in [8,9]. □

2.2 *Randomized Algorithms*

So far we have only considered deterministic online algorithms. The definition of competitiveness for randomized algorithms is a bit more subtle. In the case of a deterministic online algorithm, the adversary has complete knowledge about his opponent and can exploit this knowledge. For randomized algorithms we have to be precise in defining what kind of information about the online player is available to the adversary. This leads to different adversary models which are explained below. For an in-depth treatment we refer to [15, 39].

An *oblivious adversary* (OBL) must choose the entire request sequence in advance. He does neither have knowledge about the outcome of the random experiments of the online algorithm ALG nor about the specific actions taken by ALG as a result of the random decisions. However, the oblivious adversary knows the online algorithm ALG itself including the probability distributions guiding ALG's decisions.

An *adaptive adversary* can choose each request in the input sequence based on knowledge of all actions taken by the randomized algorithm so far, and of the outcome of all random experiments. One distinguishes different adaptive adversaries depending on how the adversary himself must serve the input sequence.

The *adaptive offline adversary* (ADOFF) defers serving the request sequence until he has generated the last request. He then uses an optimal offline algorithm. The *adaptive online adversary* (ADON) must serve the input sequence (generated by himself) online. Notice that in case of an adaptive adversary ADV, the adversary's cost $\mathsf{ADV}(\sigma)$ for serving σ is a random variable.

Definition 10 (Competitive Algorithm, Randomized Case). A randomized algorithm ALG is c-*competitive against an adversary of type* $\mathsf{ADV} \in \{\mathsf{OBL}, \mathsf{ADON}, \mathsf{ADOFF}\}$ for some $c \geq 1$, if

$$\mathbb{E}[\mathsf{ALG}(\sigma) - c\,\mathsf{ADV}(\sigma)] \leq 0 \tag{2}$$

for all request sequences σ. Here, the expectation on the left hand side is taken over all random choices made by ALG. □

In case of an oblivious adversary, the adversary's cost $\mathsf{ADV}(\sigma) = \mathsf{OBL}(\sigma)$ does not depend on any random choices made by the online algorithm. Hence, a randomized online algorithm ALG is c-*competitive* against an oblivious adversary, if for any request sequence the inequality $E[\mathsf{ALG}(\sigma)] \leq c\,\mathsf{OPT}(\sigma)$ holds.

The power of a randomized algorithm depends on the adversary it competes with. Relations between the adversaries have been studied in a general model called *request-answer games* (see [15]). It turns out that randomization does not help against an adaptive offline adversary. More precisely, it can be shown that the existence of a c-competitive algorithm against an adaptive offline adversary implies the existence of a deterministic algorithm which is c-competitive (see [15]). However, against an oblivious adversary, a randomized algorithm can "hide" its current configuration from the adversary which might enable him to achieve a better competitive ratio.

Example 11 (Ski Rental Problem Revisited). We look again at the Ski Rental Problem given in Example 1. It is easy to see that any deterministic algorithm has a competitive ratio at least $(2 - 1/B)$. Any competitive algorithm must buy skis at some point in time. The adversary simply presents skiing requests until the algorithm buys and then ends the sequence. A straightforward calculation shows that this forces a ratio of at least $2 - 1/B$ between the online and the offline cost.

We consider the following randomized algorithm RANDSKI against an oblivious adversary. Let $\rho := B/(B-1)$ and $\alpha := \frac{\rho-1}{\rho^B-1}$. At the start RANDSKI chooses a random number $k \in \{0, \ldots, B-1\}$ according to the distribution $\Pr[k = x] := \alpha\rho^k$. After that, RANDSKI works completely deterministic, buying skis after having skied k times. We analyze the competitive ratio of RANDSKI against an oblivious adversary. Note that it suffices to consider sequences σ specifying at most B days of skiing. For a sequence σ with $n \leq B$ days of skiing, the optimal cost is clearly $\mathsf{OPT}(\sigma) = n$. The expected cost of RANDSKI can be computed as follows

$$\mathbb{E}[\mathsf{RANDSKI}(\sigma)] = \sum_{k=0}^{n-1} \alpha\rho^k(k+B) + \sum_{k=n}^{B-1} \alpha\rho^k n$$

A lengthy computation shows that

$$E[\mathsf{RANDSKI}(\sigma)] = \frac{\rho^B}{\rho^B - 1}\mathsf{OPT}(\sigma).$$

Hence, RANDSKI is c_B-competitive with $c_B = \frac{\rho^B}{\rho^B-1}$. Since $\lim_{B\to\infty} c_B = e/(e-1) \approx 1.58$, this algorithm achieves a better competitive ratio than any deterministic algorithm whenever $2B - 1 > e/(e-1)$, that is, when $B > (2e-1)/2(e-1)$. □

Example 12 (Paging Revisited). It can be shown that no deterministic algorithm for the Paging Problem (see Example 2) can achieve a competitive ratio smaller than k, the size of the cache. However, there exists a randomized algorithm which is $2H_k$, competitive, where $H_k = 1 + 1/2 + \cdots + 1/k$ is the kth harmonic number. Proofs and the algorithm can be found in [15]. □

Example 13 (Machine Scheduling Revisited). The scheduling problem from Example 3 remains difficult even for randomized algorithms. Every randomized algorithm has a competitive ratio of $\Omega(\sqrt{n})$ against an oblivious adversary where again n denotes the number of jobs given in the request sequence (see [49]). □

2.3 *Alternatives to Competitive Analysis*

Competitive analysis is a type of worst-case analysis. It has (rightly) been criticized as being overly pessimistic. The competitive ratios observed in practice are usually much smaller than the pessimistic bounds provable from a theoretic point of view. Often the offline adversary is simply too powerful and allows only trivial competitiveness results. This phenomenon is called "hitting the triviality barrier" (see [24]). To overcome this unsatisfactory situation various extensions and alternatives to pure competitive analysis have been investigated in the literature.

In *comparative analysis* the class of algorithms where the offline algorithm is chosen from is restricted. This concept has been explored in the context of the Paging Problem [34] and the Online TSP [14]. Another approach to strengthen the position of an online algorithm is the concept of *resource augmentation* (see, e.g. [10, 41, 42, 46]). Here, the online algorithm is given more resources (more or faster machines in scheduling) to serve requests than the offline adversary. The *diffuse adversary* [34] model deals with the situation where the input is chosen by an adversary according to some probability distribution. Although the online algorithm does not know the distribution itself, it is given the information that this distribution belongs to a specific class of distributions. Other approaches to go beyond pure competitive analysis include the *access graph model* for paging [16, 17, 32] and the *statistical adversary* [18]. We refer to [24, Chapter 17] for a comprehensive survey.

All of the extensions and alternatives to competitive analysis have been proven to be useful for some *specific problem* and powerful enough to obtain meaningful results. However, none of these approaches has yet succeeded in replacing competitive analysis as *the* standard tool in the theoretical analysis of online algorithms. Hence, it is particularly irritating that competitive analysis can only give substantial decision support for a few "real-world problems".

3 Real-Time Issues

In real-time systems (cf. section 1), an algorithm has to deliver a solution within prescribed time constraints. The behavior of a real-time system depends of course on the quality of the solution but it depends as well as on the time needed for producing the solution. A solution provided too late may be useless or, in some cases, even dangerous because it does not fit to the current system parameters which may vary over time.

For instance, if a decision support system watching the stock market needs a long time to propose buying or selling a certain share, the price of the share (especially in a volatile market) may have changed so much that this action is no longer

reasonable. If, however, the decision support system of a pilot takes long to suggest the right action in case of an emergency the result may be fatal.

In our context, the notion *time* emphasizes the fact that the system significantly depends on the time in which answers to requests are produced. The notion *real* indicates that the system's reaction to external events must occur instantaneously. In other contexts, *real-time* reaction is just a synonym for fast reaction to external events. We have to be more precise, the speed of *real-time* reaction must correspond to the specific time requirements of the systems environment and the problem setting. The time available for computation may vary, e.g., from milliseconds to minutes. The general objective of real-time optimization is to match the problem specific timing requirements of each task and to produce a best possible solution within the incurred time constraints. Since the solution is based on the information available at the beginning of the computation, it may be necessary to check its feasibility for the state of the system at the end of the computation.

3.1 Real-Time Decision Support Systems

Real-time algorithms are often integrated into computerized decision support systems, see [44] for such examples in local transport.

Decision support can be based on the knowledge of a previously forecasted development of the real-time system. In our context, such forecasts may be obtained via offline computations of optimal solutions of some combinatorial optimization problem for real-world data describing the standard situation of the real-time system. We will thus call the presently available forecasted development of the real-time system briefly the *current solution*. Real-time decision support systems provide proposals for "quick" reactions to external unforseen events which change the current solution. Real-time decisions usually have to be made subject to and despite of severe limitations of resources: hardware, time, and information. Some fundamental components of such a decision system (according to [44]) are:

1. Information management: current update of incoming information.
2. Situation assessment: evaluation of the situation, decision whether or not a reaction of the system is required (or should be proposed).
3. Evaluation of alternatives: checking possible actions for the real-time event.
4. Decision: determining an action (or choosing to do nothing).

Real-time decision support systems for complex real-time systems are (more or less) semi-autonomous systems that support and assist human operators. Due to efficiency, responsibility, and security issues, human operators are seldom replaced by such systems. On the other hand, these systems usually require highly qualified personnel.

Decision support systems may propose actions with different degree of influence on the development of the real-time system. Three different types of decisions with increasing impact [44] are *reactive planning*, *incremental planning*, and *deliberative planning*. In reactive planning, the current solution is only locally adapted to some real-time event. Incremental planning already results in a more global update of the

current solution. Deliberate planning is a complete revision of the current solution. This is advisable when the observed situation significantly differs from the predicted state so that the current solution becomes ineffective or even infeasible. The choice of reactions on real-time events depends on the time available for computation and on the observed effects of the real-time event.

Example 14 (Dispatching Trams in Local Transport). In municipal tram dispatch, trams start from a certain depot for serving scheduled round trips. In the depot, trams are stored in several sidings one behind the other. The dispatcher has to assign the trams to a sequence of round trips each requiring a certain type of tram [13, 50, 51].

Due to unforeseen external events, e.g., delays, the pre-calculated current feasible assignment of trams to round trips has to be replaced by a new one. The dispatcher needs to find the new feasible assignment as fast as possible within a few minutes. His objective is to minimize (or prevent) shunting of trams. Otherwise, new delays may be generated or more tram drivers may be required for moving trams. □

For results on competitive analysis for online versions of tram dispatch problems, unfortunately mainly negative observations have been made, we refer to [50, 51].

In the following sections we survey some of the prominent methods for solving offline-optimization problems and comment on their usability in a real-time context.

3.2 Exact Solution Methods for Combinatorial Optimization Problems

Online and real-time algorithms try, of course, to make use of the existing machinery of combinatorial optimization. Core ingredients are, thus, fast solvers for linear, integer and mixed-integer programs.

Mixed-integer programming (MIP) provides effective tools for solving combinatorial optimization problems which arise from industrial applications. Constraints from combinatorial optimization can often easily be reformulated in terms of linear MIP-constraints though it may turn out to be difficult to find a computationally effective formulation. Modelling software like AMPL [25] or GAMS [19] is available which support modelling and problem solving. Powerful state-of-the-art solvers for linear and mixed integer programming problems such as CPLEX [12] have successfully been applied to such formulations of industrial applications.

Definition 15 (Mixed integer programming). In (linear) mixed integer programming the given (linear) objective function

$$c^T x + d^T y \tag{3}$$

has to be minimized subject to the given (linear) constraints

$$A_1 x + A_2 y = b_1 \tag{4}$$

$$A_3 x + A_4 y \leq b_2 \tag{5}$$

for integer valued vectors x and real valued vectors y. □

Solving MIPs is difficult in theory (NP-hard) and, in general, hard in practice. Nevertheless, MIP formulations and solution techniques may help under real-time constraints. Here are two examples of the MIP approach.

Example 16 (Load Balancing on Identical Machines). Consider the *Load Balancing Problem* (also called *makespan minimization*) arising in machine scheduling. One is given a sequence $I = (1, \ldots, n)$ of jobs where job i has processing time p_i. The task is to distribute the jobs on m identical machines such that the maximum load of a machine is minimized. Here, the load of a machine is defined to be the sum of job processing times assigned to the machine.

The above problem can be formulated as the following Integer Linear Program:

$$\begin{aligned}
&\text{minimize } M \\
&\text{subject to} \\
&\sum_{i=1}^{n} p_i x_{ij} \le M && \text{for } j = 1, \ldots, m && (6)\\
&\sum_{j=1}^{n} x_{ij} = 1 && \text{for } j = 1, \ldots, n && (7)\\
&x_{ij} \in \{0, 1\} && \text{for all } i, j && (8)
\end{aligned}$$

The binary (decision) variable x_{ij} has the following meaning: $x_{ij} = 1$ if and only if job i is assigned to machine j. Constraints (7) ensure that each job is assigned to exactly one machine, constraints (6) ensure that M is greater or equal to the load of any of the m machines. Since M is minimized it follows that in an optimal solution M will be exactly the maximum load of a machine. □

Example 17 (Offline Traveling Salesman Problem). In the famous symmetric *Traveling Salesman Problem* one is given a complete undirected graph $G = (V, E)$ on n vertices $V = \{1, \ldots, n\}$ with (symmetric) edge weights d_{ij} for each edge $ij \in E$. The problem consists of finding a shortest tour starting and ending at the same vertex and visiting each other vertex exactly once. The cost of a solution is the total length of all edges in the tour. □

We formulate the Traveling Salesman Problem as an Integer Linear Program. To this end define, for a subset $S \subseteq V$, the set $\delta(S) := \{ ij \in E : i \in S, j \notin S \}$ of edges incident with S. Then using the decision variables x_{ij}, with $x_{ij} = 1$ if and only if edge ij is contained in the tour, we can write the TSP as the following Integer Linear

Program

$$\text{minimize} \sum_{i,j=1}^{n} d_{ij}x_{ij}$$

subject to

$$\sum_{ij\in\delta(\{i\})} x_{ij} = 2 \qquad \text{for } i = 1,\ldots,n \tag{9}$$

$$\sum_{ij\in\delta(S)} x_{ij} \geq 2 \qquad \text{for all } S \subseteq V, 2 \leq |S| \leq |V|/2 \tag{10}$$

$$x_{ij} \in \{0,1\} \qquad \text{for all } i,j \tag{11}$$

A proof that the feasible solutions to the above Integer Linear Program are in fact exactly (incidence vectors of) tours, can be found, e.g., in [22, 36].

As already noted, there are important differences between the objectives in the Offline TSP and the Online TSP specified in Example 4. However, an algorithm for the Online TSP can make use of an (exact or approximate) algorithm for the Offline TSP to solve the following sub-problem: For a set of known but yet unserved requests R find a shortest route which serves all requests in R and returns to the origin.

Integer programming formulations are quite flexible and general. While adding or cancelling of constraints and/or variables in a MIP may severely change the complexity of the model, it still remains a MIP and thus basic methods for solving MIP's still apply. Combinatorial algorithms specially designed and tuned for some combinatorial optimization problem usually break down when such changes become necessary.

For example, integer programming methods have successfully been applied to real-time problems in transport and logistics. If solving a complete real-time model turns out to be too time-consuming, it may be decomposed into smaller parts which can be solved fast enough. Trading computing time versus solution quality helps to adapt the problem setting to the changing requirements in real-time applications.

Example 18 (Dispatching Trams in Local Transport Revisited). The task of finding shunting free assignments for the tram dispatching problem of Example 14 can be modelled as a 0-1-quadratic assignment problem [50, 51].[2] Shunting-free assignments correspond to assignments that obey certain additional side constraints [13]. After exact linearization and some model tuning, the resulting integer programming model can be solved within reasonable time [50, 51]. □

A similar approach has proved to be useful in the context of container logistics [47].

Exact solution methods, here based on mixed integer programming formulations of the combinatorial optimization model, are essential for pre-calculation of good or

[2] See [20] for a definition of the quadratic assigment problem and a comprehensive survey of solution approaches.

optimal solutions to real-time optimization problems. While computation times are reasonable, matching tight real-time requirements in real-time applications may enforce tradeoffs between solution quality and computation time. However, even then, exact methods provide indispensable information on the quality of other approaches.

3.3 Approximation Algorithms

If exact methods fail to produce answers in real-time the next step is to look for suboptimal solutions which have a guaranteed quality. *Approximation algorithms* for offline minimization problems are closely related to competitive online algorithms.

Definition 19 (Approximation algorithm). A deterministic algorithm ALG is α-*approximative* if

$$\mathsf{ALG}(I) \leq \alpha\, \mathsf{OPT}(I) \tag{12}$$

holds for any problem instance I. The quantity $\alpha - 1$ provides a worst case bound on the relative error of the approximation. The infimum of all values of α for which ALG is α-approximative is called *performance ratio* of ALG. (The remarks made on variants of the definition of c-competitiveness also apply here.) □

By the above definition, a c-competitive online algorithm is c-approximative. Conversely, if a c-approximate algorithm is also online, it is also c-competitive. In view of applications, in the design of approximation algorithms speed is of first priority since here computation time is the scarce resource. Thus, one usually restricts approximation algorithms to the class of polynomial time algorithms.[3] In contrast, time complexity is not an issue in competitive analysis: there is (at least in theory) no bound on the computation time for an answer generated by an online algorithm.

Many approximation algorithms have a simple structure and are in fact online. For NP-hard problems, polynomial time approximation algorithms offer a way to trade solution quality for computation time. Polynomial time approximation algorithms have intensively been considered within the last years. Comprehensive surveys on approximation algorithms can be found in [7, 31, 38, 48].

Example 20 (Load Balancing on Identical Machines revisited). Consider again the load balancing problem described in Exampe 16. Graham [27, 28] proposed the following greedy-type heuristic LIST: Consider the jobs in order of their occurence in the input sequence I. Always assign the next job to the machine currently with the least load (breaking ties arbitrarily). Clearly, LIST can be implemented to run in polynomial time. Moreover, LIST is also an online algorithm for the online version of the problem where jobs are revealed to an online algorithm according to the sequence model.

We are now going to analyze the performance of LIST. Obviously, the optimum load is at least as large as any job processing requirement resp. at least as large the

[3] In the literature often the notion of an approximation algorithm includes the property of the algorithm being polynomial time.

average processing time for each machine, i.e.:

$$\mathsf{OPT}(I) \geq p_i \quad \text{for } i = 1, \ldots, n \qquad \text{and} \qquad \mathsf{OPT}(I) \geq \frac{1}{m} \sum p_i. \tag{13}$$

Consider the machine j where LIST generates the maximum load when processing I. Let p_i be the load of the last job assigned to machine j and let L be the load of j before job i was assigned. With these notations we have $\mathsf{LIST}(I) = L + p_i$.

By definition of LIST, at the moment job i was assigned to machine j all other machines had load at least L. Hence, the total sum of job sizes is at least $mL + p_i$. Hence from the second inequality in (13) we get $\mathsf{OPT}(I) \geq 1/m(mL + p_i) = L + p_i/m$. This results in

$$\mathsf{LIST}(I) = L + p_i \leq \mathsf{OPT}(I) + \left(1 - \frac{1}{m}\right) p_i \leq \left(2 - \frac{1}{m}\right) \mathsf{OPT}(I),$$

where for the last inequality we have used the first inequality in (13).

This proves that LIST is $(2 - 1/m)$-approximative. Since we have already remarked that LIST is in fact an online algorithm, LIST is also $(2 - 1/m)$-competitive for the online variant of the problem in the sequence model. □

For $m \geq 2$, Albers [1] describes an online scheduling algorithm which is 1.923-competitive. Her algorithm tries to prevent schedules which distribute the load uniformly on all machines by keeping some machines with a "low" load whereas the other machines have a "high" load. For $m \geq 80$, Albers [1] derives a lower bound of 1.852 on the competitive ratio of deterministic online algorithms for the machine scheduling problem.

In the offline case, LIST can easily be improved by taking advantage of the information about I. In worst-case examples for LIST, the last job has a very long processing time. By sorting the jobs in non-increasing order according to their processing times, i.e., processing jobs with the longest processing times first, a better approximation ratio of $\frac{4}{3} - \frac{1}{3m}$ can be achieved [28]. Since sorting is quite fast, this algorithm may still be applied to real-time versions of machine scheduling problems where several jobs arrive simultaneously.

Example 21 (Offline Traveling Salesman Problem Revisited). It is easy to see that for the Traveling Salesman Problem (see Example 17), polynomial-time approximation algorithms with constant performance ratio can only exist if the edge weights satisfy the triangle inequality [26]. In this case, a 2-approximative algorithm can be constructed using a minimum spanning tree in the graph [40]. Christofides' algorithm [21] also starts with a minimum spanning tree. For the nodes with odd degree in this tree a shortest perfect matching is computed. Then, a tour following a Eulerian walk in the multi-graph formed by the spanning tree and the perfect matching is constructed. The solution found this way is $\frac{3}{2}$-approximative.

For the special case that the vertices in the input graph corresponds to points in the Euclidean plane and the edge lengths are given by the Euclidean distances,

Arora [2] and independently Mitchell [37] have devised polynomial time approximation schemes.[4] However, no practical implementation of these fairly complicated algorithms has been reported yet. □

Real-time applications require that answers are computed online *and* within tight time windows. The length of this time window is closely connected to the arrival times of the requests. In view of the discussion of polynomial approximation algorithms, one may define a real-time algorithm as an online algorithm that generates answers in constant or, at least, in "suitably" low polynomial time. A concept for the evaluation of the performance of real-time algorithms, that combines approximation aspects and time requirements in a convincing manner, would be of great value for real-time applications. Up to now, no convincing concept has been proposed.

3.4 Offline Heuristics (without Provable Worst-Case Performance Guarantees)

In some applications, optimal or approximate solutions even for small problem instances cannot be computed within the tight required real-time bounds. The typical approach in this case is to look for algorithms that quickly produce a feasible solution and iteratively keep on improving the solution. There are general principles, such as *local search* (or more fashionable: *meta heuristics*), that can be adapted to special applications and have indeed successfully been applied to many real-world applications. For a comprehensive introduction to local search we refer to [23, 43].

Local search for a combinatorial optimization problem proceeds in the following way. Let $\mathcal{F}$ be the set of all feasible solutions (also called *solution space*). For each feasible solution $x \in \mathcal{F}$, one defines a *neighborhood* $N_x \subseteq \mathcal{F}$ containing all feasible solutions which are "close" to x and which can be reached from x by applying certain modifications to x. The solution space $\mathcal{F}$ is covered by the collection of neighborhoods $\{ N_x : x \in \mathcal{F} \}$.

Starting from an initial feasible solution, local search moves from one feasible solution to another, while storing the best solution found so far. Local search can thus be stopped at any time and will always provide a feasible solution. In our context, local search algorithms may thus be called *any-time algorithms*. The initial solution for a local search algorithm is usually generated using a starting heuristic. The local search algorithm terminates either after a certain number of steps (in the context of real-time computation this may also be after a user-defined time threshold) or according to some other stopping criterion with respect to the objective function value.

The basic local search paradigm leaves open how a successor $x' \in N_x$ to the current solution x is selected. Different rules to select a successor lead to different incarnations of local search. For instance, a *Greedy-type local search* would always choose the solution with the best objective function value among all solutions in N_x.

[4] An approximation scheme consists of a collection $\{\mathsf{ALG}_\varepsilon : \varepsilon > 0\}$ of algorithms where ALG_ε ϵ-approximate and has polynomial running time.

However, since such an algorithm can get stuck at local optima, various other approaches have been suggested in the literature. In *Simulated Annealing* one accepts also successors with worse objective function value but only with a certain probality which decreases over time. *Tabu Search* is another implementation of local search which attempts to avoid a breakdown at local optima. Other approaches include so called *improvement heuristics* like k-opt. We refer to [23, 43] for comprehensive survey.

Example 22 (Dispatching Trams in Local Transport by Local Search). An example for a powerful local search algorithm is the reactive tabu search (RTS) heuristic developed by Battiti and Tecchiolli [11]. In RTS, the next solution in the neighborhood is chosen at random while recording whether and how often this solution has been visited before. If a re-visiting counter exceeds a threshold, some random steps are executed in order to leave the previously visited neighborhood in which RTS threatens to stall. RTS is known to be very effective for instances for quadratic assignment problems.

The real-time tram dispatch problem introduced in Example 14, requires to compute tram assignments within two minutes. Reactive tabu search provided optimal solutions for more than 80 percent of the considered real-world instances as well as for randomly generated instances within these tight time bounds [50, 51]. □

4 General-Purpose Online-Heuristics

There are general principles which can be used to design an online algorithm.

4.1 *First-In-First-Out* (FIFO)

The FIFO-strategy does only make sense in the time stamp model. This approach to control the order in which requests are served completely works without regard of efficiency issues: FIFO strictly serves requests in the order of appearance.

Although it is clear form the definition that this strategy is almost never cost-efficient it is incredibly popular in production-planning and control. One reason for this might be that FIFO has a desired side-effect: items in a production environment are delivered according in the order of production, so that no newer items are sold (or used) before the old items are cleared. One other reason is that a FIFO-heuristic is sometimes hidden in a control system based on priority rules. These systems usually employ FIFO as a tie-breaker inside the priority classes. Whenever there are many requests in one priority class the efficiency problem will take effect. Thus, a major problem for such controls is catching up after system break-downs.

In principle it is possible to use any of the following strategies as a tie-breaker in a priority-based control system. Therefore, FIFO– if not explicitely required – is usually an inferior strategy whenever there is a substantial number of requests available for planning.

4.2 Greedy

The greedy algorithm is a well defined algorithm in the context of matroids or independence systems in combinatorial optimization. In terms of online optimization the notion of an algorithm being "*greedy*" is used for all kinds of algorithms which have in common the following strategy: Make a "locally most promising" decision how to process the next request.

In the sequence model the GREEDY principle amounts to serving the next request that is revealed to the online algorithm by such an action that has least service cost. In the time stamp model at any time GREEDY serves that request (among the yet unserved requests) next that can be served with the least cost with respect to the current system state. This is one extremal case of local optimization: GREEDY only decides upon the next request to be served, i.e., it does not plan into the future or does not consider the system state after the service. That means, even when no other request arrives, GREEDY is very likely to be sub-optimal. Moreover, GREEDY does not take into account possible future requests.

Although the above GREEDY-strategy is very shortsighted and the solutions produced maybe sub-optimal it is very popular because it is

- easy to implement,
- usually real-time compliant, and
- it produces a stable, predictable behavior since no decision is revised.

However, if cost-efficiency is the main-goal one usually needs a more sophisticated approach.

4.3 Replan

The REPLAN-strategy for an online problem in the time stamp model assumes that we have a method that computes an optimal (or almost optimal) solution to the static optimization problem (the corresponding *offline-problem*) at a specific point in time. Note that, in a realistic environment, this imposes the restriction of real-time compliance on the algorithms used to compute the optimum of the offline-problem (see Section 3).

While the GREEDY-approach 4.2 acted as locally as one could think, for REPLAN we find the other extreme case: at any time REPLAN tries to be "as globally optimal as possible", given the information it has at that point.

More specific: REPLAN maintains a "plan" containing the information on how to serve the already known requests. This plan is followed as long as no relevant event happens. Whenever a relevant event happens (a new request arrives, a change of the system state gives rise to a new cost of the current solution, etc.), REPLAN computes a cheapest solution of all known request in the current system state. Due to its nature, REPLAN is also called REOPT in the literature.

At any point in time we compute an optimal solution that is globally optimal at that particular moment. However, with respect to the complete instance the current solution is yet only locally optimal. Whenever a new request arrives the plan maybe

revised, and the global efficiency of the old plan is never really exploited since only the first couple of requests have been served according to that plan.

A more serious problem, however, is the fact that REPLAN can completely revise all decisions for which this is still possible. This often leads to an unpredictable behavior over time. One can even produce "oscillating" solutions. This means the following: assume, e.g., at some point in time, we find an optimal solution serving some request r of type A before another request r' of type B. Before we can serve r, a new request of type B arrives. Now the optimal plan may suggest to serve r' prior to r. But then there might arrive a new request of type A changing the plan back, and so on.

4.4 Ignore

The IGNORE-strategy also assumes that we are working in the time stamp model and that we have a way of computing (sub-) optimal solutions to the static (offline) version of the problem. The main idea of this method is to make sure that the efficiency of an optimal offline-solution computed at a certain point in time be exploited completely. More important even: once we computed an optimal plan it is absolutely predictable how the system will work in the near future.

The way it works is the following: IGNORE again maintains a plan. In contrast to REPLAN, the strategy IGNORE will stubbornly serve requests according to this plan until the plan is finished. Upcoming requests are temporarily ignored and collected in a buffer. When the current plan is finished IGNORE computes a new plan optimizing the service of all not yet served requests.

Although IGNORE might give away optimization potential by temporarily ignoring requests it still exploits optimization. Moreover, the upcoming requests that fit "very well" into the old plan (i.e., with no cost) can be incorporated with no harm.

We illustrate the general purpose strategies FIFO, GREEDY, IGNORE and REPLAN for the Online Traveling Salesman Problem:

Example 23 (Online TSP Revisited). Applying FIFO to the Online Traveling Salesman Problem leads to a tour that visits all cities in the order of appearance. If not required by other constraints this is certainly not the best choice.

The GREEDY-heuristic for the Online Traveling Salesman Problem means the following: At any point in time visit the closest city next. It is known that this can lead to a very inefficient solution. In some practical applications of the Online Traveling Salesman Problem, however, the experience shows that even this simple heuristics is acceptable.

Whenever a new city becomes known the REPLAN-heuristic computes an optimal tour (according to the objective function used to model the cost) visiting all cities known so far. This tour is followed until the next city pops up.

The observed performance depends heavily on the application. In practical instances it maybe necessary to cope with the problem of *system break-downs*: the salesman has to interrupt his work at some point. During the break the number of unserved requests increases, and so does the gain of offline-optimization of all

unserved requests. For instance, an automatic storage system where transportation tasks are served by a stacker crane can be modeled as an asymmetric Online Traveling Salesman Problem (see [4, 5]). In this specific application unexpected system break-downs of the automatic storage system may occur. During the forced idle period of the server a lot of requests pile up. All these requests can be taken into account by REPLAN when the server resumes. Thus, it is plausible that REPLAN yields a good *recovery method.*

The effort to get the necessary offline-solutions is usually large and not always real-time compliant. This, however, could be achieved in cases where good approximation algorithms (see Section 3.3) exist, like in the metric case (see Example 21).

The method IGNORE waits for the first city to be "released". Then it moves its salesman to that city. Once arrived, it computes an optimal tour through all the cities that have been released during the time the salesman was underway. Then this tour is completely traveled. At the end of the tour, the cities that have become known in the meantime are planned.

Again, the success of this method in realistic systems modeled by variants of this problem is application dependent. Simulation experiments show that in single server systems there usually is a substantial gain in stability and predictability of the system behavior over REPLAN. □

4.5 Chasing the Offline Optimum and Balancing Costs

Suppose that there are n (system-) states $s_1, \ldots, s_n$ in which an algorithm can be and that the service cost for a request depends (only) on the current state. Moreover, there is a cost for changing states. (This situation can be stated more formally as a *Metrical Task System*, see [15]).

Upon arrival of a new request r_i, the strategy of *chasing the offline optimum* changes to that state s_j in which the offline optimum for the sequence $r_1, \ldots, r_i$ would process r_i. A *balancing costs* type algorithm would change from the current state s to that state s' which minimizes some function of the following two values: (i) the charge for changing from s to s', and (ii) the cost of serving r_i in s'. The most famous representative of the latter class is the *work function algorithm* which has been successfully applied to the theoretical analysis of the Paging Problem and the k-Server Problem [15, 35].

5 SIMULATION

One can view simulation as a method of checking industrial system layouts and associated algorithms by an organized sequence of computer based experiments and evaluations. This takes place, of course, on the border line of mathematics and engineering. Therefore, we cannot hope for exact mathematical definitions of all relevant objects in the realm of simulation.

In this section we informally describe the method of *discrete event based* simulation and address issues that may come up during the process of modeling and computing. More elementary information can be found in [45].

5.1 Why Simulation?

The theoretical background surveyed in Section 2 leads to mathematical problems of substantial difficulty, even for seemingly easy online optimization problems. On the other hand, the performance guarantees achieved by these methods are often very poor. This renders competitive analysis problematic for most industrial purposes. In this case, evaluation and comparison of the practical performance of online algorithms are necessary.

There are new theoretical developments – one of them in this volume – that provide some improvements in this area; the final decision about which algorithm to choose in practice, however, is usually done on the basis of simulation experiments.

5.2 Discrete Event Based Simulation

Simulating an aspect of the real world on a computer requires a quantitative definition of the relevant part of the real world. This is referred to as the *system*. The system may consist of several *components*. In order to investigate waiting time distributions in a supermarket consider, e.g., the check-out area in that supermarket as the system. This system consists of several cashiers, waiting queues, etc.

The part of the real world outside the system is usually called *environment*. Sometimes there is a feed-back between system and environment, and it is at times a difficult modeling issue to find a suitable separation. The system interacts with the environment by producing an output of the system for an input of the environment.

In the area of online optimization we are usually concerned with *dynamic systems*, i.e., the system parameters change over time. For example, the lengths of the lines at the cashiers in the supermarket are not constant. Moreover, in the realm of combinatorial online optimization it is usually possible to find discrete points in time where the system changes its state. Such systems are called *time-discrete*. In the sequel we restrict ourselves to time-discrete systems.

A *simulation model* is a translation of the relevant parameters of the system into mathematical language so that the behavior of the system over time can be investigated by a computer calculation. In this step it is necessary to specify the components and their *attributes* that one would like to keep track of. Very important attributes are *strategies* or *algorithms* that hold information about how components react on system events. Some attributes are time-dependent, some are not. In the supermarket example we could specify a component "cashier" and a component "customer". The attributes for a cashier, e.g., could be *open/closed*, *operator speed*, *length of line*.

The changes of a time-discrete system over time is described by *Events*. First, an event specifies a system transition function that assigns to every possible system

state a new system state. Second, it defines a successor function that assigns to every system state a set of succeeding events together with their time of occurrence.

In the supermarket example the event "customer arrives in line at cashier i" can be formalized as follows: for all current system states the new system state incorporates the following changes: the queue at cashier i contains a new customer, and the set of succeeding events is empty.

The event "customer is being served at cashier i" can be defined as follows: the customer is no longer in the corresponding queue, and there is one successor event, namely "customer leaves the system" in five seconds times number of items in shopping cart from now.

Simulation means computing the output of the system (over time) for a given input (over time) of the environment. In *Discrete Event Based Simulation* this is done by dynamically *processing events*, i.e., computing the system states and the successor events until no events are left or a specified time is over. To start the simulation one uses environment input events modeling the input of the environment to the system.

An example of a discrete event based simulation system is the library AMSEL [3]. It was used in the investigations in [29].

5.3 *Issues for the Practitioner*

The quality of an evaluation of algorithms by means of simulation experiments heavily depends on the input data used. The following ways of generating input data are common:

- Generate data according to a probability distribution (random data).
  Advantages: It is possible to generate an arbitrarily large set of test data.
  Draw-backs: A realistic probability distribution maybe hard to come by.
- Compile data in the system under consideration.
  Advantages: One can adjust parameters of the simulation model by comparing the outcome of the simulation experiments with the outcome in the real world operation.
  Draw-backs: Compiling the data is extremely time-consuming, often it is not clear whether the compilation contains typical or unusual data.

Although we are advertising here the use of simulation for the performance evaluation of online algorithms we are aware of the fact that simulation experiments may be misguiding. It is a nontrivial matter to come up with meaningful and representative simulation tests.

6 Conclusion

More and more industrial decision makers appear to understand the issues coming up in online and real-time systems. Solution techniques are requested in a range of applications which will certainly improve research and development in online and real-time algorithms.

We have introduced competitive analysis as a mathematical method for the evaluation of combinatorial online-algorithms resulting in provable performance guarantees. A shortcoming of this approach is that it does not take into account the real-time requirements that are present in many real-world systems. Moreover, for complex systems and complicated algorithms a rigorous competitive analysis is in most cases impossible.

Thus, using this method on elementary problems that are similar to the given complex problems seems to be the right utilization: it is possible to get an idea about what kind of strategies are promising for real-world systems and why.

There are new developments in the area of theoretical evaluation of online-algorithms [30]; this field is, however, still in its childhood.

Most online-strategies caring about cost efficiency employ offline-algorithms. Here the need for real-time compliant methods is apparent. Theoretical concepts to get a hand on the issue that a solution is computed under circumstances that might have changed when the computation finishes are not yet available. Some achievements are presented in this volume.

After all, up to now there is no way to replace the experience in simulation experiments completely by a purely theoretical concept for evaluation of combinatorial online-algorithms.

References

1. S. Albers, *Better bounds for online scheduling*, Proceedings of the 24th Annual ACM Symposium on the Theory of Computing, 1997, pp. 130–139.
2. S. Arora, *Polynomial-time approximation schemes for euclidean TSP and other geometric problems*, Proceedings of the 38th Annual IEEE Symposium on the Foundations of Computer Science, 1997, pp. 2–11.
3. N. Ascheuer, *Amsel – a modelling and simulation environment library*, Online-Documentation available[5].
4. N. Ascheuer, *Hamiltonian path problems in the on-line optimization of flexible manufacturing systems*, Ph.D. thesis, Technische Universität Berlin, 1995.
5. N. Ascheuer, M. Grötschel, N. Kamin, and J. Rambau, *Combinatorial online optimization in practice*, Optima – Mathematical Programming Society Newsletter (1998), no. 57, 1–6.
6. N. Ascheuer, S. O. Krumke, and J. Rambau, *Online dial-a-ride problems: Minimizing the completion time*, Proceedings of the 17th International Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, vol. 1770, Springer, 2000, pp. 639–650.
7. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi, *Complexity and approximation. combinatorial optimization problems and their approximability properties*, Springer, 1999.
8. G. Ausiello, E. Feuerstein, S. Leonardi, L. Stougie, and M. Talamo, *Competitive algorithms for the traveling salesman*, Proceedings of the 4th Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science, vol. 955, August 1995, pp. 206–217.

[5] `http://www.zib.de/ascheuer/AMSEL.html`

9. G. Ausiello, E. Feuerstein, S. Leonardi, L. Stougie, and M. Talamo,, *Algorithms for the on-line traveling salesman*, Algorithmica (2001), To appear.
10. B. Awerbuch, Y. Bartal, and A. Fiat, *Distributed paging for general networks*, Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, 1996, pp. 574–583.
11. R. Battiti and G. Tecchiolli, *The reactive tabu search*, ORSA journal on computing **6** (1994), no. 2, 126–140.
12. R. E. Bixby, M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling, *MIP: Theory and practice closing the gap*, ILOG Technical Report, Presented at 19th IFIP TC7 Conference on System Modelling and Optimization, Cambridge, England, July 1999.
13. U. Blasum, M. R. Bussieck, W. Hochstättler, H. H. Scheel, and T. Winter, *Scheduling trams in the morning*, Mathematical Methods of Operations Research **49** (1999), no. 1, 137–148.
14. M. Blom, S. O. Krumke, W. E. de Paepe, and L. Stougie, *The online-TSP against fair adversaries*, Proceedings of the 4th Italian Conference on Algorithms and Complexity, Lecture Notes in Computer Science, vol. 1767, Springer, 2000, pp. 137–149.
15. A. Borodin and R. El-Yaniv, *Online computation and competitive analysis*, Cambridge University Press, 1998.
16. A. Borodin, S. Irani, P. Raghavan, and B. Schieber, *Competitive paging with locality of reference*, Proceedings of the 23th Annual ACM Symposium on the Theory of Computing, 1991, pp. 249–259.
17. A. Borodin, S. Irani, P. Raghavan, and B. Schieber, *Competitive paging with locality of reference*, Journal of Computer and System Sciences **50** (1995), 244–258.
18. A. Borodin, J. Kleinberg, P. Raghavan, M. Sudan, and D. P. Williamson, *Adversarial queueing theory*, Proceedings of the 23rd Annual ACM Symposium on the Theory of Computing, 1996, pp. 376–385.
19. A. Brooke, D. Kendrick, A. Meeraus, and R. Raman, *GAMS - a user's guide*, GAMS Development Corporatio, 1998.
20. E. Çela, *The quadratic assignment problem. theory and algorithms*, Kluwer Academic Publishers, Dordrecht, 1998.
21. N. Christofides, *Worst-case analysis of a new heuristic for the traveling salesman problem*, Tech. report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA, 1976.
22. W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver, *Combinatorial optimization*, Wiley Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons, 1998.
23. J. K. Lenstra E. H. L. Aarts, *Local search in combinatorial optimizatio*, Wiley, 1997.
24. A. Fiat and G. J. Woeginger (eds.), *Online algorithms: The state of the art*, Lecture Notes in Computer Science, vol. 1442, Springer, 1998.
25. R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A modeling language for mathematical programming*, Duxbury Press, Brooks/Cole Publishing Company, 1993.
26. M. R. Garey and D. S. Johnson, *Computers and intractability (a guide to the theory of NP-completeness)*, W.H. Freeman and Company, New York, 1979.
27. R. L. Graham, *Bounds for certain multiprocessing anomalies*, Bell System Technical Journal **45** (1966), 1563–1581.
28. Ronald L. Graham, *Bounds on multiprocessing timing anomalies*, SIAM Journal on Applied Mathematics **17** (1969), 263–269.
29. M. Grötschel, S. O. Krumke, and J. Rambau, *Forschungsartikel*, ch. This book, Springer, 2001.

30. D. Hauptmeier, S. O. Krumke, and J. Rambau, *The online dial-a-ride problem under reasonable load*, Proceedings of the 4th Italian Conference on Algorithms and Complexity, Lecture Notes in Computer Science, vol. 1767, Springer, 2000, pp. 125–136.
31. D. S. Hochbaum (ed.), *Approximation algorithms for NP-hard problems*, PWS Publishing Company, 20 Park Plaza, Boston, MA 02116–4324, 1997.
32. S. Irani, A. Karlin, and S. Phillips, *Strongly competitive algorithms for paging with locality of reference*, Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms, 1992, pp. 228–236.
33. A. Karlin, M. Manasse, L. Rudolph, and D. D. Sleator, *Competitive snoopy caching*, Algorithmica **3** (1988), 79–119.
34. E. Koutsoupias and C. Papadimitriou, *Beyond competitive analysis*, Proceedings of the 35th Annual IEEE Symposium on the Foundations of Computer Science, 1994, pp. 394–400.
35. E. Koutsoupias and C. Papadimitriou, *On the* k*-server conjecture*, Journal of the ACM **42** (1995), no. 5, 971–983.
36. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys (eds.), *The traveling salesman problem*, Wiley-Interscience series in discrete mathematics, John Wiley & Sons, 1985.
37. J. S. B. Mitchell, *Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric tsp,* k*-mst, and related problems*, SIAM Journal on Computing **28** (1999), no. 4, 1298–1309.
38. R. Motwani, *Lecture notes on approximation algorithms: Volume I*, Tech. Report CS-TR-92-1435, Department of Computer Science, Stanford University, Stanford, CA 94305-2140, 1992.
39. R. Motwani and P. Raghavan, *Randomized algorithms*, Cambridge University Press, 1995.
40. C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization*, Prentice-Hall, Inc., 1982.
41. C. Phillips, C. Stein, E. Torng, and J. Wein, *Optimal time-critical scheduling via resource augmentation*, Proceedings of the 29th Annual ACM Symposium on the Theory of Computing, 1997, pp. 140–149.
42. K. Pruhs and B. Kalyanasundaram, *Speed is as powerful as clairvoyance*, Proceedings of the 36th Annual IEEE Symposium on the Foundations of Computer Science, 1995, pp. 214–221.
43. C. R. Reeves, *Modern heuristic techniques for combinatorial problems*, McGraw-Hill, 1995.
44. S. Séguin, J.-Y. Potvin, M. Gendreau, T. G. Crainic, and P. Marcotte, *Real-time decision problems: An operational research perspective*, Journal of the Operational Research Society **48** (1997), 162–174.
45. H.-J. Siegert, *Simulation zeitdiskreter systeme*, Oldenbourg, München, Wien, 1991.
46. D. D. Sleator and R. E. Tarjan, *Amortized efficiency of list update and paging rules*, Communications of the ACM **28** (1985), no. 2, 202–208.
47. D. Steenken, T. Winter, and U. T. Zimmermann, *Stowage and transport optimization in ship planning*, (2001).
48. V. Vazirani, *Approximation algorithms*, Springer, 2001.
49. A. P. A. Vestjens, *On-line machine scheduling*, Ph.D. thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1994.
50. T. Winter, *Online and real-time dispatching problems*, Ph.D. thesis, Technical University Braunschweig, 1999.

51. T. Winter and U. T. Zimmerman, *Real-time dispatch of trams in storage yards*, Annals of Operations Research **96** (2000), 287–315.