## ONLINE OPERATION

A computational problem is said to be *online* when irrevocable decisions have to be made about the output without having complete information about the input. In other words, the input data are processed as they become available, and the output data are produced in an ongoing manner depending on the input data processed so far. The output is immediately produced "without seeing the future," that is, with no knowledge of the entire input. Once produced, the output cannot be altered.

The question of making good but irrevocable decisions based on partial information often arises in the fields of computer science and engineering, especially when certain processes or systems are being controlled directly by a computer. For example, online problems have application in investment analysis, bin packing, resource allocation, processor scheduling, network routing, storage allocation, cache management, maintenance of data structures and databases, robot motion planning, file migration, facilities location, capacity expansions of networks, and navigating on the World Wide Web. A classical application of online problems arises in most interactive systems, where a human being and a computer interchange information between themselves in an almost conversational manner. This explains why online systems are often confused with interactive systems.

Online problems are the opposite of *offline* problems, where the input data are not processed as they become available, but rather are collected and held until a convenient later time for processing. Therefore, the whole output data are produced based on the whole input data.

Online problems are sometimes confused with *real-time* problems. In a real-time problem, the correctness of the output relies not only on its logical result, but also on the time at which the result is available. In other words, there are critical time requirements on the output which must be met in order to avoid catastrophic crashes (e.g., in nuclear plant control or space shuttle flight control). Of course, there are many real-time problems that are also online problems, and the time aspects of the operations are not necessarily critical in all online problems. For instance, in an airline seat reservation, it matters little whether the response time for the output be of a microsecond or of several seconds.

## ONLINE ALGORITHMS

There are many computational problems in computer science and engineering that are inherently online. Thus the design and analysis of *online algorithms* (1) has recently received an increasing attention from the scientific community. However, the classical *worst-case time-complexity* analysis (2) often fails for analyzing the performance of online algorithms. Indeed, for most online problems, one can find an arbitrarily bad sequence of input data for any online algorithm that forces the algorithm to have an arbitrarily poor performance in the worst case. There are two main techniques for analyzing the performance of online algorithms. The first, earlier technique consists of measuring the running time of the algorithm, assuming that the input sequence is generated according to some fixed distribution (3), and selecting the algorithm that minimizes the expected cost incurred in processing the input sequence. In this way, however, the choice of a particular input sequence distribution instead of another distribution is often arbitrary, while the choice among the online algorithms that solve the problem depends heavily on the choice of the input distribution itself. The second, recently introduced technique employs *competitive analysis* (4) as a way of doing meaningful worst-case analysis, by comparing an *online algorithm* with an optimal offline algorithm that solves the same problem, where an optimal offline algorithm is an algorithm that for each input sequence minimizes the cost incurred for processing the sequence. This technique does not assume any input distribution at all and allows a fair worst-case analysis to be done, since for some inherently bad input sequences for which the online algorithm performs poorly also the optimal offline algorithm performs poorly.

### Competitive Analysis

Competitive analysis was devised by Sleator and Tarjan (4) for analyzing online algorithms performing a sequence of operations on dynamic data structures. Let $A$ be an online algorithm and $S$ an input sequence. Let $A(S)$ denote the cost paid by algorithm $A$ in processing the input sequence $S$, and let $\mathrm{OPT}(S)$ denote the cost paid by an optimal offline algorithm on the same input $S$. Algorithm $A$ is said to be *c-competitive* if, for all $S$,

$$A(S) - c \cdot \mathrm{OPT}(S) \tag{1}$$

is bounded by a constant. The infimum of $c$ for which $A$ remains $c$-competitive is called *competitiveness* of $A$. In practice, an online algorithm is compared in an input-by-input manner with the best algorithm which can see the whole input sequence in advance, and the extra cost due by processing the input sequence online is evaluated. The concept of competitiveness is related to the concept of regret in game theory. Indeed, one can see the scenario as a game between an online player, who wants to select the online algorithm, and an adversary, who chooses the input sequence so as to maximize the cost of the online algorithm with respect to the optimal offline algorithm. From this point of view, the competitiveness concept is a pessimistic one, since it assumes that the input sequence is chosen by an adversary knowing the future. Indeed, *lower bounds* on the competitiveness are usually proved by employing adversary arguments, while *upper bounds* are usually derived by analyzing online algorithms by means of proper analysis techniques.

**Amortized Analysis.** In analyzing the competitiveness of many online algorithms, an analysis technique called *amortized analysis* (2,5) is often used. Amortized analysis was first devised for analyzing the cost of a sequence of operations on data structures performed by offline algorithms. Later, it has been applied also in conjunction with competitive analysis for analyzing online algorithms. However, the use of one technique does not necessarily imply the use of the other technique. Using an amortized analysis, the cost for performing a sequence of operations is averaged over all the operations in the sequence. Thus, the average cost per operation in the sequence can be proved to be small, even if the cost of a single operation in the sequence can be high in the worst case. It is worth noting that no stochastic arguments are invoked, since an amortized analysis guarantees, over all sequences, the average cost per operation in the worst case.

A common method employed in amortized analysis makes use of a *potential function,* which maintains the "potential energy" of the system. This potential represents the prepaid credit, accumulated by earlier overcharged operations, which can be used to compensate later undercharged operations. Formally, assume that a sequence of $n$ operations has to be performed on the system. A potential function $\Phi$ maps each system configuration to a real number. Let $t_i$ and $\Phi_i$ be, respectively, the actual cost of the $i$th operation and the value of the potential function after the $i$th operation is performed, for each $i = 1, 2, \ldots, n$. The *amortized cost* $a_i$ of the $i$th operation is the actual cost of the operation plus the increase in potential due to the operation:

$$a_i = t_i + \Phi_i - \Phi_{i-1} \tag{2}$$

The overall amortized cost of the sequence of $n$ operations is, by Eq. (2):

$$\textstyle\sum_i a_i = \sum_i (t_i + \Phi_i - \Phi_{i-1}) = \sum_i t_i + \Phi_n - \Phi_0 \tag{3}$$

where $\Phi_n$ and $\Phi_0$ are the potential of the final and initial configuration, respectively. If $\Phi_n - \Phi_0 \geq 0$, then the overall amortized cost of the sequence upper bounds the overall actual cost of the sequence. Observe that, if $\Phi_i - \Phi_0 \geq 0$ for all $i$, then one is guaranteed to always pay in advance the cost of each operation.

**An Example.** As an example of application of competitive analysis in conjunction with amortized analysis, consider the *list update* problem (4). In this problem, the input is a sequence of operations that have to be performed on an unordered list of elements. Each operation consists in accessing, inserting, or deleting an element. To access an element, one must linearly scan the list from the front. Thus the cost for accessing an element is equal to one plus the number of elements that precede the accessed element. While scanning the list, one can maintain a pointer to the place where the element has to be eventually moved as soon as it is found. Thus, once an element is accessed, it can be moved nearer to the front of the list in constant time. An element can also be moved anywhere at any time, and the cost is equal to the distance between the old place and the new place of the element in the list. Moreover, any old element can be deleted from the list, and any new element can be inserted into the list. The cost for a deletion/insertion of an element is also equal to one plus the number of elements that precede the deleted/inserted element.

Sleator and Tarjan have proposed the *move-to-front* (MF) online algorithm for the list update problem. The algorithm consists in moving the accessed element to the front of the list as soon as the element is accessed. Sleator and Tarjan proved that MF is a 2-competitive online algorithm, by means of a useful potential function. Assume that both MF and OPT, the optimal offline algorithm, while processing the input operations, maintain their lists of elements. Whenever the two lists are identical, any operation will cause both MF and OPT to have the same cost. Therefore, OPT has the potential to outperform MF only when the two lists are different. This suggests to define a potential function which gives the number of inverted pairs of elements, where two elements are inverted if the order in which they appear in the list produced by MF is different from the order in which they appear in the list produced by OPT. Using this potential function, it is possible to show that for each operation (i.e., access, insertion, and deletion) the amortized cost of MF is no more than twice the cost of OPT. By definition, the potential function is nonnegative. Moreover, the potential is initially 0 whenever both MF and OPT start with the same list. Consider, for instance, an access operation. Assume that the accessed element appears as the $j$th element in the OPT list after the access. Therefore, the cost of the optimal offline algorithm is at least $j$. Let the accessed element appear as the $h$th element in the MF list before the access. Finally, let $m$ be the number of elements that precede the accessed element in the MF list but follow it in the OPT list. Thus, the number of elements that precede the accessed element in both lists is $h - 1 - m$. When the accessed element is moved to the front by MF, $h - 1 - m$ inversions are added and $m$ inversions are eliminated. Thus the amortized cost of MF for an access operation is

$$h + (h - 1 - m) - m = 2(h - m) - 1 \leq 2j - 1 \tag{4}$$

Indeed, $h - m \leq j$, since of the $h - 1$ elements preceding the accessed element in the MF list only $j - 1$ elements precede it in the OPT list. Therefore, no more than twice the cost of the optimal offline algorithm is paid by the MF online algorithm.

## RELEVANT ONLINE APPLICATIONS

Some of the most relevant application areas in computer science and engineering are presented here for the study of online algorithms. Five selected computational problems are considered: (1) task systems, (2) $k$-servers, (3) paging, (4) graph coloring, and (5) real-time scheduling. Other problems can be found in (1).

### Task Systems

A *task system* is described by a set of $m$ states and an $m \times m$ distance matrix $d$. The distance $d_{ij}$ represents the cost incurred for a transition from state $i$ to state $j$, where it is assumed that $d_{ij} \geq 0$ for all $i$ and $j$, $d_{ii} = 0$ for all $i$, and $d_{ij} + d_{jk} \leq d_{ik}$ for all $i, j, k$. The input of the problem is a sequence $\boldsymbol{T}_1, \boldsymbol{T}_2, \ldots, \boldsymbol{T}_n$ of $n$ tasks, where each task $\boldsymbol{T}_j$ is a vector of $m$ components such that the $i$th component $\boldsymbol{T}_{ji}$ represents the cost of executing the task $\boldsymbol{T}_j$ in state $i$. The objective is to find a schedule for the sequence of tasks, namely, a state s($j$) in which to execute task $\boldsymbol{T}_j$, for $j = 1, 2, \ldots, n$, so as to minimize the overall cost due to the state transitions and task executions:

$$\textstyle\sum_j d_{s(j-1)s(j)} + \sum_j \boldsymbol{T}_{js(j)} \tag{5}$$

where s(0) is the initial state of the system. An on-line algorithm receives each task $\boldsymbol{T}_j$ one at a time, and must determine the state s($j$) in which to execute the task depending its decisions on the past tasks $\boldsymbol{T}_1, \ldots, \boldsymbol{T}_{j-1}$ only, but not on the future tasks $\boldsymbol{T}_{j+1}, \ldots, \boldsymbol{T}_n$. When the distance matrix is symmetric, that is, $d_{ij} = d_{ji}$ for all $i$ and $j$, a *metric* space arises. In this case, Borodin, Linial, and Saks (6) devised an optimal $2m - 1$ competitive online algorithm. The next problem is a special case of the task system problem having the nice property that a competitiveness independent of the number $m$ of states can be achieved.

### *k*-Servers

In the $k$-server problem, there are a metric space and $k$-servers which are free to move in the space. The servers are initially located at given points in the space. The input of the problem consists in a sequence of requests, that is, of points in the space to be served. Of course, a server remains stationary unless it is selected to move to a request point. The objective is to serve each request by sending any server to each requested point so as to minimize the overall distance traveled by the servers. The *k-server* problem is perhaps the most studied of all online problems. In spite of its apparent simplicity, it is not easy to achieve a good competitiveness for the $k$-server problem. For instance, a simple *greedy* algorithm, which moves the closest server to each request point, can be readily devised to solve the problem. However, such a greedy algorithm performs poorly when the requests are alternated between two sufficiently close points. The greedy algorithm will serve the two points by moving the same server forth and back forever, thus achieving an unbounded cost, while in the same situation an optimal offline algorithm would maintain two stationary servers at the two points.

Manasse, McGeoch, and Sleator (7) proved that if the metric space has at least $k + 1$ points, then the competitiveness of an online algorithm is at least $k$, and devised optimal $n - 1$ and 2-competitive algorithms when $k$ is equal to $n - 1$ and 2, respectively, where $n - 1$ is the number of points in the space. The $k$-server problem has been extensively studied and a very good performance for a very simple online algorithm, called *harmonic* algorithm, was proved by Grove (8). The harmonic algorithm is a *randomized* algorithm, that is, one which tosses a coin during its execution. The impredictability due to randomization often makes it more difficult for an adversary to construct bad input sequences. Indeed, the harmonic algorithm tends to favor servers that are close to the request points, but eliminates the predictability of the simple greedy algorithm. The main property of the harmonic algorithm is that, at each step, the probability that a given server is the one to move is inversely proportional to the distance of that server from the request point. The next problem is a special case of the $k$-server problem that arises when the distance between any pair of points is one.

### Paging

In the paging problem, there is a fast memory which may contain $k$ pages, and a slow memory with unlimited page capacity, and the input consists in a sequence of $n$ page requests. If a requested page does not reside in the fast memory, then one resident page has to be replaced by the requested page. The objective is to serve all the page requests by minimizing the number of page replacements. Sleator and Tarjan (4) showed a lower bound of $k$ on the competitiveness of any *deterministic* algorithm, that is, one not employing randomization. Their proof is based on an adversary argument. Assume there is a set of $k + 1$ pages to be maintained in the memory, with $k$ pages resident in the fast memory and 1 page in the slow memory. The adversary can produce a bad input sequence, which causes any deterministic algorithm to incur in a page replacement after every request. In contrast, for any input sequence, an optimal offline algorithm can see the entire sequence and ensure that at least $k$ requests occur between two consecutive page replacements by replacing the page resident in the fast storage for which the next request will be the latest in the future. Sleator and Tarjan also proved an upper bound of $k$ on the competitiveness of a widely used online algorithm. Indeed, they showed that the *least-recently-used* (LRU) algorithm, which consists in replacing the least recently requested page resident in the fast memory, is a $k$-competitive algorithm. Since $k$ is a lower bound on the competitiveness of any deterministic online algorithm, as seen above, the LRU algorithm is optimal with respect to the competitiveness measure. It is worth noting, however, that under certain restricted hypotheses, randomized online algorithms are more powerful than deterministic online algorithms for the paging problem (1), since they can achieve a competitiveness slightly better than $k$. Moreover, it is important to observe that the competitiveness of $k$ for the LRU algorithm is a worst-case bound. In practice, the LRU algorithm performs very well and requires indeed much less than $k$ times the optimal number of replacements.

### Graph Coloring

In the graph coloring problem, there is an undirected graph $G = (V, E)$, with $V$ being the set of *vertices* and $E$ the set of *edges* (i.e., pairs of vertices), and a set of colors. Each vertex $v$ of the graph has to be assigned a color different from the colors assigned to its *neighbor* vertices (i.e., all the vertices joined to $v$ by an edge). The objective is to color all the vertices using the minimum number of colors. The input of the online problem consists in a sequence of vertices given one at a time. When a vertex $v$ is given, all its edges to previously input neighbor vertices are also given, and $v$ has to be assigned a color before the next vertex is given. Applications of online graph coloring range from register allocation during compilation to channel assignment in wireless/mobile telecommunications.

Irani (9) considered the class of *d-inductive* graphs and analyzed the *performance ratio* of the *first-fit* (FF) algorithm. A $d$-inductive graph is a graph whose vertices can be numbered so that each vertex has at most $d$ higher numbered neighbor vertices. The FF algorithm assigns to each vertex $v$ the lowest numbered color not already assigned to any neighbor vertex of $v$. The performance ratio measures the number of colors used by the online algorithm in comparison to the minimum number $\chi$ of colors required by an offline algorithm. Irani proved that FF uses $O(d \log n)$ colors, where $n$ is the number of vertices, and that this upper bound is tight to within a constant factor. This result is strengthened for particular classes of $d$-inductive graphs, such as *planar* graphs and *chordal* graphs, which are 5-inductive and $\chi$-inductive, respectively. FF uses $O(\log n)$ colors for planar graphs and $O(\chi \log n)$ colors for chordal graphs, which yield a tight $O(\log n)$ upper bound on the performance ratio for both classes of graphs.

### Real-Time Scheduling

In a *real-time scheduling* problem, a set of real-time tasks is given, which has to be executed on one or more processors. Tasks may range from *periodic,* that is, recurring infinitely often according to a regular interarrival time, to *irregular,* that is, occurring only once at an unpredictable time, and may have time deadlines, that is to say, their execution must be completed before certain due dates. A task deadline can be either a *hard* deadline, if it has to be definitively met and missing it may lead to a catastrophic failure, or a *soft* deadline, if it is desiderable to meet it but missing it can occasionally be tolerated. The objective is to schedule all the tasks on the processor(s) so as to meet the deadlines.

The real-time scheduling problem has obvious applications in time-critical system control and was widely studied in the offline case. The scheduling algorithms used in practice are *priority-driven preemptive* algorithms. Priorities are assigned to tasks according to some policy. At each instant of time, the highest priority task ready to run is executed, preempting, if necessary, a lower priority task. The preempted task is suspended, and its execution is resumed later from the point of preemption. Two priority-driven algorithms, which are widely used when there is only one processor and all the tasks are periodic and have hard deadlines, are the *earliest-deadline-first* (EDF) and *rate-monotonic* (RM) algorithms (10–12). EDF assigns hightest priority to the ready task with the nearest deadline, while RM gives hightest priority to the task with the shortest period. When many processors are available, a common practice consists in partitioning the tasks among the processors, that is, by means of a first-fit or next-fit heuristic, and then scheduling the tasks assigned to each single processor using the EDF or RM algorithm.

## DISCUSSION AND FUTURE DIRECTIONS

The real-time scheduling problem just introduced represents perhaps the most relevant online application (13) and has some peculiarities with respect to the previous four problems, which are useful for discussing advantages and limitations of the present knowledge on the performance analysis of online algorithms.

First, the most important measure of merit to evaluate a real-time scheduling algorithm is *predictability,* namely, the ability to determine whether all the deadlines can be met. Useful parameters for predictability are (1) the degree of processor loading below which deadlines are guaranteed, (2) the latency of the system in responding to external events, and (3) the capability to meet the deadlines of the most critical tasks when it is not possible to meet all the task deadlines. Thus, although fast algorithms are, of course, helpful in satisfying the task deadlines, a predictability analysis is often used in conjunction with a worst-case time-complexity analysis. In this context, stochastic assumptions on the input distribution, as well as randomization in the online algorithm may be useless, since they cannot always guarantee that hard deadlines are met.

Second, in order to meet time deadlines, an on-line scheduling algorithm must make its decisions quickly and this could degrade its competitiveness. Indeed, although most of the on-line algorithms are efficient, a higher competitiveness is sometimes achieved when no computational restrictions are imposed.

Third, although mainly studied in the off-line setting thus far, most real-time scheduling problems are inherently online problems. Indeed, tasks usually occur infinitely many times and many of them are given one at a time. Moreover, since the purpose of real-time systems is to provide a time-critical control on its environment, a critical level of service must be guaranteed, even in the presence of hardware or software faults. Thus fault-detection and fault-recovery activities must also be managed online in order to tolerate faults. However, since the fault-tolerant real-time scheduling problem is inherently online, it could be better to directly compare online algorithms among them, instead of comparing each online algorithm to the best offline algorithm.

Fourth, as already pointed out also for the paging problem, competitive analysis is a theoretical worst-case analysis, which assumes that input sequences are generated by a fiendish adversary having unlimited computational power and complete knowledge of the future. In particular, this requires that the adversary has a complete knowledge of the algorithm to be defeated. Therefore, this kind of analysis can yield a too pessimistic evaluation of the performance of an algorithm with respect to its practical behavior.

From the above discussion, a *trade-off* arises involving (1) predictability, (2) competitiveness, (3) time-complexity, and (4) practical behavior of an online real-time scheduling algorithm. Therefore, a challenge for the future is to devise new measures for evaluating the performance of online algorithms which could balance these four factors and overcome the above-mentioned drawbacks.

## BIBLIOGRAPHY

1. R. M. Karp, On-line algorithms versus off-line algorithms: How much is it worth to know the future?, in J. van Leeuwen (ed.), *Algorithms, Software, Architectures—Information Processing 92,* Amsterdam: Elsevier, 1992, Vol. 1.
2. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms,* New York: MIT Press/McGraw-Hill, 1990.
3. G. S. Shedler and C. Tung, Locality in page reference strings, *SIAM J. Comput.,* **1**: 218–241, 1972.
4. D. D. Sleator and R. E. Tarjan, Amortized efficiency of list update and paging rules, *Commun. ACM,* **28**: 202–208, 1985.
5. R. E. Tarjan, Amortized computational complexity, *SIAM J. Algebr. Discrete Methods,* **6**: 306–318, 1985.
6. A. Borodin, N. Linial, and M. Saks, An optimal on-line algorithm for metrical task systems, *Proc. Annu. ACM Symp. Theory Comput.,* **19**: 373–382, 1987.
7. M. S. Manasse, L. A. McGeoch, and D. D. Sleator, Competitive algorithms for on-line problems, *J. Algorithms,* **11**: 208–230, 1990.
8. E. Grove, The harmonic on-line k-server algorithm is competitive, *Proc. Annu. ACM Symp. Theory Comput.,* **23**: 260–266, 1991.
9. S. Irani, Coloring inductive graphs on-line, *Algorithmica,* **11** (1): 53–72, 1994.
10. A. A. Bertossi and A. Fusiello, Rate-monotonic scheduling for hard-real-time systems, *Eur. J. Oper. Res.,* **96**: 429–443, 1997.
11. M. H. Klein, J. P. Lehoczky, and R. Rajkumar, Rate-monotonic analysis for real-time industrial computing, *Computer,* **27** (1): 24–33, 1994.
12. J. A. Stankovic and K. Ramamritham, *Hard-Real-Time Systems,* Los Alamitos, CA: IEEE Computer Society Press, 1988.
13. G. Koren and D. Shasha, $D^{over}$: an optimal online scheduling algorithm for overloaded uniprocessor real-time systems, *SIAM J. Comput.,* **24**: 318–339, 1995.

ALAN A. BERTOSSI
University of Trento

**OODBMS.** See OBJECT-ORIENTED DATABASES.
**OP AMP INTEGRATOR.** See INTEGRATING CIRCUITS.
**OP AMPS.** See OPERATIONAL AMPLIFIERS.